Problem Solving with Flowcharts and a Little Flavor of Programming with Python

Achla Agarwal
Krishna Agarwal
Laura Goadrich
Mark Goadrich

Preface

This book was written for students of any age who want to learn the concepts to enable them to program successfully. Programming is an art and this text will teach you the techniques and underlying logic that will form the foundation of future coding skills no matter what language you choose to program in. Because there are a variety of programming languages that are changing and evolving, it is important to create a solid skill-set that is adaptable to any programming language you choose to learn.

Today, computers are used in almost every aspect of our daily lives. From the alarm clock that wakes you in the morning, to the car or bus that takes you to and from school, computers are vital to maintain our current lifestyles. A computer at its core is just metal and silicon. It is the way that the metal and silicon is organized, in a logical manner that makes computers useful. One way to incorporate logic into a computer is through instructions in a program. A program is made up of instructions that execute a specific task in a language that a computer can understand.

Since programming is a multifaceted area, we use examples from a variety of areas to reach the interests of a large number of students. We focus our examples and problems on situations that arise every day. From painting a room to organizing employee information to examining DNA, we have included examples in many areas of Science, Technology, Engineering and Mathematics (STEM) to show the interconnectedness of computers.

The goal of creating a program is to solve a task. This task must be broken down into clear steps to reach your end goal. We focus on using flowcharts to allow the learner to visualize and group the steps in the programming process. Flowcharts are also helpful in organizing thoughts to create a clear task order.

In addition to flowcharts, we also focus on pseudocode and code tracing. While flowcharts are a wonderful visual tool, more complicated problems can be written in a more compact form using pseudocode. To clearly understand both the execution of the flowchart and pseudocode, it is important to be able to trace the code as well. We give detailed techniques for code tracing that will solidify the execution performed in the computer memory.

Each chapter has been organized to begin with an outline of topics needed from previous chapter sections for reference. Next we illustrate the reason for the chapter with a motivating example and then work through lessons dedicated to new concepts. Each lesson is filled with a variety of examples and ends with self-check exercises for the student to check their progress and understanding at each stage. After a summary of the main concepts at the end of the chapter, there are problem sets for more practice and application of the lesson topics.

Finally we incorporated Python, a free, straight-forward programming language, for students to test their programs on the computer. In our years of teaching, students always desired a more hands-on approach to learning programming logic and Python is a wonderful tool for applying what has been learned without requiring a programming background. Also Python can be installed in a few steps on any platform (see Appendix A for more information).

The text is outlined to cover techniques for solving problems in Chapter 1 and discusses ways to store and manipulate data in Chapter 2. The next chapter introduces flowcharts and pseudocode along with standard function references and applies these techniques to a programming language called Python. Chapter 4 augments the prior chapter using **print** and **get** statements to interact with a user, incorporating code tracing and debugging while Chapter 5 introduces operators and Boolean data types. The next chapter introduces decision and selection structures following with a detailed discussion about nested selection statements and case statements. Chapter 8 and 9 focus on looping structure and nested loops. The next two chapters look at functions and arrays. The last chapter covers file input and output.

Note that the three features, flowcharts, pseudocode, Python, can be pulled apart and used independently to meet the needs for all student and logic environments. Each chapter focuses on solving everyday problems. These problems include solutions in Python for the student and instructor to experiment with and demonstrate the chapter concepts. We feel that the combination of the three features gives a balanced understanding of programming logic and a good foundation for programming.

Table of Contents

1. Techniques for Solving Problems 1
1.1 Steps in solving a problem 1
1.2 Motivating example 2
Self-Check 1.1 3
1.3 Programming Skills 3
Key Terms 4
Exercises 4
Self-Check Solutions 6

2. Storing Data 7
2.1 Characters and Strings 7
Self-Check 2.1 7
2.2 Numerical Data Types 10
2.3 Numerical Operators 12
2.3.1 Addition and Subtraction 12
2.3.2 Multiplicative Operators 13
Self-Check 2.2 14
2.3.3 Negation 15
Self-Check 2.3 16
2.3.4 Exponentiation 16
2.4 Casting 16
Key Terms 17
Exercises 18
Self-Check Solutions 19

3. Pseudocode, Flowcharts and Python 20
3.1 Sequential Order 20
3.2 Pseudocode 20
3.3 Flowcharts 20
3.4 Python 21
Problem 3.1 21
Self-Check 3.1 23
Problem 3.2 26
Key Terms 29
Exercises 29
Self-Check Solutions 33

4. Input and Output 34
4.1 Pseudocode “print” and “get” 34
4.2 Constants 36
4.3 Problem 4.1 37
Self-check 4.1 38
Self-Check 4.2 39
4.4 Applying Python 40

4.4.1: Python “print” and ” input” 40
Key Terms 42
Exercises 42
Self-Check Solutions 44

5. Boolean Logic 47
5.1 Boolean Data Type 47
5.2 Relational Operators 47
5.3 Logical Operators 49
Self-Check 5.1 51
Self-Check 5.2 52
5.4 Assignment Operator 52
5.5 Equality Operators 53
Key Terms 54
Exercises 54
Self-Check Solutions 55

6. Selection Structures 56
6.1 If-Then-Else 56
6.2 Problem 6.1 56
6.3 Updating Flowcharts 57
6.4 Problem 6.2 60
6.5 Problem 6.3 62
Key Terms 65
Exercises 65

7. Nested If-Then-Else and Case Statements 68
7.1 Nested If-Then-Else 68
Problem 7.1 68
Self-Check 7.1 72
7.2 Case Statement 72
Problem 7.2 74
Self-Check 7.2 76
Key Terms 76
Exercises 76
Self-Check Solutions 79

8. Repetition Structures 84
8.1 While loops 84
Problem 8.1 85
Self-Check 8.1 91
Problem 8.2 92
8.2 For loops 96
Problem 8.3 97
Problem 8.4 99
8.3 Sentinels in while loops 101

Self-check 8.3 102
8.4 Do-While loops 102
Problem 8.5 104
8.5 Infinite Loops 106
Key Terms 106
Exercises 106
Self-Check Solutions 110

9. Nested Loops 114
Problem 9.1 114
Self-Check 9.1 118
Problem 9.2 119
Self-Check 9.2 121
Problem 9.3 122
Self-Check 9.3 124
Key Terms 124
Exercises 124
Self-Check Solutions 126

10. Functions 131
10.1 Function parts 131
10.2 Comments in functions 133
10.3 Value-returning functions 134
Problem 10.1 134
Problem 10.2 136
Self-Check 10.1 138
10.4 Void functions 138
Problem 10.3 139
Problem 10.4 140
Self-Check 10.2 142
Key Terms 142
Exercises 142
Self-Check Solutions 144

11. Arrays 146
Self-Check 11.1 147
Self-Check 11.2 148
Problem 11.1 148
Self-Check 11.3 154
Problem 11.2 155
11.2 Two-dimensional arrays 157
Self-Check 11.4 158
11.3 Multi-dimensional arrays 158
Exercises 159
Self-Check Solutions 161

12. File Input and Output 164
12.1 Pseudocode for File Operations 164
12.2 Problem 12.1 165
12.3 Problem 12.2 168
Self-Check 12.1 172
12.4 Problem 12.3 172
12.5 Problem 12.4 175
Exercises 179
Self-Check Solutions 182

Appendix A: Installing Python 185

References 187

Index 188

1. Techniques for Solving Problems

In life we encounter problems every day, from deciding what to wear in the morning to planning a trip for summer vacation. Some problems may be easier to solve than others because we have a preference (I love to eat eggplant parmesan. So, where should we eat tonight?) while others may be very taxing (I have only one stick of bubble gum and two friends. Who will get it?).

This chapter will give you the tools to become a great problem solver. We will look at formalizing the steps you can take to systematically arrive at a solution, and how you can use these skills to program a computer.

1.1 Steps in solving a problem

While every problem is unique, there are some core steps that you can follow to help you break down and solve any situation. The steps that are outlined here will be used throughout the entire text for you to frame your programming solutions.

After you have read or listened to the problem presented,

1. Identify your **input.**
   Input is the essential information that a program needs to solve the problem. The input may be given by an outside source (like the person using the program) or default values given by the programmer.

2. Identify the **goal** or **objective.**
   The goal is where you define your end result. This is a description of when you know your problem has been solved and your objectives have been met. Sometimes you may have more than one goal, so list each goal clearly.

3. Create a list of **tasks** to achieve your objective.
   These are the steps required to reach your goal. Make sure you enumerate your steps in order for clear execution. Also make sure each step is descriptive and clear. Your list of tasks will result in your goal from step 2.

When you have finished your tasks, make sure that you have answered the problem and met your goal(s). This is probably the most important part of the steps, but the easiest one to forget. Remember if your steps (from part 3) don't lead to your goal (from part 2) then you didn't solve the problem and you should start back at step 1.

1.2 Motivating example

Juanita is planning a trip to visit the Grand Canyon. Since she lives in Seattle, Washington, she is concerned about the cost of gas needed to make the trip. Using online mapping software, she calculated that it will take 1,224 miles to reach the Grand Canyon. Traveling on the highway, Juanita's car gets 18 miles per gallon (*MPG*). Estimating that the average cost of gas is \$4.20 per gallon (*CPG*), calculate how much money it will cost for her to travel to the Grand Canyon. Using the equation below, you can calculate the cost of travel by distance divided by MPG times CPG:

$$cost = distance \div MPG \times CPG$$

To solve this with our problem solving steps from section 1.1, we first need to identify the input (step 1):

distance: 1224 miles
MPG: 18
CPG: \$4.20

Now we can state our goal (step 2):

Find the cost

Lastly we list our tasks to reach our goal (step 3):

1. Plug the input(s) into the cost equation
2. Return the cost

Now that the problem is broken down into some clear steps, we can execute the tasks to reach our goal.

cost = 1224 / 18 * 4.20
cost = \$285.60

Self-Check 1.1

What is the purpose of the second step in solving a problem?

1.3 Programming Skills

One of the main goals of this textbook is to teach you the skills to become a good programmer. Given below, are some of the terms which a person needs to know in a programming environment.

A **programmer** is a person who creates programs that solve a problem using the computer.

A **program** is a sequence of steps that a computer understands and executes.

A **programming language** is a notation used to write instructions into a computer.

A **computer** is a logical device, created from silicon and metal, which runs on electricity.

An **algorithm** is a set of instructions designed to complete a task.

A **bug** is an error in a program.

Debugging is the process of removing errors, testing and revising a program to make sure that it performs as expected.

We encounter problems every day, many of them too complex to solve without help. By defining problems for a computer using the problem-solving components of inputs, goals, and tasks, we can start the process of programming a computer to assist us in finding solutions.

Programs have to satisfy clear rules so that a computer knows the exact specifications of the problem and solution. The rules, or **syntax**, that you use to program in will differ based on the type of programming language that you are using. In this book we will be using a programming language called

Python to apply the lessons you are learning. We will discuss more about programming techniques and Python in Appendix A and from Chapter 3 on.

Key Terms

Algorithm	Objective
Bug	Programmer
Computer	Programming language
Debugging	Python
Goal	Tasks
Input	Syntax

Exercises

For each of the problems below, identify input, goal and create a list of tasks to achieve your goal.

1. Reno, Nevada is a desert city supporting a population of over 180,480 people (in the 2000 census). The large population and desert location makes water a treasured resource. Many citizens of Reno do not own their own swimming pool because water is so dear; instead they visit the public swimming pools. If the city charges 1.2 cents per cubic feet of water, calculate the cost the city pays for water needed to fill a typical public swim pool if the pool is 20 feet wide, 30 feet long and 10 feet deep. Use the following equations:

   *Volume in cubic feet = length * width * height*
   *Cost = cost per cubic feet * Volume in cubic feet*

2. Describe how to open a bag of pretzels. Your tasks will be the actions to perform to reach your goal. Make sure you state your assumptions (i.e., what is the position of the bag and the hands). Try to limit your verbs you use in your tasks to *reach*, *grasp*, and *pull away*.

3. Jimmy works at the local air force base. On Wednesday, Jimmy worked from 8:12 hours until 16:38 hours with a lunch break from 12:02 hours until 12:24 hours. Calculate how long Jimmy worked on Wednesday.

   Time worked = ______

4. Describe how to sharpen a pencil using an electric pencil sharpener. Your tasks will be the actions to perform to reach your goal. Make sure you state your assumptions (i.e., what is the position of the pencil and the hand). Try to limit the verbs that you use in your tasks to *move, push, wait, remove, judge, toward, pull, go to*.

5. Describe how to fill a glass with water. Your tasks will be the actions to perform to reach your goal. Make sure you state your assumptions (i.e., what is the position of the glass, sink and hand). Try to limit the verbs that you use in your tasks to *move* and *hold*.

6. Describe how to make a peanut butter and jelly sandwich. Try to limit the number of verbs that you use to less than 10.

7. You have decided to repaint the four walls of your living room and need to know how many gallons of paint to buy. There is one window and one entrance to the room. Describe how you would calculate the number of gallons of paint needed taking the description of the room into account. Assume one gallon of paint covers 350 square feet.

   Hint: *Square feet = Length * Width*

8. Describe how to open a Kit-Kat bar wrapped in packaging. Your tasks will be the actions to perform to reach your goal. Make sure you state your assumptions. Try to limit the verbs that you use in your tasks to *tear* and *hold*.

9. Describe how to organize five books on a bookshelf in alphabetical order by author. Your tasks will be the actions to perform to reach your goal. Make sure you state your assumptions. Try to limit the verbs that you use in your tasks to *pick-up* and *place*.

10. Describe how to blow up a rubber balloon. Your tasks will be the actions to perform to reach your goal. Make sure you state your assumptions.

Try to limit the verbs that you use in your tasks to *blow, breathe* and *hold*.

For extra credit, describe how to create a balloon animal in the shape of a dog. Your tasks will be the actions to perform to reach your goal. Make sure you state your assumptions. Try to limit the verbs that you use in your tasks to *blow, breathe, twist* and *hold*.

Self-Check Solutions

1.1: The goal step states the end result that is accomplished via your tasks.

2. Storing Data

The computers that we use today were developed mainly in the 1950's. Computers are electrical devices built of transistors and switches, which follow specific logical rules. To be able to program a computer, you will need to understand these rules. In this chapter we will study the different ways information is stored in a computer, including alphabetical and numerical. We will also look at ways this data is manipulated.

2.1 Characters and Strings

Data is a collection of values that we store and manipulate with the computer. There are many different types of data, and we will refer to each one as a **data type**, meaning a type of storage. Let's consider the case of a message that we want to send via e-mail to a friend.

```
Hi Kendra, do you want to go out to lunch today?
```

The type of data that we want to send in this case is alphabetic text. For a computer, we call alphabetic data **characters**. Looking back at the message we have characters that are non-alphabetic as well. For example, the spaces, comma and question mark are also characters that we use.

To represent a character in the computer we use single quotes around the character. For example, the first character in our previous example is 'H', the next 'i' and so on.

Self-Check 2.1

How many characters are in the e-mail (assuming there are no spaces before and after the sentence)?

The American National Standards Institute initially standardized the entire set of characters that are allowed for computers. It created the American Standard Code for Information Interchange, commonly known by its acronym **ASCII**. In Table 2.1 you can see the entire ASCII code. The Character column shows all possible characters that you could code with and represent on the computer starting in 1963.

You should be able to recognize all the characters in this column Character. Note that there is a unique decimal number, listed as Decimal in the table, for each character in the ASCII table.

Table 2.1: The ASCII character table

US ASCII Character Codes (000-063)		**US ASCII Character Codes (064-127)**	
Decimal	Character	Decimal	Character
000	NUL	064	@
001	SOH	065	A
002	STX	066	B
003	ETX	067	C
004	EOT	068	D
005	ENQ	069	E
006	ACK	070	F
007	BEL	071	G
008	BS	072	H
009	HT	073	I
010	LF	074	J
011	VT	075	K
012	FF	076	L
013	CR	077	M
014	SO	078	N
015	SI	079	O
016	DLE	080	P
017	DC1	081	Q
018	DC2	082	R
019	DC3	083	S
020	DC4	084	T
021	NAK	085	U
022	SYN	086	V
023	ETB	087	W
024	CAN	088	X
025	EM	089	Y
026	SUB	090	Z

027	ESC	091	[
028	FS	092	\
029	GS	093	]
030	RS	094	^
031	US	095	_
032	SP	096	`
033	!	097	a
034	"	098	b
035	#	099	c
036	$	100	d
037	%	101	e
038	&	102	f
039	'	103	g
040	(	104	h
041	)	105	i
042	*	106	j
043	+	107	k
044	,	108	l
045	-	109	m
046	.	110	n
047	/	111	o
048	0	112	p
049	1	113	q
050	2	114	r
051	3	115	s
052	4	116	t
053	5	117	u
054	6	118	v
055	7	119	w
056	8	120	x
057	9	121	y
058	:	122	z
059	;	123	{
060	<	124	|
061	=	125	}
062	>	126	~
063	?	127	DEL

Unfortunately this first code was based only on English and the Roman alphabet; once computers were used more internationally, there was a need to augment the initial ASCII code. Today the entire available character set is standardized by Unicode, which includes over 65,000 different characters. Unicode includes characters for languages like Arabic, Chinese, Greek, Hindi, and Russian.

So far we know that each individual letter is a character, but many times we want to create data that contains more than an individual character. You have already seen an example of this with our e-mail message: it contained a collection of characters in a specific order called a **string**.

When we use a string in a computer, we use double quotes “ ” around the characters to show the start and end of the string. So now our e-mail message would be

```
 “Hi Kendra, do you want to go out to lunch today?”
```

Note that the Unicode table allows for any type of character. I could create a string for the cost of groceries for the week:

```
     “75.28”
```

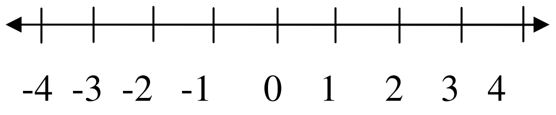
Perhaps I decide to buy a pint of ice cream too, which costs $2.50. Unfortunately I cannot just add the cost of the ice cream to the current cost of the groceries to create the string “77.78”, I would instead have to make an entirely new string.

This demonstrates that strings are rather inflexible and will not work in every situation, especially when we want to manipulate numbers algebraically. We cannot do arithmetic on strings. In this case we need another data type.

2.2 Numerical Data Types

So far we have seen two data types: characters and strings. Since grade school, you have been familiar with other types of data found in number lines. A number line shows us all the different numerical data types.

Figure 2.1: Integer Number Line

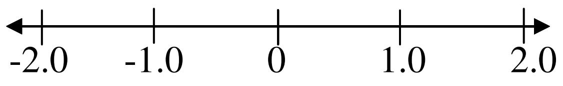


The values on the number line in Figure 2.1 show **integer** values. An integer is a positive or negative number without a decimal point (except that zero is considered to be neither positive nor negative). If there is no sign before the number, we assume it to be positive.

e.g.	-17392
e.g.	930756
e.g.	3

A larger group of numbers, which includes integers, is called **floating-point** numbers. The name floating-point gives you a clue of the difference from integer values. Floating-point numbers have decimal points.

Figure 2.2: Floating-point Number Line



The values on the number line in Figure 2.2 show some floating-point values. Like an integer, a floating-point is a number that can be positive or negative. If there is no sign before the number, we assume it to be positive.

e.g. -2.348

Note that any integer value is a floating-point number; it just has zeros after the decimal point. So 3 is the same as 3.0 and 3.0000000.

The advantage of being able to keep more decimal places is the increase in **accuracy** of the numbers that you can store. For example, let's say that I am given the task of making a duplicating a wooden table. The original table measures 7.273 feet long, but I can only store integer data. So I will round my numbers to the nearest integer, and my new table will be 7 feet long. My table will probably not be a duplicate if I cannot have the appropriate accuracy. Therefore, floating point numbers can be more accurate than integer values. The downside of using floating-point numbers is that they take more room to store in the computer than integers. To better understand the intricate details of data storage, the topic is covered in detail in a computer architecture class.

2.3 Numerical Operators

Operators are used to manipulate numerical data. The data that you manipulate with the operators are known as the **operands**. You have been using operators and the associated operands for many years in math class, and most operators that you know will be similar to the ones in the computer.

2.3.1 Addition and Subtraction

There are two additive operators, **addition** and **subtraction**. Addition is represented with the + symbol while subtraction is represented with the – symbol. For addition, the operands are called **addends**. When both addends are integer, you will get an integer result. Note that each integer could be positive or negative. If both addends have the same sign, add the two addends together and make sure the sign of the result is the same as both addends

e.g. 10 + 3 gives 13
e.g. -4 + -16 gives -20

If the addends have different signs, take the difference between the two addends and make sure the sign of the result is the same as the largest (positive or negative) addend.

e.g. -8 + 6 gives -2

When both addends are floating point, you will get a floating point or an integer result.

e.g. 4.2 + 3.1 gives 7.3
e.g. 2.6 + 5.4 gives 8

When you have one addend a floating point and the other integer, you will get a floating-point result.

e.g. -19.4 + 3 gives -16.4

If you are adding more than one value, make sure to perform your operations from left to right. Therefore you will solve the first pair, and then use this result in combination with the third number, repeating this pattern until all numbers are added.

Subtraction works similarly, you can think of it as addition where the second addend is negated.

e.g. 8 – 2 is really 8 + -2 which gives 6

These operations can be checked with a calculator, but you should be comfortable adding and subtracting by hand since we will be manipulating numbers throughout the text.

2.3.2 Multiplicative Operators

There are four multiplicative operators, multiplication and three forms of division. Multiplication is represented with the * symbol. The data type resulting after multiplying two numbers follows the same rules as addition.

e.g. 4.2 * 5 gives 21

Division is represented with the / symbol. The data type resulting after division of two numbers follows the same rules as addition and multiplication.

e.g. 6.8 / 2 gives 3.4
One type of division that you may not be as familiar with is **integer division**, represented with the // symbol. The data type resulting from integer division will always be integer no matter the data type of the numbers being divided. The result from integer division will always be the **truncated** result of the division at the decimal point (not a rounded result).

e.g. 6.8 // 2 gives 3

The final type of division is **modulus**, represented as **%**. Modulus is the remainder result after dividing two numbers. Like integer division, modulus will only return an integer result.

e.g. 7 % 2 gives 1

For // and %, we assume that the divisor is an integer and that both numbers are positive. If it is not the case, the result may be harder to predict since it depends on the rules of the particular programming language that you are using.

Self-Check 2.2

a. 10 % 5
b. 10 // 4
c. 32 / 5

When you are solving a problem with more than one operator, there are **precedence** rules that must be followed. Precedence means that there are some operations that must be conducted before others. Table 2.2 lists some of the rules of precedence.

Table 2.2: Operator Precedence

Symbols	Operator Type	Order of precedence
( )	Parentheses	Highest
-	Unary	(from **right** to **left**)
*, /, //, %	Multiplicative	(from left to right)
+, -	Additive	Lowest (from left to right)

When two operators have the same precedence, they are listed on the same row. For instance if you were solving 6/3*2 you would execute the integer division 6/3 first and the multiplication second since the rule of precedence is from left to right even though * is listed before / in the table.

2.3.3 Negation

The third type of operator is **negation**, represented by the symbol -. The multiplicative and additive operators must have two numbers (one on either side of the symbol) to be solved and are called **binary** operators ("bi" stands for two). Yet negation is a **unary** operator and only needs to have one number to the right of the symbol.

e.g. -2 is read as "negative 2"

For example, let's consider a problem that contains several of these symbols.

10/2+8*4//3

For this problem, there are two pieces to be worked through first:

10/2 and 8*4//3

The easier piece is 10/2, which gives us 5. Since 8*4//3 is a compound problem, and both * and // have the same order of precedence, we will solve the problem from left to right.

First, 8*4 gives us 32
Now 8*4//3 is 32//3, which gives us 10
Looking back at our original problem, we now have
5 + 10, this results in 15
So, 10/2+8*4//3 is 15

Another way to see this is to diagram out a solution as seen below in Figure 2.3. Each level of the diagram shows that another operation has been completed getting closer to the solution. This is an excellent method to solve a problem because your steps are clearly illustrated. If you did make a mistake while solving, it is easy to correct and update your solution when all of the solving process has been clearly displayed.

Figure 2.3: Evaluating Expressions with Mathematical Operators

```
10/2 + 8*4//3
 \_/    \_/
  5  +   32 // 3
   \      \___/
    5  +   10
     \_____/
       15
```

Self-Check 2.3

5 + 4 / 2 – 6 * (3 % 2)

2.3.4 Exponentiation

Another binary operator is the **exponent**, ^. The carat symbol is used to represent exponentiation. Exponentiation is a shortcut for multiplication, therefore 3^2 means that there are two 3's being multiplied together to give 3*3 or 9. In this example, the 3 is called the base and the 2 is the exponent. The solution is always the base multiplied by itself for exponent number of times.

Note that when an exponent is zero, you will always get 1.

e.g.   62 ^ 0 gives 1

When an exponent is negative, you will get a fractional result. We will not explore this case in class. To learn more about exponents, an algebra course would be helpful.

2.4 Casting

Casting is a useful way to convert from one data type to another. At this point there is only one type of casting that you need to be familiar with: casting an integer to/from a character. This unary operator has an order of precedence just under the unary operators, but before the multiplicative operators.

Looking back at the ASCII table in Table 2.1, you see that there is a column Character that we examined before, and another column Decimal that stands for decimal numbers. Decimal goes from the integer value 0 to 127. If I have an integer value between 0 and 127, there is an associated character representing that number. And if I have a character in the ASCII table, there is an associated integer value that represents that character. With this property, I can cast from integer to character and vice versa.

e.g. (char) 66

The above example will convert 66 to its ASCII equivalent and gives a 'B'.

e.g. (int) 'A'

The above example will give the integer value of the 65, which is the start of the capital letters in the ASCII table.

Table 2.3: Final Operator Precedence

Symbols	**Operator Type**	**Order of precedence**
( )	Parentheses	Highest
^	Exponentiation	(from **right** to **left**)
-	Unary	(from **right** to **left**)
(type)	Cast	(from **right** to **left**)
*, /, //, %	Multiplicative	(from left to right)
+, -	Additive	Lowest (from left to right)

Key Terms

Data	Numerical Operators	Casting
Data type	Additive	
Character	Multiplicative	
String	Negation	
Integer	Exponential	
Floating-point	Assignment Operator	

Exercises

1. Identify the data type of the following:

    a. “hello Sarah”
    b. -5.323
    c. “45”
    d. ‘@’
    e. “321.3”
    f. 103383994

2. What is the ASCII decimal value for ‘8’?

3. Complete the following operations:

    a. 142 – 31
    b. 5.3 + 14.62
    c. 4 * 1.2
    d. 3.5 / 2
    e. 25.4 // 7
    f. 453 % 5
    g. 3^4

4. Complete the following complex operations:

    a. 4.2 * 5 / 3 + 15//4
    b. 433 % (14 – 6) * 2.47

5. Convert the following:

    a. (int) ‘5’
    b. (char) 69

6. How many characters are there in the following string:

    “This string contains ‘@’ and ‘#’ characters”

7. Are “A” and ‘A’ the same?

8. Which operation is performed the last and what will be the result of the following expression:

   7 * 5 – (2 * 5)

9. Are the following numbers integers or floating point?

   a. -5
   b. 10.5
   c. 100
   d. -80.5

10. Which operator will give you the remainder of 25 divided by 6?

Self-Check Solutions

2.1 48

2.2

   a. 0
   b. 2
   c. 6.4

2.3

```
5 + 4 / 2 - 6 * (3 % 2)
 \   \_/    \    \___/
 5 +  2   -  6 *   1
  \     \      \__/
   5 +   2  -   6
    \____/     /
      7  -    6
       \_____/
          1
```

3. Pseudocode, Flowcharts and Python

In Chapter 2, we learned how to store information in the computer and the rules governing the manipulation of numbers and logical values. Now we will look at how to organize those rules to create simple programs.

3.1 Sequential Order

Programs are similar to books. In a book, you start reading from the top of the page and continue to the end of the page. In English, each line of text in the book contains information that is read from left to right. Likewise, we write programs for the computer to read in this order from left to right. Remember assignment statements from the previous chapter, $a = 3$ means that a stores the value of 3, not that 3 is a.

Now that we read each line appropriately, we will always start reading from the top line of our program and continuing until the last line in the program. The programs that we will look at in this chapter are all executed in **sequential** order. We will start with the first line and then continue to the next line. This **sequence control structure** can be represented with pseudocode, flowcharts, and Python code.

3.2 Pseudocode

First we will look at outlining a program using **pseudocode**. Pseudocode is a language very close to English that allows us to represent a program concisely. The only thing you need is a statement to show where you are starting and where you are ending a program. We will be using the word **Start** for the start point and the word **Stop** to show the finish point. Each program will contain statements to accomplish our goal; this will satisfy step 3 from Chapter 1.

3.3 Flowcharts

A more visual way to see the behavior of a program is a **flowchart** which is appealing to the visual learner. A flowchart uses symbols and shapes to represent an algorithm. Pseudocode can be translated into a flowchart and vice versa.

Table 3.1 shows some of the symbols used in a flowchart where text is placed inside of the symbols. The **ovals** are used when you are starting and ending a program. **Rectangles** are used when you are executing assignment statements. **Parallelograms** are used when you print statements to the screen or get information. These **print** and **get** topics will be discussed in detail in Chapter 4. **Arrows** connect the different symbols together to show the direction of flow in the program.

Table 3.1: Flowchart Symbols

Flowchart Symbol		Explanation
Arrow	→ ↓	Shows the direction of the program; what to execute next.
Oval	⬭	Used for the Start and Stop of a program. Write the word inside the shape.
Rectangle	▭	Used for assignment statements. Write the statements inside the shape.
Parallelogram	▱	Used for input and output. Write the print/get statements inside the shape.

3.4 Python

Python is a modern programming language invented by Guido von Rossum in 1995. It has a very nice and simple pseudocode-like syntax (structure). It is a powerful and an elegant language. We will be coding our solutions in Python to execute the program and confirm correctness. Coding a solution is the final stage, bringing together all of the hard work and thoughts written in pseudocode and visually interpreted in the flowchart. For more information on installing and setting up Python, read Appendix A.

Now let us look at some problems and their corresponding pseudocode, flowcharts and Python programs.

Problem 3.1

Calculate and print the average of three numbers: 5, 10, and 15.

Task 1- Identify your input:

Values of 5, 10, and 15.

Task 2- Identify the goal or objective:
Average the input values. The equation for calculating an average is to add all the numbers to create a sum. Then divide the sum by how many numbers you added.

Make sure you use **variables** to calculate the average instead of the numbers. A variable is a location in memory which holds values which can vary during program execution. Getting into the habit of creating variables now will be very helpful when you have longer more complicated programs that use at least one value multiple times. Then updating a value will take one change, where you assign the value to the variable, no matter how many times you use the value throughout your program.

Task 3- Create tasks to meet the objective:
1. Assign values for the input.
2. Calculate the sum.
3. Calculate the average.
4. Print the average.

The pseudocode of this program is shown in Figure 3.1.

Figure 3.1: Pseudocode

```
Start
  num1 = 5
  num2 = 10
  num3 = 15
  sum = num1 + num2 + num3
  average = sum/3.0
  print average
Stop
```

Note that indentation is important to clearly show the body of the program. Practice using indentation now, it will become vital as our programs get more complex.

Also note that when calculating the average, we divide by 3.0 instead of 3. This is vital to insure the accuracy of our result. To fully understand this issue, complete the following Self-Check.

This program starts by setting the value of three numbers, *num1*, *num2* and *num3,* which are needed to be able to calculate *sum*. Ensuring sequential order is vital to get the result that you expect. Note that I could not have set a variable value after calculating *sum*, as shown below:

```
num2 = 10
num3 = 15
sum = num1 + num2 + num3
num1 = 5
```

We could not calculate *sum* since it would be missing the value of *num1*. This condition occurs because *sum* is **dependent** on the values of *num1, num2*, and *num3*. This is a feature of sequential control structure that specifies that we can only execute code one line at a time from the top to the bottom of the program.

Note that the order in which *num1, num2*, and *num3*are defined does not matter. Therefore, our pseudocode could look like:

```
Start
  num3 = 15
  num2 = 10
  num1 = 5
  sum = num1 + num2 + num3
  average = sum/3.0
  print average
Stop
```

and it would still execute correctly since the variables *num1, num2*, and *num3*are defined before *sum* is calculated.

Self-Check 3.1

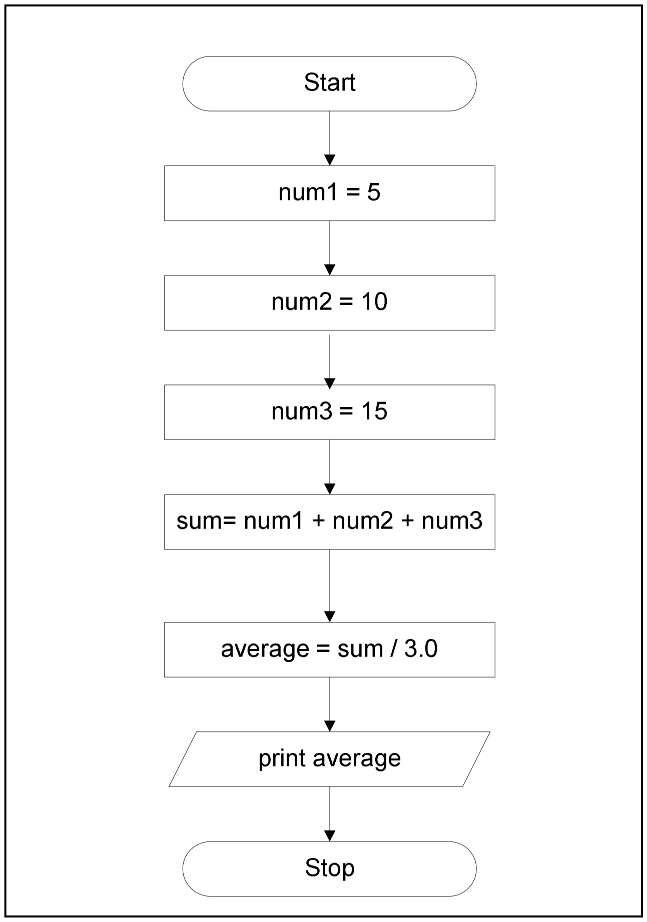
One dependency was identified in problem 3.1, can you find another dependency?

Now let's examine a more visual solution to the pseudocode problem from Figure 3.1 by creating a flowchart. Figure 3.2 begins with Start and ends with Stop, as all programs will. Following the Start, all assignment statements are in rectangles and the *print* statements are in parallelograms. Note that all statements must be placed in their appropriate flowchart symbol with arrows showing the direction of the execution.

Run through the problem by hand executing the flowchart to confirm that everything works as expected.

```
Since $num1 = 5$, $num2 = 10$, $num3 = 15$
        $sum = num1 + num2 + num3$
        $sum = 5 + 10 + 15$
            $= 30$
$average = 30/3.0$
         $= 10.0$
```

Figure 3.2: Flowchart



You can check your results with a calculator to confirm your solution.

Now let's finish the exercise with the corresponding Python program and the output. They are shown in Figure 3.3 and Figure 3.4.

To start making your program, you will need to open the IDLE (Python GUI) that you installed in Appendix A. Then click on File, New Window to open a screen that will allow you to type in your program. Note that the benefit of using Python is that the syntax is very similar to the pseudocode that we are using. See Figure 3.3 for this program.

When creating a program, it is important to document information with comments. Comments allow any user to understand the purpose of the function and how to use it appropriately. In Python, the pound sign, ‘#’ will start a comment from the ‘#’ to the end of the line.

Figure 3.3: Python program

```
# This program prints the average of three numbers
num1 = 5
num2 = 10
num3 = 15
sum = num1 + num2 + num3
average = sum/3.0
print (average)
```

Once you have typed in your Python program, press F5 (or Run, Run Module) to execute your program. Note that the computer will prompt you to save your code before it will run your program. See Figure 3.4 for the results displayed in the original IDLE screen.

Figure 3.4: Output

```
10.0
```

We will see how to create output in detail in the next chapter.

Problem 3.2

Calculate and print the square and cube of a number.

The square of a number is calculated by multiplying the number by itself. The cube is calculated by multiplying the number by itself twice.

Assuming that the number we want to square and cube is 4, let’s first look at the pseudocode to outline the steps in Figure 3.5.

Figure 3.5: Pseudocode

```
Start
  num = 4
  square = num*num
  cube = num*num*num
  print square
  print cube
Stop
```

Looking at the pseudocode, you can find two dependencies. Both *square* and *cube* require that *num* be defined before they can calculate their values.

Now let us practice again with the flowchart for this problem shown in Figure 3.6.

Again, run through the problem to confirm that your code works.

$$\begin{aligned} \textit{square} &= 4 * 4 \\ &= 16 \end{aligned}$$

$$\begin{aligned} \textit{cube} &= 4 * 4 * 4 \\ &= 64 \end{aligned}$$

You can now check your results with a calculator to confirm your solution. The corresponding Python program and the output are shown in Figure 3.7 and Figure 3.8.

