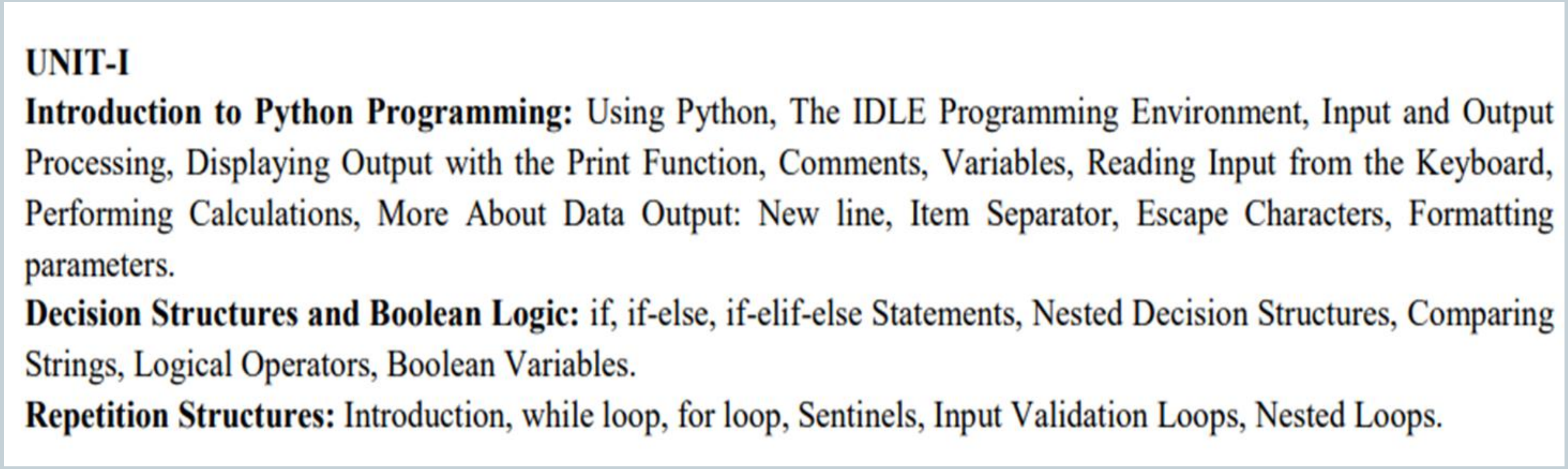
Python Programming (16ITE01)

UNIT - I

T. PRATHIMA, DEPT. OF IT, CBIT

Syllabus



UNIT-I

Introduction to Python Programming: Using Python, The IDLE Programming Environment, Input and Output Processing, Displaying Output with the Print Function, Comments, Variables, Reading Input from the Keyboard, Performing Calculations, More About Data Output: New line, Item Separator, Escape Characters, Formatting parameters.

Decision Structures and Boolean Logic: if, if-else, if-elif-else Statements, Nested Decision Structures, Comparing Strings, Logical Operators, Boolean Variables.

Repetition Structures: Introduction, while loop, for loop, Sentinels, Input Validation Loops, Nested Loops.

Python

- Python is an interpreted high-level programming language for general-purpose programming.
- Created by Guido van Rossum and first released in 1991
- Latest Python version available for windows is 3.7.0 as on date

Python

- Written in Python
- Libraries are written mostly in Python itself, with some performance critical sections written in C.
- There are other implementations:
- CPython and is written in C.
- IronPython (Python running on .NET)
- Jython (Python running on the Java Virtual Machine)
- PyPy (A fast python implementation with a JIT compiler)
- Stackless Python (Branch of CPython supporting microthreads)

Using Python

- Download and Install Python
- Python Interpreter
- Interactive Mode

Installation Instructions

- www.python.org
- Choose your OS
- Download
- Install
- Once installation is over you will find
  - Python IDLE
  - Python command prompt

The Python Interpreter

- When you install the Python language on your computer, one of the items that is installed is the Python interpreter.
- The *Python interpreter* is a program that can read Python programming statements and execute them.
- You can use the interpreter in two modes:
  - interactive mode and
  - script mode.

- In *interactive mode*, the interpreter waits for you to type Python statements on the keyboard.
- Once you type a statement, the interpreter executes it and then waits for you to type another statement.

The Python Interpreter

- In *interactive mode,* the interpreter waits for you to type Python statements on the keyboard.
- Once you type a statement, the interpreter executes it and then waits for you to type another statement.
- In *script mode,* the interpreter reads the contents of a file that contains Python statements.
- Such a file is known as a *Python program* or a *Python script*.
- The interpreter executes each statement in the Python program as it reads it.

Interactive Mode

- When the Python interpreter is running in interactive mode, it is commonly called the *Python shell*.
- When the Python interpreter starts in interactive mode, you will see something like the following displayed in a console window:
- Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53)
- [MSC v.1600 64 bit (AMD64)] on win32
- Type "help", "copyright", "credits" or "license“ for more information.
- >>>
- The >>> that you see is a prompt that indicates the interpreter is waiting for you to type a Python statement.

Examples

```
>>> print('To be or not to be')
To be or not to be

>>> print('That is the question.')
That is the question.

>>>
```

Input, Processing, and Output

- Computer programs typically perform the following three-step process:
  - Input is received.
  - Some process is performed on the input.
  - Output is produced.

Displaying Output with the print Function

- A *function* is a piece of prewritten code that performs an operation.
- Python has numerous built-in functions that perform various operations.
- When programmers execute a function, they say that they are *calling* the function
- Example – print() is a function

```
>>> print('Hello world')
Hello world
>>>
```

print() function contd…

- When you call the print function, you type the word print, followed by a set of parentheses.
- Inside the parentheses, you type an *argument*, which is the data that you want displayed on the screen.

```
print('Kate Austen')
print('123 Full Circle Drive')
print('Asheville, NC 28899')
```

Program Output

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Strings and String Literals

- In programming terms, a sequence of characters that is used as data is called a *string*.
- When a string appears in the actual code of a program it is called a *string literal*.
- In Python code, string literals must be enclosed in quote marks.
- In Python you can enclose string literals in a set of single-quote marks (') or a set of double quote marks (").

Example – string literals

```
print('Your assignment is to read "Hamlet" by tomorrow.')
```

O/p

```
Your assignment is to read "Hamlet" by tomorrow.
```

- Python also allows you to enclose string literals in triple quotes (either """ or ''').
- Triple quoted strings can contain both single quotes and double quotes as part of the string.
- The following statement shows an example:

```
print("""I'm reading "Hamlet" tonight.""")
```

O/p

```
I'm reading "Hamlet" tonight.
```

- Triple quotes can also be used to surround multiline strings, something for which single and double quotes cannot be used.
- Here is an example:

```
print("""One
Two
Three""")
```

O/p

```
One
Two
Three
```

Comments

- Comments are short notes placed in different parts of a program, explaining how those parts of the program work.
- Although comments are a critical part of a program, they are ignored by the Python interpreter.
- Comments are intended for any person reading a program's code, not the computer.
- In Python you begin a comment with the # character.
- When the Python interpreter sees a # character, it ignores everything from that character to the end of the line.

Comments…Example

```
# This program displays a person's
# name and address.
        print('Kate Austen')
        print('123 Full Circle Drive')
        print('Asheville, NC 28899')
```

O/p

```
        Kate Austen
        123 Full Circle Drive
        Asheville, NC 28899
```

end-line comment

- An *end-line comment* is a comment that appears at the end of a line of code.
- It usually explains the statement that appears in that line.

```
print('Kate Austen') # Display the name.
print('123 Full Circle Drive') # Display the address.
print('Asheville, NC 28899') # Display the city, state, and ZIP.
```

O/p

```
Kate Austen
123 Full Circle Drive
Asheville, NC 28899
```

Variables

- **A variable is a name that represents a value stored in the computer's memory.**
- Programs use variables to access and manipulate data that is stored in memory.
- A *variable* is a name that represents a value in the computer's memory.
- When a variable represents a value in the computer's memory, we say that the variable *references* the value.
- You cannot use a variable until you have assigned a value to it.
- An error will occur if you try to perform an operation on a variable, such as printing it, before it has been assigned a value.

Creating Variables

- You use an *assignment statement* to create a variable and make it reference a piece of data.
- Here is an example of an assignment statement:

  age = 25
- An assignment statement is written in the following general format:

  variable = *expression*
- The equal sign (=) is known as the *assignment operator*.
- In the general format, *variable* is the name of a variable and *expression* is a value, or any piece of code that results in a value.
- After an assignment statement executes, the variable listed on the left side of the = operator will reference the value given on the right side of the = operator.

Examples

- Assign
- >>> width = 10 # creates a variable named width and assigns it the value 10
- >>> length = 5 # creates a variable named length and assigns it the value 5
- Print - display the values referenced by these variables

```
>>> print(width)
10
>>> print(length)
5

>>> 25 = age
SyntaxError: can't assign to literal
>>>
```

Examples

- When you pass a variable as an argument to the print function, you do not enclose the variable name in quote marks.
- To demonstrate , look at the following

```
>>> print('width')
width
>>> print(width)
10
>>>
```

Variable Naming Rules

- Although you are allowed to make up your own names for variables, you must follow these rules:
- You cannot use one of Python's key words as a variable name.
- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct.
- This means the variable name ItemsOrdered is not the same as itemsordered.

Camel Case

- camelCase names are written in the following manner:
- The variable name begins with lowercase letters.
- The first character of the second and subsequent words is written in uppercase.
- For example, the following variable names are written in camelCase:

```
grossPay
payRate
hotDogsSoldToday
```

- This style of naming is called camelCase because the uppercase characters that appear in a name may suggest a camel's humps.

Sample variable names

units_per_day	-	Legal
dayOfWeek	-	Legal
3dGraph	-	Illegal
(Variable names cannot begin with a digit)		
June1997	-	Legal
Mixture#3	-	Illegal

(Variable names may only use letters, digits, or underscores)

Displaying Multiple Items with the print Function

- Python allows us to display multiple items with one call to the print function.
- We simply have to separate the items with commas

```
# This program demonstrates a variable.
room = 503
print('I am staying in room number', room)
```

O/p

```
I am staying in room number 503
```

- When the print function executed, it displayed the values of the two arguments in the order that we passed them to the function.
- Notice that the print function automatically printed a space separating the two items.
- When multiple arguments are passed to the print function, they are automatically separated by a space when they are displayed on the screen.

Variable Reassignment

- Variables are called "variable" because they can reference different values while a program is running.
- When you assign a value to a variable, the variable will reference that value until you assign it a different value.
- When a value in memory is no longer referenced by a variable, the Python interpreter automatically removes it from memory through a process known as *garbage collection*.

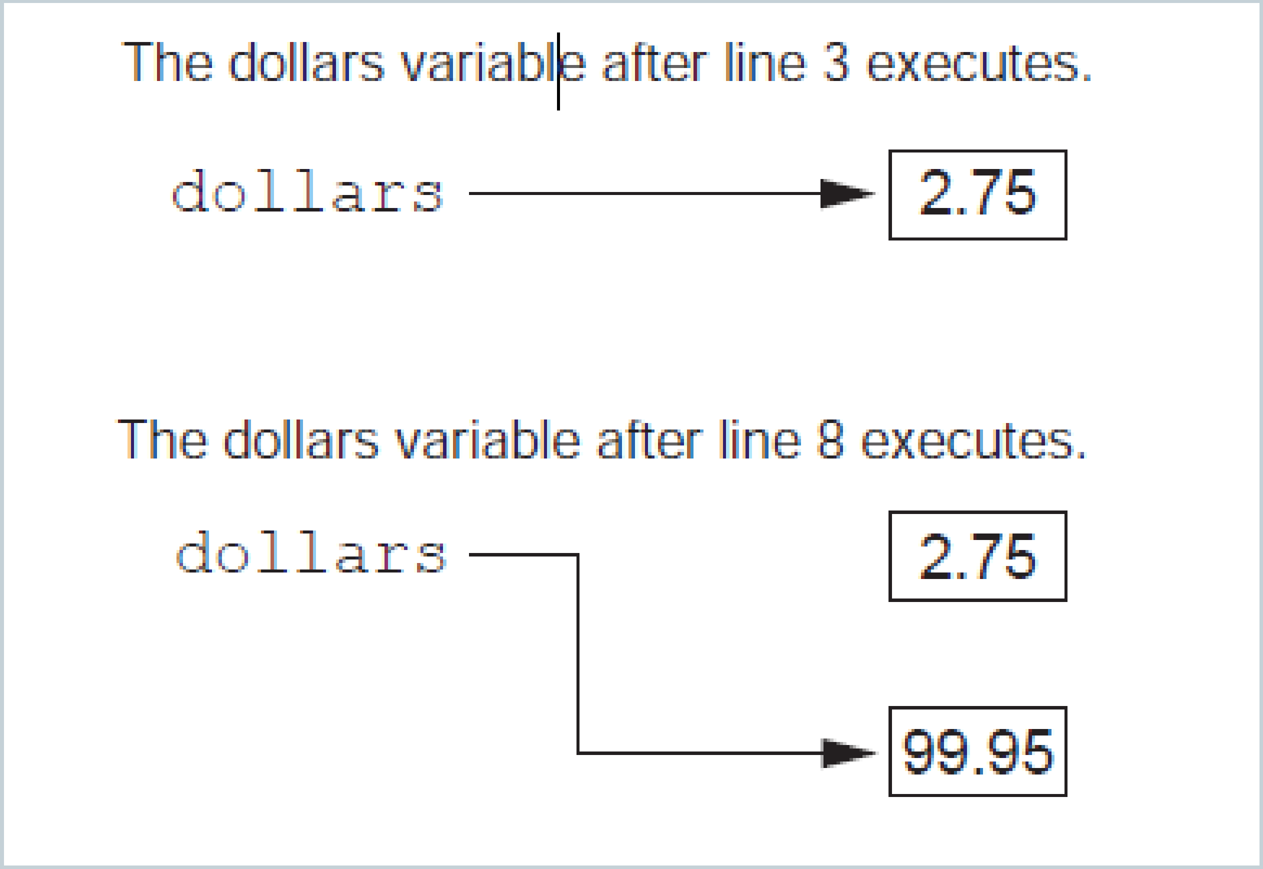
Example

```
# This program demonstrates variable reassignment.
# Assign a value to the dollars variable.
        dollars = 2.75
        print('I have', dollars, 'in my account.')
# Reassign dollars so it references
# a different value.
        dollars = 99.95
        print('But now I have', dollars, 'in my account!')
```