Figure 3.6: Flowchart

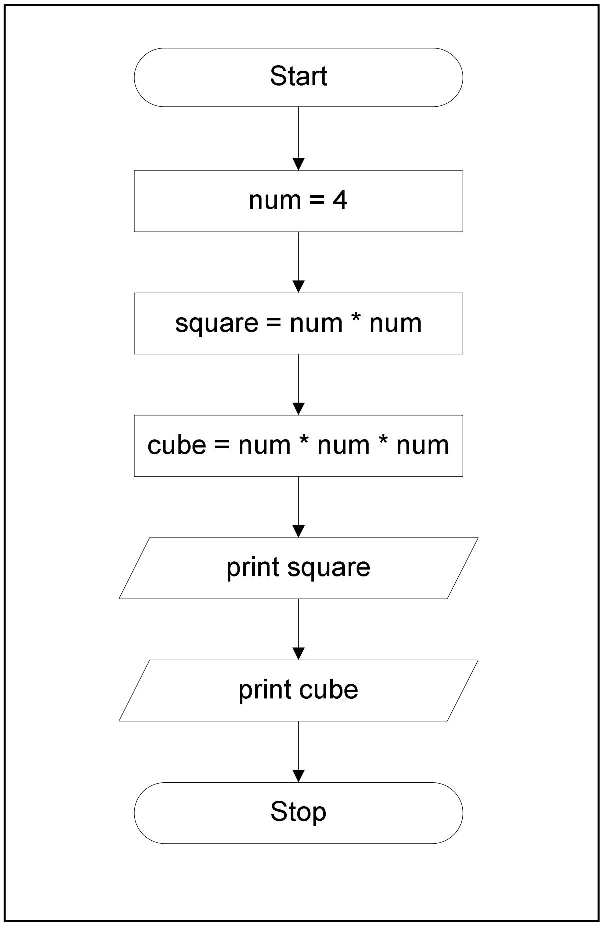


Figure 3.7: Python program

```
#This program prints the square and cube of a number
num = 4
square = num * num
cube = num * num * num
print (square)
print (cube)
```

Figure 3.8: Output

```
16
64
```

By now you should be getting comfortable using flowchart and pseudocode symbols. In the next chapter we are going to add to your pseudocode knowledge and flowchart symbols as we solve more complex problems..

Key Terms

arrows	Print
End	Pseudocode
Flowchart	Rectangle
Oval	Start
Parallelogram	Variable

Exercises

Draw flowcharts and create Python code for problems 1 and 2:

1. *num1 = 16*
   num2 = -12
   sum = num1 + num2
   print sum

2. *length = 5*
   width = 11
   *area = length * width*
   print area

3. Create a complete program that will calculate the diameter, area, and circumference of a circle with the radius of 4.25. Use the following equations (assuming *pi* = 3.14159):

*diameter = 2 * radius*
*area = pi * radius * radius*
*circumference = 2 * pi * radius*

Answer the following questions:

4. Susan wants to put wallpaper on four walls of her room. What are the three things you would like to know before you can calculate the cost?

5. Assuming 1% of your income is spent on school supplies. Create a program that will create two variables to store the amount you are paid and another to calculate the amount which is spent on school supplies.

6. Make the flowchart, and run the Python program for the following problem:

   You have decided to enter a model boat race. You put your boat at the start line next to your best friend Jill's boat. Create a program to print out the distance that both of your boats traveled given the speed your boat travels in (feet per minute), the speed that Jill's boat travels and the number of minutes in the race.

   *Distance = speed * time*

   Use the following sample Data:

   speedMe = 6.2
   speedJill = 5.9
   time = 2
   *distanceMe = speedMe * time*
   *= 6.2 * 2*
   = 12.4
   *distanceJill = speedJill * time*
   *= 5.9 * 2*
   = 11.8

7. Write the pseudocode, and run the Python program for the following problem:

A recipe you are reading states how many grams you need for the ingredient. Unfortunately, your store only sells items in ounces. Create a program to convert grams to ounces.

*ounces = 28.3495231 * grams*

Use the following sample data:

grams = 45
*ounces = 28.3495231 * grams*
*= 28.3495231 * 45*
= 1275.72854

8. Make the flowchart, and run the Python program for the following problem:

   Given the rate of pay (in dollars per hour) and the number of hours an employee has worked for a week, calculate the amount the employee should be paid.

   Use the following sample Data:

   rate = 6
   hours = 30
   *pay = 6 * 30 = 180*

9. Make the flowchart, and run the Python program for the following problem:

   Given a Fahrenheit temperature, calculate and display the equivalent centigrade temperature. The following formula is used for the conversion:

$$C = 5 / 9 * (F - 32)$$

   where F and C are the Fahrenheit and centigrade temperatures.

   Use the following sample data:

```
F = 72
C = 5 / 9 * (72 - 32) = .556 * 40 = 22.24
```

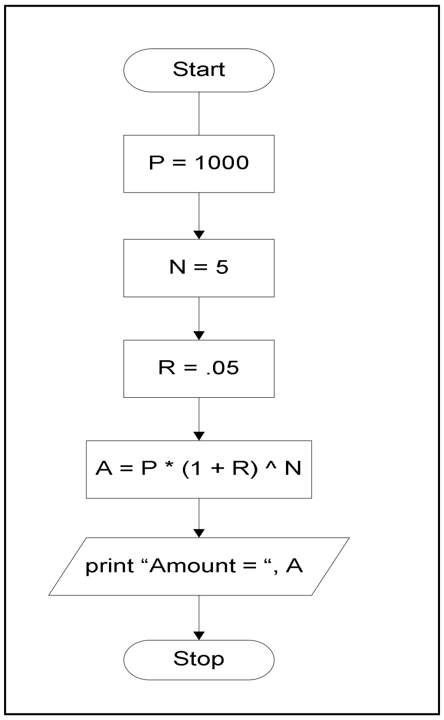
10. Write the pseudocode, and run the Python program for the following problem:

Calculate the amount obtained by investing the principal P for N years at the rate of R. The flowchart given below shows the sequence of steps necessary to accomplish this task. The following formula is used for the computation:

$$A = P * (1 + R) ^ N$$

Use the following sample Data:

P = 1000
N = 5
R = .05
A = 1276.28



11. Make the flowchart, write the pseudocode, and run the Python program for the following problem:

Given the meter reading at the beginning of the month, at the end of the month, and the price/unit of the electricity consumed, calculate the cost of the electricity consumed. The following formulas are used:

Number of units consumed: *ending meter reading – beginning meter reading*

Cost: *Number of units consumed * price/unit*

Use the following sample data:

Beginning meter reading: 11239
Ending meter reading: 20850
Price per unit = 10 cents

Self-Check Solutions

3.1 The variable *average* is dependent on *sum* being defined.

4. Input and Output

In Chapter 3 we learned how to create simple programs using logical rules. So far, all the information is in the computer and does not leave the computer; there is no interaction between the user and the information. In this chapter, we will be looking at methods to store and retrieve information in a computer to make the machine more effective and useful. We will look at how to first show information on the computer screen and then retrieve information from the user in pseudocode, then in flowcharts and finally in Python.

4.1 Pseudocode "print" and "get"

Every time you use a computer you give it **input**. Perhaps you type an e-mail to a friend or use your mouse to click on websites. Both of these are forms of input into the computer. We will be gathering input from the person using our program; this person will often be referred to as the **user** since they are using the program we created.

To make sure that we get the information we want, or to display our results on the screen, we will also need to use **output**. You get output from the computer in a variety of ways, from listening to sounds from the speakers to looking at the display on the monitor. For example, if I want to get a person's age for my program, I would need to ask for the information (output) and retrieve that information (input).

To display information on the computer screen we use the keyword ***print***. After *print*, we must tell the machine what we want to print, selecting from one of the data types that we learned about in chapter 1.

```
print "Please enter your age:"
```

Now to retrieve the information, we use the key word ***get***. After *get*, we must tell the machine where we want to store the information. The storage location (or place) will be a variable. Make sure your variable is appropriately named so that it is easy to recall its purpose later. One popular method of creating descriptive variables is **camel notation** (like the humps in a camel). To accomplish this we capitalize the first letter of the subsequent words. In this example, we want to use the words "current age" to describe the variable, but we cannot have spaces, so we will use camel notation and combine the words together as below:

```
get currentAge
```

Now when the value 22 is typed on the keyboard, it is stored in the variable *currentAge*.

With the *print* and *get* pseudocode statements, we can print and retrieve any information that we need. Note that *get* can only obtain one data type. *Print* is more flexible. I am able to print more than one data type if I use a concatenation operator, '+'. A concatenation operator does not work like addition, even though the symbol is the same. Concatenation is a binary operator that appends two operands together.

Let's write our psuedocode program with an ending line printed that will repeat the user's information.

```
print "Please enter your age."
get currentAge
print "Your current age is " + currentAge
```

If the user entered 22 for their age, then the final print statement would display:

```
Your current age is 22.
```

If I wanted to complete some complicated logical instructions inside of a *print* statement, then it is best to surround the instructions with parentheses, since by order of operations, the parentheses will always be executed first.

```
print "There are 365 days in the year and " + 365//30) +
    " months in a year."
```

The output from the above statement would be

```
There are 365 days in the year and 12 months in a year.
```

You can also have string concatenation, for example:

```
print 2+2
print "2" + "2"
print "2" + 2
```

In this case, the first line would produce 4 (because + adds the two numbers together) while the second and third lines would produce “22” (because + is now interpreted as string concatenation).

4.2 Constants

In our earlier discussion we have used variables to store information. Sometimes we want to make sure that the information does not change throughout a program. To do this we create a **constant** which stores data like a variable, but cannot be changed once it is assigned a value. A constant is represented by letters or numbers following the rules listed below:

Rules for naming a constant:

1. The first letter of a constant cannot be a number.
2. All letters must be capitalized.

We cannot use the same trick in naming a constant with more than one word as we did with a variable since all letters are capitalized. Instead you can use an underscore to represent spaces in the constant.

e.g. *DAYS_IN_YEAR* = 365

Now I can rewrite the above print statement in pseudocode using the constant and get the same result.

```
print "There are " + DAYS_IN_YEAR + “days in the year
and " + (DAYS_IN_YEAR //30) + “ months in a year."
```

In Python, the statement will need to be modified slightly using *str()* to print the numerical information as string. Note that *str()* is a function, you can tell with the () at the end of the name.

```
print ("There are " + str(DAYS_IN_YEAR) + “days in the
  year and " + str(DAYS_IN_YEAR //30) + “ months in a
  year.")
```

Here you also see that there is the possibility of wrapping your text when you run out of room on one line. This will not change your result, but by indenting your second line you can easily group the *print* statement for clarity.

Another nice feature about constants is that when I wish to change the value of the constant, I only have to change it in one location. For example, let’s say that I was talking about Venus instead of earth, assuming that there are 30 days in a Venus month and 225 days in its year. I would need my code to be:

```
DAYS_IN_YEAR =225
print “There are ”  + DAYS_IN_YEAR + “days in the year and ” +
        (DAYS_IN_YEAR //30) + “ months in a year.”
```

Now by changing one value, everything has been updated to account for Venus instead of Earth.

4.3 Problem 4.1

Now that you have a solid basis, you can create more complicated programs. Let’s create a pseudocode program that will calculate the number of miles per gallon that a car gets. Remember that the equation you will need is:

milesPerGallon = numOfMiles / numOfGallons

So the pseudocode is:

Figure 4.1: Pseudocode for Problem 4.1

```
Start
  print “Please enter the number of miles the car has driven.”
  get numOfMiles
  print “Please enter the number of gallons that the car consumed.”
  get numOfGallons
  milesPerGallon = numMiles / numGallons
  print “The number of miles per gallon your car consumed was ” +
  milesPerGallons
Stop
```

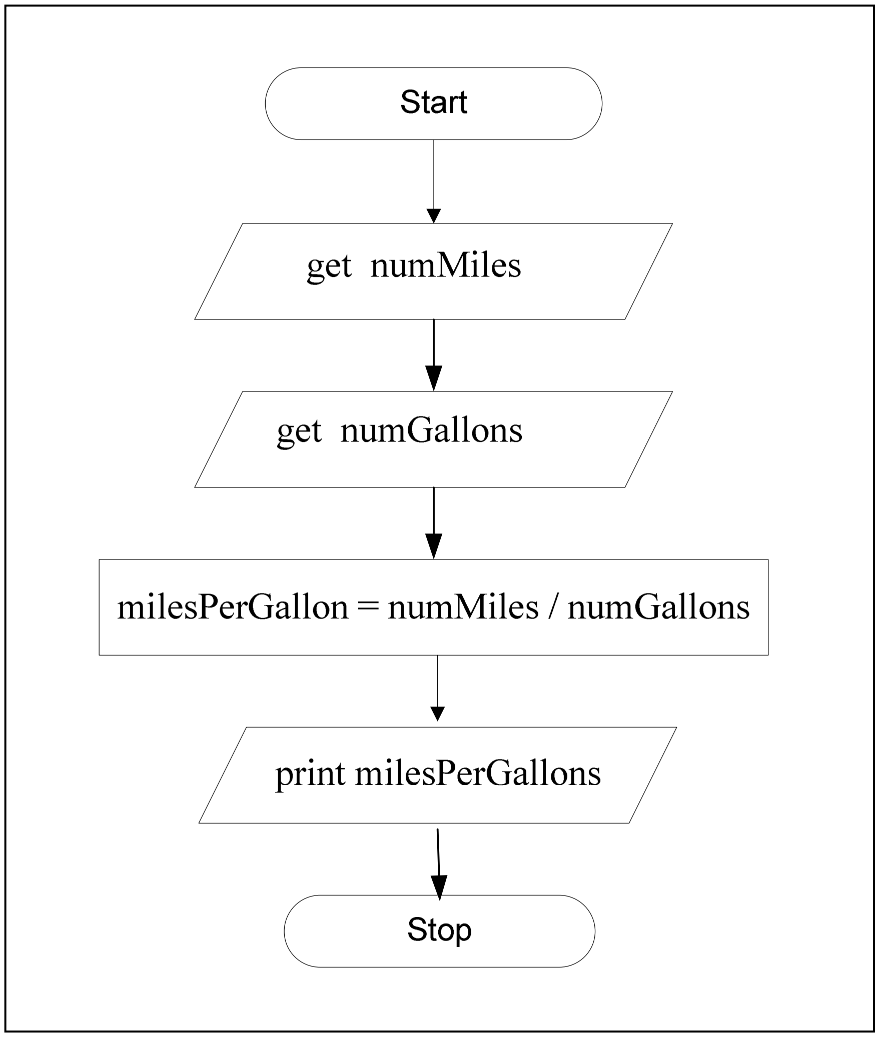
Self-check 4.1

Write Pseudocode to calculate and print out the amount of pay that an employee should be paid. First identify the input, goal and tasks using what you learned in Chapter 1.

Now we can make the flowchart for Problem 4.1. Note that we can combine pieces inside the same shape when consecutive lines of code will use the same flowchart symbol. Make sure that each code segment is on a separate line within the shape.

Note that the two "gets" could be in one box or in two separate boxes. Here we first get the number of miles and then the number of gallons.

Figure 4.1: Flowchart for Problem 4.1



Self-Check 4.2

Create a flowchart for the self-check problem in Section 4.3.

4.4 Applying Python

4.4.1: Python “print” and ” input”

In Python, the pseudocode *print* statement works just as expected, but there is no *get* in Python. Instead, we use *int(input())* to get numerical integer information. Note that *int* is a function which converts the string read in by *input()* which is also a function, you can tell with the *()* at the end of the name. We will discuss functions in detail in chapter 10. The two pseudocode statements can be written separately,

```
print (“Please enter your age:”)
currentAge = int(input())
```

Or they can be combined by putting the print statement inside the parentheses

```
currentAge = int(input(“Please enter your age:”))
```

In either case, what the user sees on the screen is the request. After the request, the cursor waits for the user’s response. In the first scenario, the cursor will blink on the next line to wait for the user input. In the second scenario, the cursor will blink at the end of the prompt to wait for the user input.

Figure 4.1: User input for Python for the first scenario

```
Please enter your age:
|
```

When the user enters their response, perhaps the age of 22, then the screen will appear as:

Figure 4.2: User input for Python using the second scenario

```
Please enter your age:22
```

Likewise in Python, concatenation needs the same data types. So we would modify the statement to be:

```
currentAge= int(input("Please enter your age: "))
print("Your current age is "+ str(currentAge))
```

By placing *str* before *currentAge*, we can convert *currentAge* into a string to print the result. If you don't do this, you will get the following error message from the code.

Figure 4.3: Python print error message

```
print("Your current age is " + currentAge)
Error: TypeError: can't convert 'int' object to str implicitly
```

Note also that it is important to include a space after "is " so that you print a sentence with appropriate spacing. Try it without the space and watch your *currentAge* get squished next to the *is*.

To accomplish the next part of Section 4.1 in Python, we just have to remember to convert the values to strings when needed.

```
print ("There are 365 days in the year and " +
   str(365//30) + " months in a year.")
```

For the last part of 4.1, we will get the same printouts in Python as in the pseudocode except for the last statement, where the conversion is needed again

```
print ("2" + str(2))
```

Figure 4.1: Python program for Problem 4.1

```
# This program inputs number of miles driven and the
# gallons used and calculates miles per gallon
miles = int(input("Enter number of miles: "))
gallons = int(input("Enter number of gallons used: "))
milesPerGallon = miles/gallons
print ("Miles per gallon: ", milesPerGallon)
```

Figure 4.1: Output for Python 4.1

```
Enter number of miles: 200
Enter number of gallons used: 20
Miles per gallon: 10.0
```

Key Terms

input	camel notation
output	constant
print	get

Exercises

Complete the following problems.

1. Given the variables below

   a = “the”
   b = “cat”
   c = “happy”
   d = “ ”

Translate the following concatenated statement: a + d + c + d + b

2. Given the variables below

```
a = “enjoy”
b = “1”
c = “life”
d = “2”
e = “happy”
f = “be”
g = “ ”
```

Translate the following concatenated statement:

```
b + g + a + g + c + g + d + g + f + g + e
```

3. What is the solution to the following concatenation?

    “3” + “2”

4. What is the value of the variable j?

    j = 01234

5. What is the value of the variable k?

    k= “01234”

6. Using problems 4 and 5, what is the result of j==k

7. Create a program to double a number’s value (where the number is given by the user). First identify the input, goal and tasks using what you learned in Chapter 1.

8. Using problem 7, create a flowchart of the tasks.

Draw flowcharts to do the following:

9. Read in two numbers and print the sum of the two numbers.

10. Read in length and width of a rectangle and print its area.

    *area = length * width*

11. Read in radius of a circle and print the diameter, area and circumference of the circle (assuming pi = 3.14159).

    *diameter = 2 * radius*
    *area = pi * radius * radius*
    *circumference = 2 * pi * radius*

12. Susan wants to put wall paper on four walls of her room. What are the three things you would like to know before you can calculate the cost?

13. 1% of one's pay is spent on school supplies. Read in pay and calculate and print the amount which is spent on school supplies.

14. Print area and perimeter of the hexagon.

    *area of hexagon = 6 * area of one triangle*
    *area of one triangle = 1 / 2 * b * h*
    *area of hexagon = 6 * 1 / 2 * b * h*
    *perimeter of this hexagon = 6 * b*

15. Read the base and the height of a right angle triangle and calculate and print the hypotenuse.

    hypotenuse = $\sqrt{(b^2+h^2)}$

Self-Check Solutions

4.1 **Input:** *hoursWorked, payRate*

Goal: *grossPay*

Tasks:

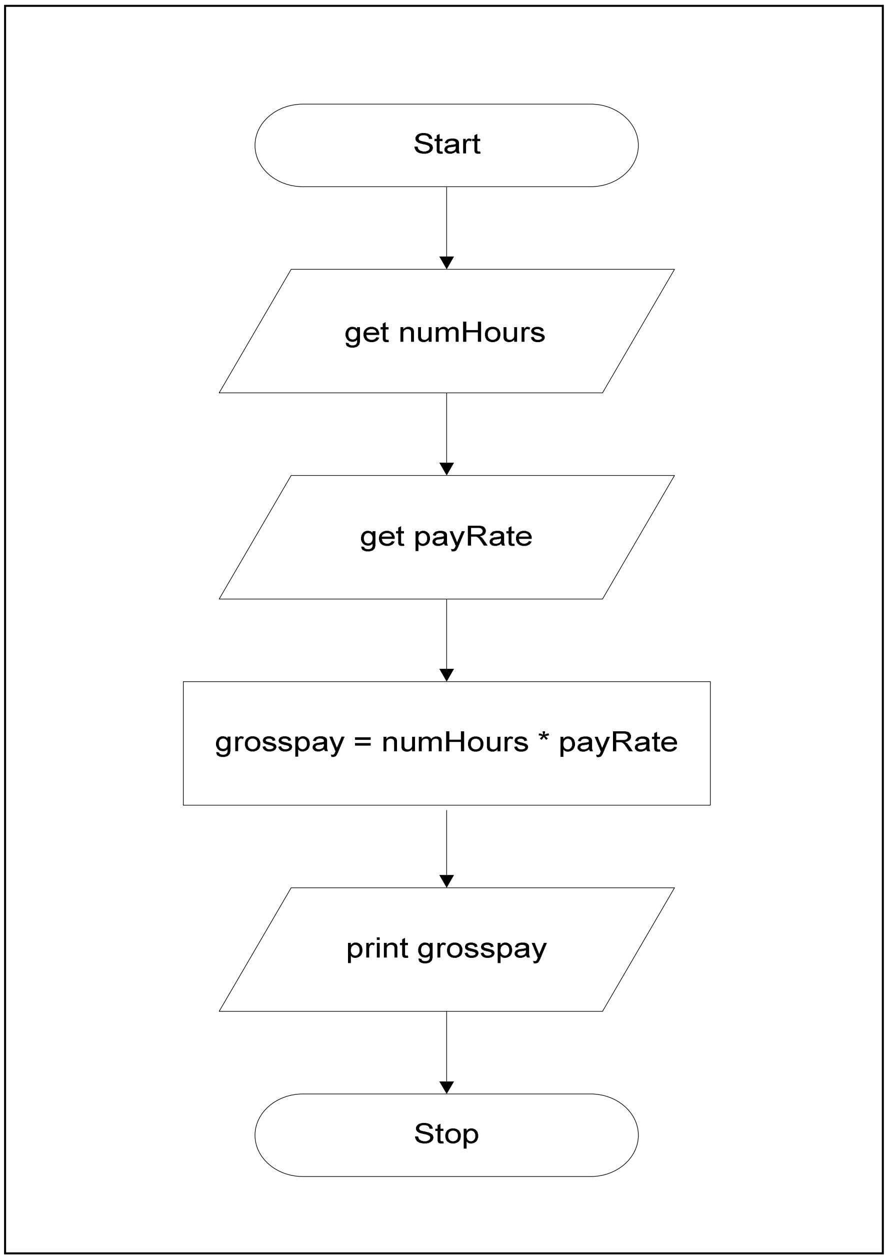
1. Enter *hoursWorked*
2. Enter *payRate*
3. Calculate grossPay
4. Print grossPay

Pseudocode:

```
Start
   print “Please enter the number of hours the
   employee worked.”
   get numHours
   print  “Please enter the employee’s pay rate.”
   get payRate
   grossPay = numHours * payRate
   print “The employee earned $” + grossPay
Stop
```

4.2 Note that we are leaving out the prompt statements here since they are implicit.

Flowchart:


Start
get numHours
get payRate
grosspay = numHours * payRate
print grosspay
Stop

5. Boolean Logic

Every decision you make can be broken down into a choice between two different options, with the question being “Do you choose the first option?” If you answer “Yes,” you choose one way, but if you answer “No,” you choose another way. This “yes” or “no” question can be rephrased as the statement “I choose the first option.” This statement is either True (instead of Yes) or False (instead of No). As a computer scientist, programming a decision that has two options is one of the most common tasks that you will perform. This chapter will introduce and explore the way Boolean logic can help formally pose these questions.

5.1 Boolean Data Type

There is one remaining data type that is commonly used in computers, a **Boolean**. Named after George Boole, a 19th century mathematician, a Boolean variable has only two possible values, either **True** or **False**. There are many times when it is convenient to use a Boolean in place of a numerical, character or string data type; to understand why we need to explore the physical foundations of a computer.

By itself, a single **bit** is not very powerful, but when you string together a long list of 0’s and 1’s, we call this machine code or binary data. In fact, everything you do on the computer must be translated into 0’s and 1’s for the machine to work. To learn more about this interaction, you will want to take a class in computer architecture. For now, we will be content to understand the origin of the 0’s and 1’s and will represent them as Booleans.

5.2 Relational Operators

There are six relational operators that you need to be familiar with for programming. Relational operators are always binary, such that they involve comparing two values. The operands will be numerical and their result will always be Boolean. Different programming languages may have different symbols for these operators, but the ideas behind the operators will be the same. These operators are “<” (less than), “<=” (less than or equal to), “>” (greater than), “>=” (greater than or equal to), “==” (equal to), “!=” (not equal to). Note that not all operators have the same precedence. The first four operators all have stronger precedence than the last two (as displayed in table 5.1).

<u>Table 5.1: Relational operator table</u>

Operator	Description	Precedence
>	Greater than	Highest
<	Less than	Highest
>=	Greater than or equal to	Highest
<=	Less than or equal to	Highest
!=	Not equal to	Lowest
==	Equal to	Lowest

In Table 5.2 you can see the results of using all of the possible relational operators when you have set values for the variables.

<u>Table 5.2: Examples of Boolean expressions for *x* = 10 and *y* =5</u>

Expression	Result of the expression
x > y	True
x < y	False
x >= y	True
x <= y	False
x != y	True
x == y	False

For the above table, substitute the values of x=10 and y = 5 to get the result. For example, *x > y* is *10 > 5* which is True. Similarly for another example, *x != y* is *10 != 5* which is also True.

It is possible to combine other operators with relational operators by following the rules of precedence. Note that the relational operators have a lower precedence than all other operators we have seen so far, see Table 5.3 for a summary of the operator precedence.

Table 5.3: Operator Precedence

Symbols	Operator Type	Order of precedence
( )	Parentheses	Highest
^	Exponentiation	(from right to left)
-	Unary	(from right to left)
*, /, //, %	Multiplicative	(from left to right)
+, -	Additive	(from left to right)
<, <=, >, >=	Relational	(from left to right)
==, !=	Relational	Lowest

5.3 Logical Operators

So far we have only considered problems that result in a numerical solution from numerical input. Now that we have a new data type, we can solve problems which involve Boolean values as inputs or intermediary computations. There are three types of **logical** operators that allow us to handle Boolean input: AND, OR, NOT. For each of these operators, since they have a Boolean input, their result will also be Boolean.

The following order of precedence applies: NOT is the strongest operator, followed by AND, then by OR. So our table of precedence for these operators is seen in Table 5.4. But logical operators are not as strong as the other operators that we have learned so far, such as the additive and multiplicative operators, and we will integrate them all at the end of this chapter.

Table 5.4: Logical Operator Precedence

Operator	Description	Precedence
!	NOT	Highest
&&	AND	Middle
||	OR	Lowest

To understand the rules for logical operators, we create **truth tables**. A truth table displays all possibilities for the inputs of the logical operator for every possible input. First let's look at a truth table for NOT:

<u>Table 5.5: Truth Table for NOT</u>

input	**NOT input**
True	False
False	True

Starting with the first row in Table 5.5, we see that the result of NOT True is False and the second row of the table shows us that the result of NOT False is True. Therefore, NOT changes or flips the value of its Boolean input.

Now let's look at AND, the next strongest logical operator in Table 5.6. Note that this operator is binary, so it needs two inputs. Instead of using the term input as was done before for NOT, let's generalize to *P* for the first input and Q for the second input. This way we can see the truth table for the statement P AND Q. The result of the AND operator is True only if both inputs P and Q are True**.**

<u>Table 5.6: Truth Table for AND</u>

P	**Q**	**P AND Q**
True	True	True
True	False	False
False	True	False
False	False	False

Finally, let's look at OR, the weakest of the logical operators. This operator is also binary, so it will need two inputs just like AND. The result of the OR operator is False only if both inputs P and Q are False. Note that this is not an exclusive condition; both P and Q could be True, and P OR Q will still be True.

Table 5.7: Truth Table for OR

P	Q	P OR Q
True	True	True
True	False	True
False	True	True
False	False	False

Our table of precedence says that the two logical operators AND and OR are executed from left to right. This is important when there are more than two inputs, as seen in the figures below.

Figure 5.1: Evaluating Expressions with Logical Operators

```
True AND False AND True
  \ _______/         |
False           AND True
    \_______________/
         False
```

Self-Check 5.1

Evaluate the following expression:
```
     False OR True OR False
```

As with numerical operators, we need to also be comfortable with using parentheses, which have a stronger precedence. This is shown in Figure 5.2.

Figure 5.2: Evaluating Expressions with Logical Operators

```
NOT (False OR True) AND True
       \________/         |
NOT       True      AND True
   \__________/           |
      False         AND True
         \_______________/
               False
```

It doesn't matter how many ANDs, ORs and NOTs are strung together, the same rules will apply. If you follow the rules of precedence and the truth tables for the operators, you will come to the correct conclusion.

Figure 5.3: Evaluating Expressions with many Operators

```
6+3 > 8 OR False
\__/     |    |
  9 > 8 OR False
  \___/       |
   True OR False
      \_______/
        True
```

Self-Check 5.2

Evaluate the following expression:

```
True AND 16 % 5 < 3 OR False
```

5.4 Assignment Operator

The assignment operator is represented with a single equal symbol, "=". This is the weakest of all operators and will be performed after all other operators have been evaluated. Assignment is particularly helpful when you want to assign a variable to have a certain value and then remember this value later. A **variable** is a letter or group of letters and numbers that holds a value.

For example, let's store a value into the variable *num1* using assignment.

num1 = 6 (This can also be read as *num1* gets *6*.)

Now I can use *num1* in an equation.

*answer = num1 * 3 + 2*

This results in the value of *20* being stored in *answer*.

Rules for variable names

1. The first letter is a lowercase alphabet.
2. Numbers 0 to 9 can be used after the first letter.
3. There are no spaces or non-alphanumeric characters (Note alphanumeric means numbers or alphabetic characters).

Example	**Validity**	**Reason**
payRate	valid	
pay Rate	invalid	Space is not allowed
numHoursWorked	valid	
4numHoursWorked	invalid	A number is not allowed for the first character
time@Day	invalid	Special characters are not allowed
region-sales	invalid	Hyphen not allowed
car_color	invalid	_ is not allowed

As mentioned in Chapter 3, if you want to combine two or more words together for a variable name, note that you cannot use a space. But one handy trick is to use camel notation where you capitalize the first letter of every word after the first word in the name.

A variable can store a value of any of the five data type as discussed in Chapter 2.

e.g. `currentValue = True`

5.5 Equality Operators

Both operands of "==" (and of"!=") must be Boolean or both must be numerical, but the resulting answer will always be Boolean. Note that there is a difference between using a single equals sign, "=", used in assignment of values to variables. Whereas a double equals sign, "==", is used in comparing two values. If both of the operands for "==" have the same value, then the result will be *True*, otherwise the value will be *False*. When using "!=", the result will be *True* only when the operands are different.

e.g. True == False results in False
e.g. 5 == 5 results in True

Table 5.8: Revised Operator Precedence

Symbols	Operator Type	Order of precedence
( )	Parentheses	Highest
^	Exponentiation	(from right to left)
-	Unary	(from right to left)
*, /, //, %	Multiplicative	(from left to right)
+, -	Additive	(from left to right)
<, <=, >, >=	Relational	(from left to right)
==, !=	Relational	(from left to right)
NOT	Logical	(from right to left)
AND	Logical	(from left to right)
OR	Logical	Lowest (from left to right)
=	Assignment	Lowest

Key Terms

Assignment operator	Logical
Boolean	True
False	Truth Table
	Variable

Exercises

1. Calculate the following (True or False):

    a. 23 != 15
    b. 5 + 3 < 10
    c. 6 > 10 == 10 < 2
    d. 4 ≤ 4 AND True == NOT False

2. Given that *a* = *3* and *b* = *8*, what are the results of the following statements

    a. a < b
    b. 6 ≥ a
    c. b > a == False

   d. True != (a==b)

3. Given that c = True and d = False, what are the results of the following statements:

   a. *c* AND *d*
   b. NOT *d* OR *c*
   c. *c* == *d* AND True
   d. (NOT *c* OR NOT *d*) == NOT (*c* AND *d*)
      Note that this is also known as DeMorgan's law.

4. Using the shell in IDLE, convert all parts of problem 3 into Python code (using table 5.4). Note that your answers to problem 3 should be the same as what you get from Python.

5. Which of the following are valid variable names? If it is not a valid variable name, state why.

   a. Donkey
   b. Game board
   c. myFirstCounter
   d. 4months

Self-Check Solutions

1. True
2. True

6. Selection Structures

We make choices in the world every day, from what color of shirt you wear to what station you watch on television. The previous chapter explored Boolean logic and relational operators. In this chapter we will look at how to create more flexible programs containing decisions based on Boolean logic.

6.1 If-Then-Else

One of the most common cases that we encounter every day is the choice between two objects: Should I get vanilla or chocolate ice cream? Should I turn left or right at this intersection? These choices can be represented with the following form in pseudocode:

```
If BooleanExpression Then
  trueChoice
Else
  falseChoice
EndIf
```

There are four keywords that must be included: *If, Then, Else* and *EndIf*. To distinguish these words from possible variables, we capitalize the first letter of each word. It is a common convention to indent the code as shown above to make it easy to understand.

6.2 Problem 6.1

Let’s look at an example:

```
a = 4
b = 3
If a > b Then
  print "Variable a is greater than variable b"
Else
  print "Variable b is greater than or equal to variable a"
EndIf
```

In this case, our Boolean expression is *“a > b”*. Since the variable *a* has a value greater than the value in *b*, we have a True statement. With our

Boolean expression satisfied, we would print “Variable a is greater than variable b” to the computer screen.

Now suppose we changed the values of our variables to be:

> *a = 2*
> *b = 3*

With the same *If-Then-Else* statement, we would no longer have a True statement for the Boolean expression, so we would print “Variable b is greater than or equal to variable a” to the computer screen.

Note that the *trueChoice* and the *falseChoice* can be single or multiple lines of code, including *print* or *get* statements, assignment statements, and even other *If* statements.

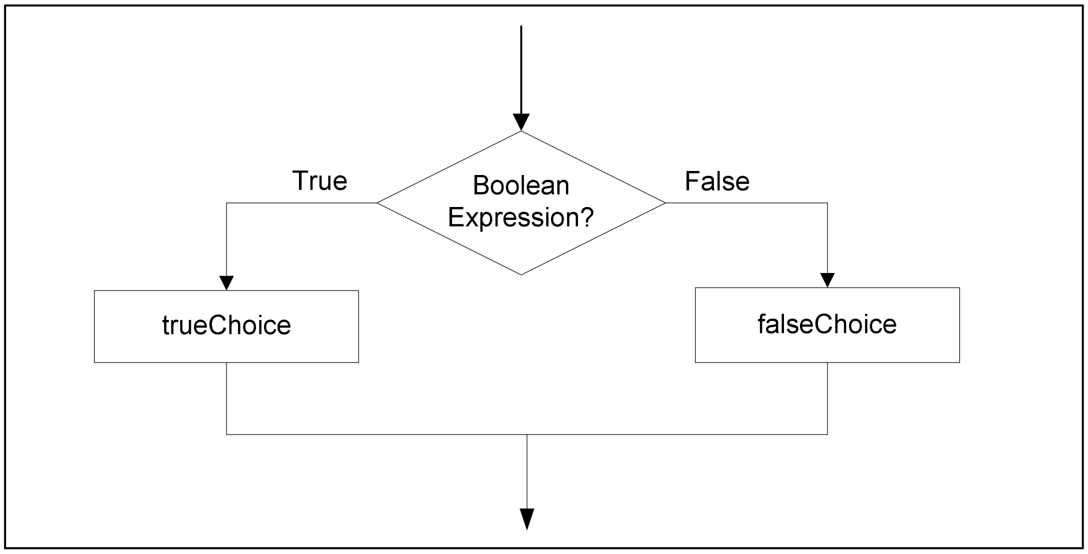
6.3 Updating Flowcharts

Now that we have examined the pseudocode for making decisions with *If-Then-Else*, we need to update the symbol chart for our flowcharts to include the ability to have a Boolean expression.