O/p

```
        I have 2.75 in my account.
        But now I have 99.95 in my account!
```

Example


The dollars variable after line 3 executes.
dollars
2.75
The dollars variable after line 8 executes.
dollars
2.75
99.95

Numeric Data Types and Literals

- Different types of numbers are stored and manipulated in different ways.
- Python uses *data types* to categorize values in memory.
- When an integer is stored in memory, it is classified as an int, and when a real number is stored in memory, it is classified as a float.

Examples

```
room = 503
```

- This statement causes the value 503 to be stored in memory, and it makes the room variable reference it.

```
dollars = 2.75
```

- This statement causes the value 2.75 to be stored in memory, and it makes the dollars variable reference it.
- A number that is written into a program's code is called a *numeric literal.*
- When the Python interpreter reads a numeric literal in a program's code, it determines its data type according to the following rules:
  - A numeric literal that is written as a whole number with no decimal point is considered an int. Examples are 7, 124, and −9.
  - A numeric literal that is written with a decimal point is considered a float. Examples are 1.5, 3.14159, and 5.0.

type() function

- you can use the built-in type function in interactive mode to determine the data type of a value.
- For example, look at the following session:

```
>>> type(1)
<class 'int'>
>>>
```

- In this example, the value 1 is passed as an argument to the type function.
- The message that is displayed on the next line, <class 'int'>, indicates that the value is an int.
- Here is another example:

```
>>> type(1.0)
<class 'float'>
>>>
```

- indicates that the value is a float

Storing Strings with the str Data Type

- Python also has a data type named str, which is used for storing strings in memory

```
# Create variables to reference two strings.
        first_name = 'Kathryn'
        last_name = 'Marino'
# Display the values referenced by the variables.
        print(first_name, last_name)
```

O/p

```
        Kathryn Marino
```

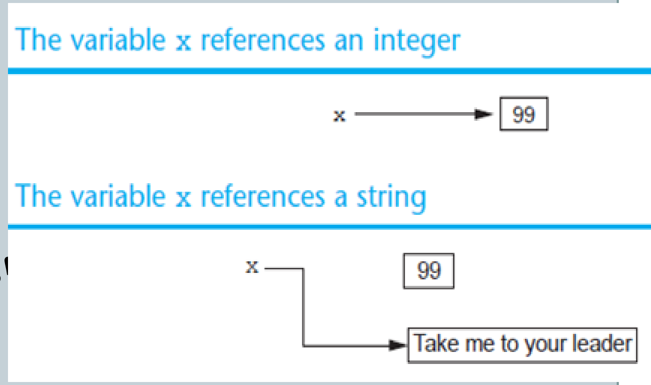
Reassigning a Variable to a Different Type

- A variable is just a name that refers to a piece of data in memory.
- It is a mechanism that makes it easy for the programmer, to store and retrieve data.
- Internally, the Python interpreter keeps track of the variable names that you create and the pieces of data to which those variable names refer.
- Any time you need to retrieve one of those pieces of data, you simply use the variable name that refers to it.

Reassigning a Variable to a Different Type

- A variable in Python can refer to items of any type.
- After a variable has been assigned an item of one type, it can be reassigned an item of a different type.
- To demonstrate, look at the following interactive session. (We have added line numbers for easier reference.)

```
1 >>> x = 99
2 >>> print(x)
3 99
4 >>> x = 'Take me to your leader'
5 >>> print(x)
6 Take me to your leader.
7 >>>
```


The variable x references an integer
x
99
The variable x references a string
x
99
Take me to your leader

Reading Input from the Keyboard

- When a program reads data from the keyboard, usually it stores that data in a variable so it can be used later by the program.
- Python's built-in input function to read input from the keyboard
- The input function reads a piece of data that has been entered at the keyboard and returns that piece of data, as a string, back to the program.

  variable = input(*prompt*)

- *prompt* is a string that is displayed on the screen.
- The string's purpose is to instruct the user to enter a value;
- *variable* is the name of a *variable* that references the data that was entered on the keyboard.

Example

```
name = input('What is your name? ')
```

- When this statement executes, the following things happen:
- The string 'What is your name? ' is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard and then to press the Enter key.
- When the Enter key is pressed, the data that was typed is returned as a string and assigned to the name variable.

Example

```
# Get the user's first name.
        first_name = input('Enter your first name: ')
# Get the user's last name.
        last_name = input('Enter your last name: ')
# Print a greeting to the user.
        print('Hello', first_name, last_name)
```

O/p (with input shown in bold)

```
        Enter your first name: CBIT
        Enter your last name: MGIT
        Hello CBIT MGIT
```

Reading Numbers with the input Function

- The input function always returns the user's input as a string, even if the user enters numeric data.
- Python has built-in functions that you can use to convert a string to a numeric type.

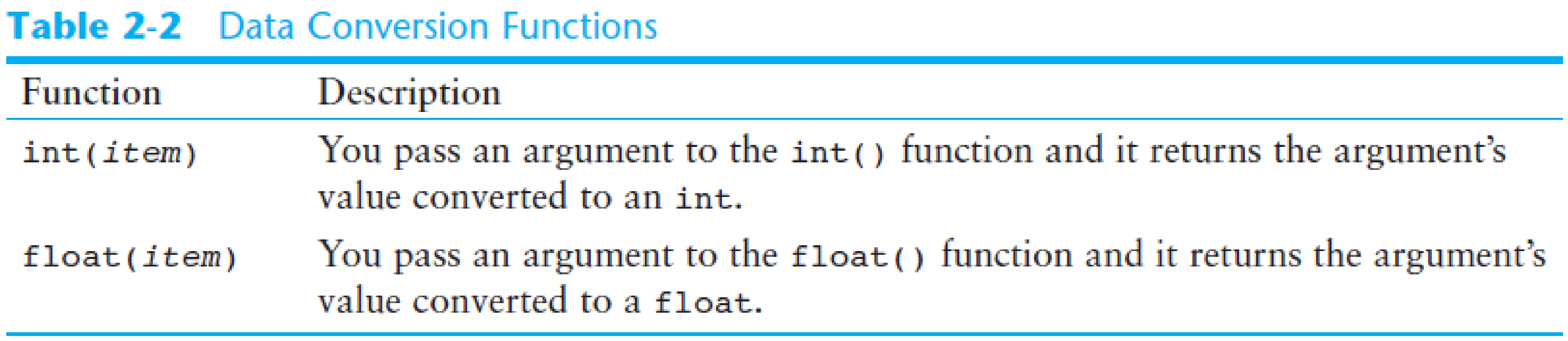


Table 2-2 Data Conversion Functions

Function	Description
`int(`*`item`*`)`	You pass an argument to the `int()` function and it returns the argument's value converted to an `int`.
`float(`*`item`*`)`	You pass an argument to the `float()` function and it returns the argument's value converted to a `float`.

Example

```
string_value = input('How many hours did you work? ')
hours = int(string_value)
```

- The first statement gets the number of hours from the user and assigns that value as a string to the string_value variable.
- The second statement calls the int() function, passing string_value as an argument.
- The value referenced by string_value is converted to an int and assigned to the hours variable.
- Above is inefficient because it creates two variables:
- one to hold the string that is returned from the input function and another to hold the integer that is returned from the int() function.

nested function calls - Example

```
hours = int(input('How many hours did you work? '))
```

- This one statement uses *nested function* calls. The value that is returned from the input
- function is passed as an argument to the int() function. This is how it works:
- It calls the input function to get a value entered at the keyboard.
- The value that is returned from the input function (a string) is passed as an argument to the int() function.
- The int value that is returned from the int() function is assigned to the hours variable.

Example

```
pay_rate = float(input('What is your hourly pay rate? '))
```

- This is how it works:
- It calls the input function to get a value entered at the keyboard.
- The value that is returned from the input function (a string) is passed as an argument to the float() function.
- The float value that is returned from the float() function is assigned to the pay_rate variable.

Performing Calculations

- Python has numerous operators that can be used to perform mathematical calculations.
- Programmers use the operators to create math expressions.
- A *math expression* performs a calculation and gives a value

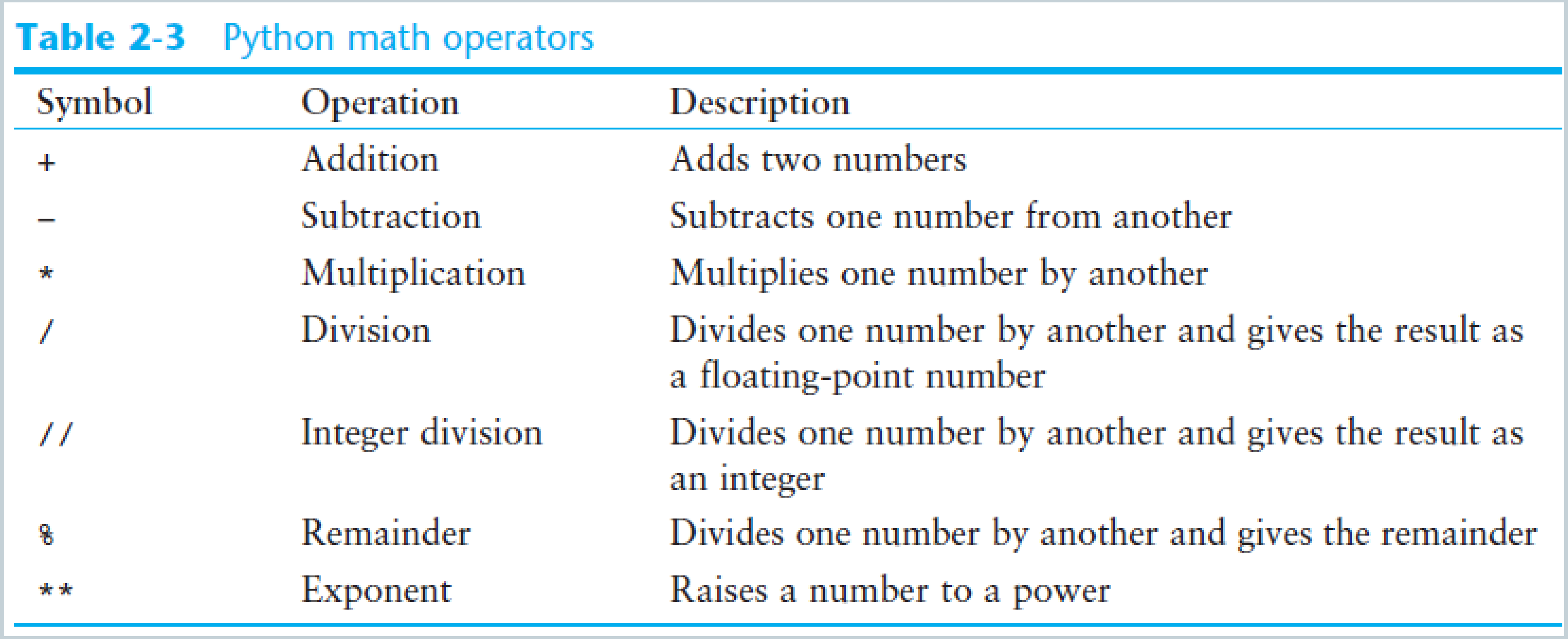


Table 2-3 Python math operators

Symbol	Operation	Description
+	Addition	Adds two numbers
–	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the result as a floating-point number
//	Integer division	Divides one number by another and gives the result as an integer
%	Remainder	Divides one number by another and gives the remainder
**	Exponent	Raises a number to a power

Examples…

- # Assign a value to the salary variable.
- salary = 2500.0

- # Assign a value to the bonus variable.
- bonus = 1200.0

- # Calculate the total pay by adding salary
- # and bonus. Assign the result to pay.

- pay = salary + bonus

- # Display the pay.
- print('Your pay is', pay)

O/p

- Your pay is 3700.0

Calculating percentage –
(Sales price – discount – algorithm)

- 1. *Get the original price of the item.*
- original_price = float(input("Enter the item's original price: "))
- 2. *Calculate 20 percent of the original price. This is the amount of the discount.*
- discount = original_price * 0.2
- 3. *Subtract the discount from the original price. This is the sale price.*
- sale_price = original_price – discount
- 4. *Display the sale price.*
- print('The sale price is', sale_price)

Sales price – discount – Program

- # This program gets an item's original price and
- # calculates its sale price, with a 20% discount.
- # Get the item's original price.
- original_price = float(input("Enter the item's original price: "))

- # Calculate the amount of the discount.
- discount = original_price * 0.2

- # Calculate the sale price.
- sale_price = original_price – discount

- # Display the sale price.
- print('The sale price is', sale_price)

- **O/p** (with input shown in bold)
- Enter the item's original price: **100.00**
- The sale price is 80.0

Floating-Point and Integer Division

- Python has two different division operators.
- The / operator performs floating-point division, and
- The // operator performs integer division.
- Both operators divide one number by another.
- The difference between them is that the / operator gives the result as a floating-point value, and the // operator gives the result as an integer.
- >>> 5 / 2
- 2.5
- >>>

Division - Example

- >>> 5 // 2
- 2
- >>>

- When the result is positive, it is *truncated*, which means that its fractional part is thrown away.
- When the result is negative, it is rounded *away from zero* to the nearest integer.

- >>> 5 / 2
- 2.5
- >>>

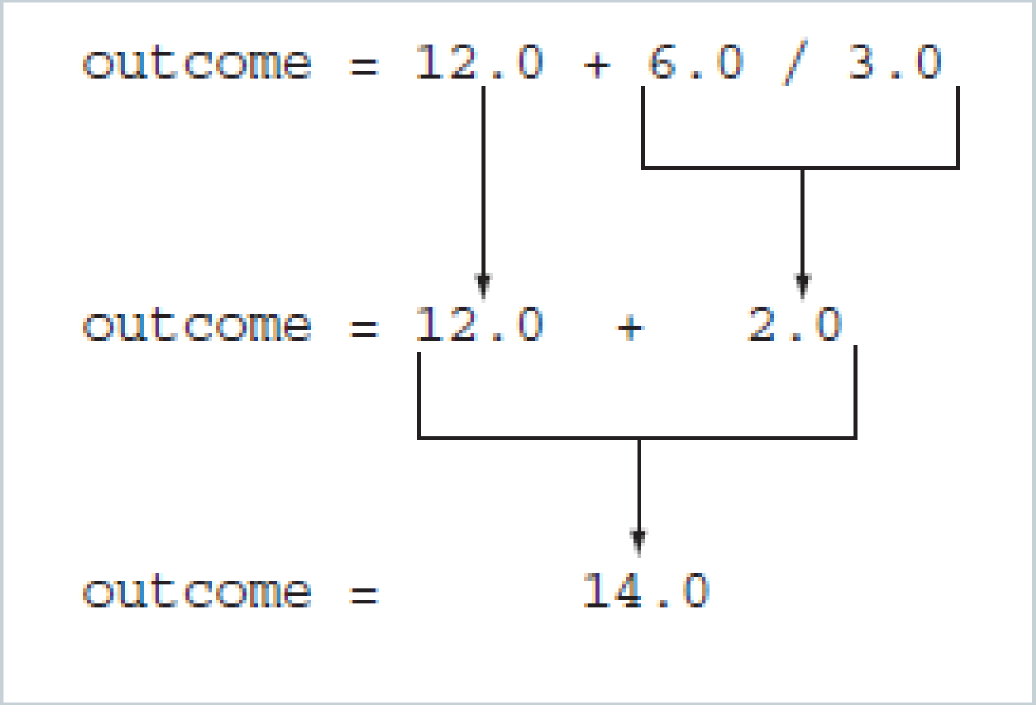
- >>> −5 // 2
- −3
- >>>

Operator Precedence

- First, operations that are enclosed in parentheses are performed first.
- Then, when two operators share an operand, the operator with the higher *precedence* is applied first.
- The precedence of the math operators, from highest to lowest, are:
  - Exponentiation: **
  - Multiplication, division, and remainder: * / // %
  - Addition and subtraction: + –
- There is an exception to the left-to-right rule.
- When two ** operators share an operand, the operators execute right-to-left.
- For example, the expression 2**3**4 is evaluated as 2**(3**4).

Example

- outcome = 12.0 + 6.0 / 3.0



Grouping with Parentheses

- Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others.
- In the following statement, the variables a and b are added together, and their sum is divided by 4:

  ```
  result = (a + b) / 4
  ```

- Without the parentheses, however, b would be divided by 4 and the result added to a.

To calculate average

- # Get three test scores and assign them to the
- # test1, test2, and test3 variables.
- test1 = float(input('Enter the first test score: '))
- test2 = float(input('Enter the second test score: '))
- test3 = float(input('Enter the third test score: '))

- # Calculate the average of the three scores
- # and assign the result to the average variable.
- average = (test1 + test2 + test3) / 3.0

- # Display the average.
- print('The average score is', average)

The Exponent Operator

- Two asterisks written together (**) is the exponent operator, and its purpose it to raise a number to a power.
- area = length**2
- >>> 4**2

O/p 16

- >>> 5**3

O/p 125

- >>> 2**10

O/p 1024

- >>>

The Remainder Operator

- In Python, the % symbol is the remainder operator. (This is also known as the *modulus operator*.)

- The remainder operator performs division, but instead of returning the quotient, it returns the remainder.

Example

- # Get a number of seconds from the user.
- total_seconds = float(input('Enter a number of seconds: '))

- # Get the number of hours.
- hours = total_seconds // 3600

- # Get the number of remaining minutes.
- minutes = (total_seconds // 60) % 60

- # Get the number of remaining seconds.
- seconds = total_seconds % 60

- # Display the results.
- print('Here is the time in hours, minutes, and seconds:')
- print('Hours:', hours)
- print('Minutes:', minutes)
- print('Seconds:', seconds)

O/p (with input shown in bold)

```
Enter a number of seconds: 11730
Here is the time in hours, minutes, and
seconds:
Hours: 3.0
Minutes: 15.0
Seconds: 30.0
```

Converting Math Formulas to Programming Statements

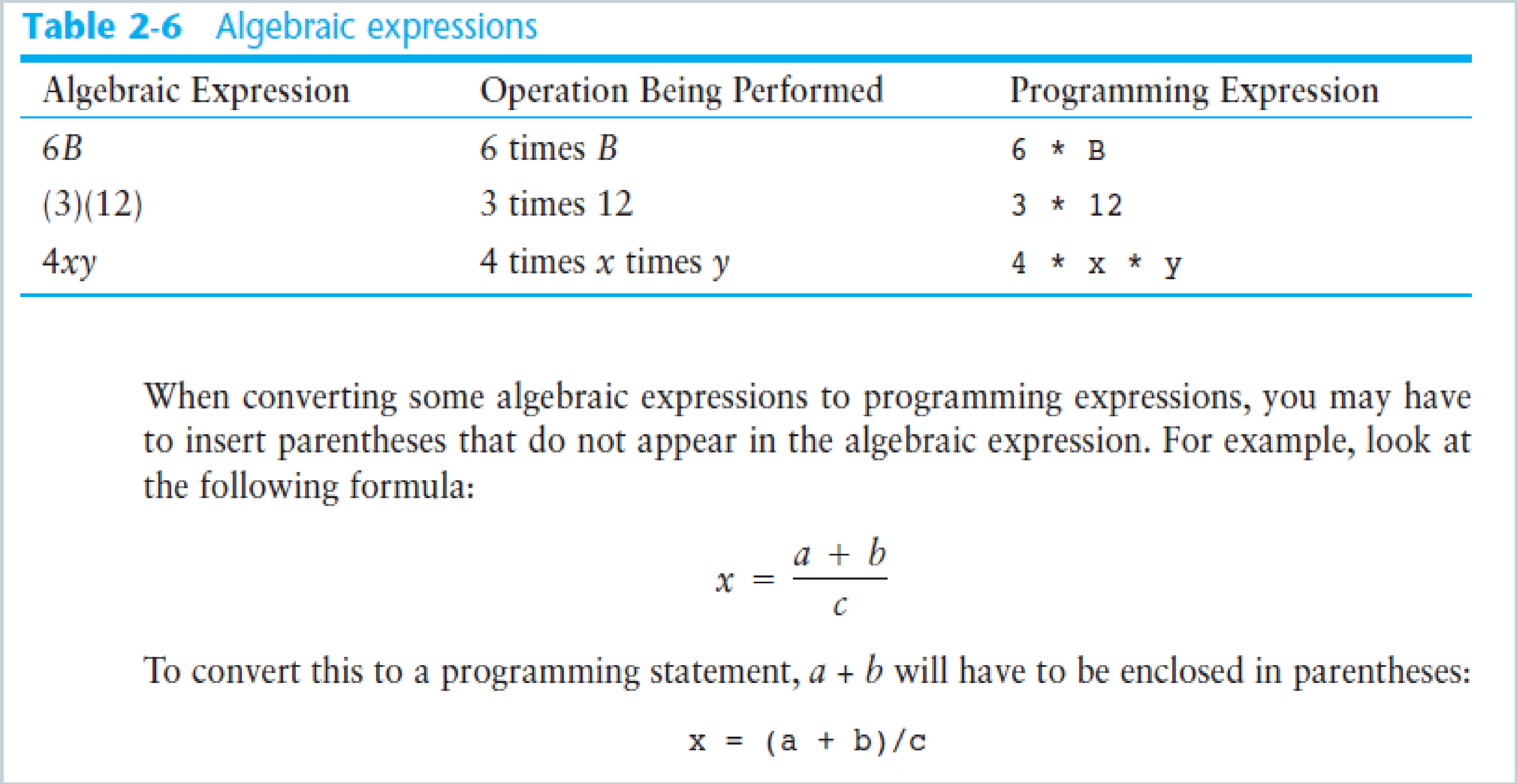


Table 2-6 Algebraic expressions

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	`6 * B`
(3)(12)	3 times 12	`3 * 12`
$4xy$	4 times x times y	`4 * x * y`

When converting some algebraic expressions to programming expressions, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following formula:

$$x = \frac{a + b}{c}$$

To convert this to a programming statement, $a + b$ will have to be enclosed in parentheses:

```
x = (a + b)/c
```

Converting a Math Formula to a Programming Statement

Suppose you want to deposit a certain amount of money into a savings account and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have \$10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the present value, or the amount that you need to deposit today.
- F is the future value that you want in the account. (In this case, F is \$10,000.)
- r is the annual interest rate.
- n is the number of years that you plan to let the money sit in the account.

It would be convenient to write a computer program to perform the calculation because then we can experiment with different values for the variables. Here is an algorithm that we can use:

1. *Get the desired future value.*
2. *Get the annual interest rate.*
3. *Get the number of years that the money will sit in the account.*
4. *Calculate the amount that will have to be deposited.*
5. *Display the result of the calculation in step 4.*

Example

```python
# Get the desired future value.
future_value = float(input('Enter the desired future value: '))

# Get the annual interest rate.
rate = float(input('Enter the annual interest rate: '))

# Get the number of years that the money will appreciate.
years = int(input('Enter the number of years the money will grow: '))

# Calculate the amount needed to deposit.
present_value = future_value / (1.0 + rate)**years

# Display the amount needed to deposit.
print('You will need to deposit this amount:', present_value)
```

Program Output

```
Enter the desired future value: 10000.0 Enter
Enter the annual interest rate: 0.05 Enter
Enter the number of years the money will grow: 10 Enter
You will need to deposit this amount: 6139.13253541
```

Mixed-Type Expressions and Data Type Conversion

- When you perform a math operation on two operands, the data type of the result will depend on the data type of the operands.

- Python follows these rules when evaluating mathematical expressions:
  - When an operation is performed on two int values, the result will be an int.
  - When an operation is performed on two float values, the result will be a float.
  - When an operation is performed on an int and a float, the int value will be temporarily converted to a float and the result of the operation will be a float.
  - (An expression that uses operands of different data types is called a *mixed-type expression*.)