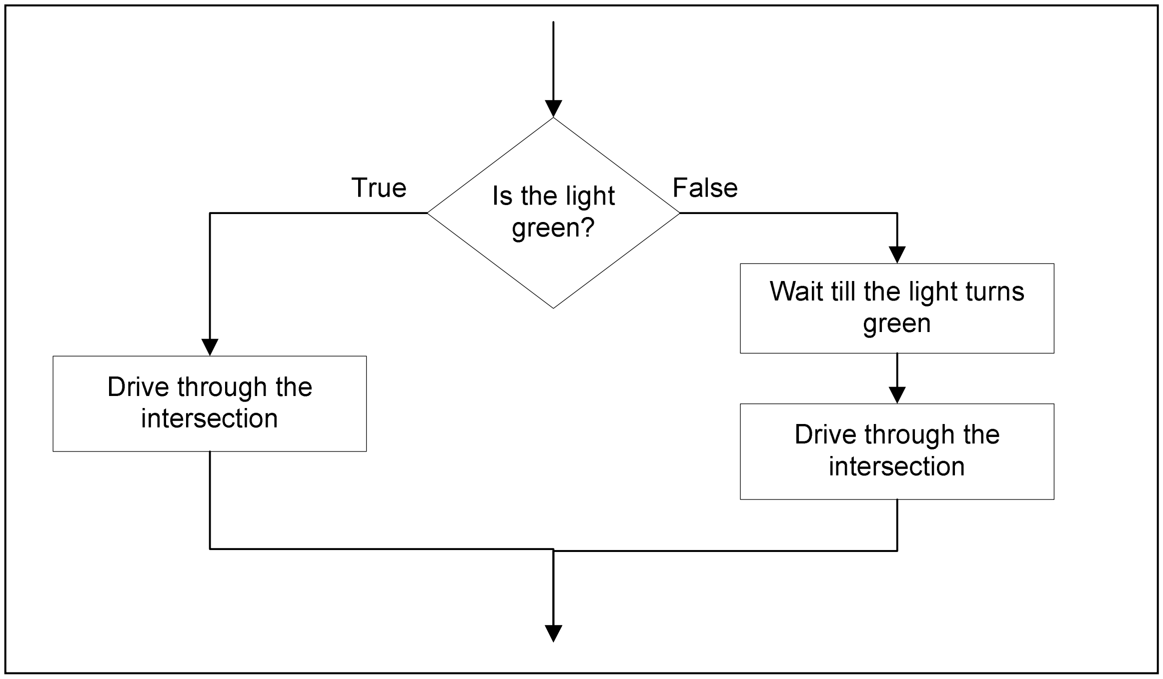
Figure 6.1: Updated Flowchart Symbols

Flowchart Symbol		Explanation
Arrow	→ ↓	Shows the direction of the program; what to execute next
Oval	⬭	Used for the *Start* and *Stop* of a program. Write the word inside the shape.
Rectangle	▭	Used for assignment statements. Write the statements inside the shape.
Rhombus	▱	Used for input and output. Write the get/print statements inside the shape.
Diamond	◇	Used for a Boolean condition. Write the Boolean statement inside the shape.

In the general case of an *If-Then-Else*, you have the following layout for a flowchart:

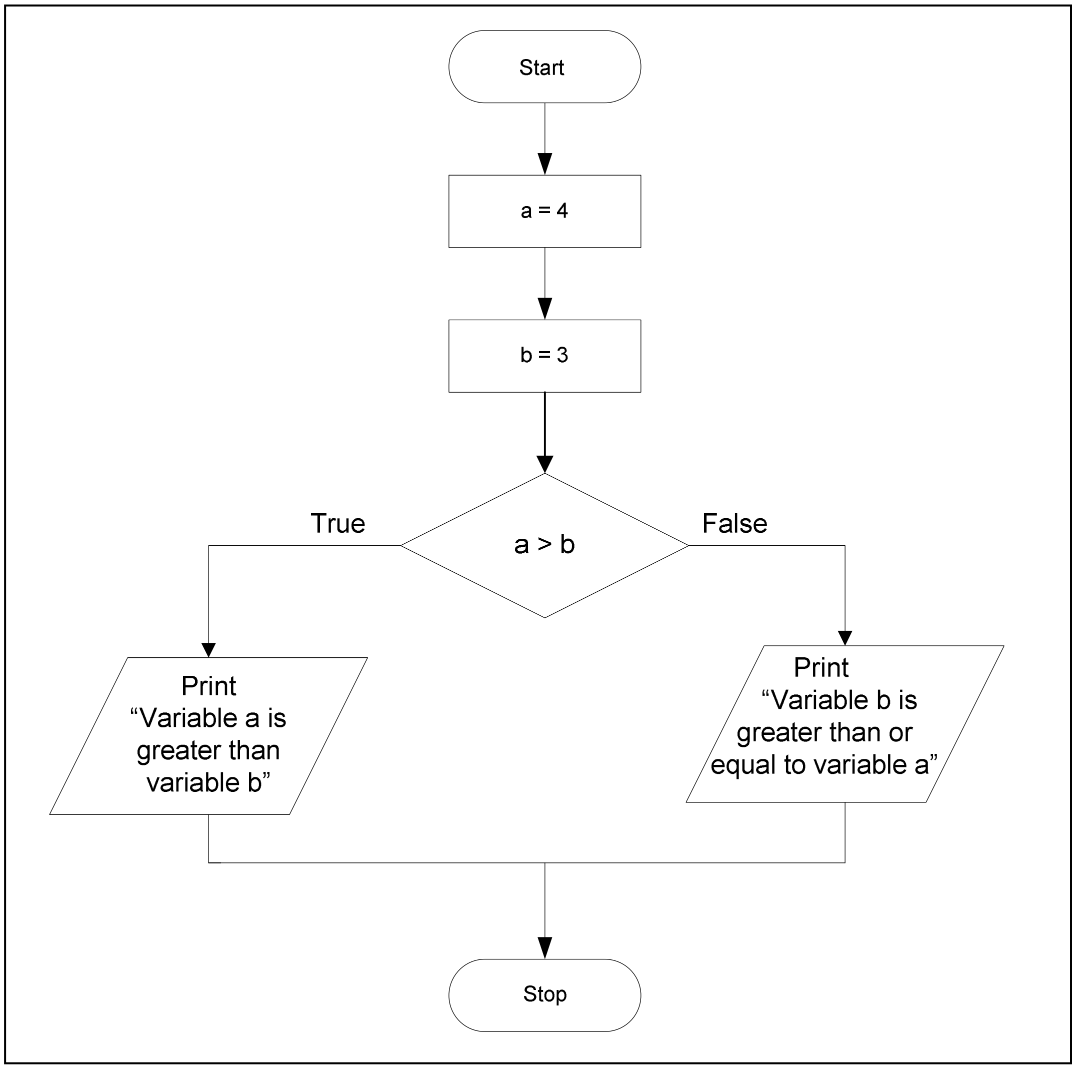


To give you an idea of how this behavior is used in a flowchart, let's look at the example for a traffic light. When you see a traffic light you automatically ask yourself "Is the light green?" Based on the answer, you decide to either drive though the intersection or wait until the light turns green to continue. This decision structure is illustrated in the example below.


True
Is the light green?
False
Wait till the light turns green
Drive through the intersection
Drive through the intersection

Notice that there are some distinct differences between the flowchart and the pseudocode. The flowchart does not contain the *If* or *EndIf* statements. In a flowchart, these statements are not necessary since the arrows help us to direct the control and an *If-Then-Else* will be the only decision statement with this flowchart format. Also notice that the words True and False are included on the arrows coming from the diamond. This is vital to include in your diagram; this is the only reference in the flowchart to tell you which branch is to be executed when the condition is True and which one when the condition is False.

Let's get some practice by converting the previous problem 6.1, into a flowchart.


Start
a = 4
b = 3
a > b
True
False
Print "Variable a is greater than variable b"
Print "Variable b is greater than or equal to variable a"
Stop

We can use this decision pattern to create a complete program. Let us discuss some problems to illustrate this point.

6.4 Problem 6.2

Determine whether a number entered by the user is even or odd.

To determine whether a number is even or not, you divide the number by two. If the remainder is zero, it is an even number otherwise it is odd. Remember from Chapter 2 that the modulus operator returns the remainder of a division. So, if the result of *number % 2* equals zero, it would be an even number, otherwise it would be an odd number.

Figure 6.2 shows the pseudocode, Figure 6.3 shows the flowchart and the Figure 6.4 shows the Python code for the problem.

Figure 6.2: Pseudocode

```
Start
  print “Please input a number”
  get num
  If (num % 2) == 0 Then
    print “Even Number”
  Else
    print “Odd Number”
  EndIf
Stop
```

Figure 6.3: Flowchart

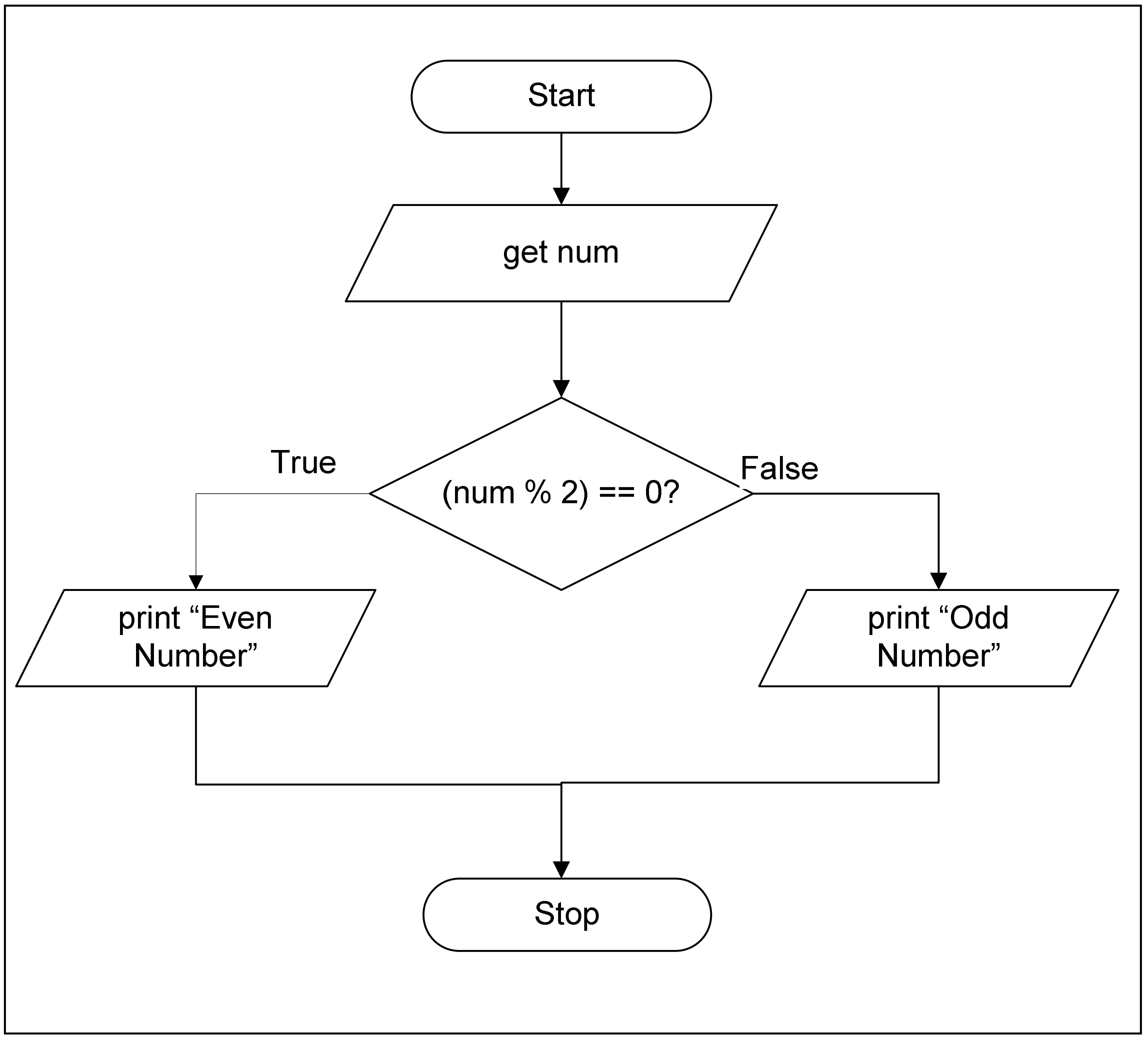


Figure 6.4: Python program

```
#This program inputs a number and prints if it is odd
#or even
num = int(input("Enter a number: "))
if num % 2 = = 0:
  print ("Even number")
else:
  print ("Odd number")
```

Figure 6.5: Output

Sample Data 1:

```
Enter a number: 6
Even number
```

num = 6
Since 6 % 2 = 0, "Even number" is printed.

Sample Data 2:

```
Enter a number: 17
Odd number
```

num = 17
Since 17 % 2 = 1, "Odd number" is printed.

To give you more practice with flowcharts, let's look at one more problem.

6.5 Problem 6.3

Read in the rate of pay (in dollars per hour) and the number of hours an employee has worked for a week. Calculate the amount the employee should be paid according to the following rules:

(a) Regular pay: up to 40 hours, at the given rate.
(b) Overtime pay: for each hour above 40, at 1.5 times the given rate.

Figure 6.6: Pseudocode

```
Start
  print "Please enter rate of pay: "
  get rate
  print "Please enter hours worked: "
  get hours
  If hours > 40 Then
    pay = 40 * rate + (hours - 40) * 1.5 * rate
  Else
    pay = hours * rate
  EndIf
  print "Pay=$" + pay
Stop
```

Figure 6.7: Flowchart

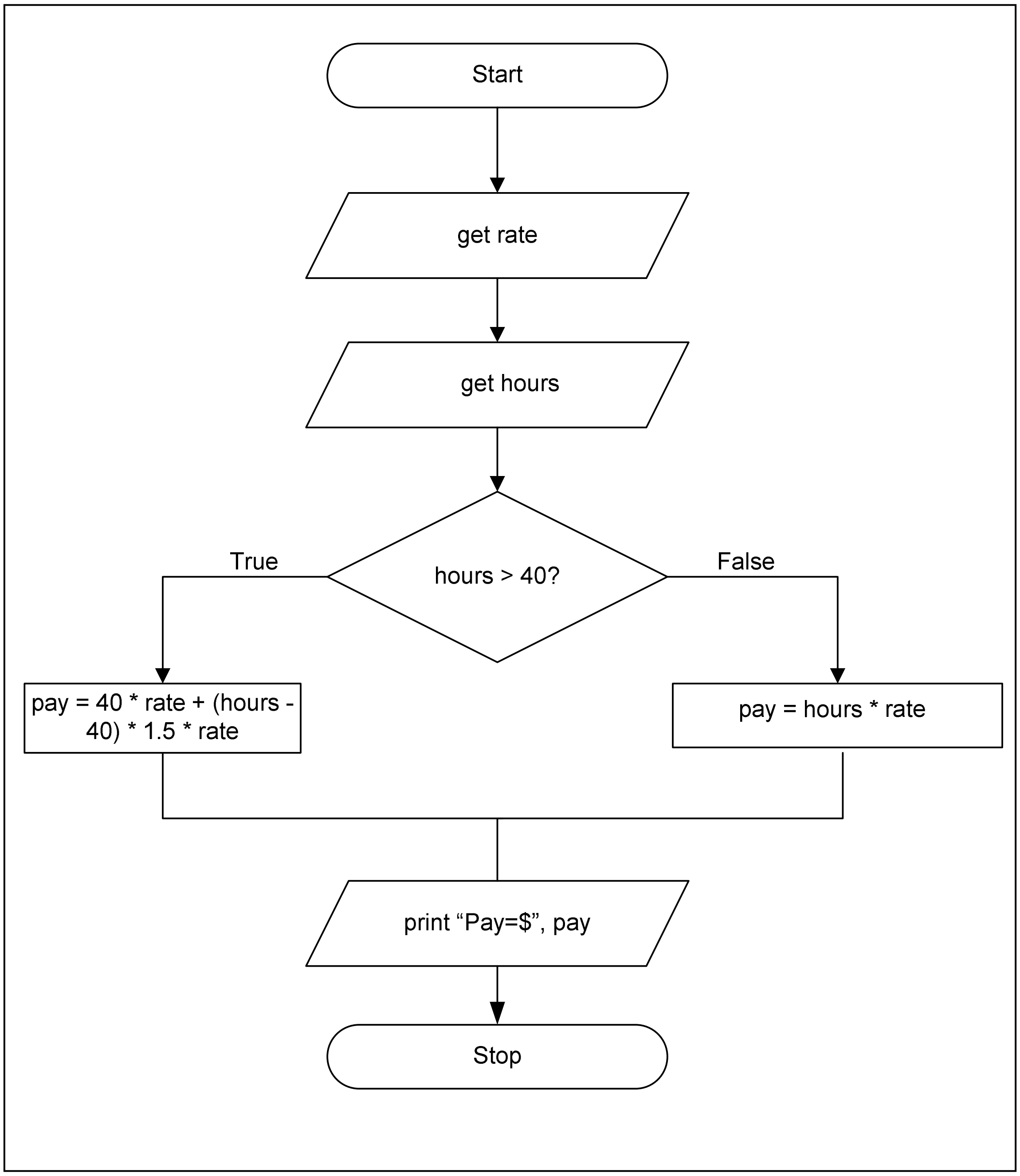


Figure 6.8: Python program

```
#This program calculates and prints pay given
#rate and hours worked
rate = float(input("Enter rate: "))
hours = float(input("Enter hours worked: "))
if hours > 40:
  pay = 40 * rate + (hours - 40) * 1.5 * rate
else:
  pay = hours * rate
print ("Pay=$", pay)
```

Figure 6.9: Output

Sample Data 1:

rate = 6
hours = 30
*pay = 6 * 30 = 180.*

```
Enter rate: 6
Enter hours worked: 30
For this week you earned $ = 180.0
```

Sample Data 2:

rate = 6
hours = 50
*pay = 40 * 6 + (50 – 40) * 1.5 * 6*
*= 240 + 10 * 1.5 * 6*
= 240 + 90
= 330

```
Enter rate: 6
Enter hours worked: 50
For this week you earned $ = 330.0
```

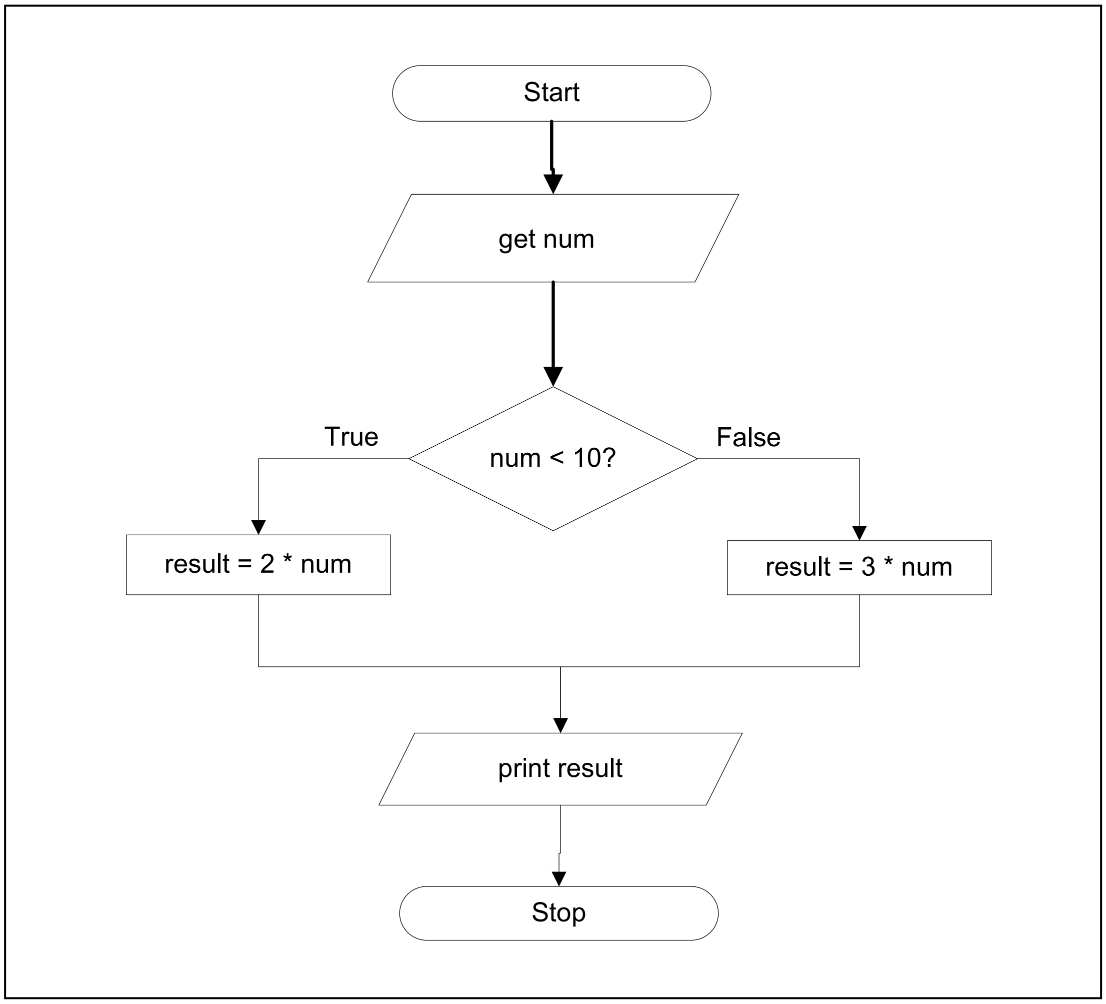
Key Terms

If	falseChoice
Else	Then
Endif	trueChoice

Exercises

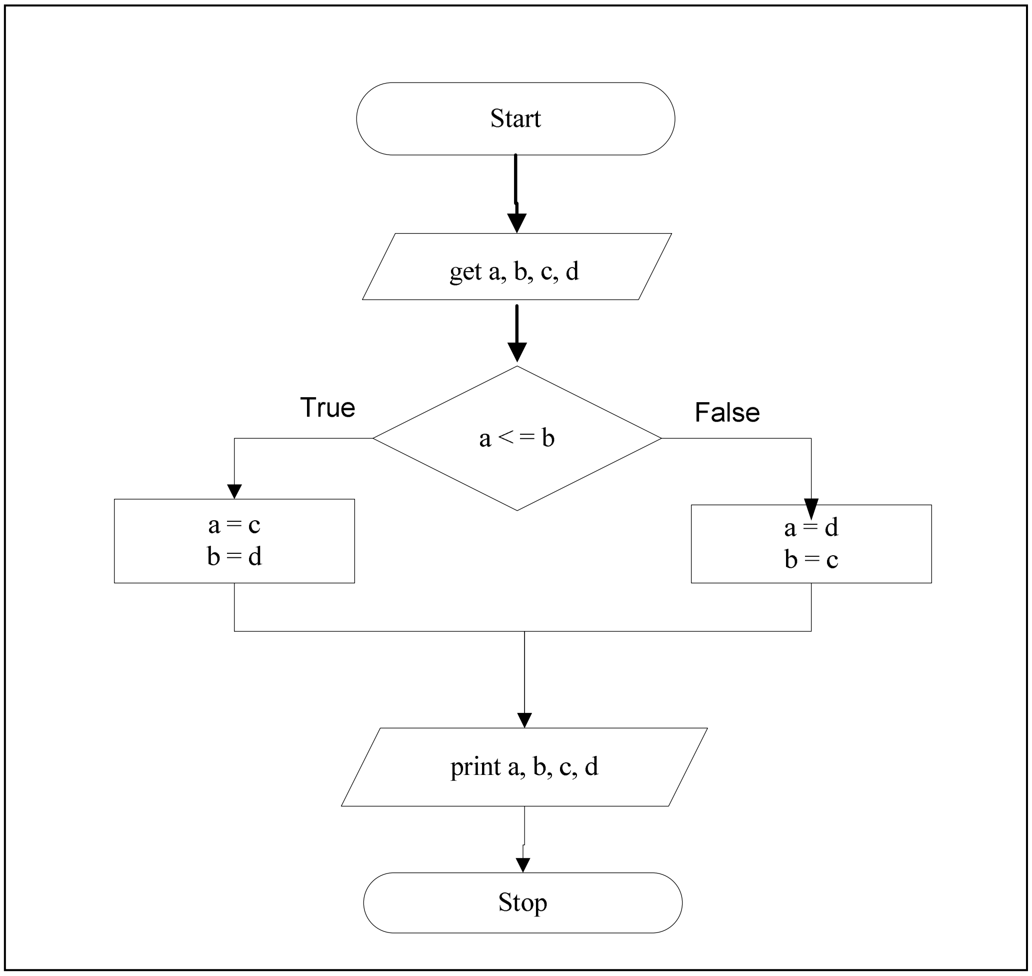
1. What will be printed for each case in the flowchart below? Assume the following values:

    a. *num = 8*
    b. *num = 25*
    c. *num = 10*



2. What will be printed for each case in the flowchart below? Assume the following values:

a. *a*= 8, b=9, c=4, d=8
b. *a*= 12, b=5, c=4, d=8
c. *a*= 8, b=8, c=4, d=8



3. Create a flowchart that will tell the user if they need to take their jacket based on the temperature outside.

a. First ask the user what the temperature is.
b. If the temperature is over 65 degrees then tell the user that they will NOT need a jacket
c. Otherwise, if the temperature is 65 or less then encourage them to bring a jacket.

4. Create pseudocode for a tip program based on if the user thought the restaurant service was acceptable or not.

a. Ask the user for the amount on the bill.
b. Ask the user if they thought that the restaurant service was acceptable or not.
c. If they didn't like the service, then give a 10% tip.
d. If they did like the service, then give a 20% tip.
e. Your program should return the final bill value with tip included that the patron should pay.

5. In the guessing game "Hi/Low", you see if the user can guess your number. If the user does not guess your number, then you print out if they were Hi or Low.

a. Create a variable yourNumber and assign it a value.
b. Ask the user for their guess.
c. If the number guessed is above yourNumber, print out "You guessed too high."
d. In a separate *If* statement, if the number guessed is below yourNumber, print out "You guessed too low."
e. In a separate *If* statement, if the number guessed is equal to yourNumber, print out "Congratulations, you guessed my number."

6. You are working as a cashier for the fundraiser to raise money for your computer science club. Assuming that the patron always gives the exact change or overpays and requires change, complete the following program to help you out.

a. Ask for the amount of money that the patron owes.
b. Ask for the amount money that the patron gives you.
c. If the amount of money given is more than the amount of money owed, then return the amount of money they overpaid.

7. Nested If-Then-Else and Case Statements

Most real-world problems are more complicated than we have shown so far. These complicated problems are based on the same ideas that have been expressed in previous chapters, but we just need the tools to expand those ideas. That is what we will examine in this chapter.

7.1 Nested If-Then-Else

When you have a choice that is dependent on another choice, you cannot use a simple *If-Then-Else* statement. We need to use a **multiple-alternative selection structure**. In this structure, an *If-Then* is followed by a dependent *If-Then*. This style **nests** the *If-Then-Else* inside one another to solve more complex problems. Let us examine this scenario with multiple problems discussed below.

Problem 7.1

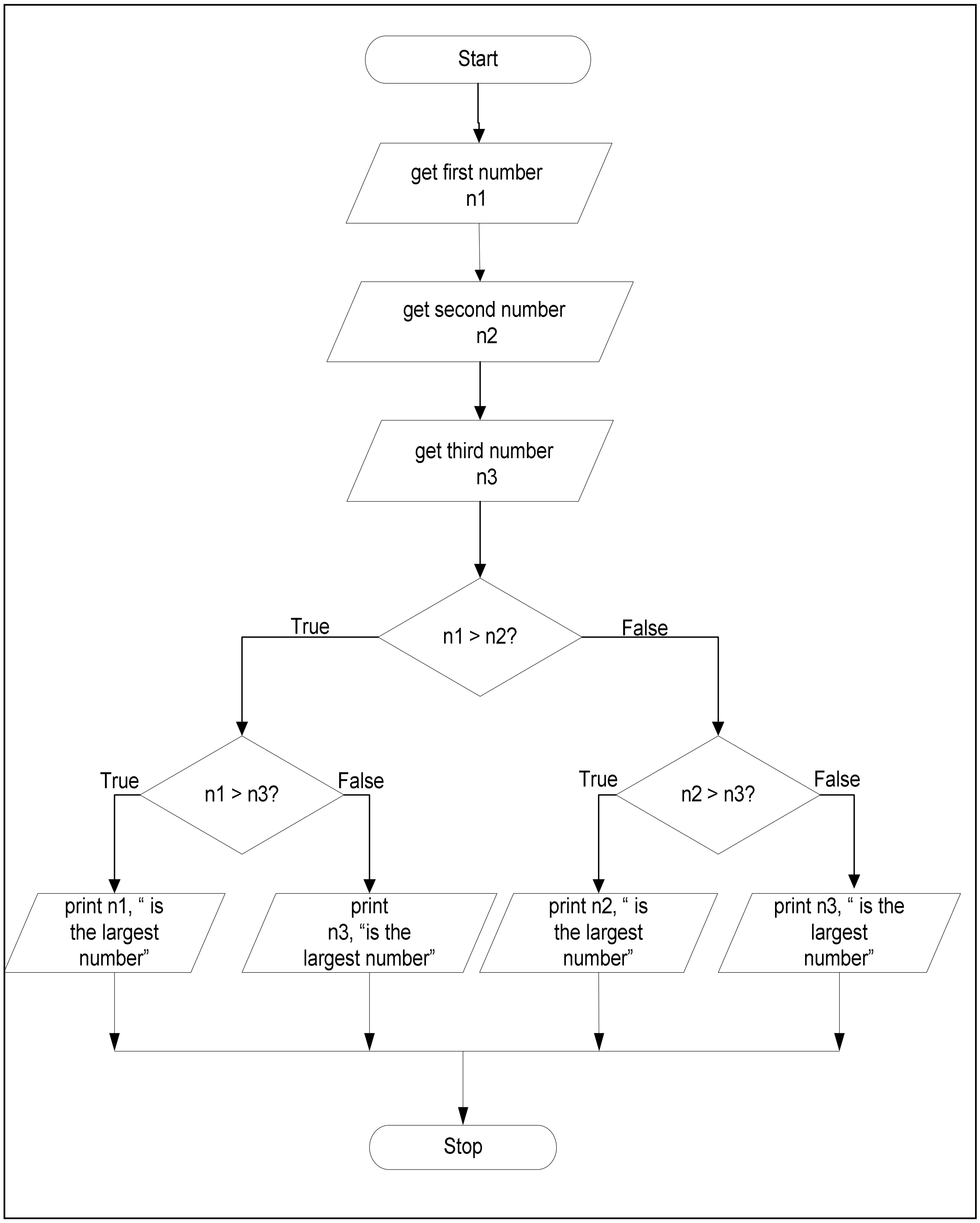
Read three numbers and print the largest among them.

Figure 7.1 shows the pseudocode and the Figure 7.2 shows the flowchart necessary to accomplish this task.

Figure 7.1: Pseudocode

```
Start
  print “Enter first number: ”
  get n1
  print “Enter second number: ”
  get n2
  print “Enter third number: ”
  get n3
  If n1 > n2 Then
    If n1 > n3 Then
      print n1 + “ is the largest number”
    Else
      print n3 + “ is the largest number”
    EndIf
  Else
    If n2 > n3 Then
      print n2 + “ is the largest number”
    Else
      print n3 + “ is the largest number”
    EndIf
  EndIf
Stop
```

Figure 7.2: Flowchart


Start
get first number n1
get second number n2
get third number n3
n1 > n2?
True
False
n1 > n3?
True
False
n2 > n3?
True
False
print n1, " is the largest number"
print n3, "is the largest number"
print n2, " is the largest number"
print n3, " is the largest number"
Stop

Sample Data 1:

n1 = 10
n2 = 25
n3 = 5

Using the above sample data in the first comparison, *n1* is compared with *n2* (note *n1* > *n2*, i.e. 10 > 25). Since the statement is False, the False path is followed. Now *n2* is compared with *n3* (note *n2* > *n3*, i.e., 25 > 5). Since the statement is True, the True path is followed and “25 is the largest number” is printed to the computer screen.

Sample Data 2:

n1 = 15
n2 = 12
n3 = 18

Using the sample data 2 in the second comparison, *n1* is compared with *n2* (note *n1* > *n2*, i.e. 15 > 12). Since the statement is True, the True path is followed. Now *n1* is compared with *n3* (*n1* > *n3* is done, i.e., 15 > 18). Since the statement is False, the False path is followed and “18 is the largest number” is printed.

Figure 7.3: Python program

```
#This program calculates and prints the largest of the
#three numbers
N1 = int(input("Enter first number: "))
N2 = int(input("Enter second number: "))
N3 = int(input("Enter third number: "))

if N1 > N2:
  if N1 > N3:
    print (N1, " is the largest number")
  else:
    print (N3, " is the largest number")
  else:
    if N2 > N3:
      print (N2, " is the largest number")
    else:
      print (N3, " is the largest number")
```

Figure 7.4: Output

```
Enter first number: 10
Enter first number: 25
Enter first number: 5
25 is the largest number
```

Self-Check 7.1

Read a score and print the corresponding grade. The grade is calculated as follows:

Score	*Grade*
90 or above	*'A'*
>= 80 but less than 90	*'B'*
>= 70 but less than 80	*'C'*
>= 60 but less than 70	*'D'*
less than 60	*'F'*

Create the flowchart, pseudocode and Python code for this problem.

7.2 Case Statement

When there are lots of choices that require multiple *Else If*s in an *If-Then-Else*, there is another condensed alternative to writing the conditions; using a **case** statement. Note that a case statement will only work when you have an exact value that the variable can be equal to.

Figure 7.5: General case statement

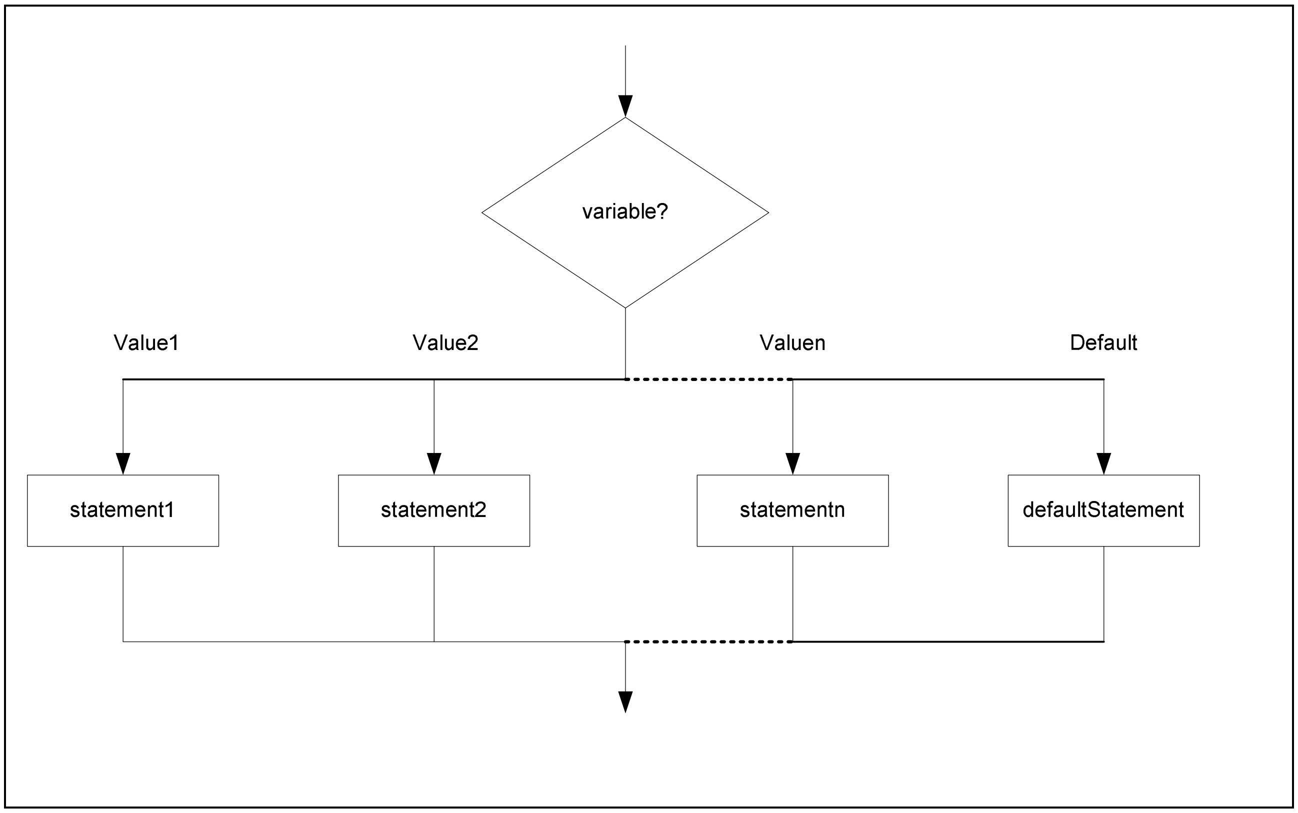
```
Case of variable:
  Case value1: statement1: break
  Case value2: statement2: break


  Default: defaultStatement
EndCase
```

Here *value1*, *value2* and other possible values would be the same data type as the *variable*. When the *variable* has a particular value, it would print out the associated statement. So if *variable* = *value2* then *statement2* would be executed. In this case the *Default* is just like the *Else* in an *If-Then-Else*; the default statement will be executed if *variable* does not match the other values given. Also note that there can be one or more statements after every case and *default*. Flowchart 7.2 shows the flowchart for the general case statement.

When all the lines of code are executed for one case, a **break** is required to let the computer know when the case is finished. Without a break you end up with **fall-through** code, where all statements will be executed until a break, or the *EndCase* is found.

Figure 7.6: General case statement



Problem 7.2

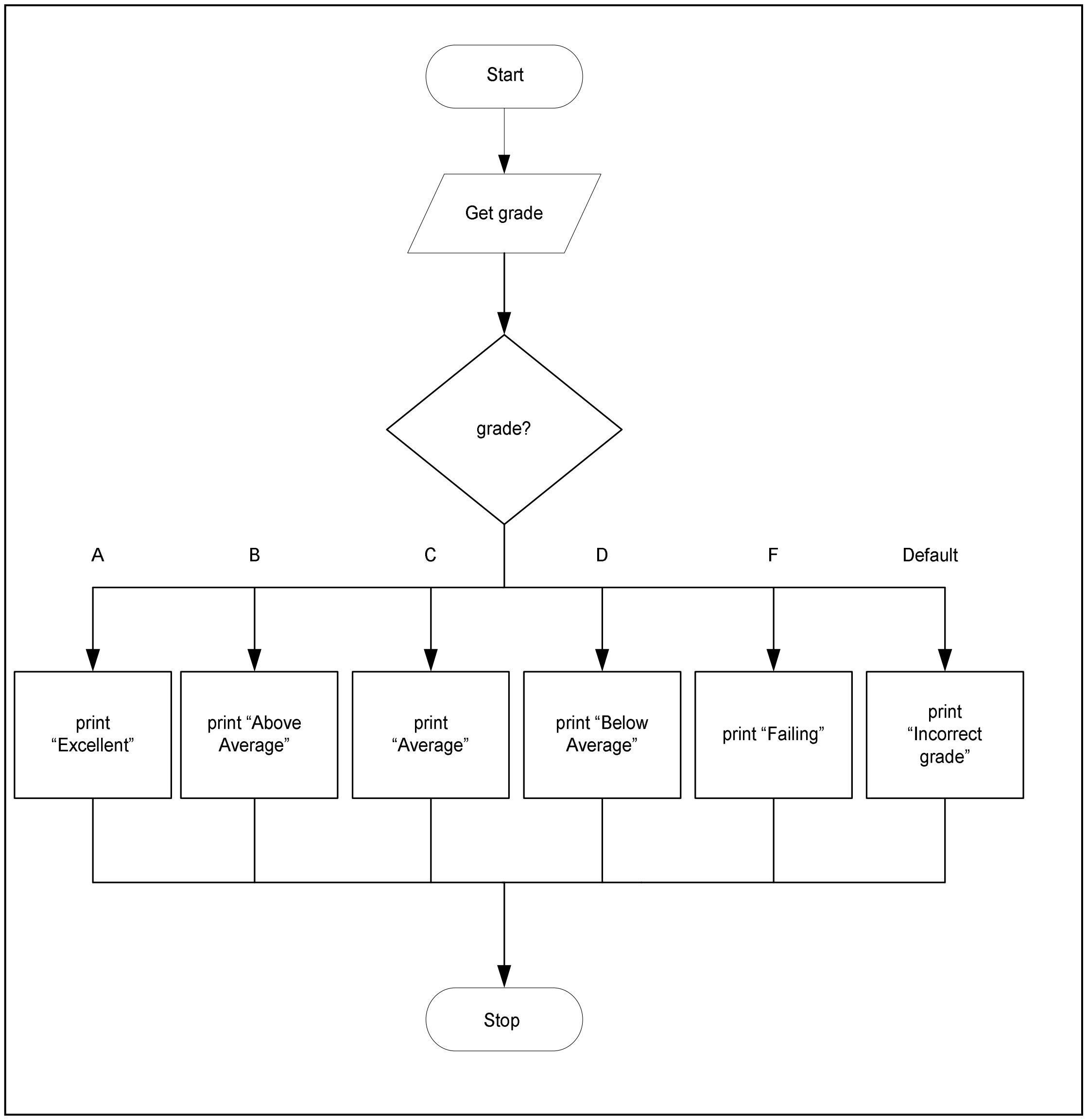
Create a grade program, assuming that we know the student’s letter grade.

Figure 7.7: Pseudocode

```
Start
  print “Please enter the student’s grade.”
  get grade
  Case of grade:
    Case ‘A’: print “Excellent”: break
    Case ‘B’: print “Above Average”: break
    Case ‘C’: print “Average”: break
    Case ‘D’: print “Below Average”: break
    Case ‘F’: print “Failing”: break
    Default: print “Incorrect grade”
  EndCase
Stop
```

The flowchart for a case statement branches on all the choices and condenses the space that would be needed for an *If-Then-Else*.

Figure 7.8: Flowchart



Python doesn't have a case statement- instead you will need to use the *nested If-Then-Else* structure to perform the task.

Self-Check 7.2

Get two numbers and a code for the operation to be performed on those two numbers. Use the following table for valid "op" codes types and their meaning:

OP CODE	MEANING
A	Addition
S	Subtraction
M	Multiplication
D	Division
%	Remainder (MOD)

Perform the operation and print the answer.

Create the pseudocode, flowchart and Python code for the problem.

Since Python doesn't have a case statement, the code will use If-Then-Else as you have seen before.

Key Terms

break	multiple-alternative selection structure
case statement	nest
fall-through	

Exercises

1. Draw a flowchart to do the following:

   a. Get total *income* and total *expense* from a user.
   b. Print "It is a profit" if *income* > *expense*
      Print "It is neither profit nor loss" if *income* equals *expense*
      Print "It is a loss" if *income* < *expense*.

2. Create a case statement in pseudocode for a calendar program that will take in user input as a number from 1 to 12 and print out the associated month's name.

3. Draw a flowchart which will print either "0 week vacation", or "2 week vacation" or "3 week vacation" depending on the following rule:

Number of years employed	Weeks of vacation
0 - 2	0
2 - 5	2
More than 5	3

4. Draw a selection structure which will read a number and:

   Print "Number is positive" if the number is > 0.
   Print "Number is zero" if the number is 0.
   Print "Number is negative".

5. Create a flowchart for a program to display a menu as shown below. Based on the user's choice, display the volume of a cube, or a rectangular solid, or a cylinder, or a sphere, or a cone.

   <u>Menu</u>
   1 = volume of a cube
   2 = volume of a rectangular solid
   3 = volume of a cylinder
   4 = volume of a sphere
   5 = volume of a cone

   Hints:

   If user's choice = 1, prompt the user for the side of a cube (*s*), use the following formula:

   $$volume = s^3$$

   If user's choice = 2, prompt the user for the length of the rectangular solid (*l*) , width of the rectangular solid (*w*), height of the rectangular solid (*h*) , use the following formula:

$$volume = l * w * h$$

If user's choice = 3, prompt the user for the height of the cylinder (*h*), radius of the cylinder (*r*). Note that *pi* is 3.14. Use the following formula:

$$volume = pi * r^2 * h$$

If user's choice = 4, prompt the user for the radius of the sphere (*s*), Use the following formula:

$$volume = 4/3 * pi * r^3$$

If user's choice = 5, radius of the cone (r), height of the cone (h), Use the following formula:

$$volume = 1/3 * pi * r^2 * h$$

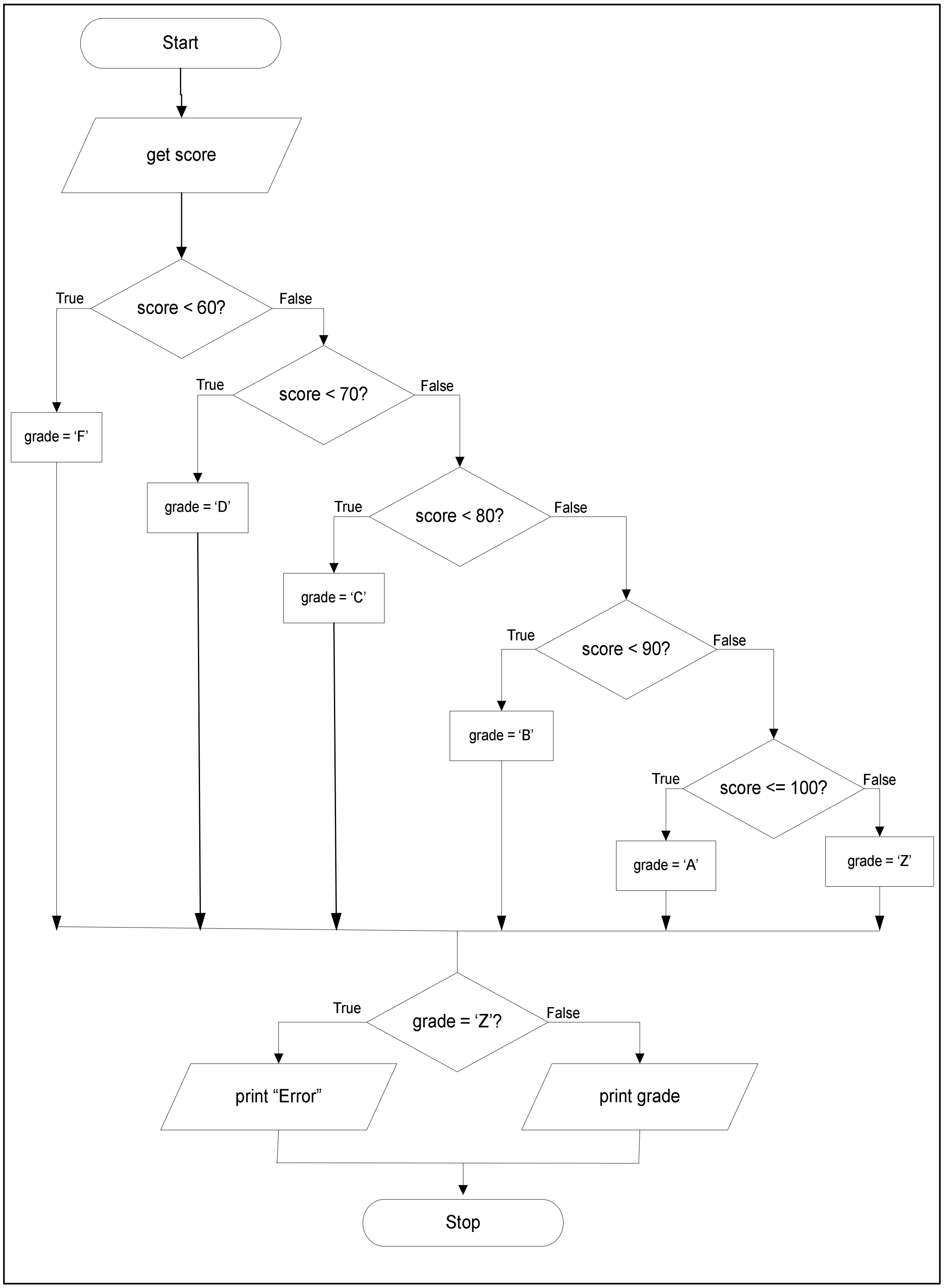
6. Write the pseudocode for the following scenario:
   There is a sale at your local clothing store. This store is selling short sleeved blouses for $14.59 dollars each, tank tops for $9.99 each and sweaters for $19.99. Based on what you choose to buy, the amount you owe is different. Include the *Else* just in case the user typed in the wrong choice.

Self-Check Solutions

7.1
Pseudocode:

```
Start
  print “Please enter a score “
  get score
  If score < 60 Then
    grade = ‘F’
  Else If score < 70 Then
    grade = ‘D’
  Else If score < 80 Then
    grade = ‘C’
  Else If score < 90 Then
    grade = ‘B’
  Else If score <= 100 Then
    grade = ‘A’
  Else
    grade = ‘Z’
  EndIf
  If grade = ’Z’ Then
    print “Error”
  Else
    print grade
  EndIf
Stop
```

Flowchart:


Start
get score
score < 60?
True
False
grade = 'F'
score < 70?
True
False
grade = 'D'
score < 80?
True
False
grade = 'C'
score < 90?
True
False
grade = 'B'
score <= 100?
True
False
grade = 'A'
grade = 'Z'
grade = 'Z'?
True
False
print "Error"
print grade
Stop

Python code for self-check exercises:

```
#This program prints grade given score

score = int(input("Enter score: "))

if score < 60:
  grade = 'F'
elif score < 70:
  grade = 'D'
elif score < 80:
  grade = 'C'
elif score < 90:
  grade = 'B'
elif score <= 100:
  grade = 'A'
else: grade = 'Z'
if grade == 'Z':
  print ("ERROR")
else:
  print ("Grade = ", grade)
```

Corresponding output:

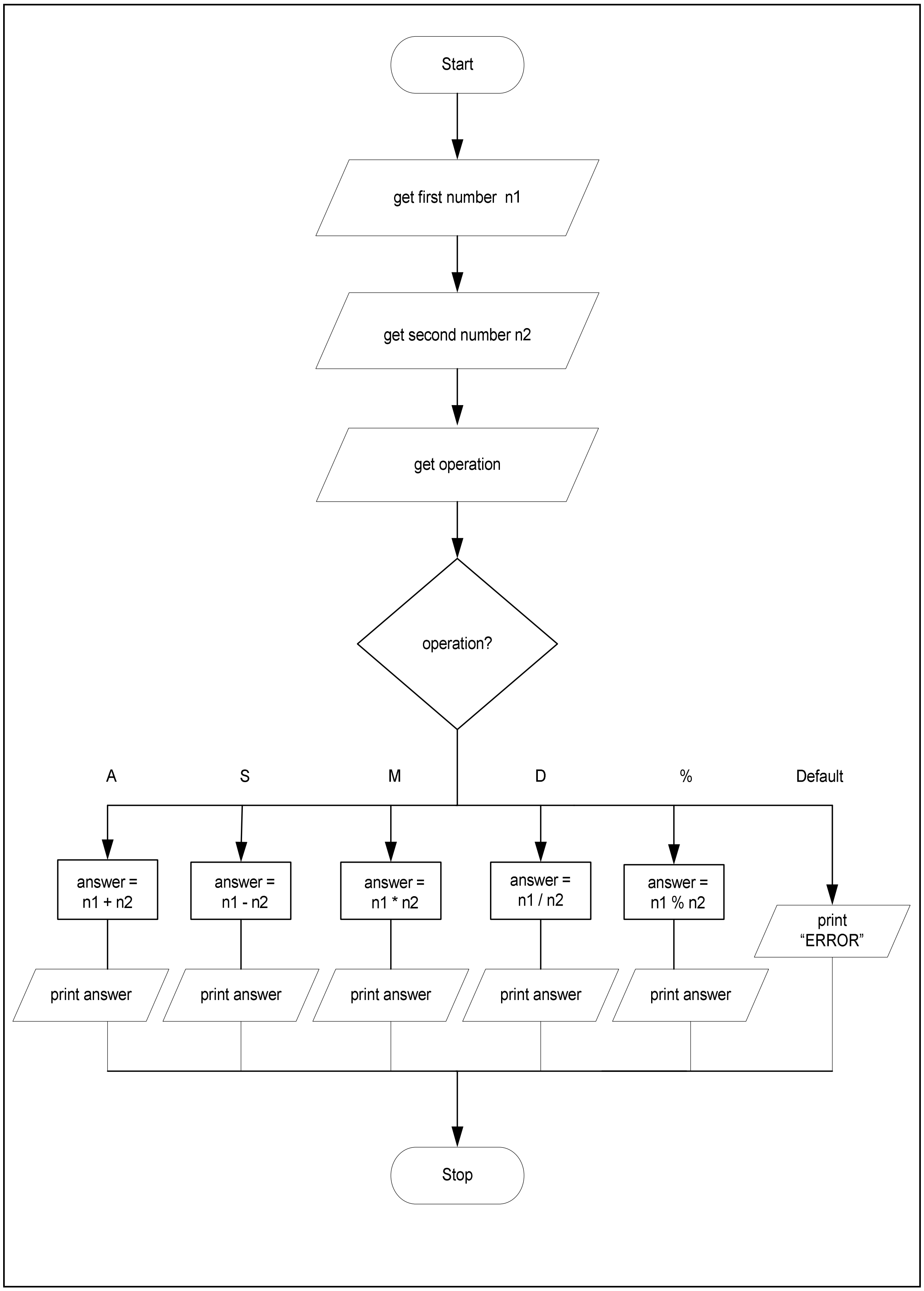
```
Enter score: 75
Grade = C
```

7.2

Pseudocode:

```
Start
  print “Enter first number: ”
  get n1
  print “Enter second number: ”
  get n2
  get operation
  Case of operation:
  Case ‘A’: answer = n1 + n2
    print answer
    break
  Case ‘S’: answer = n1 – n2
    print answer
    break
  Case ‘M’: answer = n1 * n2
    print answer
    break
  Case ‘D’: answer = n1 / n2
    print answer
    break
  Case ‘%’: answer = n1 % n2
    print answer
    break
  Default: print “ERROR”
  EndCase
Stop
```

Flowchart:


Start
get first number n1
get second number n2
get operation
operation?
A
S
M
D
%
Default
answer = n1 + n2
answer = n1 - n2
answer = n1 * n2
answer = n1 / n2
answer = n1 % n2
print "ERROR"
print answer
print answer
print answer
print answer
print answer
Stop

8. Repetition Structures

In life we encounter tasks that we must repeat every day. Let's consider when you walk from your car to a building. To make progress toward your goal of reaching the building you will need to put one foot in front of the other, reducing the distance to the building at each step. When you repeat an action or process, you have a **loop**. Placing your right foot in front of the left foot until you reach the building is a loop, in this case a loop of steps. All loops must have a starting place and ending location; in this case you start at your car and end at the building.

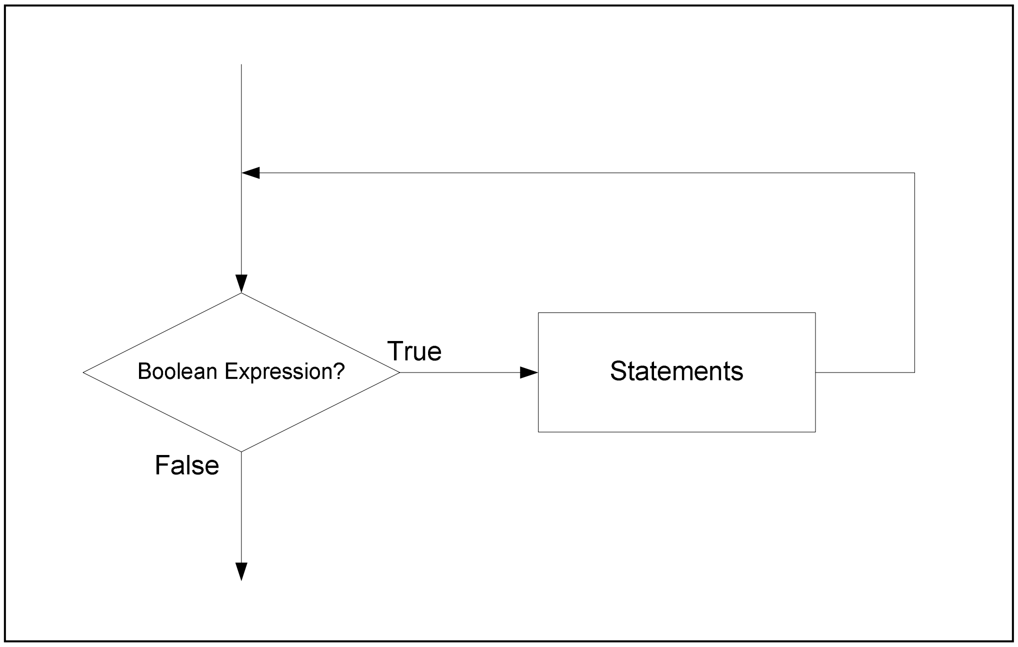
8.1 While loops

The basic loop structure is the ***while*** loop. From this one type of loop we will see that the other loop types are based on very similar ideas. Within a *while* loop there are some very important pieces that need to be determined before you create a loop.

a. The **initialization** will assign a starting value for a variable.
b. The **condition** is a Boolean expression that will execute the statements in the body of the loop while the condition is True.

First let's look at the flowchart for our *while* loop. We already have all the shapes needed to illustrate the concept of a *while* loop.

Figure 8.1: Flowchart for a general while loop



Note that the translation from flowchart to pseudocode is very direct; we just need to add a note for when the loop is finished.

Figure 8.2: Pseudocode for general while loop

```
initialization
While condition Do
   statements
   …
EndWhile
```

One type of loop is a **counter controlled loop**. In this case, there is a designated variable to update the condition when the loop executes. One type of update could be an **increment** or **decrement** which will change the value of the variable in the *condition*. If the value of the variable doesn't change, then the loop will never end- creating an **infinite** loop.

Let us look at an example of a *counter controlled loop*.

Problem 8.1

Print out the numbers from 1 to 3.

First let's create the pseudocode for this problem. We will need to initialize a variable, *count*, to be the first number that we want to print, 1.

```
Start
  count = 1
```

Starting the loop, we need to state our **ending condition**, the condition that will make us leave the loop. In this case, the ending condition is after we have printed the value of *count* as 3, which is the expression *count* <= 3.

```
Start
  count = 1
  While count <= 3 Do
```

The body of the loop contains the statements. For this loop, we only need to print the value of count, so we have:

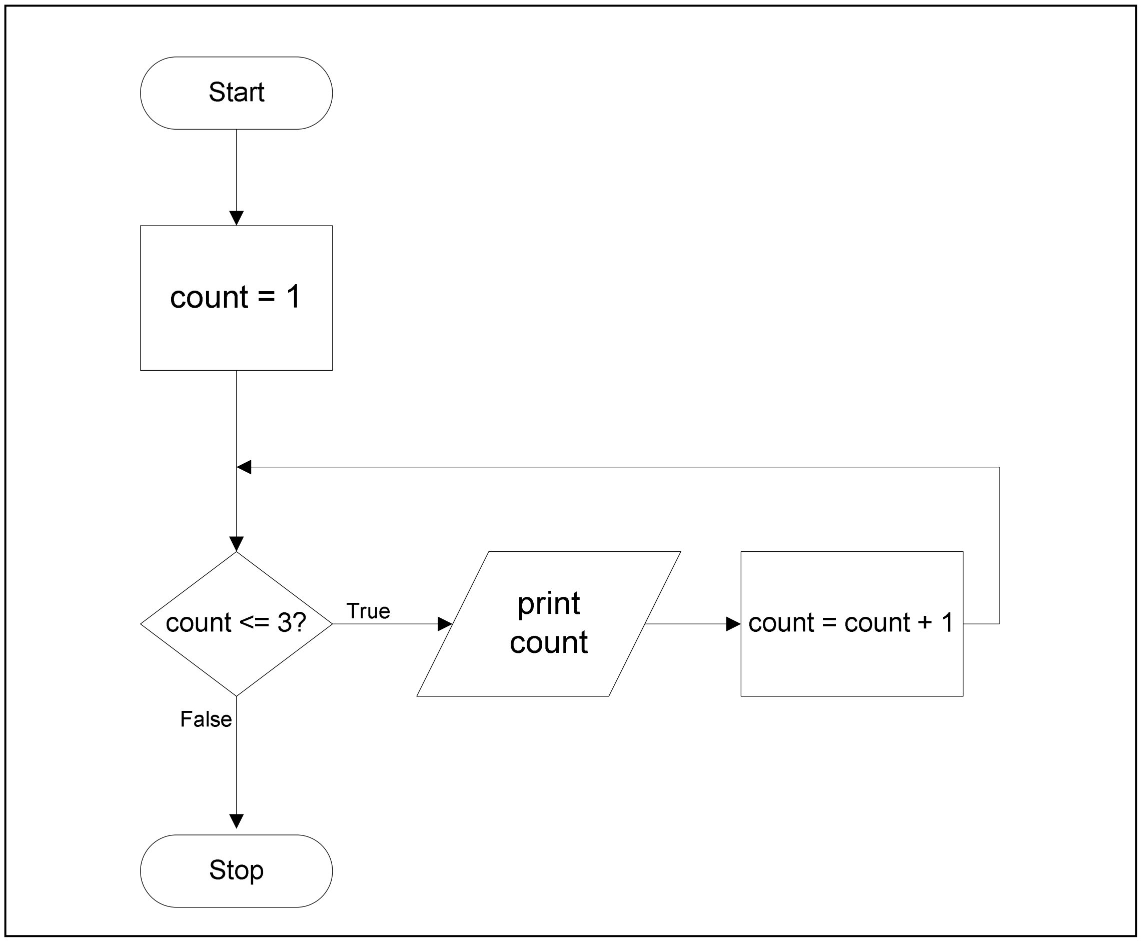
```
Start
  count = 1
  While count <= 3 Do
    print count
```

Finally we complete the loop with the increment or decrement line. In this case we want *count* to increment by 1 each time in the loop. Now our completed pseudocode is:

```
Start
  count = 1
  While count <= 3 Do
    print count
    count = count + 1
  EndWhile
Stop
```

And the translation into the flowchart follows directly:

Figure 8.3: Flowchart



To follow how the pseudocode and flowchart are behaving, we often use the technique of tracing the code. **Code tracing** requires you to step through a program as a computer would and update the variables as appropriate.

To look at this process we are going to first create a table to keep track of the variables and output, in this case our only variable is *count*.

count	Output

Now as we execute each line (indicated in bold), the chart will be updated. First we execute the first line of the code

```
count = 1
While count <=3 Do
  print count
  count =count + 1
EndWhile
```

count	Output
1	

As we go to the second line, we check to make sure that our condition, *count* <= *3* is met. Since the condition is True, we continue to the third line and update the output of the chart.

```
count = 1
While count <=3 Do
  print count
  count =count + 1
EndWhile
```

count	Output
1	1

On the fourth line we increase the value of *count* by one by crossing out the old value and updating the chart with the new value. By keeping the old values around, we can see possible errors in the behavior of our code when we debug.

```
count = 1
While count <=3 Do
  print count
  count =count + 1
EndWhile
```

count	Output
~~1~~	1
2	

Now that we have hit the end of the loop, we return to the second line and check our condition again. Since *count* = *2*, and *2* <= *3*, our condition is met and we can continue to the third line and update our output. Note that the previous value of the output will remain since we are adding to the 1 we already printed to the screen.

```
count = 1
While count <=3 Do
  print count
  count =count + 1
EndWhile
```

count	Output
~~1~~	1
2	2

Continuing to the fourth line, we update the value of *count*, crossing out the previous value.

```
count = 1
While count <=3 Do
  print count
  count =count + 1
EndWhile
```

count	Output
~~1~~	1
~~2~~	2
3	

We have hit the end of the loop and return to the second line to check our condition. Now *count* = *3*, and *3* <= *3* is still True, so we will continue with the body of the loop again, continuing to the third line and updating the output.

```
count = 1
While count <=3 Do
  print count
  count =count + 1
EndWhile
```

count	Output
~~1~~	1
~~2~~	2
3	3

Continuing to the fourth line, we update the *count*.

```
count = 1
While count <=3 Do
  print count
  count =count + 1
EndWhile
```

count	Output
~~1~~	1
~~2~~	2
~~3~~	3
4	

Now we return to the second line to check our condition to find that
count = *4*, and *4* <= *3* is no longer satisfied. Since the condition returns False, we must exit the loop. Therefore the output to the screen, as displayed in our trace table is shown below:

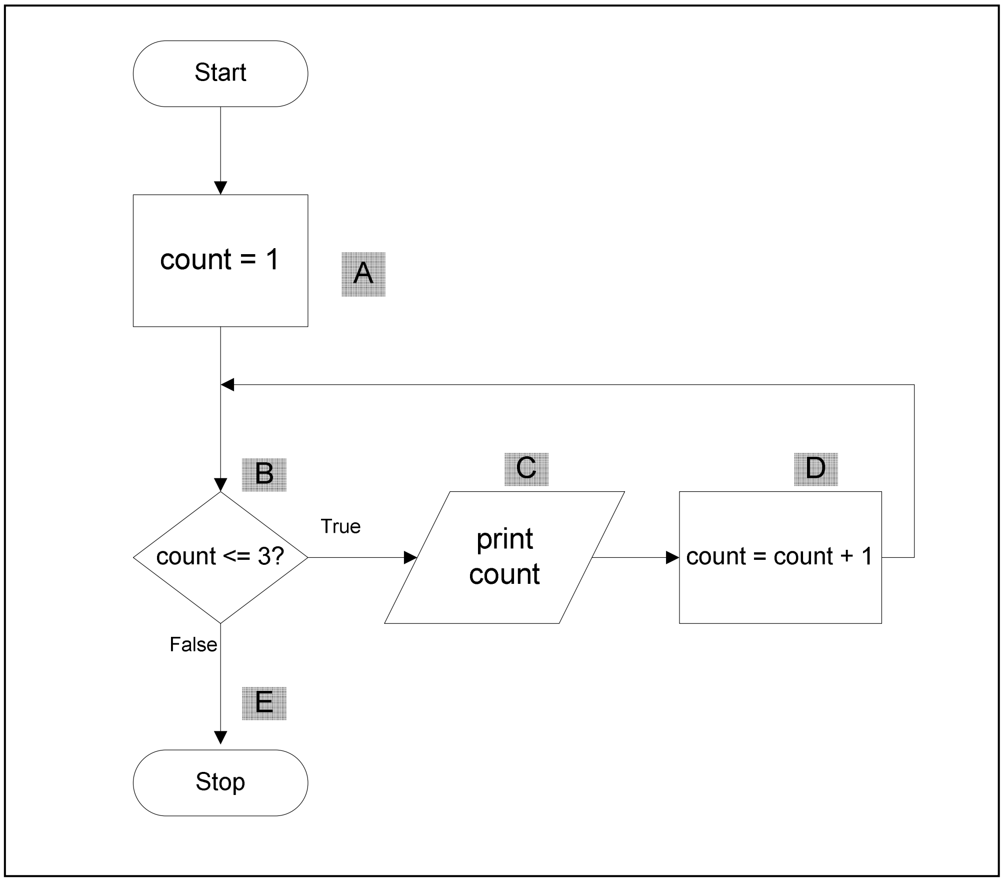
```
1
2
3
```

This output is exactly what we wanted the program to accomplish: print out the numbers from 1 to 3.

Sometimes it is more convenient to keep more information about the progress of executing a code segment. To do this, we need a more formalized charting system to keep track of the variable updates. Let's label our flowchart cells in Table 8.1 to keep track of where we are at in the code.

The Figure 8.4 is reintroduced below with the reference letters to help illustrate the execution process of the flowchart.

Figure 8.4: Flowchart



The following trace table will be using the references (A-E) in the above flowchart to refer to different locations within the program.

Table 8.1: Trace table for problem 8.1

Iteration #	Box #	count	count<= 3?	Output
Start				
	A	1		
1	B		True	
	C			1
	D	2		
2	B		True	
	C			2
	D	3		
3	B		True	
	C			3
	D	4		

4	B		False	
Stop	E			

Before the loop, the value of *count* is set to *1*, as seen in the first white row of the trace table. The condition in box B is *count* <= *3* i.e., *1* <= *3*, the condition is True and the value of 1 is printed to the screen (box C) and recorded in the Output column. Continuing to box D, the value of *count* is incremented to 2 and the loop brings us back to check the condition in box B. This process is repeated and the results recorded in the trace table (see Table 8.1) until the condition returns a False value and we leave the loop.

Note that the column "Iteration #" helps keep track of how many times we have been in the loop. For this loop, the iterations match the value of count. We update the value in the iteration column every time we re-evaluate the condition for the *while* loop.

Finally, let's create our Python program. Our code would be modified to the following and we can check that we do get the correct output. The only changes from the pseudocode to Python are that the condition is followed by a colon (without Do) and there is no need for an *EndWhile* since the indented code is in the loop.

Figure 8.5: Python program

```
#Print numbers from 1 to 3
count = 1
while count <= 3:
  print (count)
  count = count + 1
```

Figure 8.6: Output

```
1
2
3
```

Self-Check 8.1

Using a *While* loop, create the pseudocode, flowchart and Python code to calculate and print squares of numbers between 1 and 4 inclusive. Show the trace table and the final output from the program to the screen.

Problem 8.2

Calculate and print average of scores. An average is calculated by dividing the sum by the count. Note that a score of zero will end the loop.

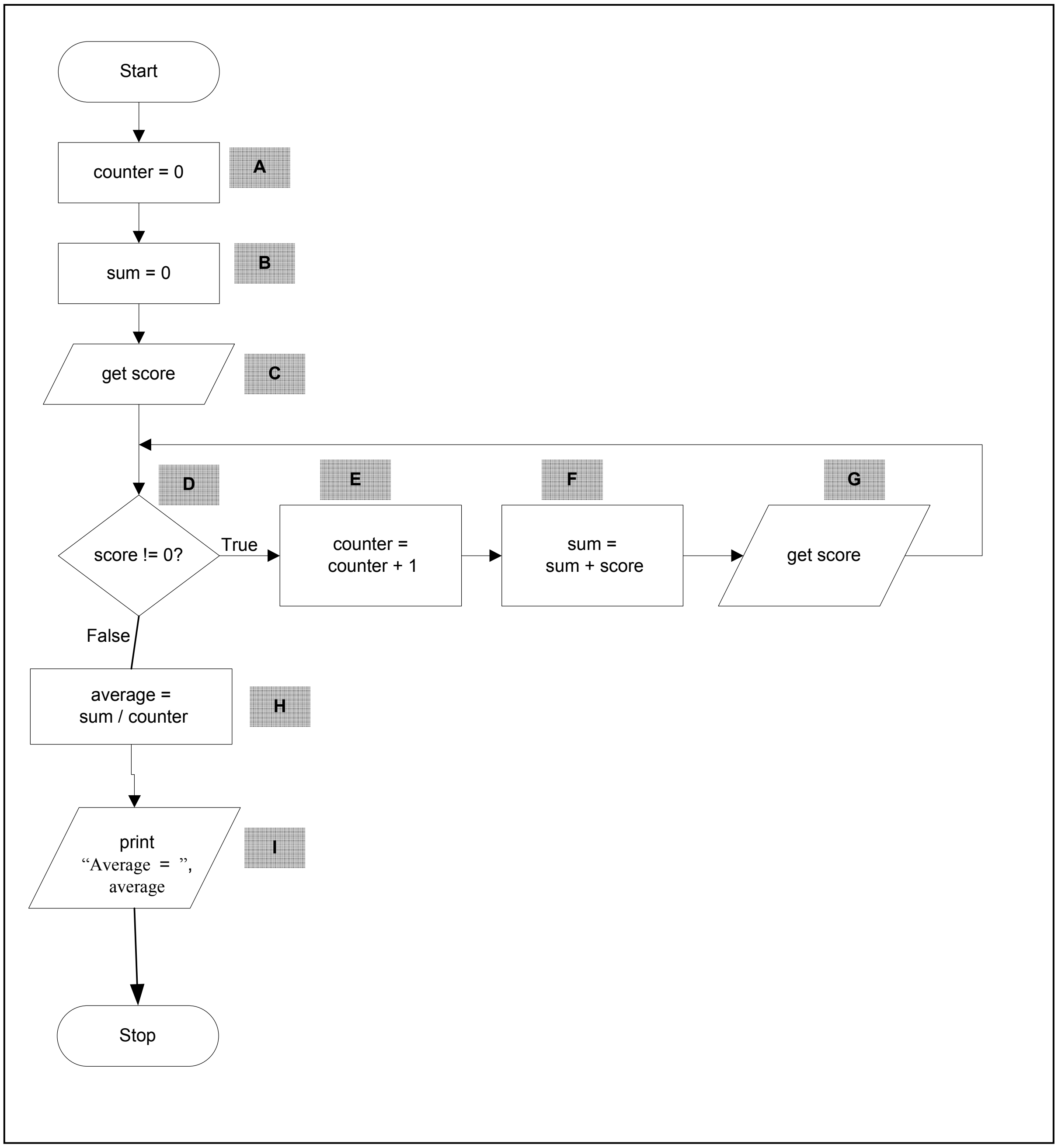
Sample Data:

```
90
80
75
85
0
```

For the data shown above, the average would be: 330/4 = 82.5.

First let’s create the flowchart for this problem.

Figure 8.7: Flowchart



And the translation into the pseudocode follows directly:

Figure 8.8: Pseudocode

```
Start
  counter = 0
  sum = 0
  get score
  While score != 0 Do
    counter = counter + 1
    sum = sum + score
    get score
  EndWhile
  average = sum/counter
  print "Average = ", average
Stop
```

Remember from Chapter 2 that if you divide an integer by another integer, your result will also be an integer. That is not what we want in this problem. Here we want to be as accurate as possible with our division, so we need to declare sum as a float by using the statement sum = 0.0.