Examples

```
my_number = 5 * 2.0
```

- When this statement executes, the value 5 will be converted to a float (5.0) and then multiplied by 2.0.
- The result, 10.0, will be assigned to my_number.
- The int to float conversion that takes place in the previous statement happens implicitly.
- If you need to explicitly perform a conversion, you can use either the int() or float() functions.
- For example, you can use the int() function to convert a floating-point value to an integer, as shown in the following code:

```
fvalue = 2.6
ivalue = int(fvalue)            - ivalue=2

fvalue = −2.9
ivalue = int(fvalue)            - ivalue=-2

ivalue = 2
fvalue = float(ivalue)          - fvalue=2.0
```

Breaking Long Statements into Multiple Lines

- Most programming statements are written on one line.
- If a programming statement is too long, however, you will not be able to view all of it in your editor window without scrolling horizontally.
- Python allows you to break a statement into multiple lines by using the *line continuation character*, which is a backslash (\)
- Here is a print function call that is broken into two lines with the line continuation character:

```
print('We sold', units_sold, \
        'for a total of', sales_amount)
result = var1 * 2 + var2 * 3 + \
        var3 * 4 + var4 * 5

print("Monday's sales are", monday, \
        "and Tuesday's sales are", tuesday, \
        "and Wednesday's sales are", wednesday)
```

More About Data Output:

Suppressing the print Function's Ending Newline

- The print function normally displays a line of output. For example, the following three
- statements will produce three lines of output:

```
print('One')
print('Two')
print('Three')
```

- If you do not want the print function to start a new line of output when it finishes displaying its output, you can pass the special argument end=' ' to the function, as shown in the following code:

```
print('One', end=' ')
print('Two', end=' ')
print('Three')
```

O/p: One Two Three

- If that is the case, you can pass the argument end='' to the print function, as shown in the following code:

```
print('One', end='')
print('Two', end='')
print('Three')
```

O/p: OneTwoThree

Specifying an Item Separator

- >>> print('One', 'Two', 'Three')
- One Two Three
- >>>

```
>>> print('One', 'Two', 'Three',
sep='*')
One*Two*Three
>>>
```

- >>> print('One', 'Two', 'Three', sep='')
- OneTwoThree
- >>>

```
>>> print('One', 'Two', 'Three',
sep='~~~')
One~~~Two~~~Three
>>>
```

Escape Characters

- An *escape character is a special character that is preceded with a backslash (\), appearing* inside a string literal.
- When a string literal that contains escape characters is printed, the escape characters are treated as special commands that are embedded in the string.

```
print('One\nTwo\nThree')
```

- When this statement executes, it displays

```
One
Two
Three
```

Escape Sequences

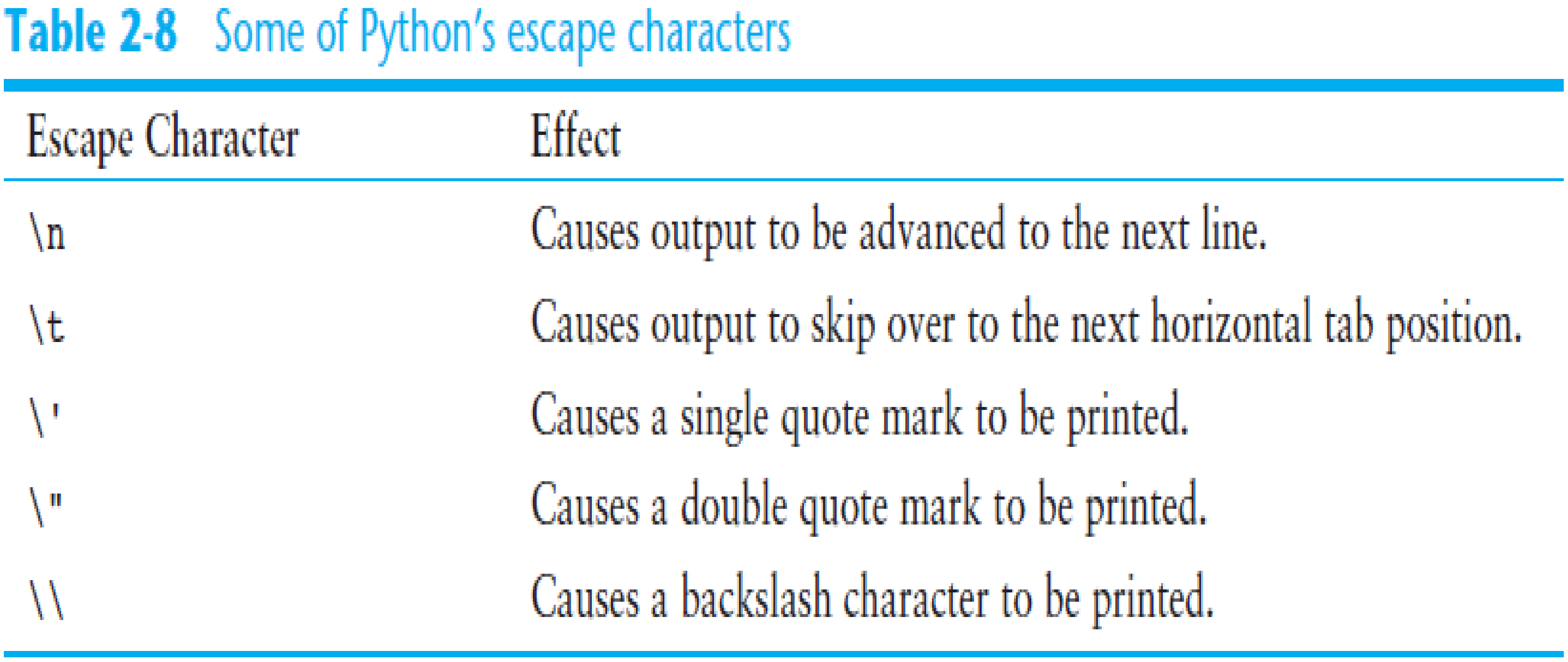


Table 2-8 Some of Python's escape characters

Escape Character	Effect
`\n`	Causes output to be advanced to the next line.
`\t`	Causes output to skip over to the next horizontal tab position.
`\'`	Causes a single quote mark to be printed.
`\"`	Causes a double quote mark to be printed.
`\\`	Causes a backslash character to be printed.

Examples

- print('Mon\tTues\tWed')
- print('Thur\tFri\tSat')

O/p

```
Mon    Tues    Wed
Thur   Fri     Sat
```

- print("Your assignment is to read \"Hamlet\" by tomorrow.")
- print('I\'m ready to begin.')

O/p

```
Your assignment is to read "Hamlet" by tomorrow.
I'm ready to begin.
```

- You can use the \\ escape character to display a backslash, as shown in the following:
- print('The path is C:\\temp\\data.')

O/p

```
The path is C:\temp\data.
```

Displaying Multiple Items with the + Operator

- When the + operator is used with two strings, however, it performs *string concatenation*.
- *This means* that it appends one string to another.

```
print('This is ' + 'one string.')
```

- This statement will print

```
This is one string.
```

```
print('Enter the amount of ' + \
      'sales for each day and ' + \
      'press Enter.')
```

- This statement will display the following:

```
Enter the amount of sales for each day and press Enter
```

Formatting Numbers

- When a floating-point number is displayed by the print function, it can appear with up to 12 significant digits.

```
# This program demonstrates how a floating-point
# number is displayed with no formatting.
amount_due = 5000.0
monthly_payment = amount_due / 12.0
 print('The monthly payment is', monthly_payment)
```

O/p

- The monthly payment is 416.666666667

Example

- When you call the built-in format function, you pass two arguments to the function: a numeric value and a format specifier.
- The *format specifier is a string that contains special* characters specifying how the numeric value should be formatted
- format(12345.6789, '.2f')
- The first argument, which is the floating-point number 12345.6789, is the number that we want to format.
- The second argument, which is the string '.2f', is the format specifier.
- Here is the meaning of its contents:
  - The .2 specifies the precision. It indicates that we want to round the number to two decimal places.
  - The f specifies that the data type of the number we are formatting is a floating-point number. (If you are formatting an integer, you cannot use f for the type.)

```
>>> print(format(12345.6789, '.2f'))
12345.68
>>>
```

Formatting in Scientific Notation

- If you prefer to display floating-point numbers in scientific notation, you can use the letter e or the letter E instead of f.
- Here are some examples:

```
>>> print(format(12345.6789, 'e'))
1.234568e+04

>>> print(format(12345.6789, '.2E'))
1.23E+04
>>>
```

Inserting Comma Separators

- If you want the number to be formatted with comma separators, you can insert a comma into the format specifier, as shown here:

```
>>> print(format(12345.6789, ',.2f'))
12,345.68
>>>
```

- Here is an example that formats an even larger number:

```
>>> print(format(123456789.456, ',.2f'))
123,456,789.46
>>>
```

- Notice that in the format specifier the comma is written before (to the left of) the precision designator.
- Here is an example that specifies the comma separator but does not specify precision:

```
>>> print(format(12345.6789, ',f'))
12,345.678900
>>>
```

Specifying a Minimum Field Width

- The format specifier can also include a minimum field width, which is the minimum number of spaces that should be used to display the value.
- The following example prints a number in a field that is 12 spaces wide:

```
>>> print('The number is', format(12345.6789, '12,.2f'))
The number is 12,345.68
>>>
```

- In this example, the 12 that appears in the format specifier indicates that the number should be displayed in a field that is a minimum of 12 spaces wide.
- In this case, the number that is displayed is shorter than the field that it is displayed in.
- The number 12,345.68 uses only 9 spaces on the screen, but it is displayed in a field that is 12 spaces wide.
- When this is the case, the number is right justified in the field.
- If a value is too large to fit in the specified field width, the field is automatically enlarged to accommodate it.

Examples

Program

- # This program displays the following
- # floating-point numbers in a column
- # with their decimal points aligned.
- num1 = 127.899
- num2 = 3465.148
- num3 = 3.776
- num4 = 264.821
- num5 = 88.081
- num6 = 799.999

- Display each number in a field of 7 spaces
- with 2 decimal places.
- print(format(num1, '7.2f'))
- print(format(num2, '7.2f'))
- print(format(num3, '7.2f'))
- print(format(num4, '7.2f'))
- print(format(num5, '7.2f'))
- print(format(num6, '7.2f'))

Output

```
127.90
3465.15
3.78
264.82
88.08
800.00
```

Formatting a Floating-Point Number as a Percentage

- Instead of using f as the type designator, you can use the % symbol to format a floating point number as a percentage.
- The % symbol causes the number to be multiplied by 100 and displayed with a % sign following it.
- Here is an example:

```
>>> print(format(0.5, '%'))
50.000000%
>>>
```

- Here is an example that specifies 0 as the precision:

```
>>> print(format(0.5, '.0%'))
50%
>>>
```

Formatting Integers

- There are two differences to keep in mind when writing a format specifier that will be used to format an integer
  - You use d as the type designator.
  - You cannot specify precision.

```
>>> print(format(123456, 'd'))
123456


>>> print(format(123456, ',d'))
123,456

>>> print(format(123456, '10d'))
    123456

>>> print(format(123456, '10,d'))
   123,456
```

Unit # 1

DECISION STRUCTURES AND BOOLEAN LOGIC

The if Statement

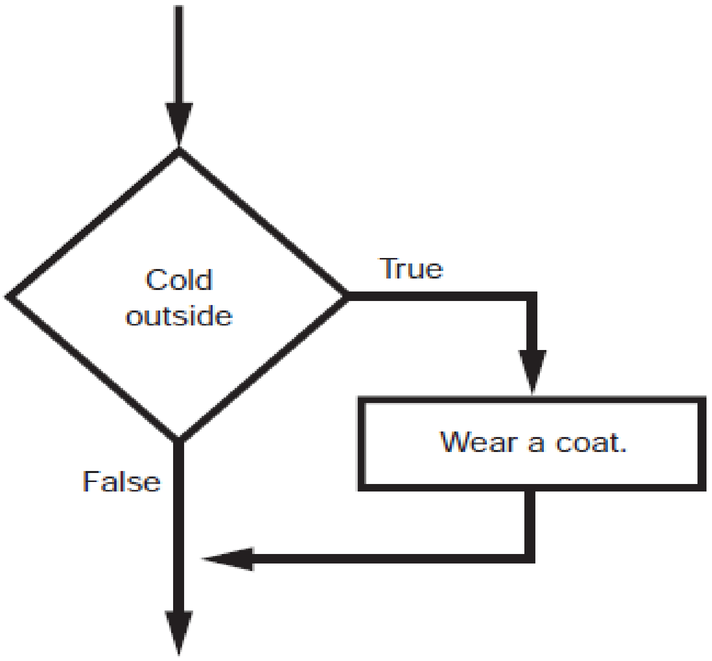
- The if statement is used to create a decision structure, which allows a program to have more than one path of execution.
- The if statement causes one or more statements to execute only when a Boolean expression is true.
- A ***control structure*** is a logical design that controls the order in which a set of statements execute.
- A ***sequence structure*** is a set of statements that execute in the order that they appear.