Figure 8.9: Python program

```
#This program calculates and prints average of scores
#entered by the user.

counter = 0
sum = 0.0

score = int(input("Enter score: "))
while score != 0:
  counter = counter + 1
  sum = sum + score
  score = int(input("Enter score: "))

average = sum / counter
print ("Average = ", average)
```

Figure 8.10: Output

```
Enter score: 90
Enter score: 80
Enter score: 75
Enter score: 85
Enter score: 0
Average = 82.5
```

Table 8.2: Trace table

Iteration #	Box	counter	sum	score	score !=0	Average	Output
1	A	0					
	B		0				
	C			90			
	D				True		
	E	1					
	F		90				
	G			80			
2	D				True		
	E	2					
	F		170				
	G			75			
3	D				True		
	E	3					
	F		245				
	G			85			
4	D				True		
	E	4					
	F		330				
	G			0			
5	D				False		
1	H					82.5	
1	I						Average = 82.5

Note that the Box letters refer to the same letters in the flowchart. The trace table starts with Box A and B initializing the values of *counter* and *sum* to 0. Box C retrieves the score from the user (in this case *score* is 90) and we begin the loop with Box D. Since the *score* is not 0, we update *counter* to 1, add the *score* to *sum*, to make *sum* 90 and get the next *score* from the user.

This is repeated until a score of 0 is retrieved from the user, at this point we exit the loop, proceeding to Box H and I to print the Average of 82.5.

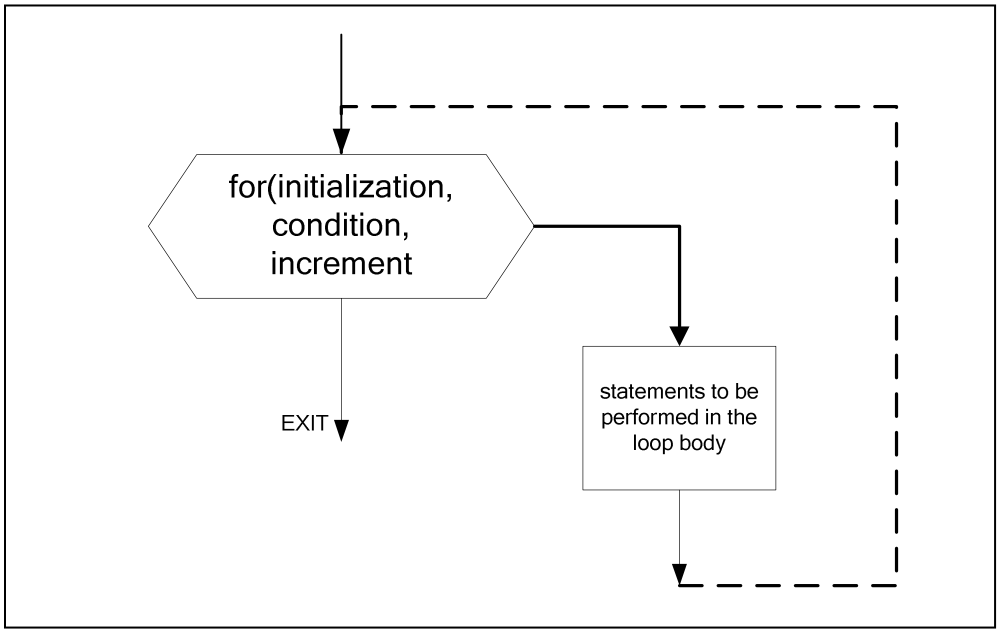
8.2 For loops

Since while loops are used so often, there is a short-cut called a ***for*** loop that incorporates the three key pieces of a *while* loop in a concise style. A *for* loop accomplishes the same task as a *while* loop, it is just considered more succinct by most programmers.

Figure 8.11: Pseudocode for general for loop

```
For (initialization, condition, increment)
  statements
EndFor
```

Figure 8.12: Flowchart for general for loop



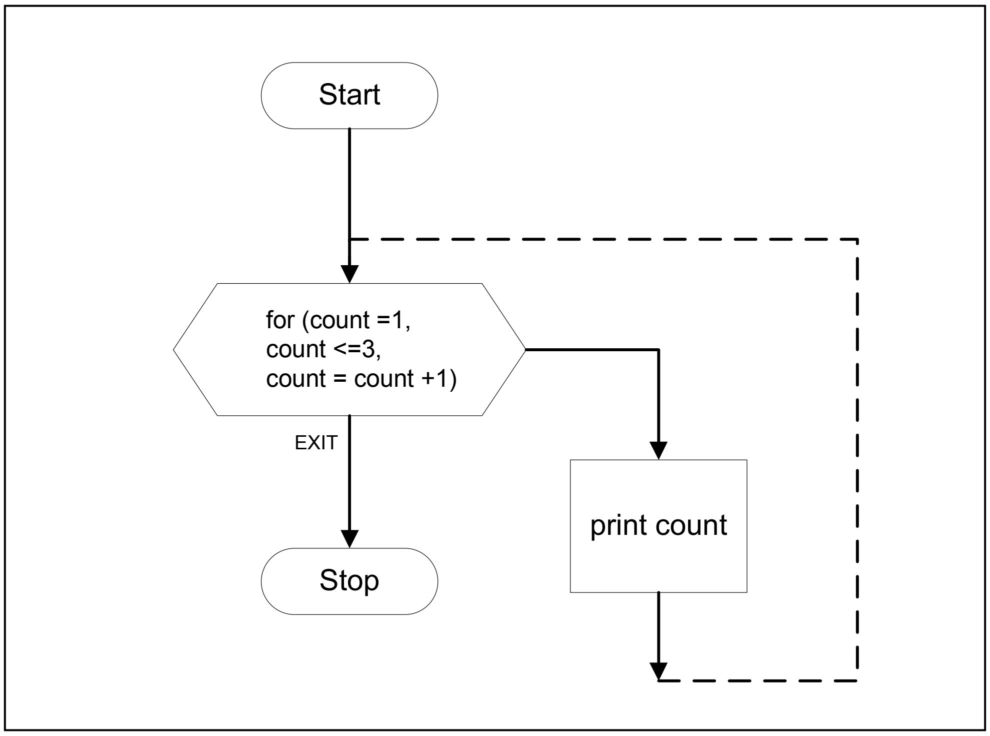
Problem 8.3

This problem is a modification of Problem 8.1. Any *while* loop can be converted to a *for* loop, and vice-versa. So let’s look at Problem 8.1 again and rewrite it as a *for* loop.

Our task remains the same: print out numbers from 1 to 3.

First let’s create the flowchart for this problem.

Figure 8.13: Flowchart



And the translation into the pseudocode follows directly:

Figure 8.14: Pseudocode

```
For (count = 1, count <= 3, count = count + 1)
  print count
EndFor
```

In walking through the loop for the code trace, we start with the initialization, just like in the *while* loop and obtain

```
For (count=1,count<=3,count=count+1)
  print count
EndFor
```

count	Output
1	

Now we check for the condition, again like the *while* loop, and since the *count* = 1, and 1<= 3, we can continue to the second line of code and print the current value of *count*.

```
For (count=1,count<=3,count=count+1)
  print count
EndFor
```

count	Output
1	1

Now we return to the first line and execute the increment, updating our chart appropriately.

```
For (count=1,count<=3,count=count+1)
  print count
EndFor
```

Count	Output
~~1~~	1
2	

Note, like the *while* loop, we now check the condition again (the initialization is executed only once). Since *count* = 2, and 2<= 3 is True, then we continue to the second line and print the updated count to the screen.

```
For (count=1,count<=3,count=count+1)
  print count
EndFor
```

Count	Output
~~1~~	1
2	2

Returning to the first line, we execute the increment and update our chart.

```
For (count=1,count<=3,count=count+1)
  print count
EndFor
```

Count	Output
~~1~~	1
~~2~~	2
3	

Checking our condition, we find that *count* = 3, and 3<= 3 is still True, and continue to execute the second line.

```
For (count=1,count<=3,count=count+1)
  print count
EndFor
```

Count	Output
~~1~~	1
~~2~~	2
3	3

Returning to the first line, we complete the increment and update our chart accordingly.

```
For (count=1,count<=3,count=count+1)
  print count
EndFor
```

Count	Output
~~1~~	1
~~2~~	2
~~3~~	3
4	

Now when we check our condition, we find that *count* = 4, and 4 <= 3 is False, and so we jump out of the loop.

<u>Figure 8.15: Python program</u>

```
#An example of a simple for loop which prints
#out numbers from 1 to 3
for count in range(1, 4):
  print (count)
```

<u>Figure 8.16: Output</u>

```
1
2
3
```

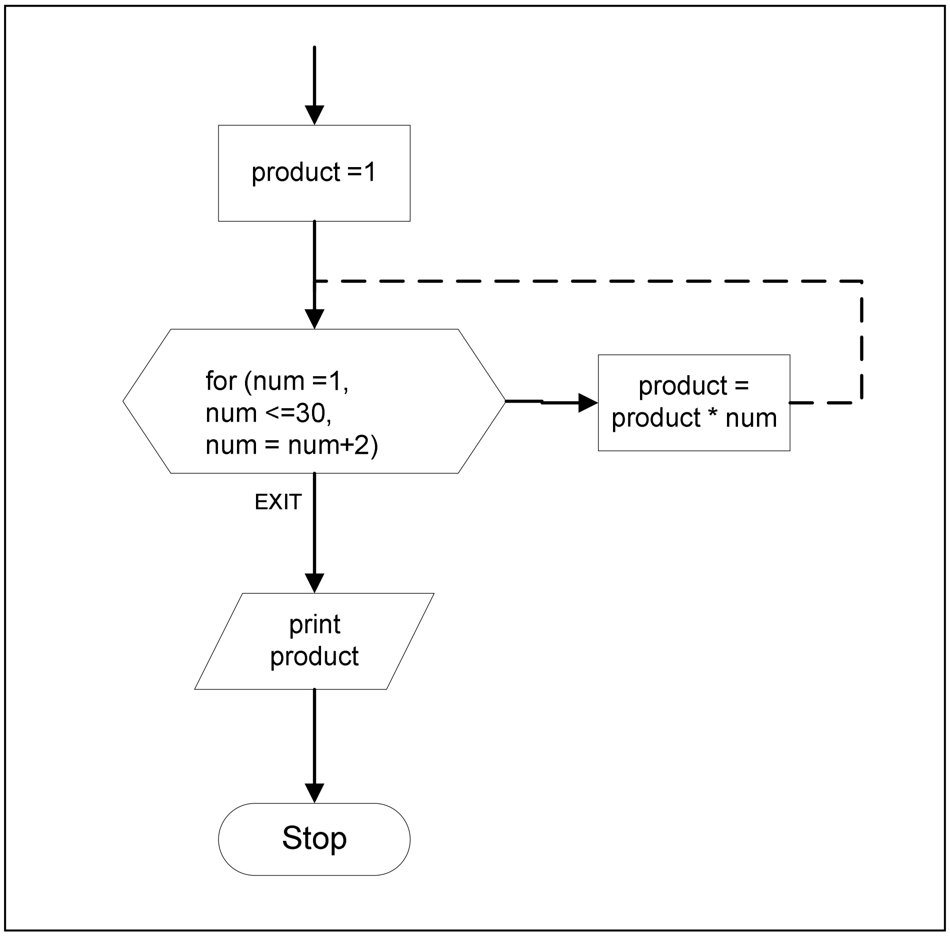
Self-check 8.2

To get some practice, convert the previous self-check problem (to calculate and print squares of numbers between 1 and 4 inclusive) into a *for* loop for the pseudocode, flowchart, and Python code showing the final output from the program to the screen.

Problem 8.4

Compute and print the product of odd numbers between 1 and 30.

Figure 8.17: Flowchart



Let's analyze the flowchart above. First we set the product to be equal to 1. Note that anything times 1 will give you back the same thing (i.e. 5*1 = 5), if product were equal to zero, then we would only be getting back zero (i.e. 5*0 = 0). The *for* loop will go through all the odd values from 1 (*num* = 1) up to 30 (*num* <= 30). Because we only want odd values, and we are starting with 1, then the next odd value would be 3 (or 1 + 2) and the subsequent odd value would be 5 (or 3 + 2). Therefore, we want *num* to increase by two each time we are in the loop (*num* = *num* + 2).

Figure 8.18: Pseudocode

```
Start
  product =1
  For (num = 1, num <= 30, num = num + 2)
    product = product * num
  EndFor
  print product
Stop
```

Figure 8.19: Python program

```
#This program prints the product of odd
#numbers between 1 and 30
product = 1
for num in range (1, 30, 2):
  product = product * num
print (product)
```

Figure 8.20: Output

```
6190283353629375
```

8.3 Sentinels in while loops

Values that are used to end loops are referred to as **sentinel** values. Consider the following example

```
get sales
While (sales > 0)
     bonus= sales * 0.1
     display bonus
     get sales
```

All values of sales that are 0 or less than 0 will stop this loop. All of these values are known as sentinel values. In problem 8.2, 0 will be the sentinel value as it will stop the loop.

Let's look at an example of a simple game below. In this game the user will have three options chosen by the user entering the values of 1 to 3. If the value of 1 or 2 is entered by the user, then we will state that they selected that particular option. Since the value of *more* will not change for the input of 1 or 2, the user will then be asked what their option is. When the user selects option 3, they will be told their option and then *more* is updated to False. Now the loop ends.

```
more = True
While more Do
  print "Please enter an option from 1 to 3: "
  get option
  If option == 1 Then
    print "You selected option 1."
  Else If option == 2 Then
    print "You selected option 2."
    Else
      print "You selected option 3. Good Bye."
      more = False
    EndIf
EndWhile
```

Note that we will stay in the loop as long as the condition, *more*, is True. When the user selects option 3, *more* is updated to False and subsequently, we leave the loop. Therefore, the value False is the sentinel value.

Self-check 8.3

1. Convert the above pseudocode into Python code.
2. Sentinel values do not always need to be Boolean; they could also be a Boolean expression. How could you replace the Boolean *more* with a Boolean expression?

Now that the code is in Python, try it out and think of the games that you have played where you get a menu of options to choose between. Now you know the code that makes the menu work.

8.4 Do-While loops

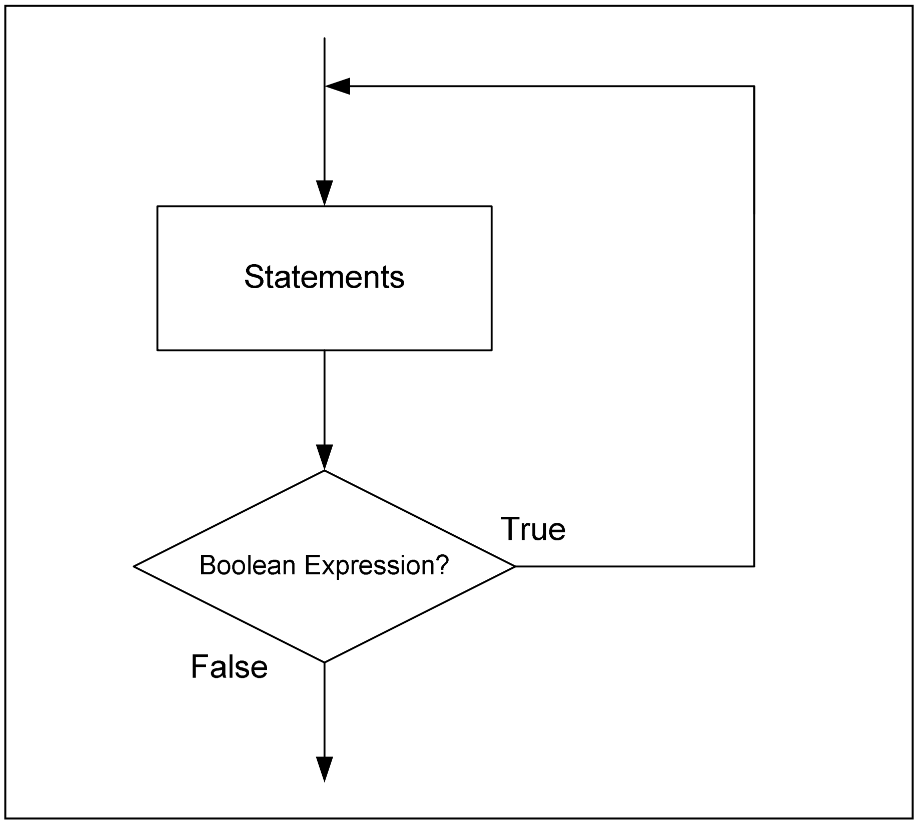
Sometimes we want to execute the body of a loop first and then check the condition later. To accomplish this task, the condition will need to be at the end of the loop rather than at the beginning. This structure change is called a **Do-While** loop. This means that the body of loop will execute at least one time.

```
Do
  statement1
```

```
      statement2
    While condition
```

The flowchart for this type of loop looks like:

Figure 8.21: Flowchart for general Do-While loop



The one place where a *While* loop and *For* loop are not as helpful is with checking for user input correctness. To accomplish this task with a *While* loop you must repeat code. For example, look at the problem below.

Problem 8.5

Create a pseudocode segment to have the user enter a positive value in both a *While* loop and a *Do-While* loop.

```
print "Please enter a positive value."
get value
While value < 0 Do
    print "You entered a negative value, try again"
    get value
EndWhile
```

Note that the repeated sections of code are the *print* and *get* for *value*. You must *print* a request for the user to enter a *value* before and inside of the loop. If we convert the code to a *do-while* loop, this repetition disappears.

```
Do
  print "Please enter a positive value."
  get value
While value < 0
```

We have gone from 6 lines of code in the *While* loop to 4 in the *Do-While*. Note that our code is also more concise with only one *print* and one *get* request to the user.

Let's convert the above code to Python and test for user correctness by running the program. Note that Python does not have a *Do-While* structure, but the same can be accomplished initializing *negative* outside of the *While* loop.

Python code:

```
negative = True
while negative:
  value = int(input("Please enter a positive value: "))
  if value >= 0:
    negative = False
```

Viewing some sample runs for this:

```
Please enter a positive value: -4
Please enter a positive value: -10
Please enter a positive value: 5
```

8.5 Infinite Loops

There are cases when your loop does not end, we call these **infinite** loops. Infinity is a value that has no bounds; it is very large. You can easily tell when you have an infinite loop in your program, because your program will not perform as expected.

An example of an infinite loop would be

```
While true
  print "hello"
EndWhile
```

Note that the above code segment will continually print "hello". One way that you can end an infinite loop is to type *Ctrl-X* or to type *Ctrl-C*. Sometimes for a Windows machine, you must end the program you are running by selecting *Ctrl-Alt-Delete* and ending the current task.

Key Terms

Condition	Initialization
Decrement	Loop
Do-While	Sentinel
For-loop	Trace table
Increment	While
Infinite Loop	

Exercises

For each of the problems below, show the pseudocode, flowchart, and Python code. Run the Python code to ensure your code runs as desired.

1. Print numbers from 5 to 1 in descending order. Use a *While* loop.

2. Print numbers from 5 to 1 in descending order. Use a *For* loop.

3. Read in a number and print its factorial. The factorial of a number N is:

```
1 * 2 * 3 *...* N
```

Sample Data :

num =5

Factorial of num is:

```
1 * 2 * 3 * 4 * 5 = 120
```

4. Print the maximum and the minimum of a set of positive numbers.

   Sample Data :

   50
   40
   60
   85
   -1 Sentinel Value

   Corresponding output:

   The maximum number is: 85
   The minimum number is: 40

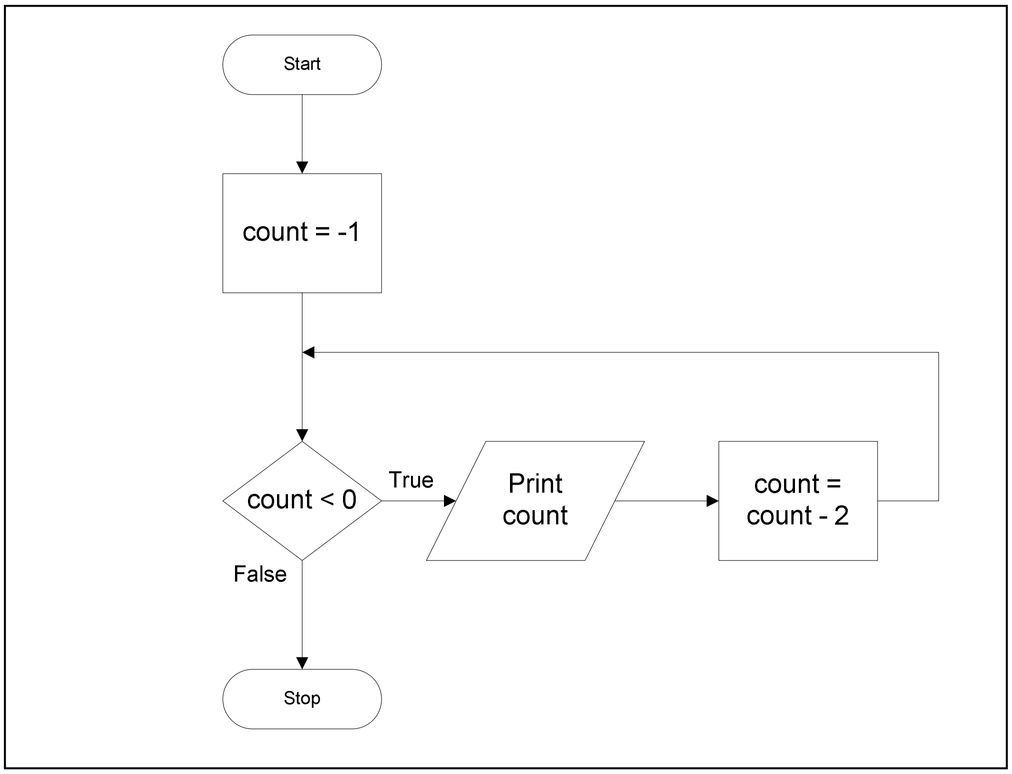
5. Read a set of numbers and print whether each number is even or odd.

   Sample Data :

   31
   40
   25
   85
   -1 Sentinel Value

   Corresponding output:
   31 is odd
   40 is even
   25 is odd
   85 is odd

6. Walk through the following flowchart and describe what happens using a trace table.



7. Calculate and print the sum of the first ten numbers (i.e., 1 to 10). Use a *While* loop.

8. Calculate and print the sum of first ten numbers. Use a *For* loop.

9. Calculate and print the following sum:

   1 + 1/2 + 1/3 + 1/4 + 1/5

   Show pseudocode for both a *For* loop and a *While* loop.

10. Given a positive integer, determine whether or not it is a “perfect number”.

Explanation:

A number is “perfect” if all its divisors (including the number 1, but not including the number itself) add up to the number. For example, 6 is perfect since the sum of its divisors 1, 2 and 3 is 6. 15 is not perfect since its divisors 1, 3 and 5 add up to 9 and not 15. 28 is perfect since its divisors 1, 2, 4, 7 and 14 add up to 28. 14 is not perfect since its divisors 1, 2 and 7 add up to 10 and not 14.

The technique for one possible solution can be explained in words as follows: (a) Read in the number n. (b) Try out the divisors 1, 2, 3,…., (n-1). If any of these divisors divide n, update sum by divisor.

Hint:

It is very important to start out with a 0 value for sum, so that the operation sum + divisor is done properly. If either sum or divisor did not have a known value, it is meaningless to add them together. Note that sum is set to 0 before testing out the divisors. In most programming languages, MOD is a built-in function that “returns” the remainder obtained after dividing the first integer value by the second. If the remainder returned is 0, then the second integer divides the first. Note that if divisor was not incremented within the loop, we would get an “infinite” loop, i.e. a loop that will never end. Flowcharts with infinite loops are completely meaningless.

Finally, a check is made to see if the number is perfect. This is done by using sum==n?, which corresponds precisely to the definition of a perfect number.

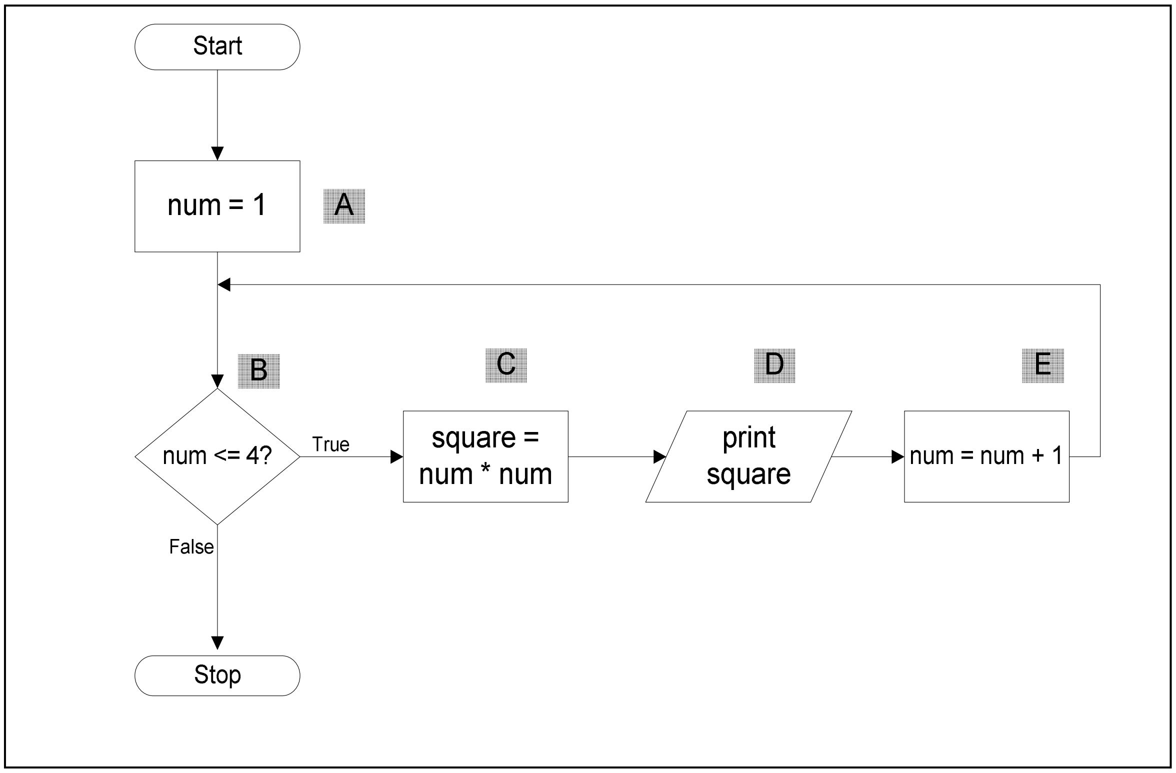
Self-Check Solutions

8.1

Pseudocode:

```
Start
  num = 1
  While num <= 4 Do
    square = num * num
    print square
    num = num + 1
  EndWhile
Stop
```

Flowchart:



Python code:

```
#This program prints squares of numbers from 1 to 4.
num = 1
while num <= 4:
  square = num * num
  print (square)
  num = num + 1
```

Corresponding Output:

```
1
4
9
16
```

Trace Table for Self-check problem:

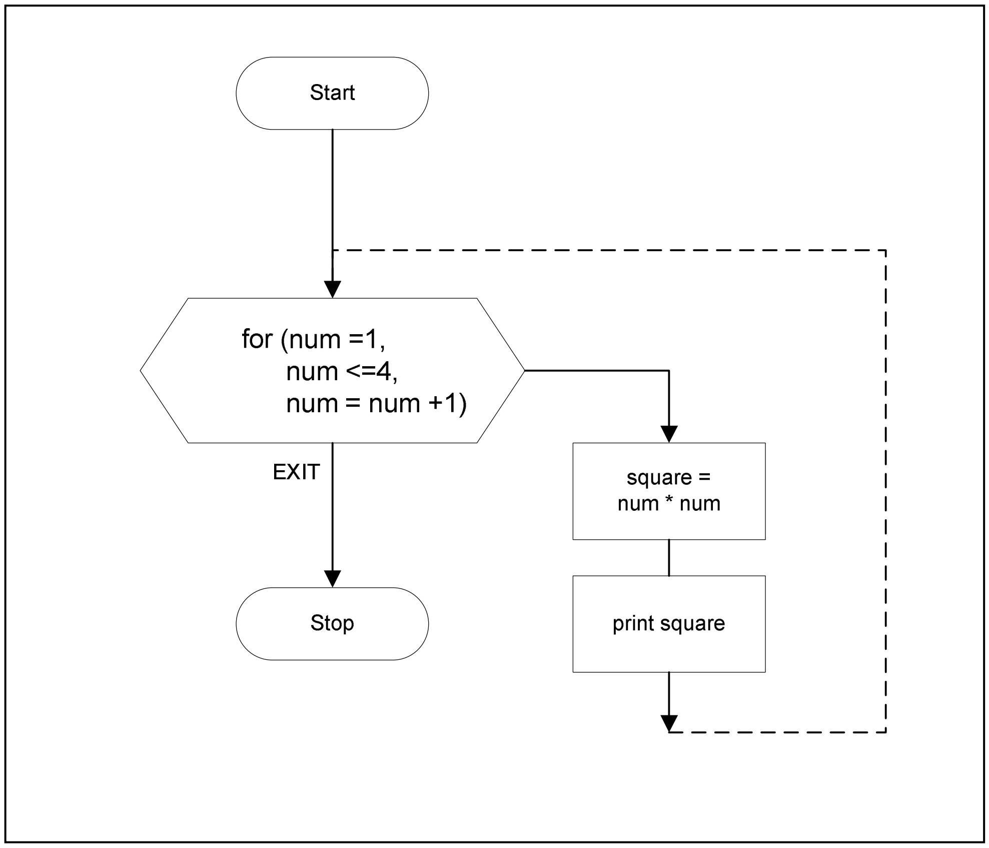
Iteration #	Box #	num	num<= 4?	square	Output
Start					
1	A	1			
	B		True		
	C			1	
	D				1
	E	2			
2	B		True		
	C			4	
	D				4
	E	3			
3	B		True		
	C			9	
	D				9
	E	4			
4	B		True		
	C			16	
	D				16
	E	5			
5	B		False		
Stop					

8.2

Pseudocode:

```
Start
  For (num = 1, num <= 4, num = num + 1)
    square = num * num
    print square
  EndFor
Stop
```

Flowchart:



Python code:

```
#This program prints the square of numbers from 1 to 4
for num in range(1,5):
  square = num * num
  print (square)
```

Output:

```
1
4
9
16
```

8.3

1.

```
more = True
while more:
  option = int(input("Please enter an option from 1 to 3:"))
  if option == 1:
    print ("You selected option 1.")
  elif option ==2:
    print ("You selected option 2.")
  else:
    print ("You selected option 3. Good Bye.")
    more = False
```

2. option <> 3

Note that this statement will be False only when the user has entered the value of 3.

Corresponding output:

```
Please enter an option from 1 to 3: 2
You selected option: 2.
Please enter an option from 1 to 3: 1
You selected option: 1.
Please enter an option from 1 to 3: 3
You selected option: 3. Good Bye.
```

9. Nested Loops

Single loops are great when you want to continue repeating one task as we saw in the previous chapter. But in life, we often have situations where our tasks are more complicated and we cannot complete them in such a straight forward manner. A ***nested loop*** is similar to a nested *If-Then-Else* where we have at least one loop inside another loop.

When you get up in the morning, you may start the day by brushing your teeth. Exploring this concept deeper, perhaps you brush up and down with your toothbrush on the right side of your mouth 15 times and 18 times on the left side. Now we have created a nested loop (similar to a nested *If* statement seen in the previous chapter) where you brush your teeth everyday and for every day you brush each side many times each day.

Let us look at some of the problems which use nested loops.

Problem 9.1

When we have one loop nested inside another, we can create two dimensional objects like a box of stars. We will let the user decide the size of the box and then create the box to those specifications using nested loops. For example, if the user chooses the height to be 3, then we want to print:

```
***
***
***
```

In looking at the problem, you need to see the pieces that are needed for the variables. A computer can only print one line at a time, starting at the top line and working its way down. So the first thing that we want to print is:

```
***
```

This will require a loop to print a * three times. From the last chapter, we can create the single loop,

```
    column = 0
    While column < 3 Do
        print '*'
        column = column + 1
    EndWhile
```

Now we need to print that row two more times. Therefore the above code becomes our **inner loop**, the **outer loop** will print it a total of three times. Each time we print one row of three *'s, we need to end the line with a carriage return. The symbol for a carriage return is '\n' (refer to Chapter 2 ASCII code). Note that we include comments in the pseudocode to aide in understanding by using '#' symbol and highlighting the code in bold.

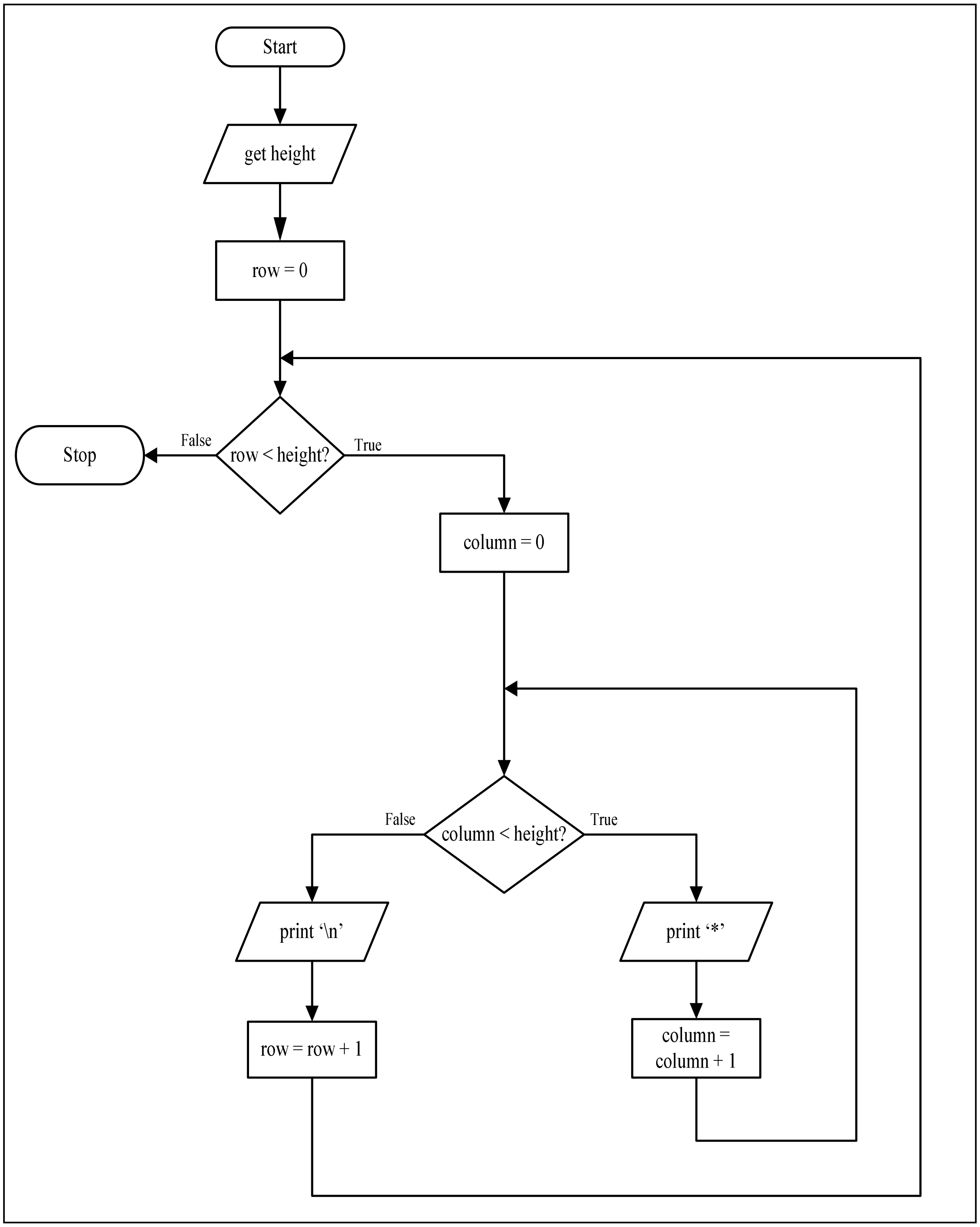
```
    row = 0
    While row < 3 Do
      column = 0
      While column < 3 Do
        print '*'
        column = column + 1
      EndWhile
      #wrapping up the outer loop by using a carriage return
      print '\n'
      row = row + 1
    EndWhile
```

Now we only need to get the user to enter the height of the square we are creating and use that variable to terminate the loop.

Figure 9.1: Pseudocode

```
Start
  print "Please enter a number"
  get height
  row = 0
  While row < height Do
    column = 0
    While column < height Do
      print '*'
      column = column + 1
    EndWhile
    #wrapping up the outer loop by using a carriage return
    print '\n'
    row = row + 1
  EndWhile
Stop
```

Figure 9.2: Flowchart



In Python, you can print on separate lines by using the *print* statement as shown below:

```
print(“a”)
print(“b”)
```

will result in the output on two separate lines.

```
a
b
```

You can print to one line if you include a comma after the first argument.

```
print ("a",  "b")
```

or you can use a comma at the end of the first print statement.

```
print(“a”),
print(“b”)
```

Either approaches will result in the same output

```
ab
```

To print the end of a line by itself, you can use the command

```
print ()
```

Now let’s look at the Python code to print the solution to our initial problem.

Figure 9.3: Python program

```
height = int(input("Please enter height: "))
row = 0
while row < height:
  column = 0
  while column < height:
    print ('*', end='')
    column = column + 1
  print()
  row = row + 1
```

Figure 9.4: Output

```
Please enter height: 3
***
***
***
```

Self-Check 9.1

Use a nested loop to create a triangle of stars. We will let the user decide the height of the triangle and then create a triangle to those specifications.

If the user chose the triangle height to be 4, then our printed triangle would be:

```
*
**
***
****
```

Create the pseudocode, flowchart and Python code to gain practice with nested loops.

Problem 9.2

Nested loops can have more than one loop nested inside of them. Let’s modify the triangle created in the self-check to align to the right instead of the left, so if the height was again 4, our triangle would look like:

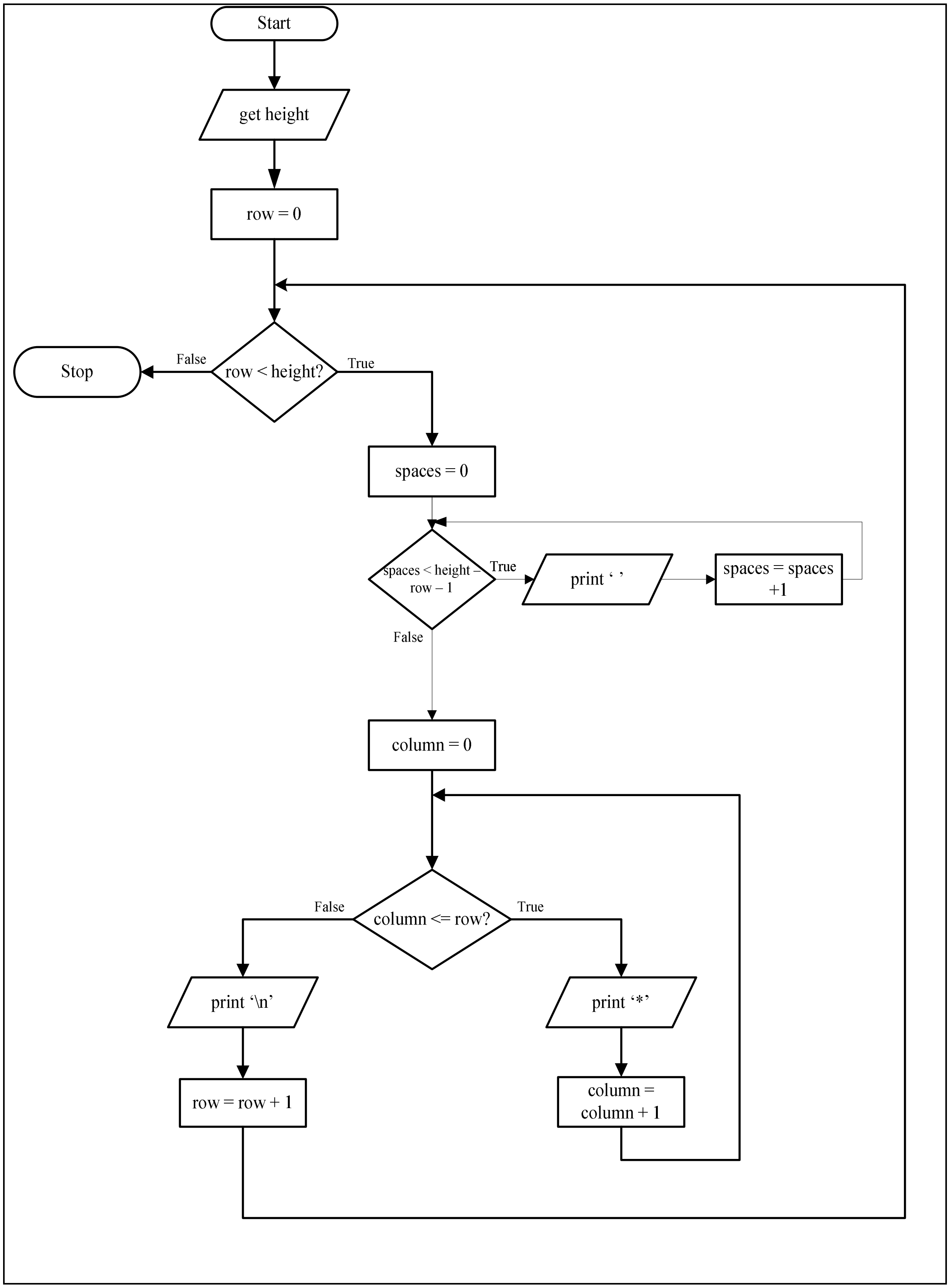
```
   *
  **
 ***
****
```

The difference is that, for each row, we want to print out spaces before we print the stars for that row. Note that the number of spaces is different at each line; this decreasing amount of stars can be accomplished with a loop to print the spaces before we have the loop to print the stars.

Figure 9.5: Pseudocode

```
Start
  print “Please enter the height.”
  get height
  row = 0
  While row < height Do
    # Create spaces before the stars on each row
    spaces = 0
    While spaces < height – row – 1 Do
      print ‘ ’
      spaces = spaces + 1
    EndWhile
    #Create the correct number of stars per row
    column = 0
    While column <= row Do
      print ‘*’
      column = column + 1
    EndWhile
    # Ends the current row and starts a new row
    print ‘\n’
    row = row + 1
  EndWhile
Stop
```

Figure 9.6: Flowchart



<u>Figure 9.7: Python program</u>

```
height = int(input(“Please enter height: “))
row = 0
while row < height:
  spaces = 0
  while spaces < (height – row – 1)
    print(‘ ’, end=’’)
    spaces = spaces + 1
  column = 0
  while column <= row:
    print(‘*’, end=’’)
    column = column + 1
  print ()
  row = row + 1
```

<u>Figure 9.8: Output</u>

```
Please enter height: 4
   *
  **
 ***
****
```

Self-Check 9.2

A small modification to the right aligned triangle code can create a pyramid. If the user wants a height of 4, then the resulting pyramid would be

```
   *
  ***
 *****
*******
```

Note that the only difference between the previous triangle and the above pyramid is the number of stars that are printed on each line. Write the pseudocode to create a triangle with user specified height.

By now you have gained a good amount of experience manipulating nested loops. Let’s try two more before we leave these *’s and spaces.

Problem 9.3

Now that you have a pyramid, let's create a diamond which is just two pyramids sandwiched together. If the user wants a height of 4, then the resulting diamond would be:

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

Note that the only difference between the previous pyramid and the above diamond is that an upside down pyramid has been attached to the bottom. Therefore, all of the code that we have created before will be used as is, we will just need to add additional code to create the reversed pyramid. Namely we need to count the rows down to zero rather than up to *height*. The modifications from the previous problem are highlighted in gray.

Figure 9.9: Pseudocode

```
Start
  print “Please enter height”
  get height
  # Create the upper half of the diamond
  row = 0
  While row < height Do
    # Create spaces before the stars on each row
    spaces = 0
    While spaces < height – row – 1 Do
      print ‘ ’
      spaces = spaces + 1
    EndWhile
    # Create the correct number of stars per row
    column = 0
    While column < row*2 + 1 Do
      print ‘*’
      column = column + 1
    EndWhile
    # Ends the current row and starts a new row
    print ‘\n’
    row = row + 1
  EndWhile
  # Create the lower half of the pyramid
  row = height - 1
  While row > 0 Do
    # Create spaces before the stars on each row
    spaces = 0
    While spaces < height – row  Do
      print ‘ ’
      spaces = spaces + 1
    EndWhile
    # Create the correct number of stars per row
    column = 0
    While column < row*2 + 1 Do
      print ‘*’
      column = column + 1
    EndWhile
    # Ends the current row and starts a new row
    print ‘\n’
    row = row – 1
  EndWhile
Stop
```

Self-Check 9.3

Using everything that you have learned in this chapter, you have created substantial code segments to accomplish a wide variety of tasks. Now you should be able to create a nested loop to print a hollow diamond. If the user wants a height of 4, then the resulting diamond would be:

```
   *
  * *
 *   *
*     *
 *   *
  * *
   *
```

Write the pseudocode to create a diamond as stated above with a user specified height.

Note that this task is not as complicated as the filled diamond due to the smaller number of stars that must be printed.

Key Terms

Nested loop
Inner loop
Outer loop

Exercises

1. Create a multiplication table that will have 12 rows and 12 columns. The output should appear as below:

```
 1    2    3    4    5    6    7    8    9   10   11   12
 2    4    6    8   10   12   14   16   18   20   22   24
 3    6    9   12   15   18   21   24   27   30   33   36
 4    8   12   16   20   24   28   32   36   40   44   48
 5   10   15   20   25   30   35   40   45   50   55   60
 6   12   18   24   30   36   42   48   54   60   66   72
 7   14   21   28   35   42   49   56   63   70   77   84
 8   16   24   32   40   48   56   64   72   80   88   96
 9   18   27   36   45   54   63   72   81   90   99  108
10   20   30   40   50   60   70   80   90  100  110  120
11   22   33   44   55   66   77   88   99  110  121  132
12   24   36   48   60   72   84   96  108  120  132  144
```

2. You want to invest your money in a local bank. They are currently offering several plans, one with simple interest and the other with compound interest. The rates offered at the bank are 5%, 10% and 15%. Your task is to vary the principle invested from $10,000 to $15,000 with an increment of $1,000 for each rate.

   a) Create a flowchart to accomplish the task for problem2 using nested loops.
   b) Create a Python program to accomplish the task for problem 2 using nested loops.

   Hint: The pseudocode is:

```
N=5            #where N is number of years
For (rate = 0.05, rate <= 0.15, rate = rate + 0.05)
  For (principal=10000, principal <=15000, principal=principal+1000)
    simple = principal * (1 + rate * N) #where N is number of years
    compound = principal * (1 + rate) ^ N
    print simple + " " + compound
   EndFor
 EndFor
```

3. How many times will "Hello World" be printed for the pseudocode below?

```
For (i= 1, i<=5, i = i + 1)
  For (j= 1, j <=3, j = j + 1)
    print"Hello World"
  EndFor
EndFor
```

4. What will be the conditions for the inner and the outer loop for the following pattern:

```
* * * * * * *
  * * * * *
    * * *
      *
```

5. What will be the conditions for the inner and the outer loop for the following pattern:

```
* * * * * * *
* * * * *
* * *
*
```

6. What will be the conditions for the inner and the outer loop for the following pattern:

```
* *
* * * *
* * * * * *
* * * * * * * *
```

7. What will be the conditions for the inner and the outer loop for the following pattern:

```
* * * * * * * *
* * * * * *
* * * *
* *
```

8. What will be the output of the following pseudocode:

```
For (i= 1, i<=5, i = i + 1)
  For (j= 1, j <=4, j = j + 1)
    print i + j
  EndFor
EndFor
```

Self-Check Solutions

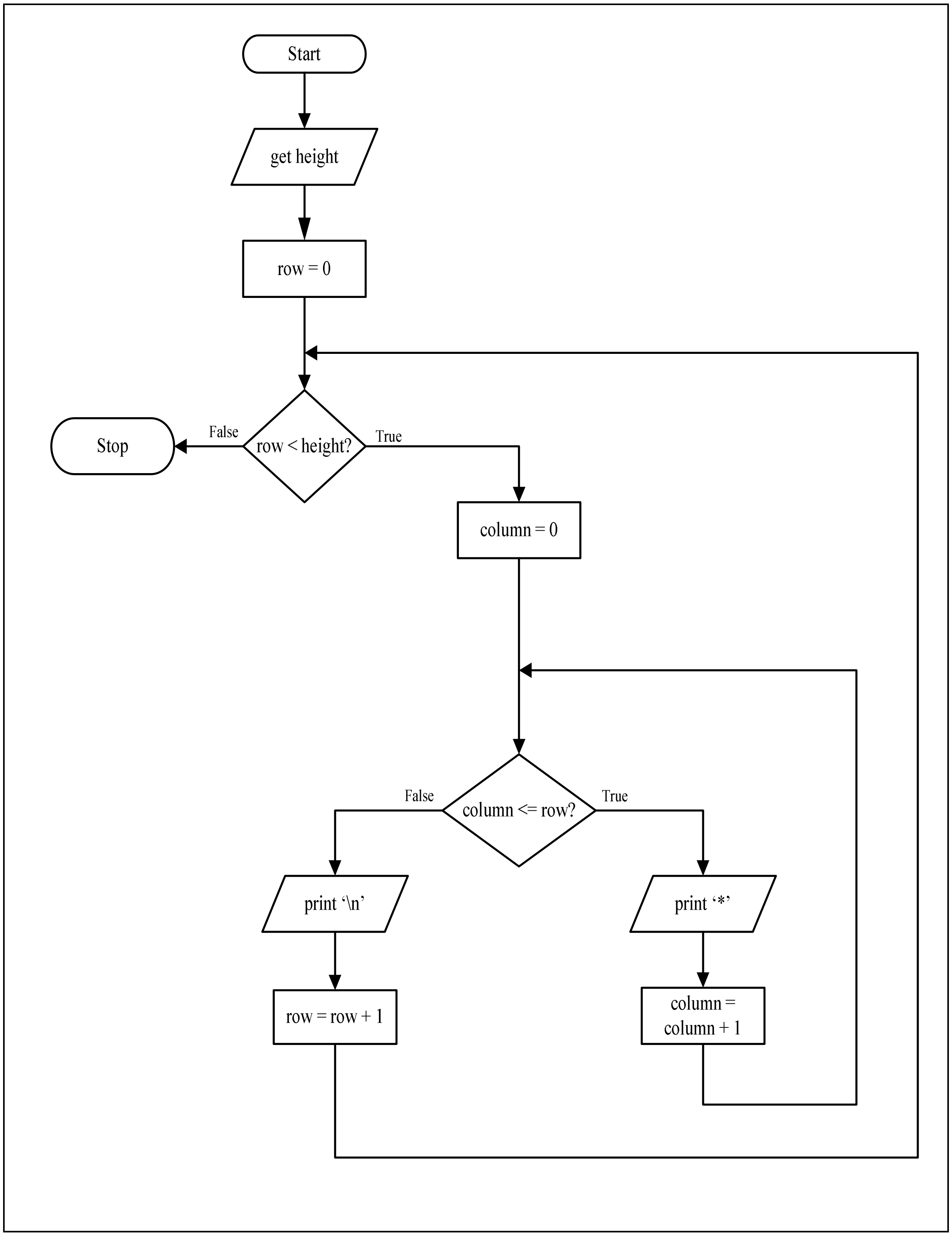
9.1

Note that the outer loop will be the same as when we created a box of stars, but the inner loop will be different because we don’t want so many stars to print. There is really only one small change that will need to be made, modifying the second condition for the loop.

Pseudocode:

```
Start
  Get height
  row = 0
  While row < height Do
    column = 0
    While column <= row Do
      print ‘*’
      column = column + 1
    EndWhile
    print ‘\n’
    row = row + 1
  EndWhile
Stop
```

Flowchart:


Start
get height
row = 0
row < height?
False
True
Stop
column = 0
column <= row?
False
True
print '\n'
print '*'
row = row + 1
column = column + 1

Python code:

```
height = int(input("Please enter height: "))
row = 0
while row < height:
  column = 0
  while column < row:
    print('*', end='')
    column = column + 1
  print ()
  row = row + 1
```

Corresponding output:

```
Please enter height: 4
*
**
***
****
```

9.2

Only the second inner loop needs to be modified to print more stars. Let's look at a small chart to see the number of stars needed per row:

row	*stars*
1	1
2	3
3	5
4	7

Perhaps you already see the pattern, but if not, look at an added column for *row*2* below

row	*row*2*	*stars*
1	2	1
2	4	3
3	6	5
4	8	7

Now you can see that the only difference between *row*2* and stars is a 1. Therefore, we can create our solution.

Pseudocode:

```
Start
  print "Please enter height"
  get  height
  row = 0
  While row < height Do
    # Create spaces before the stars on each row
    spaces = 0
    While spaces < height - row - 1 Do
      print ' '
      spaces = spaces + 1
    EndWhile
    # Create the correct number of stars per row
    column = 0
    While column < row*2 + 1 Do
      print '*'
      column = column + 1
    EndWhile
    # Ends the current row and starts a new row
    print '\n'
    row = row + 1
  EndWhile
Stop
```

9.3

Pseudocode:

```
Start
  print “Please enter height”
  get height
  row = 0
  While row < height Do
    spaces = 0
    While spaces < height – row – 1 Do
      print ‘ ’
      spaces = spaces + 1
    EndWhile
    print ‘*’
    row = row + 1
  EndWhile
  # Now make the bottom half
  row = height –1
  While row > 0 Do
    spaces = 0
    While spaces < height – row  Do
      print ‘ ’
      spaces = spaces + 1
    EndWhile
    print ‘*\n’
    row = row – 1
  EndWhile
Stop
```

10. Functions

So far we have been creating and using **scripts** to perform tasks. These scripts are independent pieces of code that require no prior knowledge to execute. While scripts are flexible, they require the data values to be set or to request the values from the user before they can work. When you want to perform a task multiple times, possibly changing the data values each time, then a **function** is a better choice than a script. In working with a group of programmers to complete a task, functions are vital so that problems can be broken down into smaller tasks that each group can handle individually and then come together with the completed project.

10.1 Function parts

A function has several parts. Figure 10.1 shows the syntax of a value returning function.

Figure 10.1: Syntax of a value-returning function

```
Function returnValue functionName(parameter1,parameter2, …)
     Statement1
     Statement2
     return Value
EndFunction

# Main program will call the function
variable = functionName(parameter1, parameter2, …)
```

The first line of the function is known as the **header.** In Figure 10.1, the header is

```
Function returnValue functionName(parameter1,parameter2, …)
```

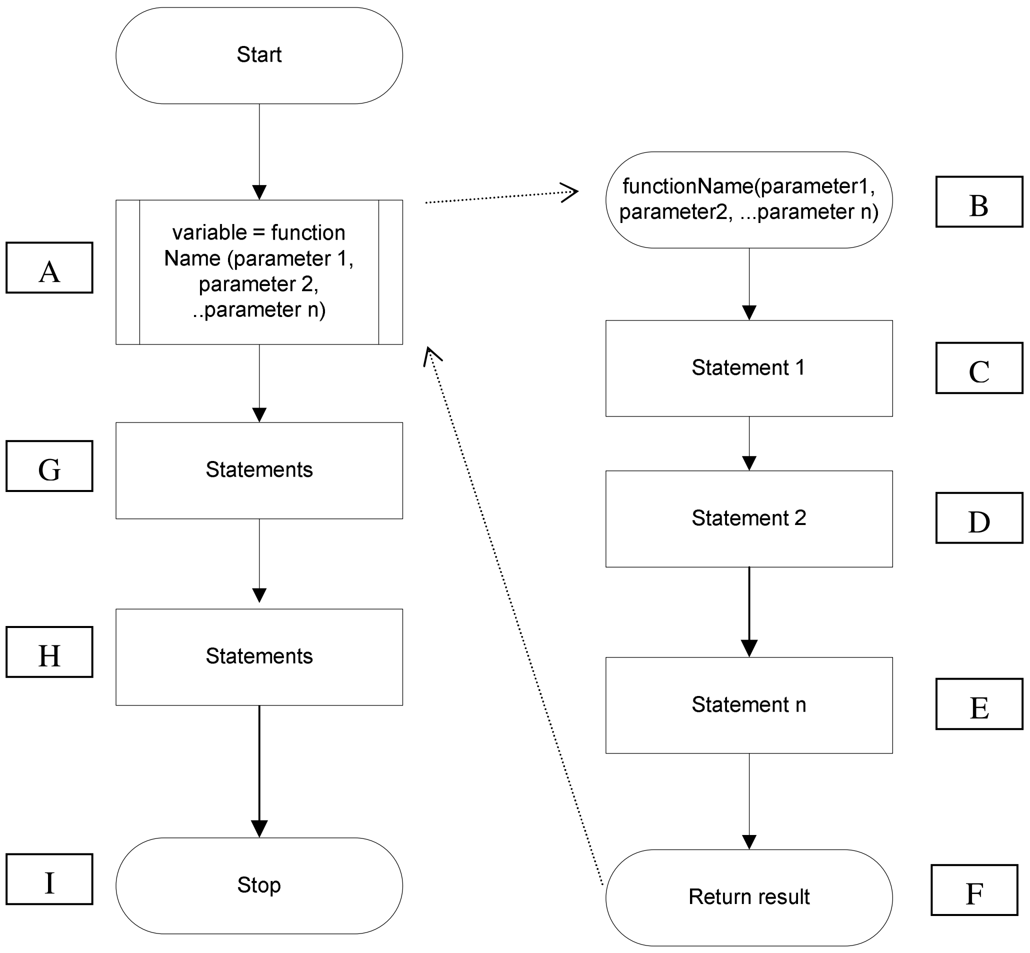
The next three lines comprise the **body** of the function. In Figure 10.1, the body is

```
     Statement1
     Statement2
     return Value
```

In the header, *returnValue* is the data type of the value returned by the function. Note that the function will not be executed unless it is called by its name. In Figure 10.1, *functionName* is the name of the function. Therefore, the main program, or another function, or script, must call the function with the appropriate parameters to use the code created. The function's parameters must match the data type and number of parameters passed with the function call.

Referring to Figure 10.2, when a statement in Box A is encountered, the execution branches to Box B. Statements in Box C-F are executed and the execution branches back to the main program in Box A. The *result* of Box F is assigned to the *variable* in Box A of the main program. Then execution continues with Box G –I.

Figure 10.2: Flowchart of a value-returning function


Start
A
variable = function Name (parameter 1, parameter 2, ..parameter n)
G
Statements
H
Statements
I
Stop
functionName(parameter1, parameter2, ...parameter n)
B
Statement 1
C
Statement 2
D
Statement n
E
Return result
F

Within the parentheses above, there is a description of *parameter1*, *parameter2*, etc. These are **parameters**, variables that **must** be supplied to execute the function. There are two types of parameters that may be passed into a function: **call-by-reference** and **call-by-value**. A call-by-value parameter passes the value to be used by the function while a call-by-reference parameter's value may be updated and used in the program that called the function. The way to distinguish between the two different types of parameters is different based on which language you are using. For our pseudocode, we will use a star before the parameter's name to distinguish call-by-reference; i.e. **parameter2*.

Note that there do not have to be any parameters, in which case you would only have *functionName()* or there could be a large number of parameters in which case you would separate them with commas. There can also be more or fewer statements inside the function to be executed.

In most cases, you want to have each function have a unique name. But a function can have the same name as another function as long as the parameters passed are different.

For example, if I have a function *summation()* and another function *summation(x,y)* then the computer knows which function I want to use when in my code I state, *summation(3,16)*. In this case I want to use the second summation function that was created because the number of parameters match.

If I had two functions called *summation()*, then the computer would not know which function I want to use when I call the function *summation()* in my program. Hence it is not a good idea to have the same name for two different functions.

10.2 Comments in functions

When you create a function, it is important to document information about your function with comments. Remember that comments allow any user to understand the purpose of the function and how to use it appropriately. In pseudocode and Python, the pound sign, '#' will start a comment from the '#' to the end of the line.

Below is the minimal information that you should include *before* any function that you create.

```
# Purpose: state the purpose of the function
# Parameter1: expected type of this parameter
# Parameter2: expected type of this parameter
# Return type: state the return type
```

10.3 Value-returning functions

You have already used functions when you gather input from the user for Python.

a = int(input(String))
b = float(input(String))

In this case, *input* is a **built-in** function that has one parameter, a string, and returns a string to store in *a*. A built-in function is a function that is supplied by a library of functions in the language (here Python) that we are using. When you create a function yourself, that is called a **user-defined** function. Note, *int(input(String))* is a combination of two functions, where the *int()* function converts the value read-in by *input(String)* to an integer

Note that in order to return a value, you must use the key word *return* in your function followed by the variable or value that you want the function to return. Let us see some problems to understand value-returning functions.

Problem 10.1

Write a function *calcSquare* which receives a number and returns its square.

Figure 10.3: Pseudocode

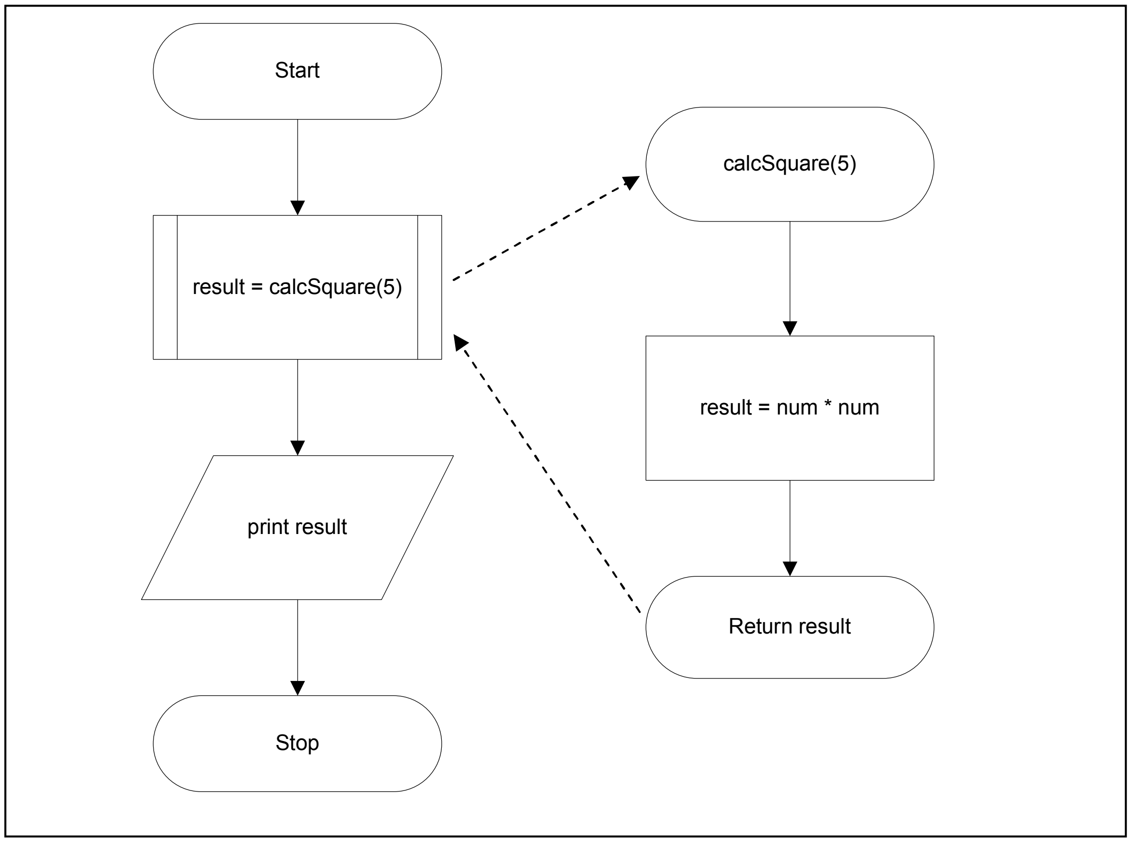
```
# Purpose: To square a value
# Parameter1: value to be squared
# Return type: numerical
Function numerical calcSquare(number)
  return number * number
EndFunction
```

To call this function, we could use the following code.

```
num = calcSquare(5)
```

After the above line of code, *num* would contain 25.

Figure 10.4: Flowchart



For Python, a function is defined as *def* and there is a ":" after the function definition. The statement inside of a function is indented to the right. In the code below, the function *calcSquare(num)* is called on the fourth line (including the blank line). Also, note that Python begins execution at line 4 after skipping over the function definition.

Figure 10.5: Python Program

```
def calcSquare(num):
  return num * num

result = calcSquare(5)
print (result)
```

Note that *result* will have the value of 25 after the line:

```
result = calcSquare(5)
```

In order to see the value of *result*, we need to print the *variable*. Hence the last line of code is:

```
print(result)
```

Corresponding output:

```
25
```

Problem 10.2

Write a function *calcAverage* which receives three numbers and returns their average.

Figure 10.6: Pseudocode

```
# Purpose: To return the average of three numbers
# Parameter1, Parameter2, Parameter3: numbers to be averaged
# Return type: numerical
Function numerical calcAverage(num1, num2, num3)
  return (num1 + num2 + num3) / 3.0
EndFunction

# call the function with the values of 2, 3, and 4
result = calcAverage(2,3,4)
print result
```

Figure 10.7: Flowchart

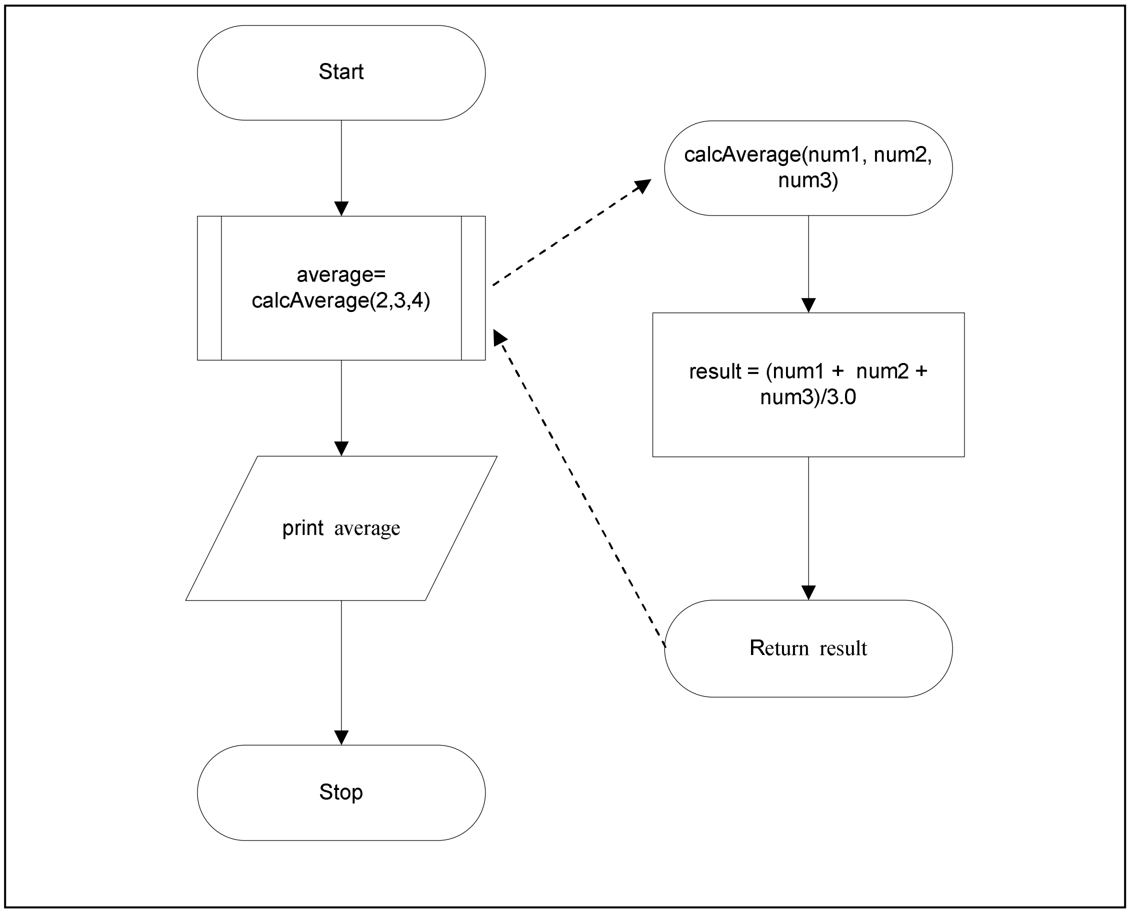

Start
average= calcAverage(2,3,4)
print average
Stop
calcAverage(num1, num2, num3)
result = (num1 + num2 + num3)/3.0
Return result

Figure 10.8: Python program

```
def calcAverage(x, y, z):
  return (x+y+z)/3.0

result = calcAverage(2, 3, 4)
print (result)
```

Figure 10.9: Output

```
3.0
```

Self-Check 10.1

Write pseudocode for a function called minimum that will return the minimum of two values given as parameters. Also draw its flowchart and write the corresponding Python code.

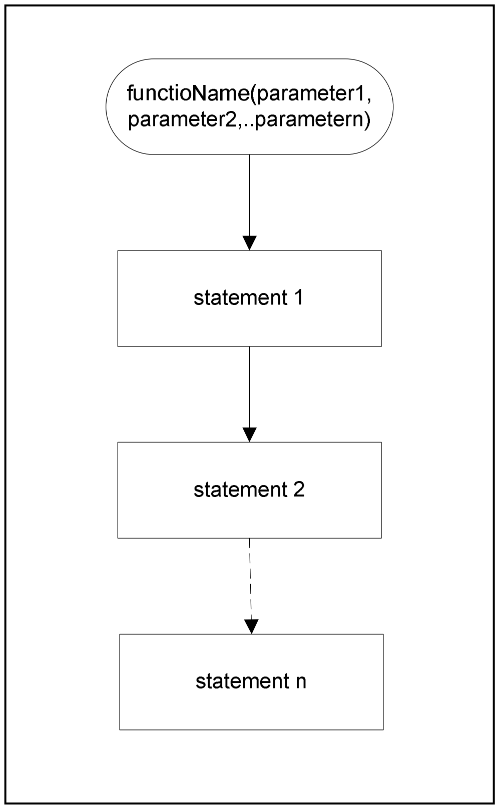
10.4 Void functions

When a function does not return a value, then we call it a **void** function. Some functions that don't return a value have tasks to print out information to the screen. The only changes from the *pseudocode* that we saw before, is that the comment for return type is now void, the *returnType* is replaced with *void* in the header.

Figure 10.10: Syntax of a void function

```
# Purpose: state the purpose of the function
# Parameter1: expected type of this parameter
# Parameter2: expected type of this parameter
# Return type: void
Function void functionName(parameter1, parameter2, …)
  statement1
  statement2
EndFunction
```

Figure 10.11: Flowchart of a void function



Problem 10.3

Write a function *calcFahrenheit* which receives a Celsius temperature as input and converts and prints its equivalent Fahrenheit temperature.

Figure 10.12: Pseudocode

```
# Purpose: To display a Fahrenheit temperature
# Parameter1: Celsius
# Return type: void
Function void calcFahrenheit(celsius)
  print (9/5.0)*celsius + 32
endFunction

# call the function with the Celsius value of 40
calcFahrenheit(40)
```

Figure 10.13: Flowchart

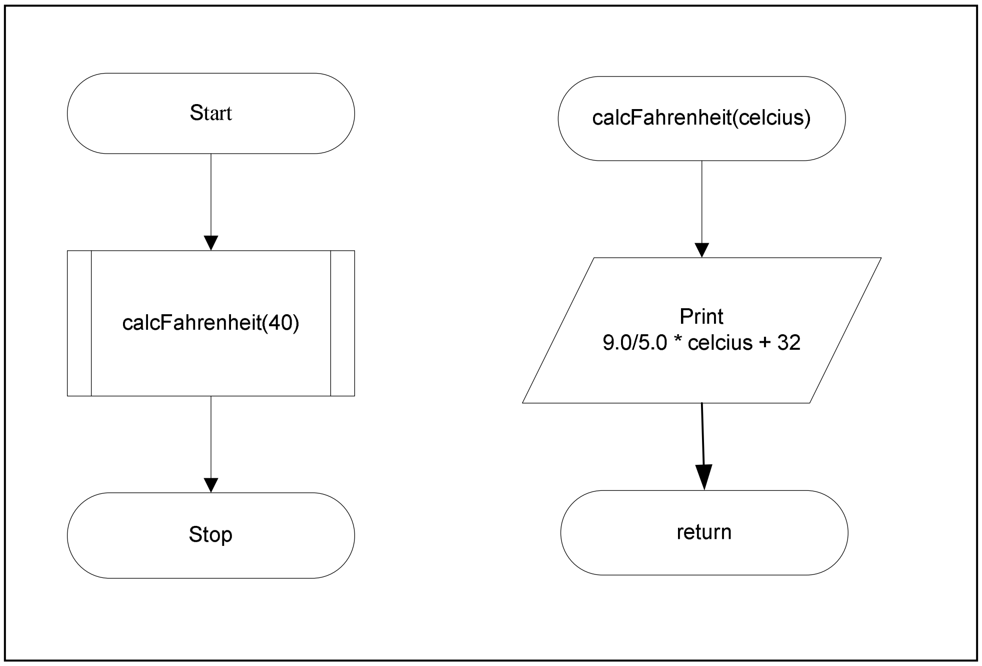


Figure 10.14: Python program

```
def calcFahrenheit(celsius):
  print (9.0/5.0 * celsius + 32)

calcFahrenheit(40)
```

Figure 10.15: Output

```
104.0
```

Problem 10.4

Write a void function printInfo to print the name, date and project number.