Decision Structures

- Can execute a set of statements only under certain circumstances.
- This can be accomplished with a *decision structure*.
- *Decision structures are also known as selection structures*.
- In a decision structure's simplest form, a specific action is performed only if a certain condition exists.
- If the condition does not exist, the action is not performed.
- If the condition is true, we follow one path, which leads to an action being performed.
- If the condition is false, we follow another path, which skips the action.

Example Decision Structure - *single alternative decision structure*

3-1 A simple decision structure



In the flowchart, the diamond symbol indicates some condition that must be tested. In this case, we are determining whether the condition `Cold outside` is true or false. If this condition is true, the action `Wear a coat` is performed. If the condition is false, the action is skipped. The action is *conditionally executed* because it is performed only when a certain condition is true.

General Format

- In Python we use the if statement to write a single alternative decision structure.
- Here is the general format of the if statement:

```
if condition:
        statement
        statement
        etc.
```

if ….explained

- For simplicity, we will refer to the first line as the *if clause*.
- *The if clause begins with* the word if, followed by a *condition, which is an expression that will be evaluated* as either true or false.
- A colon appears after the *condition*.
- *Beginning at the next line* is a block of statements.
- A *block is simply a set of statements that belong together as a* group.
- Notice in the general format that all of the statements in the block are indented.
- This indentation is required because the Python interpreter uses it to tell where the block begins and ends.
- When the if statement executes, the *condition is tested*.
- *If the condition is true, the* statements that appear in the block following the if clause are executed.
- If the condition is false, the statements in the block are skipped

Boolean Expressions and Relational Operators

- Boolean expression that is tested by an if statement is formed with a relational operator.
- A *relational operator determines whether a specific relationship exists* between two values.

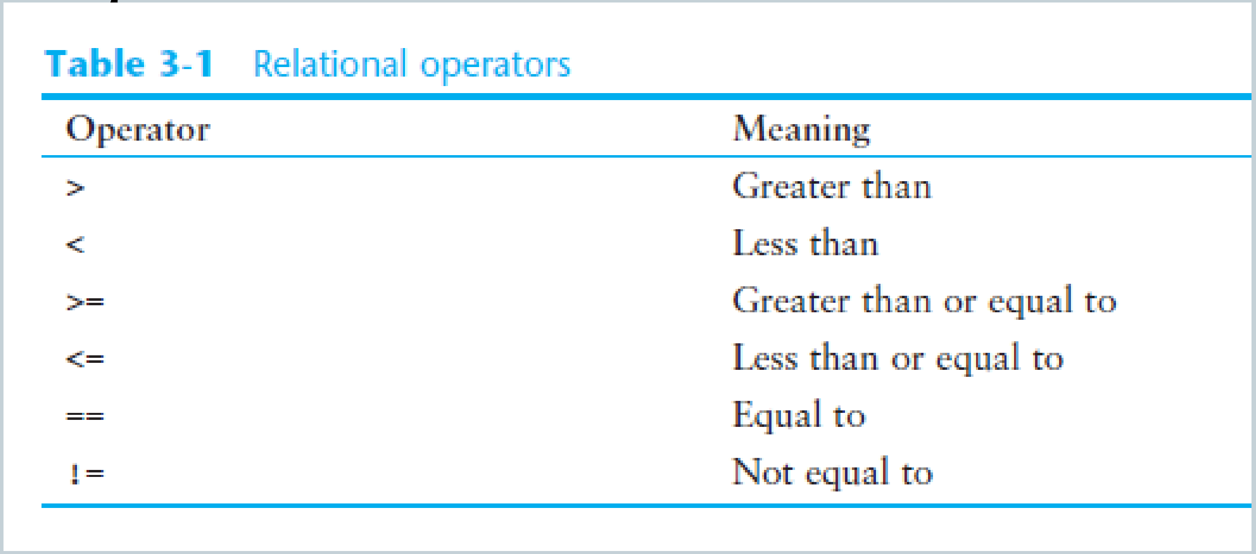


Table 3-1 Relational operators

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Example

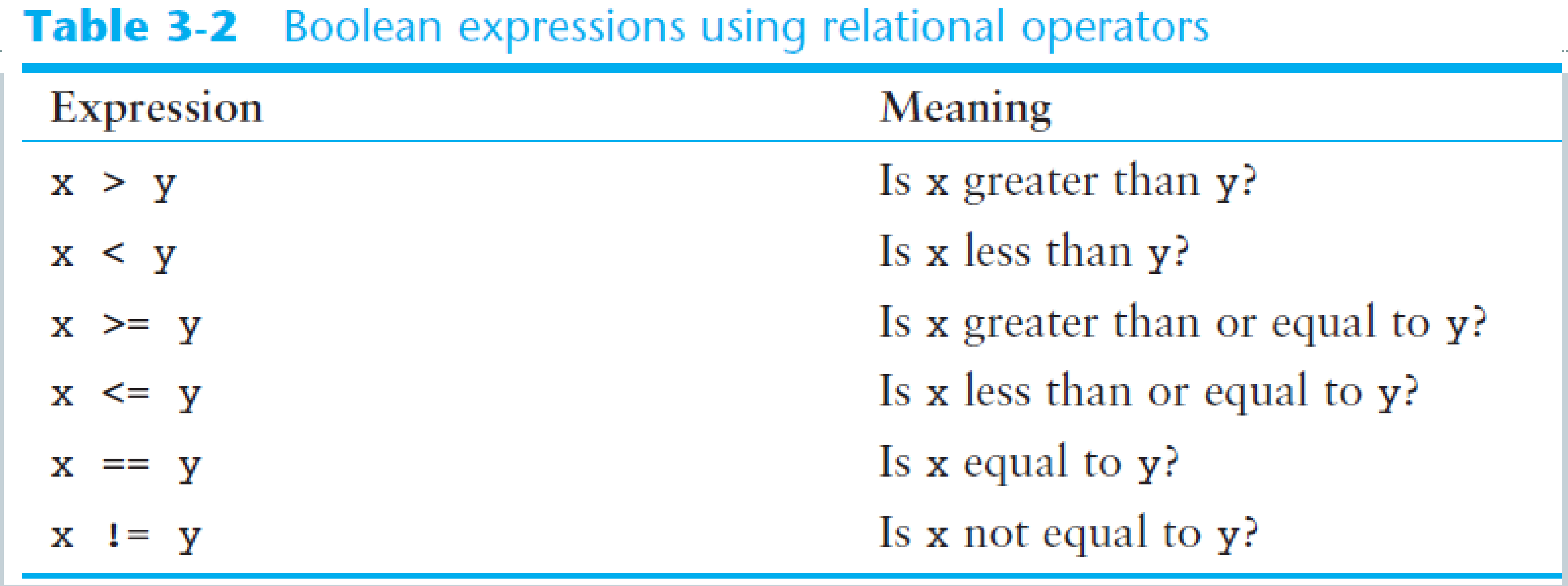


Table 3-2 Boolean expressions using relational operators

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y?
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

```
>>> x = 1
>>> y = 0
>>> y < x
True
>>> x < y
False
>>>
```

The >=, <=, == and != Operators

- Two of the operators, >= and <=, test for more than one relationship.
- The >= operator determines whether the operand on its left is greater than *or equal to the operand on its* right.
- The <= operator determines whether the operand on its left is less than *or equal to* the operand on its right
- The == operator determines whether the operand on its left is equal to the operand on its right.
- Note: The equality operator is two = symbols together. Don't confuse this operator with the assignment operator, which is one = symbol.
- The != operator is the not-equal-to operator.
- It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the == operator

Example

```
# This program gets three test scores and displays
# their average. It congratulates the user if the
# average is a high score.

# The high score variable holds the value that is
# considered a high score.
high_score = 95

# Get the three test scores.
test1 = int(input('Enter the score for test 1:' ))
test2 = int(input('Enter the score for test 2:' ))
test3 = int(input('Enter the score for test 3:' ))
```

Program Output (with input shown in bold)

```
Enter the score for test 1: 82
Enter the score for test 2: 76
Enter the score for test 3: 91
The average score is 83.0
```

```
# Calculate the average test score.
average = (test1 + test2 + test3) / 3

# Print the average.
print('The average score is', average)

# If the average is a high score,
# congratulate the user.
if average >= high_score:
        print('Congratulations!')
        print('That is a great average!')
```

Program Output (with input shown in bold)

```
Enter the score for test 1: 93
Enter the score for test 2: 99
Enter the score for test 3: 96
The average score is 96.0
Congratulations!
That is a great average!
```

The if-else Statement

- An if-else statement will execute one block of statements if its condition is true, or another block if its condition is false.
- *Dual alternative decision structure, which has two possible paths of execution—one path is taken if a condition* is true, and the other path is taken if the condition is false

General format

- In code we write a dual alternative decision structure as an if-else statement. Here is the
- general format of the if-else statement:

```
if condition:
        statement
        statement
        etc.
else:
        statement
        statement
        etc.
```

- When this statement executes, the *condition is tested.*
- *If it is true, the block of indented statements* following the if clause is executed, and then control of the program jumps to the statement that follows the if-else statement.
- If the condition is false, the block of indented statements following the else clause is executed, and then control of the program jumps General Format to the statement that follows the if-else statement.

General Format

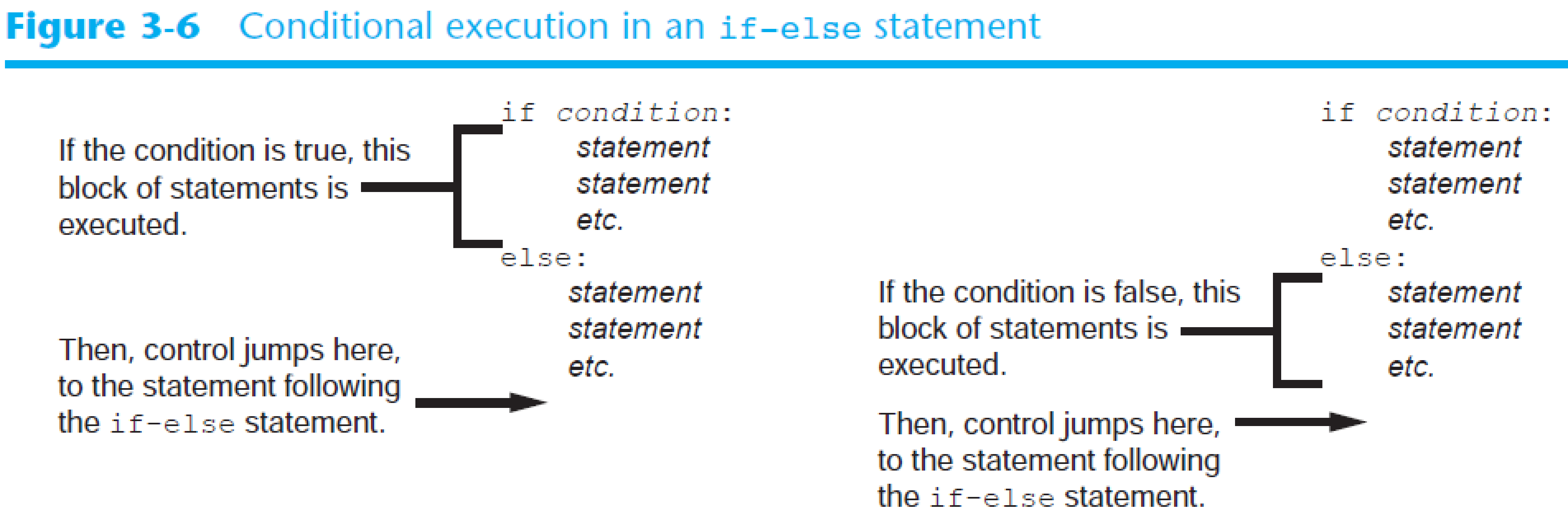


Figure 3-6 Conditional execution in an if-else statement
If the condition is true, this block of statements is executed.
Then, control jumps here, to the statement following the if-else statement.
If the condition is false, this block of statements is executed.
Then, control jumps here, to the statement following the if-else statement.

```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

- **Indentation in the if-else Statement**
- When you write an if-else statement, follow these guidelines for indentation:
  - Make sure the if clause and the else clause are aligned.
  - The if clause and the else clause are each followed by a block of statements. Make sure the statements in the blocks are consistently indented

Comparing Strings

- Python allows you to compare strings
- This allows you to create decision structures that test the value of a string.

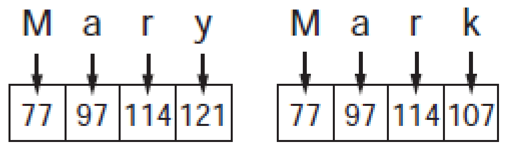
```
name1 = 'Mary'
name2 = 'Mark'
if name1 == name2:
        print('The names are the same.')
else:
        print('The names are NOT the same.')
if month != 'October':
        print('This is the wrong time for Octoberfest!')
```

Other String Comparisons

- ASCII (the American Standard Code for Information Interchange) is a commonly used character coding system
- The uppercase characters A through Z are represented by the numbers 65 through 90.
- The lowercase characters a through z are represented by the numbers 97 through 122.
- When the digits 0 through 9 are stored in memory as characters, they are represented by the numbers 48 through 57.
- (For example, the string 'abc123' would be stored in memory as the codes 97, 98, 99, 49, 50, and 51.)
- A blank space is represented by the number 32.