Figure 10.16: Pseudocode

```
# Purpose: To print out basic header information
# Parameter1: String for the name of the user
# Parameter2: String for the date of the project
# Parameter3: String for the project name
# Return type: void
Function void printInfo(name, date, project)
  print “Name: ” + name
  print “Date: ” + date
  print “Project: ” + project
EndFunction

printInfo("John Doe", "6/10", "19")
```

Figure 10.17: Flowchart

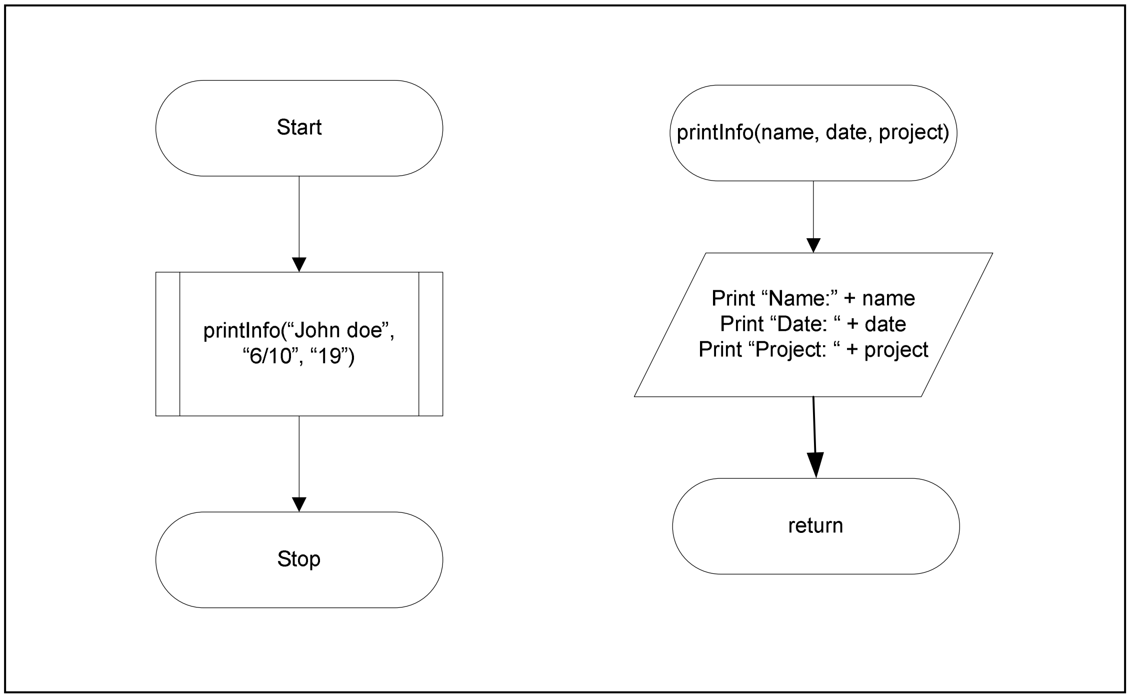

Start
printInfo(“John doe”, “6/10”, “19”)
Stop
printInfo(name, date, project)
Print “Name:” + name
Print “Date: “ + date
Print “Project: “ + project
return

Figure 10.18: Python program

```
def printInfo(name, date, project):
  print ("name:" + name)
  print ("date:" + date)
  print ("project:" + project)

printInfo("John Doe", "6/10", "19")
```

Note that void Python functions do not need a *return*.

Figure 10.19: Output

```
Name:John Doe
Date:6/10
Project:19
```

Self-Check 10.2

Create a void function called *triangle* that will print out a left aligned triangle of stars based on the height specified in the numerical parameter.

Key Terms

Script
Function
Parameter
Call-by-reference
Call-by-value
Built-in function
User-defined function
Void function

Exercises

For each problem, create the pseudocode, flowchart and Python code.

1. Create a function called *minimum* that will return the minimum of three numerical parameters.

2. Create a *void* function that will create a pyramid of stars whose height is based on the parameter.

3. Create a conversion function named *ASCII* that will take an integer value and return the associated ASCII character. Note that there are only 168 ASCII characters, so your program will need to return “There is no ASCII character associated with that value.” if the user inputs a value below or above the possible ASCII decimal values.

4. Create a *void* function that will output the ASCII decimal and associated character values.

5. Create a *power* function that will take the base and exponent and return the base to power of the exponent.

6. Create a *summation* function that will take three numbers as parameters and return the sum of the numbers.

7. Create a function to play the game hi/low. In this function there are no parameters and no return value. The program flow is:

   a) Create a random integer from 1 to 100. Note that Python already has a random number generator. To access it, you need to *import math* at the start of your function.

   b) Ask the user to guess the number.

   c) If the number is too high, state "You guessed too high."
   If the number is too low, state "You guessed too low."
   If the number is just right, state "Congratulations, you guessed my number."

   You should repeat b) and c) until the user guesses the number correctly.

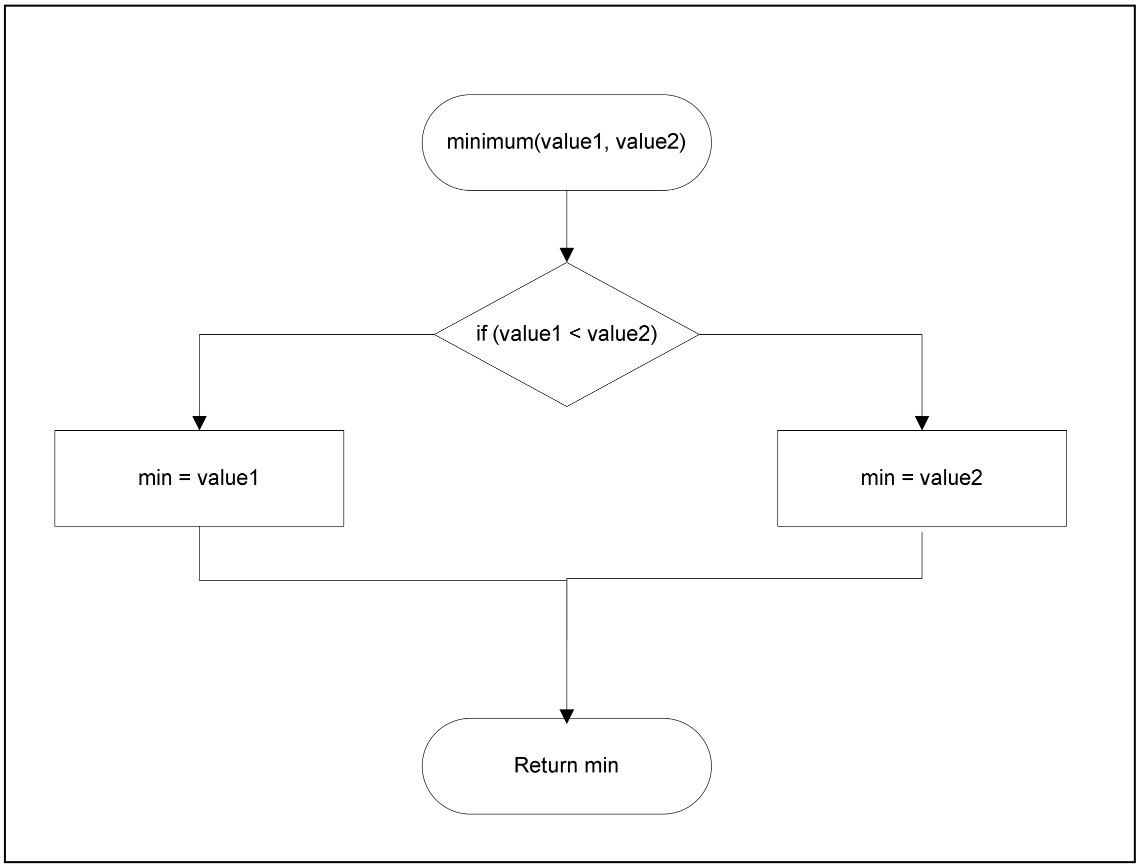
Self-Check Solutions

10.1

Pseudocode:

```
# Purpose: this function will return the smallest of
# the two parameters passed
# Paremeter1: numerical
# Paremter2: numerical
# Return value: numerical
Function numerical minimum(value1, value2)
  If value1 < value2 Then
    min =  value1
  Else
    min =  value2
  EndIf
  return min
EndFunction
```

Flowchart:



Python:

```
#Function min receives two parameters and
# returns the minimum of the two values

def min(value1, value2):
  if value1 < value2:
    min = value1
  else:
    min = value2
  return min

#Main program
#The line below calls the function min
minimum = min(5,15)
print (minimum)
```

```
5
```

10.2

Pseudocode:

```
# Purpose: To create a left-aligned
# triangle of stars
# Parameter1: height of the triangle
# Return type: void
Function void triangle(height)
row = 0
While row < height Do
  column = 0
  While column <= row Do
    print '*'
    column = column + 1
  Endwhile
  print '\n'
  row = row + 1
  Endwhile
return
EndFunction
```

11. Arrays

We have come a long way in the text so far. Among many tasks, two tasks that you learned were to create variables and to perform repetitious tasks with loops. As of now, each variable you create can hold one value. When each variable holds a different data type, this is necessary, but when each variable holds the same data type, there is a better way to store and update your values - an **array**. An array is a collection of variables with the same data type.

In pseudocode, we define an array as

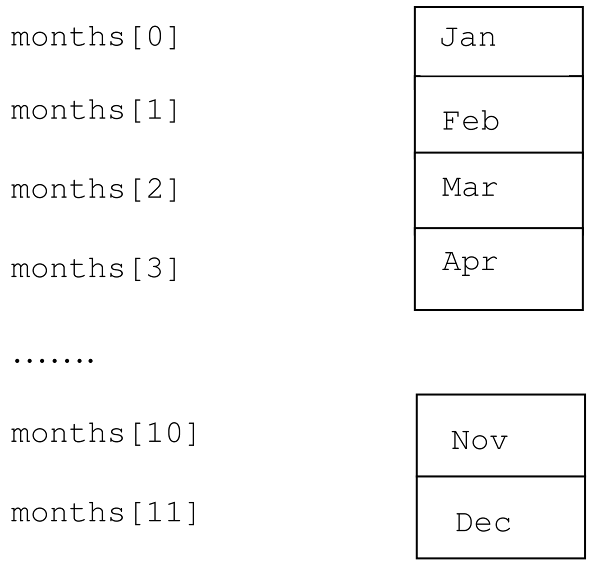
```
    arrayName [numberOfElements] = {values}
```

In the above case, *arrayName* is the name of the array. The brackets enclose an integer for the total number of **elements**, *numberOfElements*, in the array. The curly braces state the initial value of the elements of the array. Note that there are two ways to assign values to an array, you can either use commas to separate the values or you can state one value for all the elements of the array.

For example, let's create an array of months:

```
    months[12]=  {"Jan",  "Feb",  "Mar",  "Apr",  "May",
    "June", "July", "Aug", "Sep", "Oct", "Nov", "Dec"}
```

You can visualize the array in a tabular form



To access the first month of the year, we use:

```
months[0]
```

We refer to the value 0 as the **index** or **subscript** specifying the location into the array and it is read "month sub zero". In most programming languages, the index starts with 0.

Self-Check 11.1

How can you access the last month of the year in the previous example?

To **initialize** all elements of an array to the same value, we can just include one value in the curly braces. For example, let's create an array to keep track of the final grades in a course of three students.

```
finalGrades[3] = {0}
```

0	0	0
finalGrades[0]	*finalGrades [1]*	*finalGrades [2]*

This is called initializing your array, so that every element in the array starts with the same value in the case above, 0.

Let's create a **parallel array** to match the student names with their final grades. For example, Jennifer's grade is stored in *finalGrades[0]*, Juan's in *finalGrades*[1], and Kerry's in *finalGrades*[2].

```
studentNames[3] = {"Jennifer", "Juan", "Kerry"}
```

Jennifer	Juan	Kerry
studentNames[0]	*studentNames[1]*	*studentNames[2]*

A **one-dimensional array** can be thought of as a table of values.

Self-Check 11.2

If we know that Juan got an 87 on his exam, show the pseudocode that assigns his grade to the appropriate value in the array of finalGrades.

In the prior examples, you see that arrays usually have an underlying theme that connects all of the array variables. Let's say that we want to create a variable to keep track of class sizes for all classes in an entire school. Then we would need an array that has the same number of elements as the number of classes. So each class has a separate spot to store its class sizes.

Now imagine that we didn't have just three students, but we had three hundred. In this case, it would take three hundred lines of code to reference every student, but we can reduce this by using information we already know- a loop.

Let us look at some examples of how to access individual elements of an array with a loop.

Problem 11.1

Read five numbers into an array then print the contents in the reverse order.

Figure 11.1: Pseudocode

```
# Purpose: To read and reverse print five numbers in an array
Start
  A[5] = {0.0}
  For (index = 0; index < 5; index = index + 1)
    A[index]= input("Please enter a number to store.")
  EndFor
  print "The reversed numbers are: "
  For (index = 4; index >=0; index = index -1)
    print A[index]
  EndFor
Stop
```

Note that the only changes needed for going through an array forward or backward is the starting place, ending place and increment (or decrement). Fortunately a *for* loop structure provides a great organization to place all of the modifications on the first line of the loop.

To clearly understand how this program works, let's examine the following flowchart and trace table to see how we can store the variables in the array using the first *for* loop created with the user-entered data below:

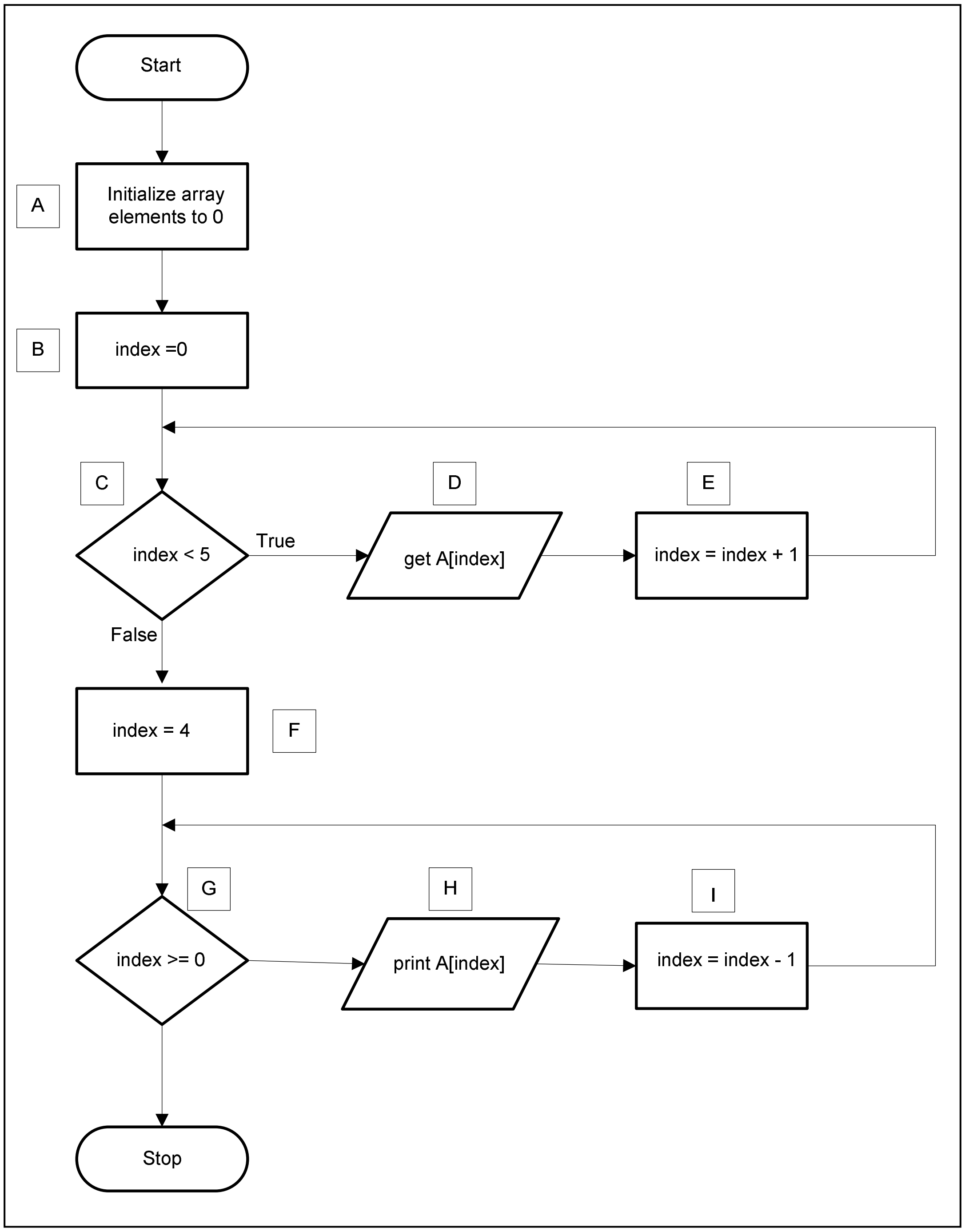
Sample Data:

```
14
30
15
12
18
```

To enter the data above as requested by the program, the user must first enter the value and then press enter after each value.

To understand the behavior of the code completely, let's examine the flowchart of the pseudocode and then run the program in Python.

Figure 11.2: Flowchart



In executing the first loop of the Figure 11.1, we will be looking at the boxed areas for A-E. Initially we begin with an index of 0 at Box B. Since the condition is True, *index* is 0 < 5 in Box C, we continue to Box D and read the value entered by the user for A[0] which is 14. The updated memory diagram is:

A[0]	A[1]	A[2]	A[3]	A[4]
14	**0.0**	**0.0**	**0.0**	**0.0**

For Box E, the *index* is increased to 1 and we return to check the condition in Box C. Since *index* is *1 < 5* is True, we continue to Box D and the user enters 30 to be stored in A[1]. The updated memory diagram is:

A[0]	A[1]	A[2]	A[3]	A[4]
14.0	**30.0**	**0.0**	**0.0**	**0.0**

For Box E, the index is increased to 2 and we return to check the condition in Box C. Since *index* is *2 < 5* is True, we continue to Box C and the user enters 15 to be stored in A[2]. The updated memory diagram is:

A[0]	A[1]	A[2]	A[3]	A[4]
14.0	**30.0**	**15.0**	**0.0**	**0.0**

For Box E, the *index* is increased to 3 and we return to check the condition in Box C. Since the *index* is *3 < 5* is True, we continue to Box D and the user enters 12 to be stored in A[3]. The updated memory diagram is:

A[0]	A[1]	A[2]	A[3]	A[4]
14.0	**30.0**	**15.0**	**12.0**	**0.0**

For Box E, the *index* is increased to 4 and we return to check the condition in Box C. Since the *index* is *4 < 5* is True, we continue to Box D and the user enters 18 to be stored in A[4]. The updated memory diagram is:

A[0]	A[1]	A[2]	A[3]	A[4]
14.0	**30.0**	**15.0**	**12.0**	**18.0**

For Box E, the *index* is increased to 5 and we return to check the condition in Box C. Since the *index* is *5 < 5* is False, we do not enter the loop and continue past the first loop.

For the second loop, we will be focusing on the boxed areas for F-I. Similarly, the second loop will start with *index* as 4 and loops through all the values in the array in reverse order until the *index* is negative. Note that the array is not modified in this loop, it is only viewed.

Table 11.1: Trace table for the first loop

Iteration #	Box #	index	index< 5?	A[index]
Start				
1	B	0		
	C		True	
	D			14
	E	1		
2	C		True	
	D			30
	E	2		
3	C		True	
	D			15
	E	3		
4	C		True	
	D			12
	E	4		
5	C		True	
	D			18
	E	5		
6	C		False	
Stop				

Now we can examine what has been stored in the array by looking at a graphical picture of the storage:

A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**
14.0	**30.0**	**15.0**	**12.0**	**18.0**

To see the behavior of the second *for* loop, let's look at the trace table below and see how the information will be printed back out to the screen.

Table 11.2: Trace table for the second loop

Iteration #	Box #	index	Index >= 0?	Output A[index]
Start				
1	F	4		
	G		True	
	H			18
	I	3		
2	G		True	

	H			12
	I	2		
3	G		True	
	H			15
	I	1		
4	G		True	
	H			30
	I	0		
5	G		True	
	H			14
	I	-1		
6	G		False	
Stop				

Therefore, our resulting output will be:

Figure 11.3: Output

```
18
12
15
30
14
```

In Python, no distinction is made between arrays and lists. To declare an array with 5 elements in Python, we can say:

```
A = [14, 30, 15, 12, 18]
```

The first element is at location 0:

```
A[0]
```

which will return the value of 14. To access the next element use:

```
A[1]
```

which will return the value of 30.

When you want to print the reverse list of all the elements using a loop, a *for* loop provides an excellent structure using the keyword *range*. Let’s first examine the pseudocode again:

```
print "The reversed numbers are: "
For (index = 4; index >=0; index = index -1)
  print A[index]
EndFor
```

In the *for* loop we used the condition *index*>= 0; for Python there is only the condition *index*>*value*, so we must change the condition to be *index*>*-1* to accomplish the same task. In Python, *index* will initially be 4 and decreases by 1 in each iteration until *index*>-1. Therefore, the pseudocode for the *for* loop described above will give the same output as the for loop in the Python program fragment shown below.

```
for index in range(4, -1, -1)
```

Figure 11.4: Python program

```
A = []
for i in range(0,5):
  x = int(input("Enter a number: "))
  A.append(x)
for i in range (4, -1, -1):
  print (A[i])
```

Figure 11.5: Output

```
Enter a number: 14
Enter a number: 30
Enter a number: 15
Enter a number: 12
Enter a number: 18
18
12
15
30
14
```

Self-Check 11.3

Convert the previous problem to read in 10 values and then print every other value in reverse order starting with the last value.

Problem 11.2

Create a program that will print the smallest value in an array containing numbers.

Figure 11.6: Pseudocode

```
# Purpose: Print the smallest value in a user
# entered array(main program)
Start
  A[10]={0}
  index = 0
   While index < 10 do
    get A[index]
    index = index + 1
  EndWhile
  min= A[0]
  index = 1
  While index < 10 Do
    If (min > A[index]) Then
      min = A[index]
    EndIf
    index = index + 1
  EndWhile
  print min
Stop
```

By initializing *min* = *values[0]* we are guaranteed that the initialized value of *min* is not smaller than all the other elements in the values *array*. Since we have this initialization, I can start my index for the *for* loop with 1.

Figure 11.7: Flowchart

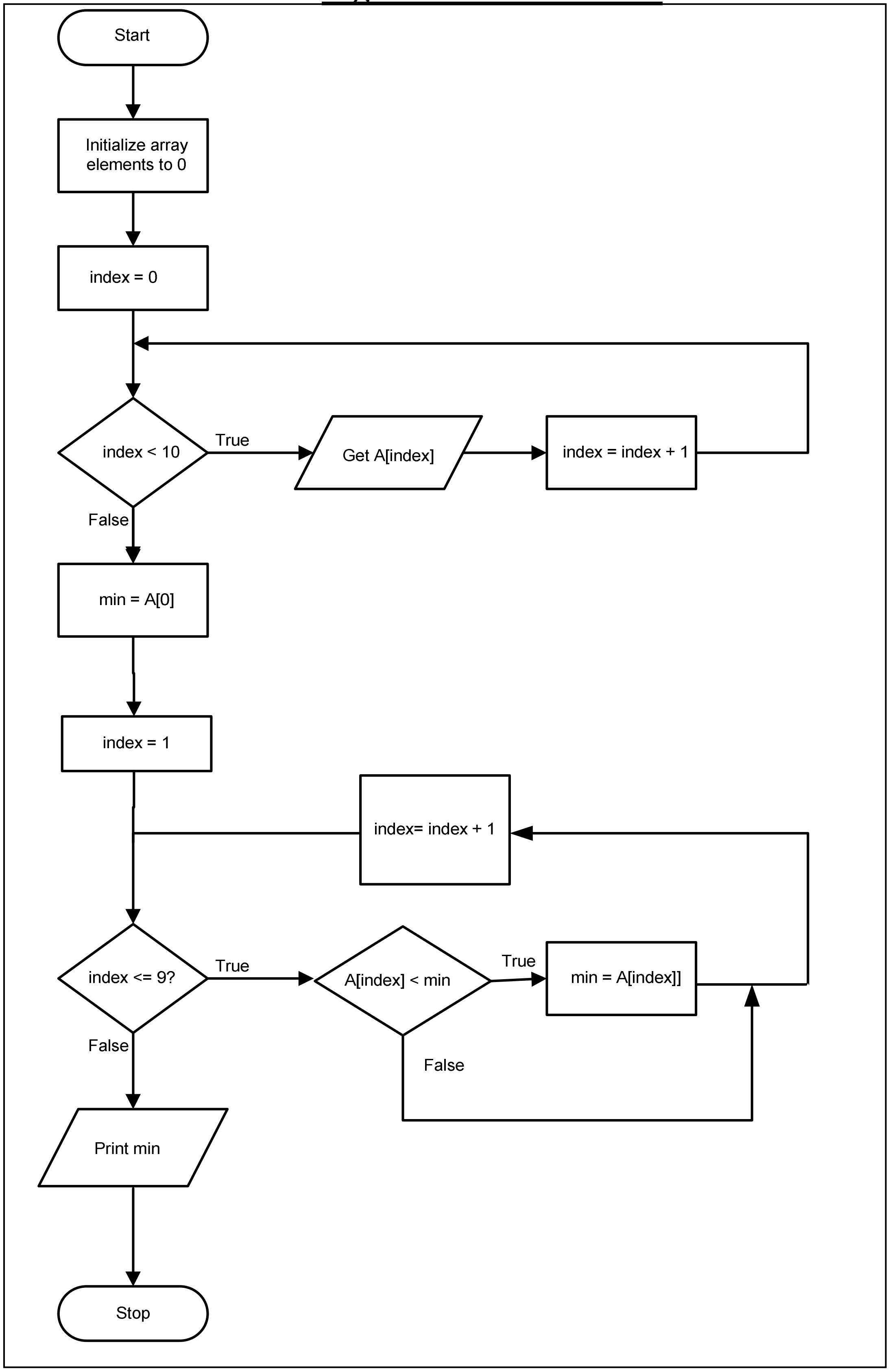


Figure 11.8: Python program

```
A = []
for index in range(0,10):
  x = int(input("Enter a number: "))
  A.append(x)
min = A[0]
index =1
while index <=9:
  if A[index] < min:
    min = A[index]
  index = index + 1
print (min)
```

Figure 11.9: Output

```
Enter a number: 72
Enter a number: 95
Enter a number: 56
Enter a number: 84
Enter a number: 93
Enter a number: 45
Enter a number: 69
Enter a number: 35
Enter a number: 25
Enter a number: 78
25
```

11.2 Two-dimensional arrays

Arrays don't have to be just one dimensional. When you calculate your bills for the month, you have both rows and columns. In the example below, you have two columns and two rows:

Bills for the month

Company	**Cost**
Water and Sewer	$15.87
Electricity	$35.67

In creating a two-dimensional array, you will need to make sure that all elements of the array are of the same data type. The pseudocode will look very similar, but now you will need to have a value for the second dimension. For example:

```
arrayName[numOfRows][numOfColumns]
     = {{row1col1,row1col2}, {row2col1, row2col2}, …}
```

Let’s say that we want to create the following table:

1	2
3	4
5	6

Your declaration will be:

```
     numberArray[3][2]= {{1,2},{3,4},{5,6}}
```

Let’s say that you wanted to access the element in the second row, first column, then you would say:

```
     numberArray[1][0]
```

which would return the value of *‘3’*.

In Python you can create a list of lists to simulate a **two-dimensional array**.

Self-Check 11.4

Create the following array:

1	2	3
4	5	6

Write the pseudocode to print the element with the value of 5.

11.3 Multi-dimensional arrays

In theory, there is no limit to the number of dimensions that an array can have. When you have more than two dimensions, we refer to the arrays as **multi-dimensional array**. It is beyond the scope of this text to cover this topic in more detail, but is important to be aware of the concept.

Key Terms

Array	One-dimensional array
Index	Elements
Subscript	Multi-dimensional array
Initialize	Two-dimensional array
Parallel array	

Exercises

1. Assume that there are ten numerical elements in an array that has already been populated with values. Draw the flowchart to print the largest of all the elements of one-dimensional array.

2. Draw a flowchart to create an array of six elements. Your flowchart should prompt the user to populate the array. Sample data is shown below:

3	5	7	11	13	17

3. Write a Python program to create a multiplication table for the number 5 as shown below:

5	10	15	20	25

4. What will following program display?

```
num = []
for i in range(0,5):
  num.append(i)
for i in range(0,5):
  print (num[i] - 2)
```

5. Write a Python program that creates an array of 6 elements and using a loop stores the ascending values from 5 to 10 in the array. Then print the contents of each element of the array in reverse order.

6. Examining the Python program below, show what will be printed to the monitor.

```
num = []
for i in range(0,5):
  num.append(2 * i)
for i in range(0,5):
  print (num[i])
for i in range(4,-1,-1):
  print (num[i])
```

7. What is printed out from the code below if the user enters the following numbers for the array: 7, 12, -3, 6, 15, 17, 19, 14, 2, -7 ?

```
A = []
for i in range(0,10):
  x = int(input("Enter a number: "))
  A.append(x)
sum = 0
i=0
while i<= 9:
  sum = sum + A[i]
  i = i + 1
print ("Average = ", sum/10.0)
```

8. Complete the following parts:
   a. Draw a flowchart to increment each element of an array by 12. Assume there are 5 elements.

   b. Write the corresponding Python program. Run and test your program with the following sample data:

      10 5 20 30 12

      After the run, your program should display:

      22 17 32 42 24

9. Complete the following parts:
   a. Draw a flowchart to display the sum of all the elements of the array. Assume there are five elements.

   b. Write the corresponding Python program. Run and test your program with the following sample data:

10   5   15   20   30

Your program should display 80.

Self-Check Solutions

11.1

Since we start counting at zero, the last month must be at *numberOfElements – 1*, i.e. *month*[11]

11.2

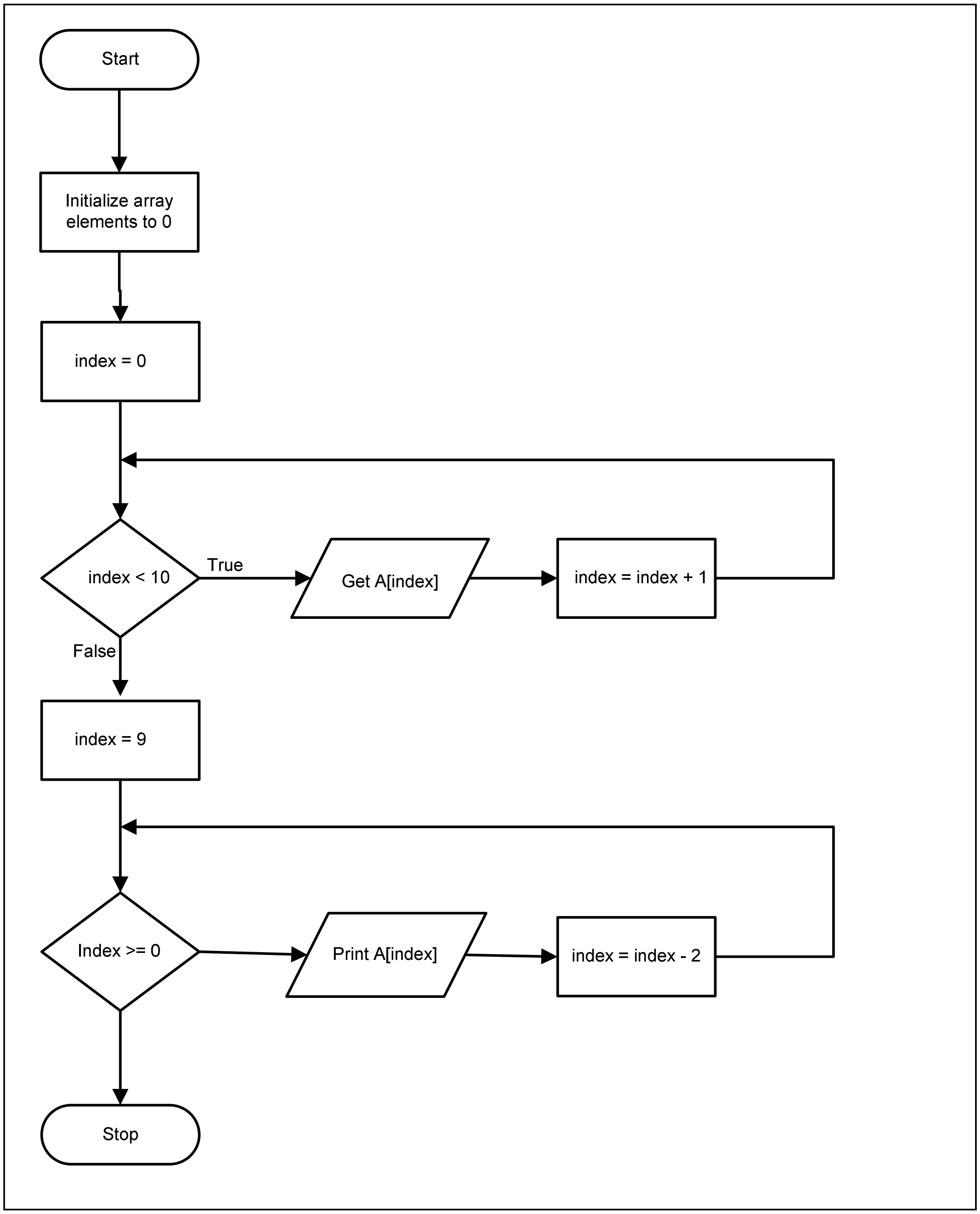
finalGrades[1] = 87

11.3

Pseudocode:

```
Start
  # Purpose: To read ten numbers and reverse print every other
  # number in an array.
  A[10] = {0.0}
  For (index = 0; index < 10; index = index + 1)
     A[index]= input(“Please enter a number to store.”)
  EndFor
  print “Every other reversed number is: ”
  For (index = 9; index >= 0; index = index - 2)
     print A[index] + “,”
  EndFor
Stop
```

Flowchart


Start
Initialize array elements to 0
index = 0
index < 10
True
False
Get A[index]
index = index + 1
index = 9
Index >= 0
Print A[index]
index = index - 2
Stop

11.4

```
numberArray[2][3]= {{1,2,3},{4,5,6}}
Print numberArray[1][1]
```

12. File Input and Output

Did the last chapter give you the idea that you could be creating a large amount of data easily using loops? Imagine a loop that goes through all of a company's employees to print their monthly time sheets. Would you want to type in all their names and addresses manually every month? What happens if you make a mistake when you manually input the information?

When you have large amounts of information that you want to keep track of, the best approach is to store the information in a file. You have already been storing your code for functions and scripts in a file. In this chapter we are going to examine how to store data in files too.

When you are reading information from a file, we call the file an **input file**. Likewise, when you are sending information to be stored in a file, we call the file an **output file**.

12.1 Pseudocode for File Operations

To send or retrieve information from a file, you need to **open** the file first. We use the command below to accomplish this task.

```
open fileName
```

When you want to send output to a file, use the key word *write* followed by the *fileName* and what you want to go into the file (i.e. a string)

```
write filename "This information is for my file."
```

You may also use a variable

```
write filename newPay
```

To **read** from a file that has already been opened, you need to use the keyword *read* along with the *fileName* and the variable to store the information that you just read.

```
read fileName variable
```

As you read information from a file, you continue reading until you encounter the end of the file, **EOF**. The end of file can be detected by a simple test. It is usually a special character attached to the end of a file. Therefore you can create a loop that will continue until the EOF.

When you have finished editing or reading from a file, you will need to **close** the file. The command for this is:

```
    close fileName
```

12.2 Problem 12.1

This section demonstrates a file I/O problem without a loop. We write a program to read a name and three scores from an input file named *data.txt*. An **input** file is a file that contains data you will be reading. Then send the name and the average of three scores to the output file. To create a data file for input, you can use the **New File** feature of IDLE. In IDLE, go to *File -> New Window* and type the file data. Just make sure you save your file with the appropriate name. In Windows, you may also use Notepad to create a data file. Do NOT use Word or WordPad to create a data file because the file is saved in a format that is not easy to read.

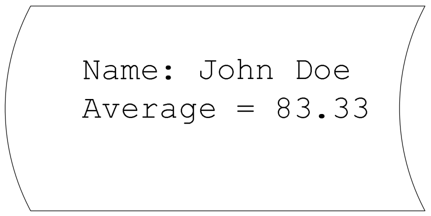
The input file contains the following data:

Figure 12.1: Input file (data.txt)

```
John Doe
70
90
85
```

We want to send the output to another file. An **output** file is a file where you can write your results. *sol.txt* is the name of the output file. The output file is created under the same directory where the Python program is located. Similarly, the input file should be located in the same directory as the Python program.