```
if 'a' < 'b':
        print('The letter a is less than the letter b.')
```

Character codes for the strings 'Mary' and 'Mark'

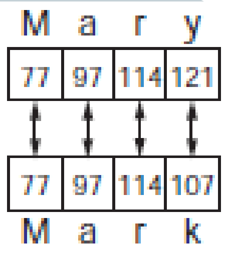


When you use relational operators to compare these strings, the strings are compared character-by-character. For example, look at the following code:

```
name1 = 'Mary'
name2 = 'Mark'
if name1 > name2:
    print('Mary is greater than Mark')
else:
    print('Mary is not greater than Mark')
```

The > operator compares each character in the strings `'Mary'` and `'Mark'`, beginning with the first, or leftmost, characters. This is shown in Figure 3-9.

Comparing each character in a string



Here is how the comparison takes place:

1. The `'M'` in `'Mary'` is compared with the `'M'` in `'Mark'`. Since these are the same, the next characters are compared.
2. The `'a'` in `'Mary'` is compared with the `'a'` in `'Mark'`. Since these are the same, the next characters are compared.
3. The `'r'` in `'Mary'` is compared with the `'r'` in `'Mark'`. Since these are the same, the next characters are compared.
4. The `'y'` in `'Mary'` is compared with the `'k'` in `'Mark'`. Since these are not the same, the two strings are not equal. The character `'y'` has a higher ASCII code (121) than `'k'` (107), so it is determined that the string `'Mary'` is greater than the string `'Mark'`.

Examples

- If one of the strings in a comparison is shorter than the other, only the corresponding characters will be compared.
- If the corresponding characters are identical, then the shorter string is considered less than the longer string.
- For example, suppose the strings 'High‘ and 'Hi' were being compared.
- The string 'Hi' would be considered less than 'High‘ Explaiendbecause it is shorter.

Sort Names Example

```
# This program compares strings with the < operator.
# Get two names from the user.
name1 = input('Enter a name (last name first):')
name2 = input('Enter another name (last name first):')

# Display the names in alphabetical order.
 print('Here are the names, listed alphabetically.')

if  name1 < name2:
            print(name1)
            print(name2)
else:
            print(name2)
            print(name1)
```

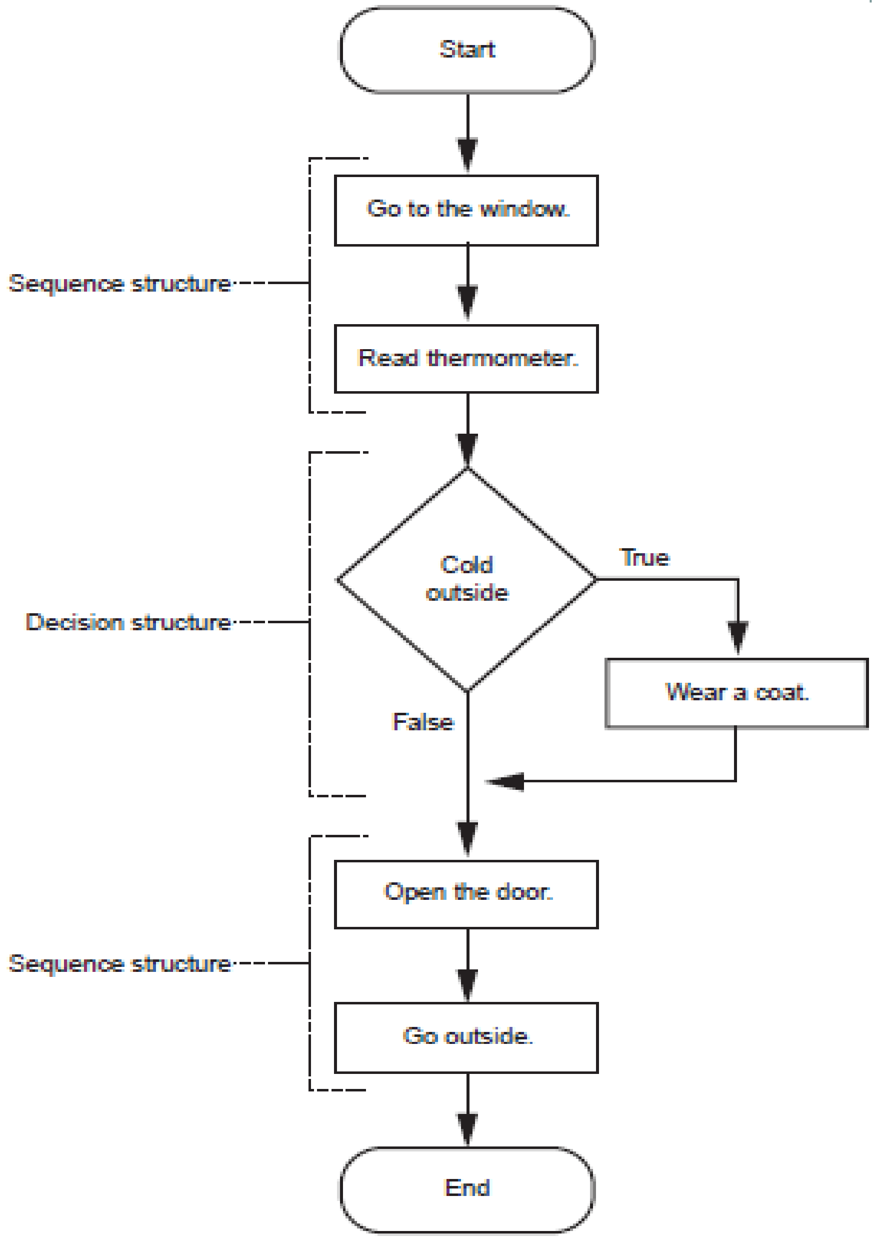
- **Program Output (with input shown in bold)**

```
Enter a name (last name first): Jones, Richard
Enter another name (last name first) Costa, Joan
Here are the names, listed alphabetically:
Costa, Joan
Jones, Richard
```

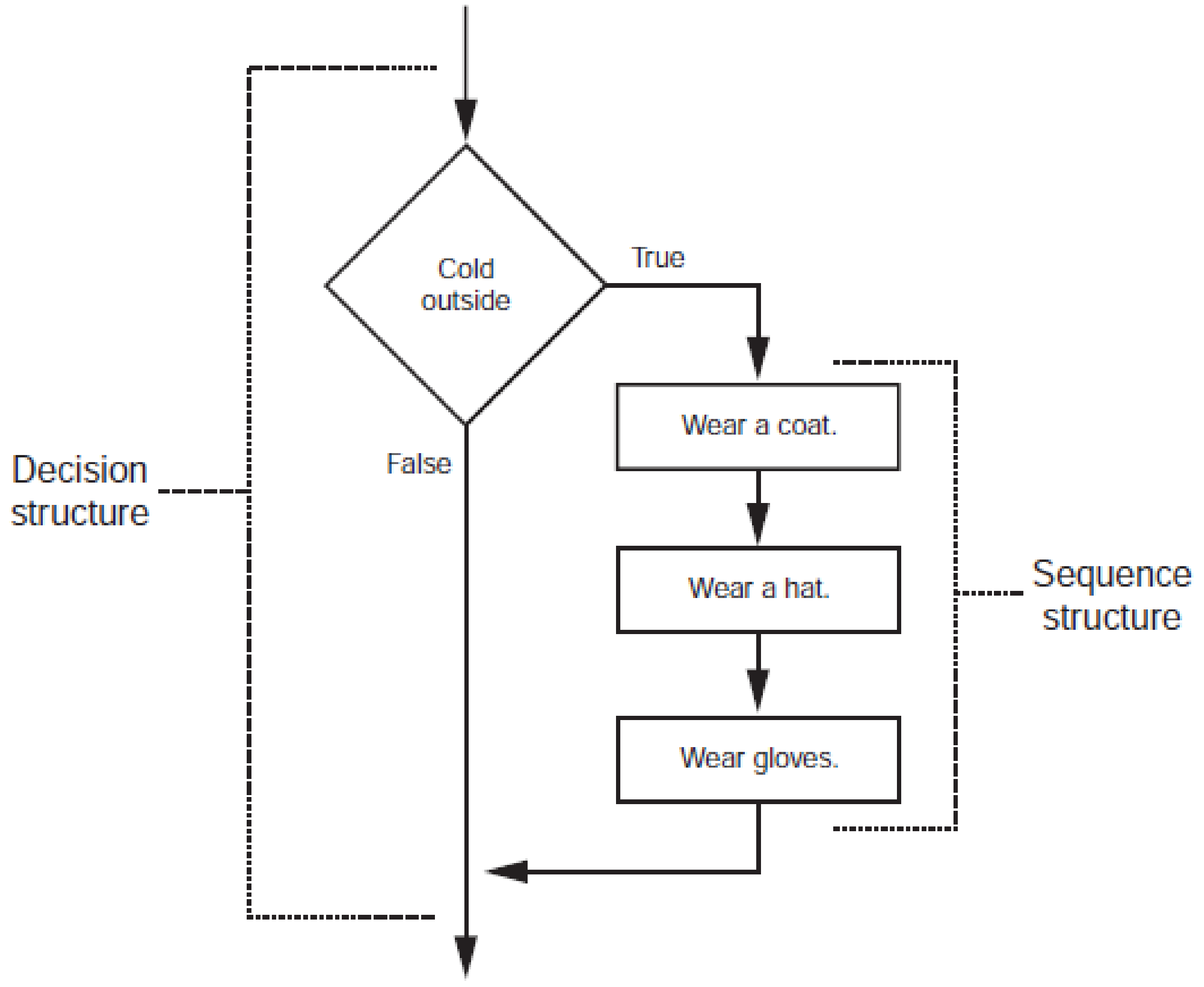
Nested Decision Structures and the if-elif-else Statement

- To test more than one condition, a decision structure can be nested inside another decision structure.

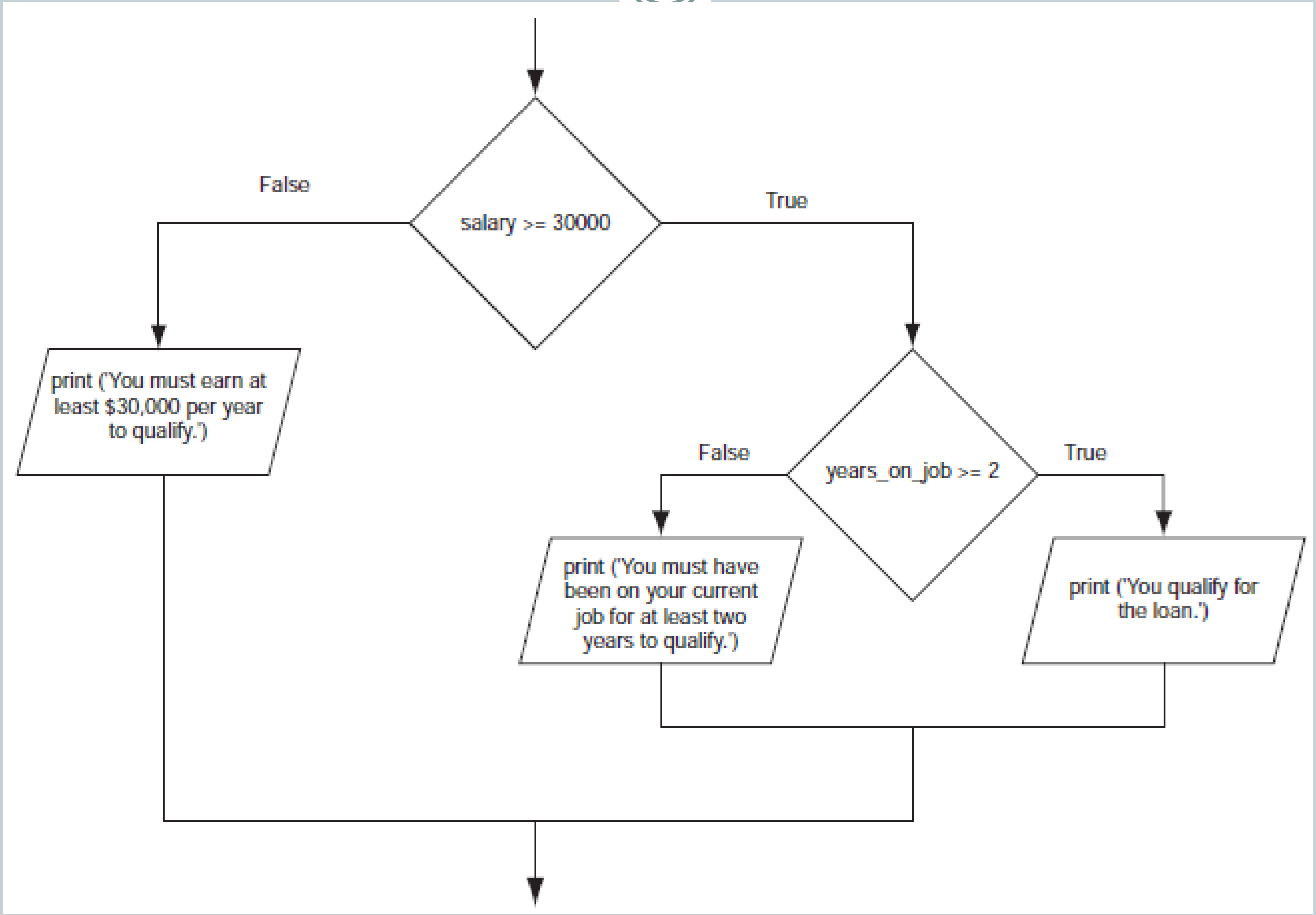
Combining sequence structures with a decision structure



Sequence structure nested inside a decision structure

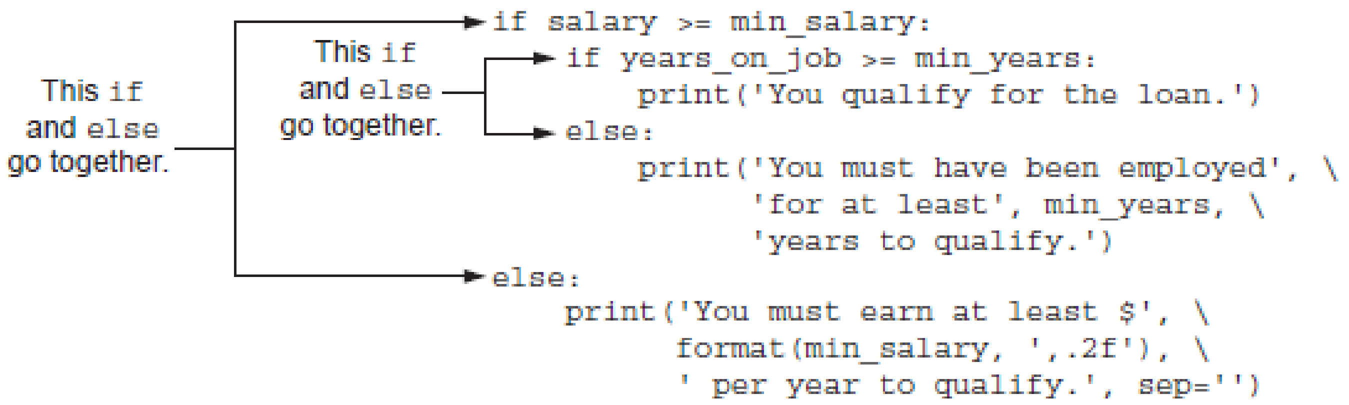


A nested decision structure


False
salary >= 30000
True
print ('You must earn at least $30,000 per year to qualify.')
False
years_on_job >= 2
True
print ('You must have been on your current job for at least two years to qualify.')
print ('You qualify for the loan.')

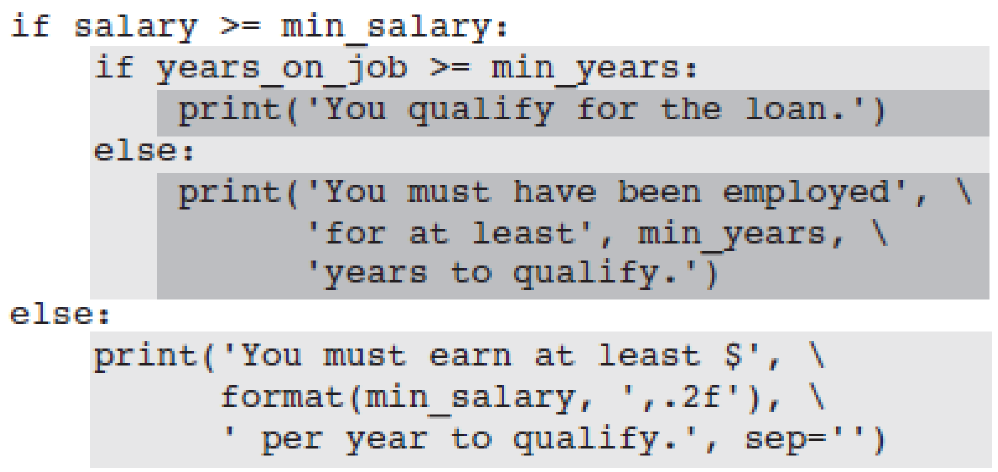
Alignment of **if** and **else** clauses; Nested Blocks

Figure 3-13 Alignment of `if` and `else` clauses



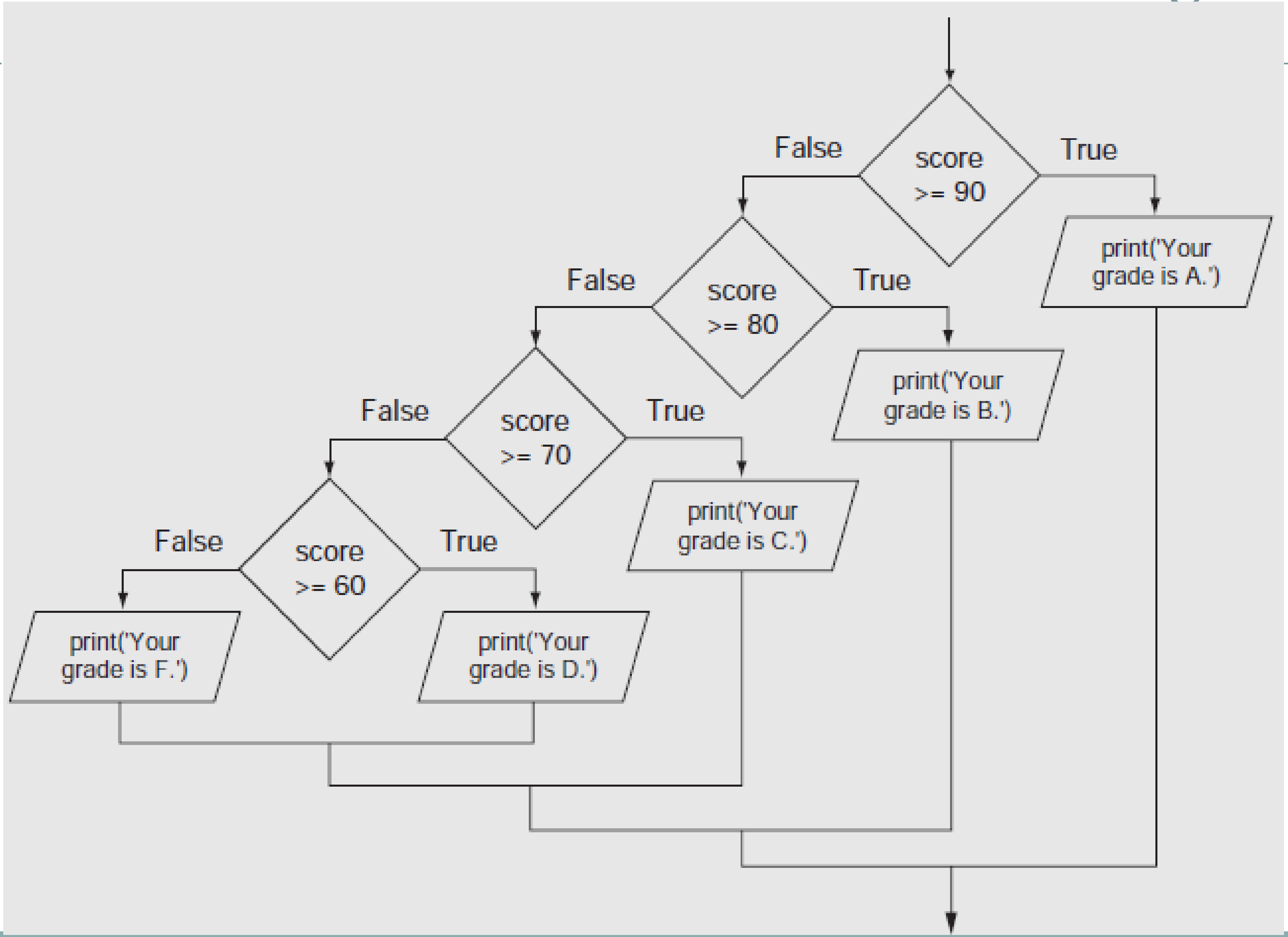
```
if salary >= min_salary:
    if years_on_job >= min_years:
        print('You qualify for the loan.')
    else:
        print('You must have been employed', \
              'for at least', min_years, \
              'years to qualify.')
else:
    print('You must earn at least $', \
          format(min_salary, ',.2f'), \
          ' per year to qualify.', sep='')
```

Figure 3-14 Nested blocks



```
if salary >= min_salary:
    if years_on_job >= min_years:
        print('You qualify for the loan.')
    else:
        print('You must have been employed', \
              'for at least', min_years, \
              'years to qualify.')
else:
    print('You must earn at least $', \
          format(min_salary, ',.2f'), \
          ' per year to qualify.', sep='')
```

Nested decision structure to determine a grade



score >= 90
False
True
print('Your grade is A.')
score >= 80
False
True
print('Your grade is B.')
score >= 70
False
True
print('Your grade is C.')
score >= 60
False
True
print('Your grade is F.')
print('Your grade is D.')

The if-elif-else Statement

- Python provides a special version of the decision structure known as the if-elif-else statement, which makes this type of logic simpler to write.
- Here is the general format of the if-elif-else statement:

```
if condition_1:
        statement
        statement
        etc.
elif condition_2:
        statement
        statement
        etc.
#Insert as many elif clauses as necessary . . .
else:
        statement
        statement
        etc.
```