Figure 12.2: Output file (sol.txt)



```
Name: John Doe
Average = 83.33
```

The average is calculated by adding the three scores and dividing by 3.

average = (70 + 95 + 85) /3 = 250/3 = 83.33

One solution to problem 12.1 is shown in figures 12.3 – 12.6 below.

Figure 12.3: Pseudocode

```
Start
  # open and read info from input file:data.txt
  open data.txt
  read data.txt name
  read data.txt grade1, grade2, grade3
  close data.txt
  # sending information to the output  file:sol.txt
  open sol.txt
  write sol.txt “Name: ” + name
  average = (grade1 + grade2 + grade3)/3.0
  write sol.txt “Average= ” + average
  close sol.txt
Stop
```

Figure 12.4: Flowchart

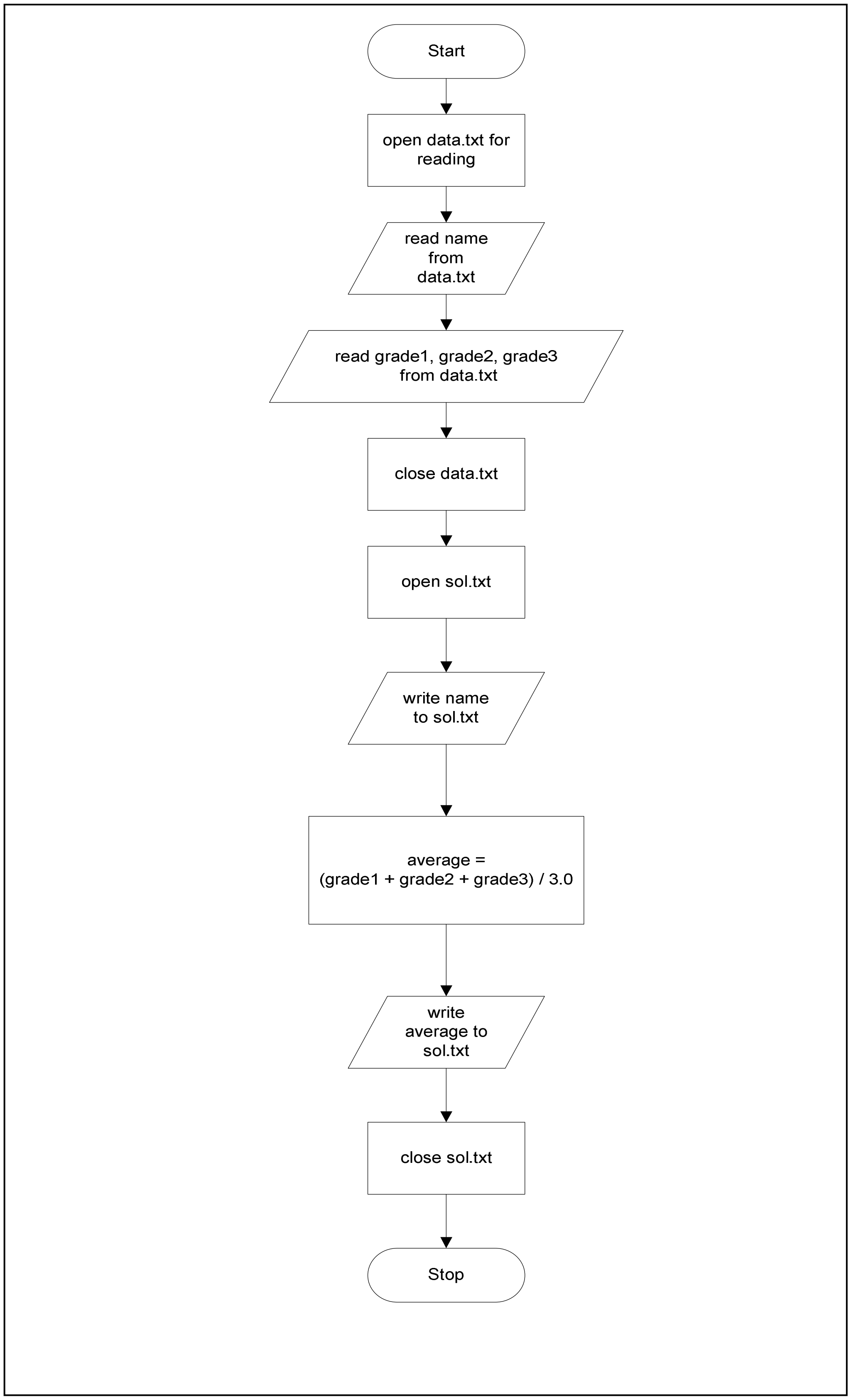
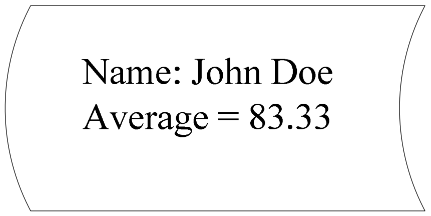


Figure 12.5: Python program

```
inFile=open(“data.txt”,'r')
name = inFile.readline()
score1=inFile.readline()
score2=inFile.readline()
score3=inFile.readline()
inFile.close()
outFile=open(“sol.txt”,'w')
s1=int(score1)
s2=int(score2)
s3=int(score3)
average = (s1+s2+s3)/3.0
outFile.write("Name:")
outFile.write(name)
outFile.write("Average = ")
outFile.write(str(average))
outFile.close()
```

Note that input files should be in the same directory as the Python program. Output files are also created in the same directory. The first *inFile.readline()* reads “John Doe” from the file and assigns it to the *name* variable. The second *inFile.readline()* assigns “70” to *score1*, the third *inFile.readline()* assigns “90” to *score2* and the fourth *inFile.readline()* assigns*“85”*to *score3*. Note that *score1*, *score2* and *score3* are of type *string* and they need to be converted to type *integer* before they can be used for the calculation. The contents of output file are shown below.

Figure 12.6: Output



```
Name: John Doe
Average = 83.33
```

12.3 Problem 12.2

This section demonstrates a file I/O problem with a loop. Modifying the above example, create a program to read a name and three scores from an input file known as data.txt. We may have any number of students. This will require us to read names and grades until we encounter the EOF.

One solution to problem 12.2 is shown in figures 12.7 – 12.11 below.

Figure 12.7: Pseudocode

```
Start
  # open and read info from input file:data.txt
  open data.txt
  open sol.txt
  read data.txt name
  While(not EOF)
    read data.txt grade1, grade2, grade3
    # sending information to the output file:sol.txt
    write sol.txt “Name: ” + name
    average = (grade1 + grade2 + grade3)/3.0
    write sol.txt “Average= ” + average
    read data.txt name
  EndWhile
  close data.txt
  close sol.txt
Stop
```

Figure 12.8: Flowchart

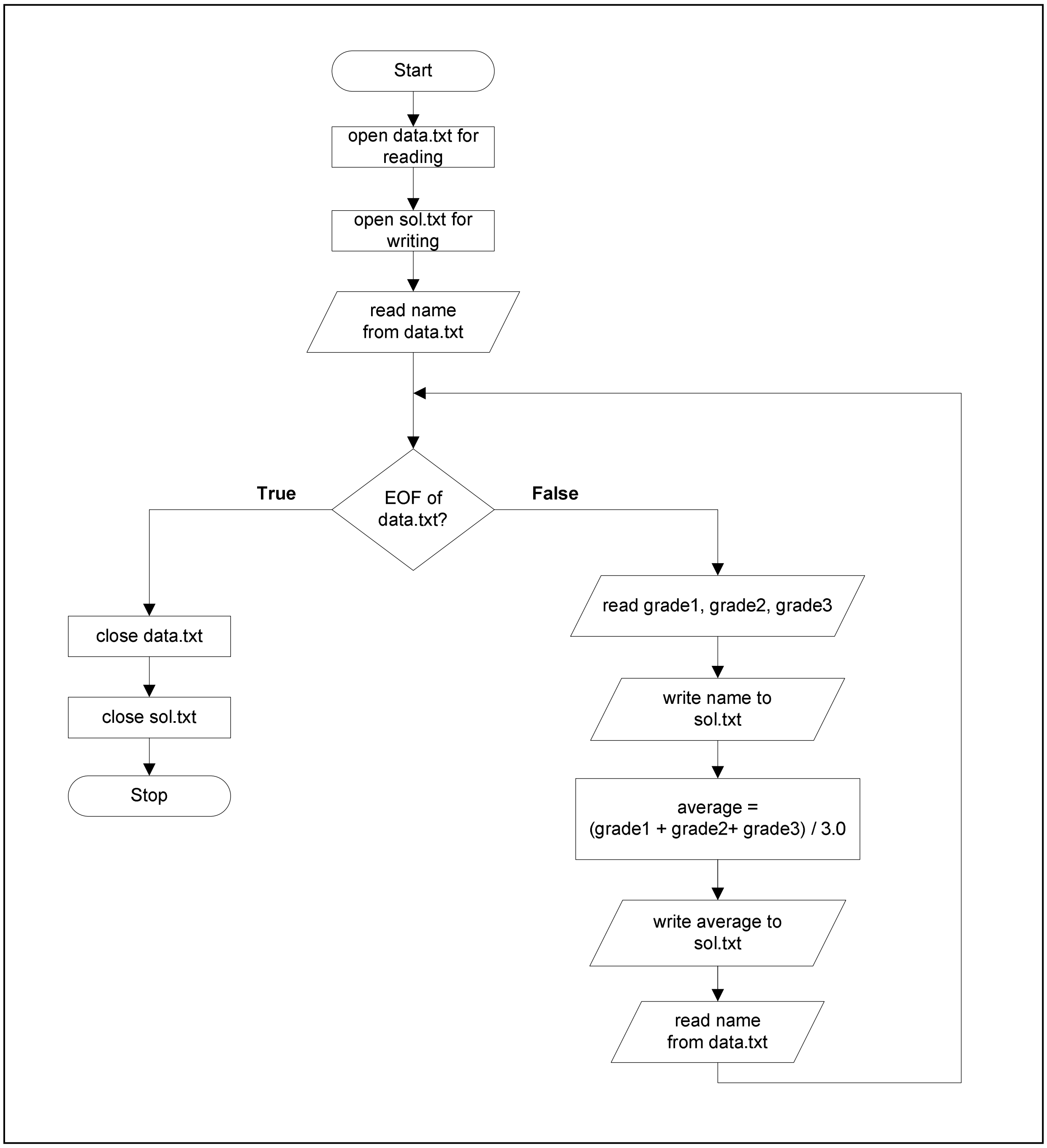


Figure 12.9: Python program

```
inFile=open(“data.txt”,'r')
outFile=open(“sol.txt”,'w')
name = inFile.readline()
while (name != ""):
  grade1=inFile.readline()
  grade2=inFile.readline()
  grade3=inFile.readline()
  outFile.write(name + "\n"))
  s1=int(grade1)
  s2=int(grade2)
  s3=int(grade3)
  average = (s1+s2+s3)/3.0
  outFile.write(str(average)+"\n")
  name = inFile.readline()
inFile.close()
outFile.close()
```

Figure 12.10: Input file (data.txt)

```
Pat Smith
60
70
75
Chris White
80
90
95
Jerry Waldon
60
90
80
Josh Cathey
77
99
88
```

Figure 12.11: Output file sol.txt

```
Pat Smith
68.3333333333
Chris White
88.3333333333
Jerry Waldon
76.6666666667
Josh Cathey
88.0
```

Self-Check 12.1

Rewrite the problem in Section 12.2 by storing and retrieving the numerical information in a grade array.

12.4 Problem 12.3

This section discusses another file I/O problem with a loop. Modifying the problem of 12.2, we read name and grades. We calculate the average of grades but do not know how many grades we are averaging. This will require us to sum all the grades until we encounter the EOF.

Figure 12.12: Pseudocode

```
Start
  # open and read info from file
  open data.txt
  read data.txt name
  # we must read the first value to see if we are at the EOF
  read data.txt grade
  count = 0
  sum = 0.0
  While !EOF of data.txt Do
    count = count + 1
    sum = sum + grade
    read data.txt grade
  EndWhile
  close data.txt
  # printing information to the solution file
  open sol.txt
  write sol.txt “Name: ” + name
  average = sum/count
  write sol.txt “Average= ” + average
  close sol.txt
Stop
```

Figure 12.13: Flowchart

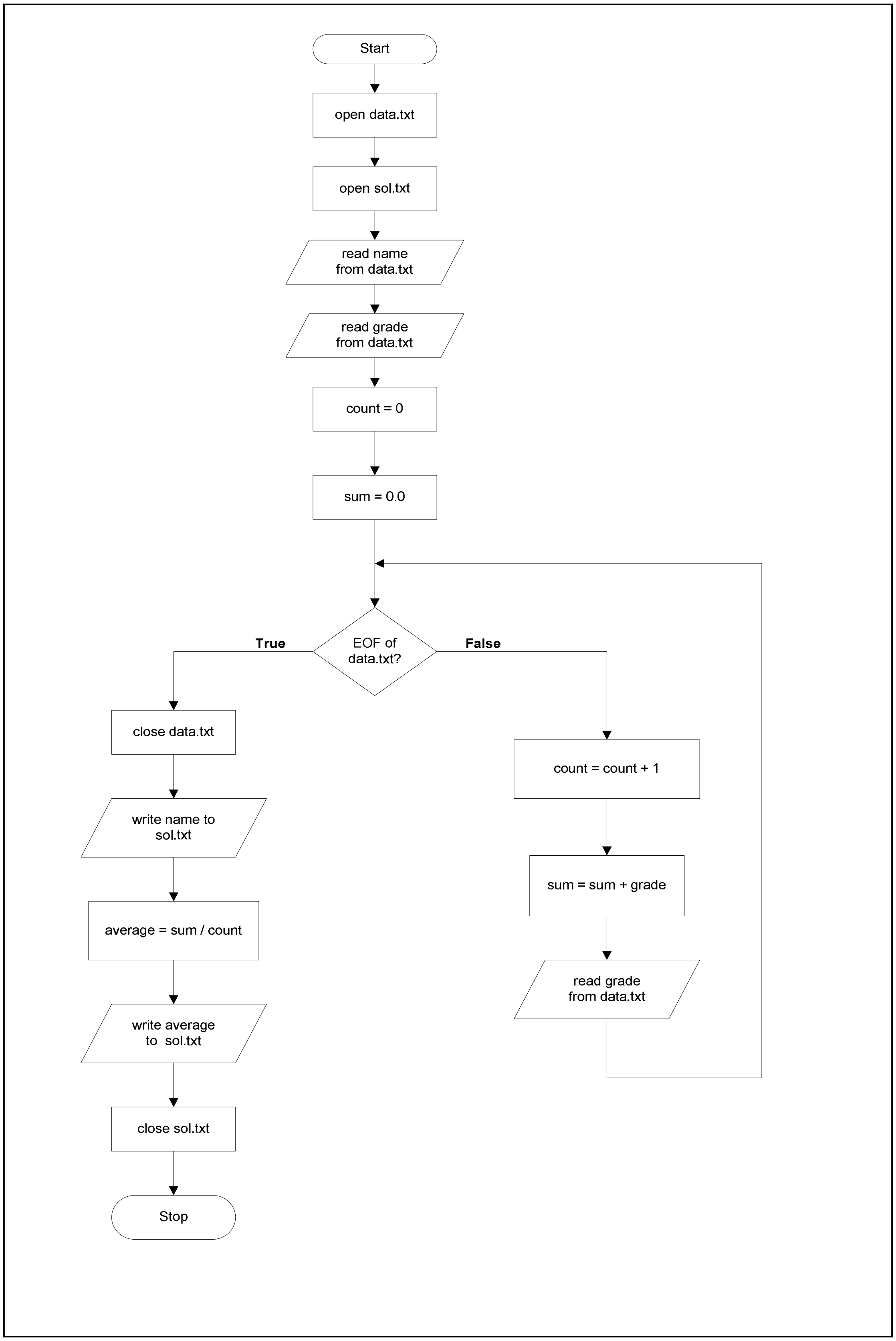


Figure 12.14: Python program

```
inFile=open(“data.txt”,'r')
outFile=open(“sol.txt”,'w')
name = inFile.readline()
grade = inFile.readline()
count = 0
sum = 0.0
while (grade != ""):
  count = count + 1
  sum = sum + float(grade)
  grade = inFile.readline()
inFile.close()
outFile.write("Name = " + name + "\n")
average = sum / count
outFile.write("Average = " + str(average))
outFile.close()
```

Figure 12.15: Input file (data.txt)

```
Pat Smith
60
70
75
80
90
95
```

Figure 12.16: Output file sol.txt

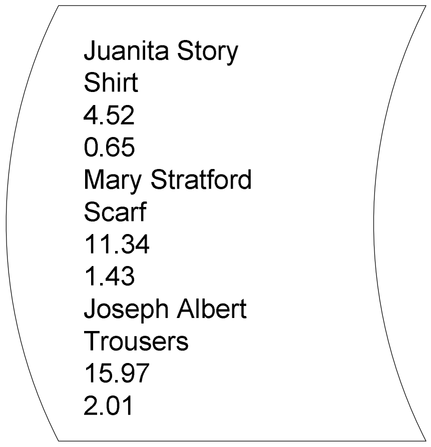
```
Name = Pat Smith
Average = 78.3333333333
```

12.5 Problem 12.4

Given the following information in a *purchase.txt* file, create a program that will read the information and send the contents to the output file print.txt.

The file, *purchase.txt,* will contain the following records mentioned above in the following format:

<u>Figure 12.17: Input file purchase.txt</u>



```
Juanita Story
Shirt
4.52
0.65
Mary Stratford
Scarf
11.34
1.43
Joseph Albert
Trousers
15.97
2.01
```

Note that the name of the patron is followed by the item purchased. This is followed by the price and then the tax on that item.

We want output to be in the following format:

```
Patron 1:  Juanita Story
Purchase: Shirt
Cost: $5.17

Patron 2:  Mary Stratford
Purchase: Scarf
Cost: $12.77

Patron 3:  Joseph Albert
Purchase: Trousers
Cost: $17.98
```

Note that the cost is the sum of the price and the tax of the item.

Figure 12.18: Pseudocode

```
Start
  # open and read info from input file:purchase.txt
  open purchase.txt
  open print.txt
  i=1
  read purchase.txt name
  While(not EOF) Do
    read purchase.txt type, cost, tax
    # sending information to the output  file:print.txt
    write print.txt “Patron ” + i + ": " + name
    write print.txt “Purchase: ”  + type
    write print.txt “Cost: ”  + (cost+tax)
    i = i+ 1
    read purchase.txt name
  EndWhile
  close purchase.txt
  close print.txt
Stop
```

Figure 12.19: Flowchart

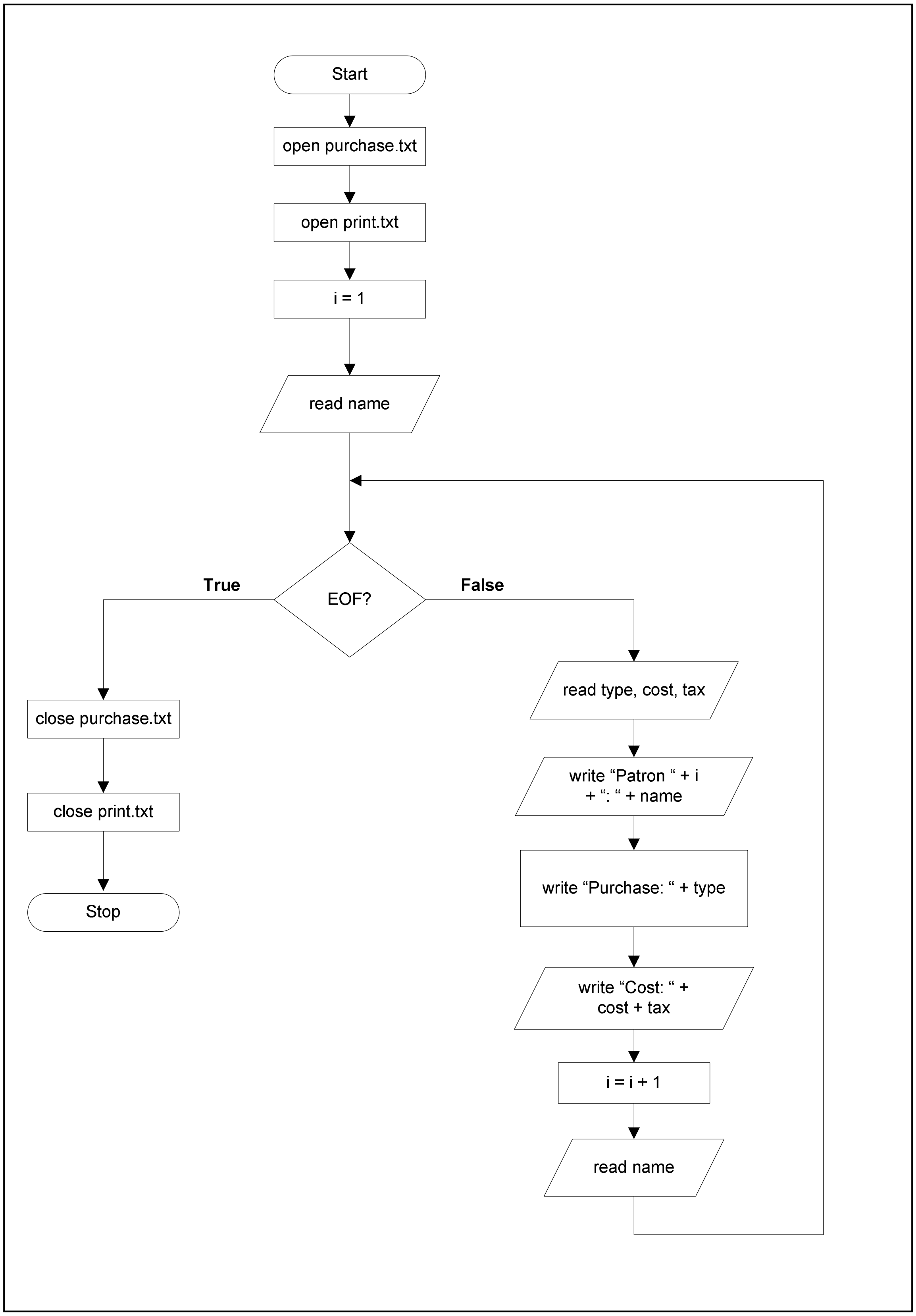
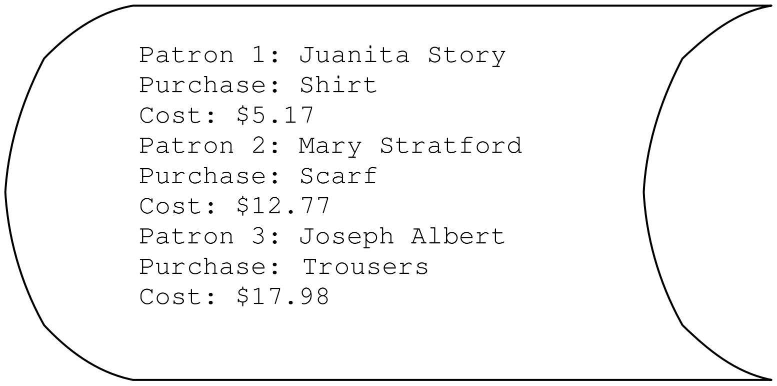


Figure 12.20: Python program

```
inFile=open(“purchase.txt”,'r')
outFile=open(“sol.txt”,'w')
name = inFile.readline()
grade = inFIle.readline()
count = 0
sum = 0.0
while(grade != "")
  count = count + 1
  sum = sum + float(grade)
  grade = inFile.readline()
inFile.close()
outFile.write("Name = " + name)
average = sum / count
outFile.write("Average = " + str(average))
outFile.close()
```

Figure 12.21: Output file sol.txt



```
Patron 1: Juanita Story
Purchase: Shirt
Cost: $5.17
Patron 2: Mary Stratford
Purchase: Scarf
Cost: $12.77
Patron 3: Joseph Albert
Purchase: Trousers
Cost: $17.98
```

Normally, when you open a file for writing, you start writing at the beginning of the file and all previously written information in the file is erased. If you wanted to add information to an existing output file, you need to **append** the information to the end of the file. The only difference is that you use the keyword append when opening the file.

```
    open fileName append
```

Everything is performed as before, but all the information is now automatically written at the end of the file.

Key Terms

Open	Input
Read	New File
EOF	Output
Close	Append

Exercises

1. You are a teaching assistant to Mr. Couch and he has asked you to create a file with the records of all student grades for his History 1 class. In addition, you have been asked to create a python program to read in the input file and send out the class average to an output file. Note that you will need a loop to read the data from the input file until EOF is reached.

   Below is the information that Mr. Couch needs in his input grade file. Create the grade file (i.e. using Notepad) and send the class average to an output file.

   ```
   Mandy Bloom
   88
   Jeff Davis
   81
   Hector Juarez
   92
   Sandra Manning
   89
   Thomas Nuez
   78
   Karen Tunes
   81
   ```

2. Write a Python program to read pairs of numbers from an input file and calculate their sum. Send the sum of each pair with the appropriate message to an output file. See example below.

   inputFile.txt

   ```
   10    20
   5     15
   30    80
   ```

```
75      10
```

outputFile.txt

```
The sum of 10 and 20 is 30.
The sum of 5 and 15 is 20.
The sum of 30 and 80 is 110.
The sum of 75 and 10 is 85.
```

3. You have been thinking about creating a home theatre and want to keep all your notes in a file to be able to quickly calculate the total cost of the system. The input file should contain the data shown below; send the appropriate message along with the total cost to the output file. Note that you will need a loop to read the data from the input file until EOF is reached.

   Below is the information that has been gathered for the input file. Create a python program to calculate the total cost of the system and send the answer to an output file.

```
Receiver
310
Flat Screen TV
959
Center Speaker
109
Side Speakers
289
Rear Speakers
199
HDMI cable for DVD to TV
39
Cables for speaker
26
DVD
34.99
```

4. Create a Python program which asks the user for a number. Then the program determines whether that number is even or odd. This program should continue to ask the user for input until the user enters a zero. Send the numbers and your findings to an output file.

User input from the keyboard:

```
5
10
17
0
```

outputFile.txt

```
5 is an odd number.
10 is an even number.
17 is an odd number.
```

5. In order to organize your monthly bills, you want to create a file listing all of your July expenses with the amount due for the month. Below is the information entered by the user via the keyboard that will need to end up in the output file along with the total amount spent for the month.

```
Gas
22.10
Electric
69.31
GrocWk1
52.12
GrocWk2
48.61
GrocWk3
60.92
GrocWk4
54.74
Water&Sewer
15.19
```

6. Looking over your personal library, you want a record of all the books you own. You want to create a file with at least five books titles and author and another file that can read your collection and print it in a table using the tab character, '\t', as needed. Make sure you print titles for your table to include the Author and Title columns.

7. You are helping your local environmental group keep track of their expenses for the year to create a file using Notepad with the information below and another file to print the total cost for the club for the year.

Below you will find the information containing the yearly costs accrued by the club. Create a file with this information and print the total cost for the year to the screen with the comment before “Total Yearly Club Cost:” followed with the cost.

```
Arbor day trees
83.48
Awareness Flyers
34.99
Roadside Clean-up supplies
12.51
Reseeding Wildflower Area
36.22
Toxic chemical Reclamation Supplies
43.33
Elementary Awareness Supplies
14.31
```

8. Create a Python program to store a class of students with their associated grade information in a file called classGrades.txt. Ask the user for the name of the students and for each students homework and exam grades. Note that you will need the user to state how many students are in the class, how many homework grades each student will have and how many exam grades each student will have. Note that each student will have the same number of homework and exam grades.

Self-Check Solutions

12.1

Pseudocode:

```
Start
  open data.txt
  read name
  read grade[0], grade[1], grade[2]
  close data.txt
  # Printing result to screen
  print “Name: ” + name
  average = (grade[0]+ grade[1] + grade[2])/3.0
  print “Average= ” + average
Stop
```

Python:

```
grade = []
inFile=open("someFile.txt",'r')
name = inFile.readline()
score1=inFile.readline()
grade.append(int(score1))
score2=inFile.readline()
grade.append(int(score2))
score3=inFile.readline()
grade.append(int(score3))
inFile.close()
average = (grade[0]+grade[1]+grade[2])/3.0
print ("Name = " + name)
print ("Score1 = ", grade[0])
print ("Score2 = ", grade[1])
print ("Score3 = ", grade[2])
print ("average = ", average)
```

<u>Input File (data.txt)</u>

```
John Doe
70
90
85
```

<u>Corresponding output:</u>

```
Name = "John Doe"
Score1 = 70
Score2 = 90
Score3 = 85
Average = 81.6666666667
```

Appendix A: Installing Python

Python is a wonderful language to use for a first programming language. In teaching introduction to programming classes, we have found that students greatly enjoy applying the skills they learn to a real application, like Python.

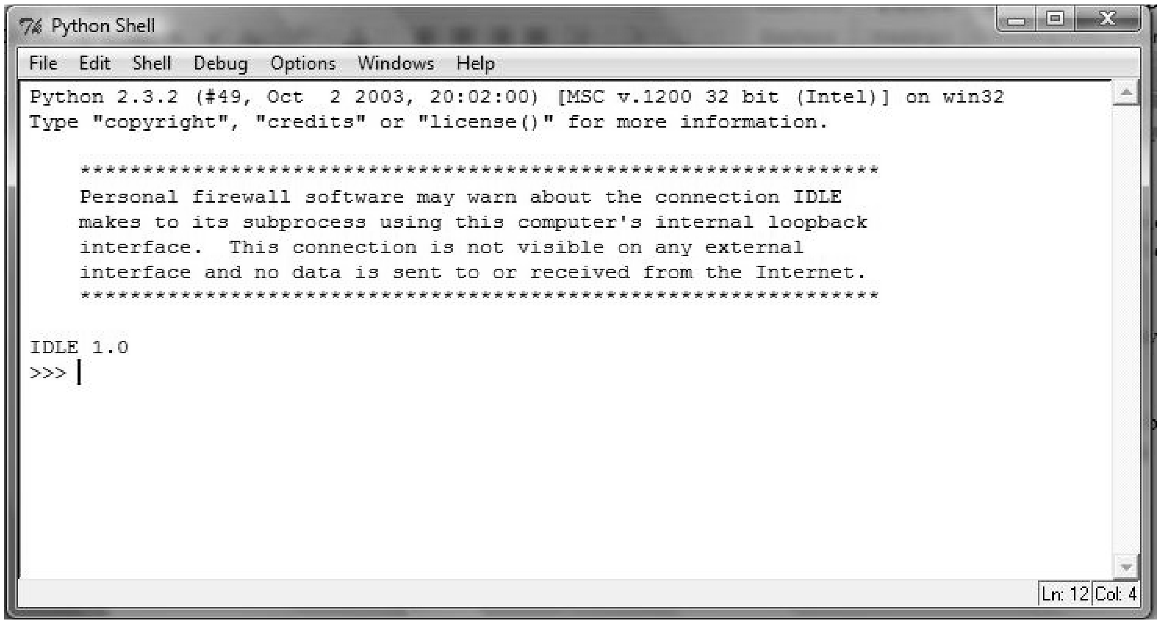
There are many programming languages available to you, but we have found that the beginning programmer can use Python to accomplish tasks that other languages make more challenging; such as reading and writing values to the console window. Also, unlike other languages, Python does not require you to understand, or ignore, advanced concepts before you can write your first line of code.

Since Python is a free language, there are many free materials for you to use available on the Internet. An Internet search will come up with many hits for Python resources. Some great resources include

1. Python is available free online from http://www.python.org/downloads Just follow the instructions given.

2. Think Python: an Introduction to Software Design by Allen B. Downey is a free book under the GNU License found at: http://www.greenteapress.com/thinkpython/ which will delve deeper into the mysteries of Python.

The following figure shows the Python shell that you will see when you first start Python.

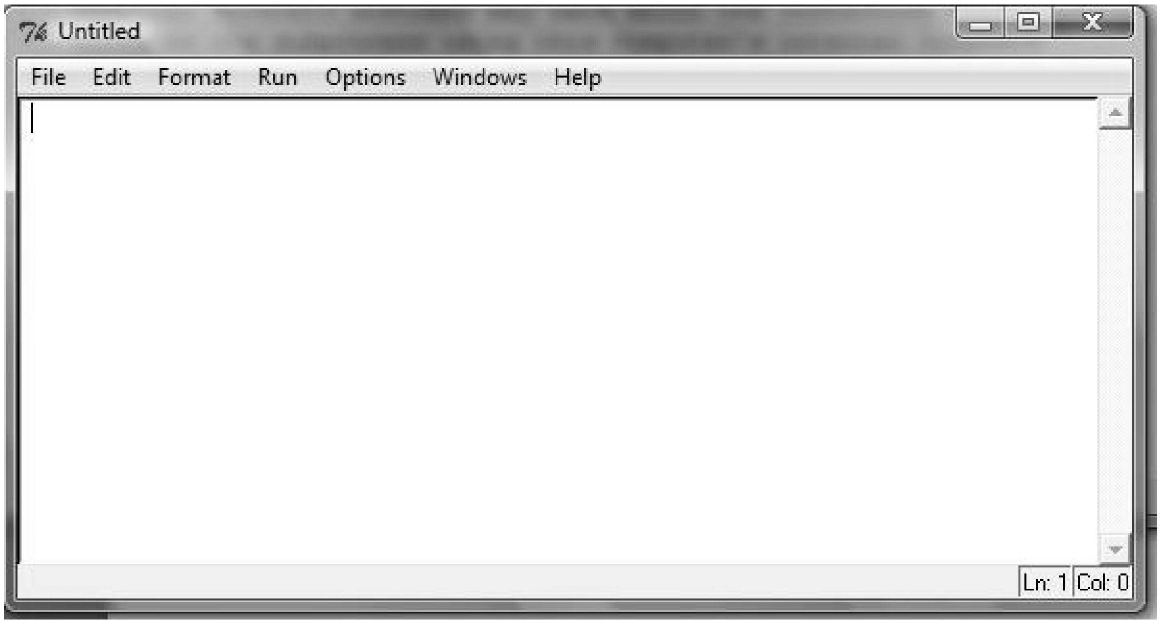


```
Python 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0
>>>
```

If you select “File” and then “New” in the Python Shell, you can enter a Python program in the window shown below.



Once you have entered a Python program, program file can be executed by selecting “Run” and then “Run Module”. You should be prompted to save your file before the code is executed. You can save your program file at any time by selecting “File” and then “Save”. Please see the Python documentation on the website for more details.

References

We have included books that we have found helpful in creating our courses over the years and broken them down by area.

Programming Logic:
Starting out with Programming Logic and Design by Tony Gaddis, Addison Wesley, first edition, 2008.

Language Specific:
An Introduction to Programming with C++ by Diane Zak, Thompson Course Technology, fifth edition, 2005.

There are always new books being added to the Internet. For more resources, please search for “Python book” online.

Index

A

accuracy, 11
addition, 12
algorithm, 3
AND, 50
append, 176
array, 145
arrow, 21
ASCII, 8

B

binary, 15
bit, 47
Boolean, 47
break, 73
bug, 3
built-in function, 133

C

call-by-reference, 131
call-by-value, 131
camel notation, 34
case statement, 72
casting, 16
character, 7
close, 162
code tracing, 87
computer, 3
condition, 84
constant, 36

D

data, 7
data type, 7
debugging, 3
decrement, 84
dependent, 23
Do-While, 102

E

Else, 56
EndIf, 56
ending condition, 86
EOF, 162
exponent, 16
exponentiation, 16

F

fall-through, 73
False, 47
falseChoice, 57
floating-point, 11
flowchart, 20
for loop, 96
function, 130

G

get, 21, 34
goal, 1
Guido von Rossum, 21

I

If, 56
increment, 84
index, 146
infinite loop, 84
infinite loops, 105
initialization, 84
inner loop, 114
input, 1, 34
input file, 163
integer, 11
integer division, 13

L

logical operator, 49

M

modulus, 14
multiple-alternative selection structure, 68

N

negation, 15
nest, 68
nested If-Then-Else, 75
nested loop, 113
NOT, 49

O

objective, 1
open, 162
operand, 12
operator, 12
OR, 50

outer loop, 114
output, 34
output file, 163
oval, 21

P

parallelogram, 21
parameter, 131
print, 21, 34
program, 3
programmer, 3
programming language, 3
pseudocode, 20
Python, 4, 21

R

read, 162
rectangle, 21
relational operators, 47

S

scripts, 130
sentinel, 101
sequence control structure, 20
sequential, 20
Start, 20
Stop, 20
string, 10
subtraction, 12
syntax, 3

T

task, 1
Then, 56
True, 47
trueChoice, 57
truncated, 13
truth table, 49

U

unary, 15
user-defined function, 133

V

variable, 22, 52
void functions, 137

W

while, 84