```
if score >= A_score:
        print('Your grade is A.')
elif score >= B_score:
        print('Your grade is B.')
elif score >= C_score:
        print('Your grade is C.')
elif score >= D_score:
        print('Your grade is D.')
else:
        print('Your grade is F.')
```

- Notice the alignment and indentation that is used with the if-elif-else statement:
- The if, elif, and else clauses are all aligned, and the conditionally executed blocks are indented.

The if-elif-else Statement Contd..

- a long series of nested if-else statements has two particular disadvantages when you are debugging code:
  - The code can grow complex and become difficult to understand.
  - Because of the required indentation, a long series of nested if-else statements can become too long to be displayed on the computer screen without horizontal scrolling.
- Also, long statements tend to "wrap around" when printed on paper, making the code even more difficult to read.
- The logic of an if-elif-else statement is usually easier to follow than a long series of nested if-else statements.
- And, because all of the clauses are aligned in an if-elif-else statement, the lengths of the lines in the statement tend to be shorter.

Logical Operators

- The logical and operator and the logical or operator allow you to connect multiple Boolean expressions to create a compound expression.
- The logical not operator reverses the truth of a Boolean expression.

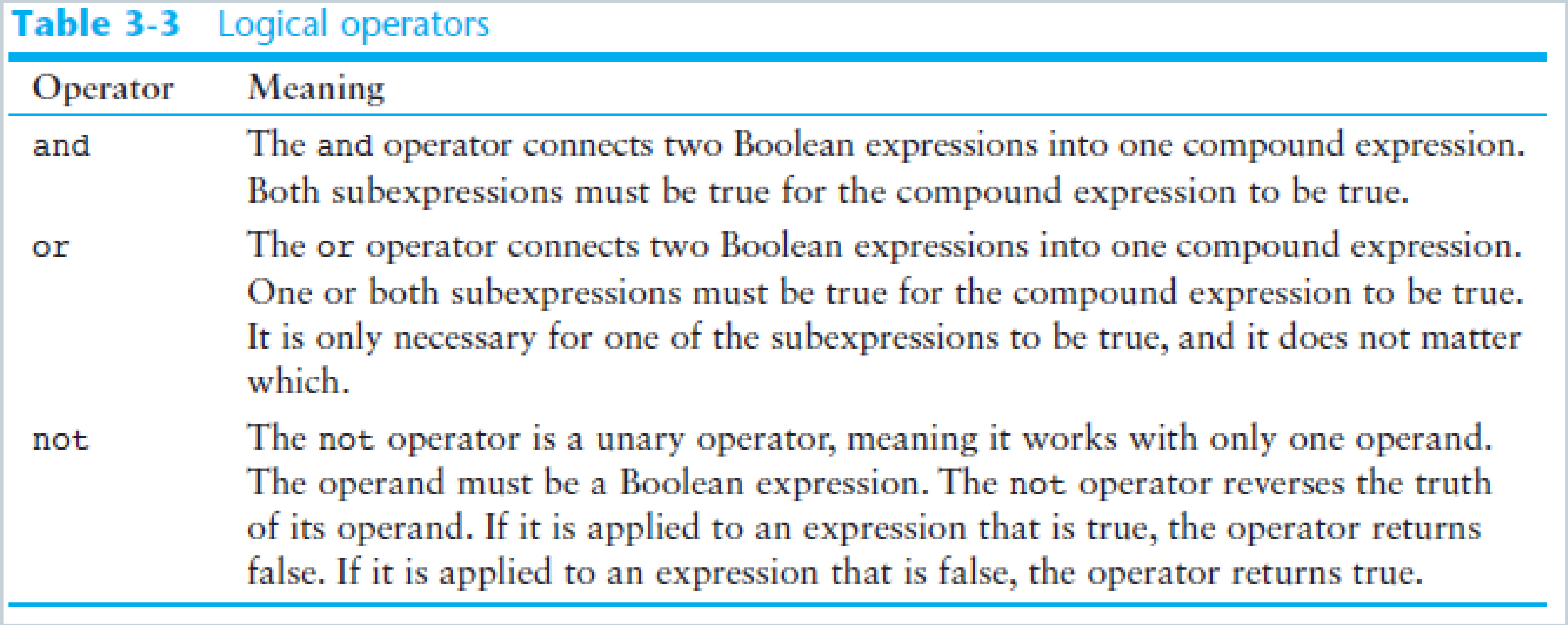


Table 3-3 Logical operators

Operator	Meaning
`and`	The `and` operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
`or`	The `or` operator connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
`not`	The `not` operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The `not` operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

Examples

Table 3-4 Compound Boolean expressions using logical operators

Expression	Meaning
`x > y and a < b`	Is x greater than y AND is a less than b?
`x == y or x == z`	Is x equal to y OR is x equal to z?
`not (x > y)`	Is the expression `x > y` NOT true?

Table 3-5 Truth table for the `and` operator

Expression	Value of the Expression
true and false	false
false and true	false
false and false	false
true and true	true

Table 3-7 Truth table for the `not` operator

Expression	Value of the Expression
not true	false
not false	true

Table 3-6 Truth table for the `or` operator

Expression	Value of the Expression
true or false	true
false or true	true
false or false	false
true or true	true

Short-Circuit Evaluation - and

- Both the and and or operators perform *short-circuit evaluation*.
- *Here's how it works with* the and operator: If the expression on the left side of the and operator is false, the expression on the right side will not be checked.
- Because the compound expression will be false if only one of the sub expressions is false, it would waste CPU time to check the remaining expression.
- So, when the and operator finds that the expression on its left is false, it short circuits and does not evaluate the expression on its right

Short-Circuit Evaluation - or

- Here’s how short-circuit evaluation works with the or operator:
- If the expression on the left side of the or operator is true, the expression on the right side will not be checked.
- Because it is only necessary for one of the expressions to be true, it would waste CPU time to check the remaining expression.

The not Operator

- The not operator is a unary operator that takes a Boolean expression as its operand and reverses its logical value.
- In other words, if the expression is true, the not operator returns false, and if the expression is false, the not operator returns true

Boolean Variables

- A Boolean variable can reference one of two values: True or False.
- Boolean variables are commonly used as flags, which indicate whether specific conditions exist.
- So far we have worked with int, float, and str (string) variables.
- In addition to these data types, Python also provides a bool data type.
- The bool data type allows you to create variables that may reference one of two possible values: True or False.

Example

- Boolean variables are most commonly used as flags.
- A *flag is a variable that signals when* some condition exists in the program.
- When the flag variable is set to False, it indicates the condition does not exist.
- When the flag variable is set to True, it means the condition does exist.
- For example, suppose a salesperson has a quota of $50,000. Assuming sales references the amount that the salesperson has sold, the following code determines whether the quota has been met:

```
if sales >= 50000.0:
        sales_quota_met = True
else:
        sales_quota_met = False
```

- Later in the program we might test the flag in the following way:

```
if sales_quota_met:
        print('You have met your sales quota!')
```

- This code displays 'You have met your sales quota!' if the bool variable sales_quota_met is True.
- This code is equivalent to the following:

```
if sales_quota_met == True:
        print('You have met your sales quota!')
```

Repetition Structures

Introduction to Repetition Structures

- Concept: A repetition structure causes a statement or set of statements to execute repeatedly.
- Programmers commonly have to write code that performs the same task over and over.
- Instead of writing the same sequence of statements over and over, a better way to repeatedly perform an operation is to write the code for the operation once, and then place that code in a structure that makes the computer repeat it as many times as necessary.
- This can be done with a *repetition structure, which is more commonly known as a loop*.

Condition-Controlled and Count-Controlled Loops

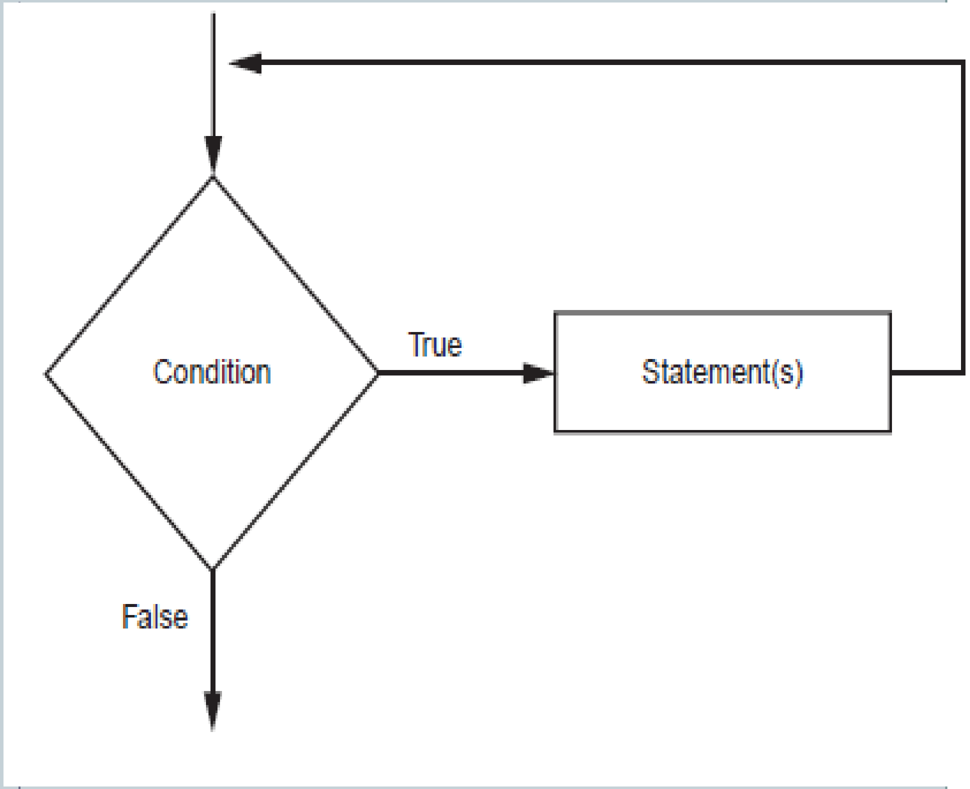
- A *condition-controlled loop uses a true/false condition to control the* number of times that it repeats.
- A *count-controlled loop repeats a specific number of times*.
- In Python you use the while statement to write a condition-controlled loop, and you use the for statement to write a count-controlled loop

The while Loop: A Condition-Controlled Loop

- A condition-controlled loop causes a statement or set of statements to repeat as long as a condition is true.
- In Python you use the *while statement* to write a condition-controlled loop.
- The while loop gets its name from the way it works: *while a condition is true, do some task.*
- *The loop has two parts:*
  - *a condition that is tested for a true or false value, and*
  - a statement or set of statements that is repeated as long as the condition is true.

Logic of a while loop

- The diamond symbol represents the condition that is tested.
- Notice what happens if the condition is true: one or more statements are executed and the program's execution flows back to the point just above the diamond symbol.
- The condition is tested again, and if it is true, the process repeats. If the condition is false, the program exits the loop.
- In a flowchart, you will always recognize a loop when you see a flow line going back to a previous part of the flowchart.



General format of the while loop in Python

```
while condition:
        statement
        statement
        etc.
```

- For simplicity, we will refer to the first line as the *while clause.*
- *The while clause begins* with the word while, followed by a Boolean *condition that will be evaluated as either* true or false.
- A colon appears after the *condition.*
- *Beginning at the next line is a block of* statements.
- This indentation is required because the Python interpreter uses it to tell where the block begins and ends
- When the while loop executes, the *condition is tested.*
- *If the condition is true, the statements* that appear in the block following the while clause are executed, and then the loop starts over.
- If the *condition is false, the program exits the loop.*
- The while loop is known as a *pretest loop, which means it tests its condition before performing* an iteration.

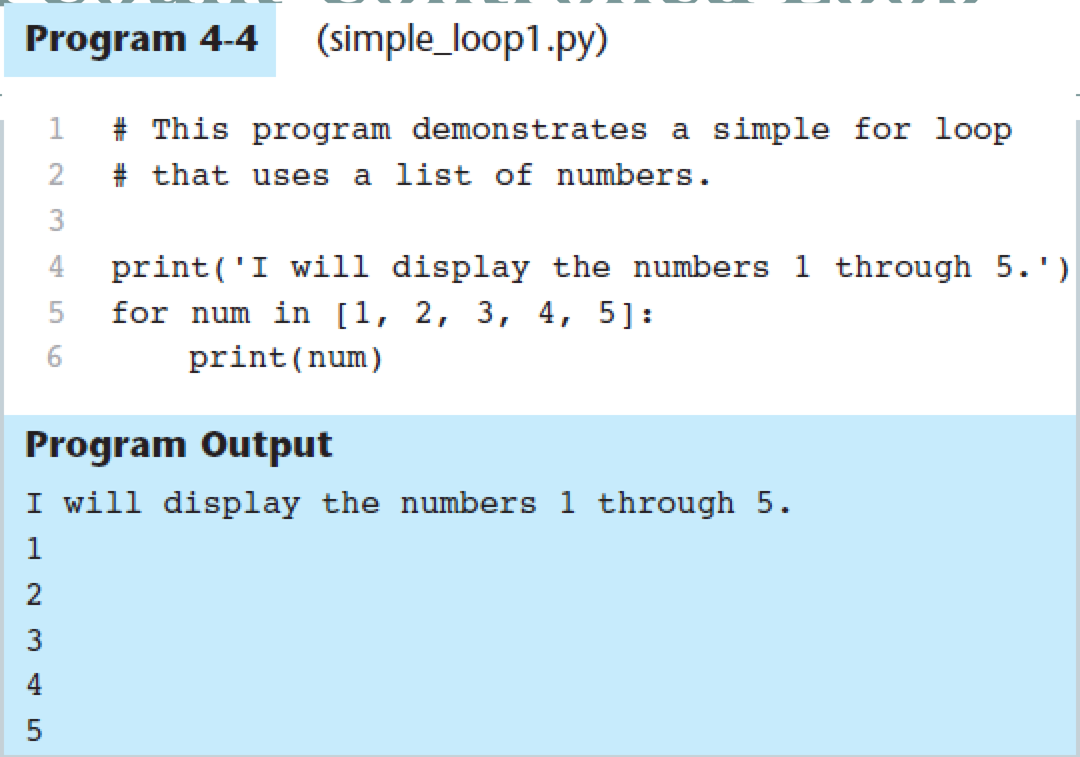
The for Loop: A Count-Controlled Loop

- A count-controlled loop iterates a specific number of times.
- In Python you use the for statement to write a count-controlled loop.
- You use the for statement to write a count-controlled loop.
- In Python, the for statement is designed to work with a sequence of data items.
- When the statement executes, it iterates once for each item in the sequence.
- Here is the general format:

```
for variable in [value1, value2, etc.]:
        statement
        statement
        etc.
```

- We will refer to the first line as the *for clause*.
- *In the for clause, variable is the name of* a variable.
- Inside the brackets a sequence of values appears, with a comma separating each value.
- In Python, a comma-separated sequence of data items that are enclosed in a set of brackets is called a *list*.

The for Loop: A Count-Controlled Loop



Program 4-4 (simple_loop1.py)

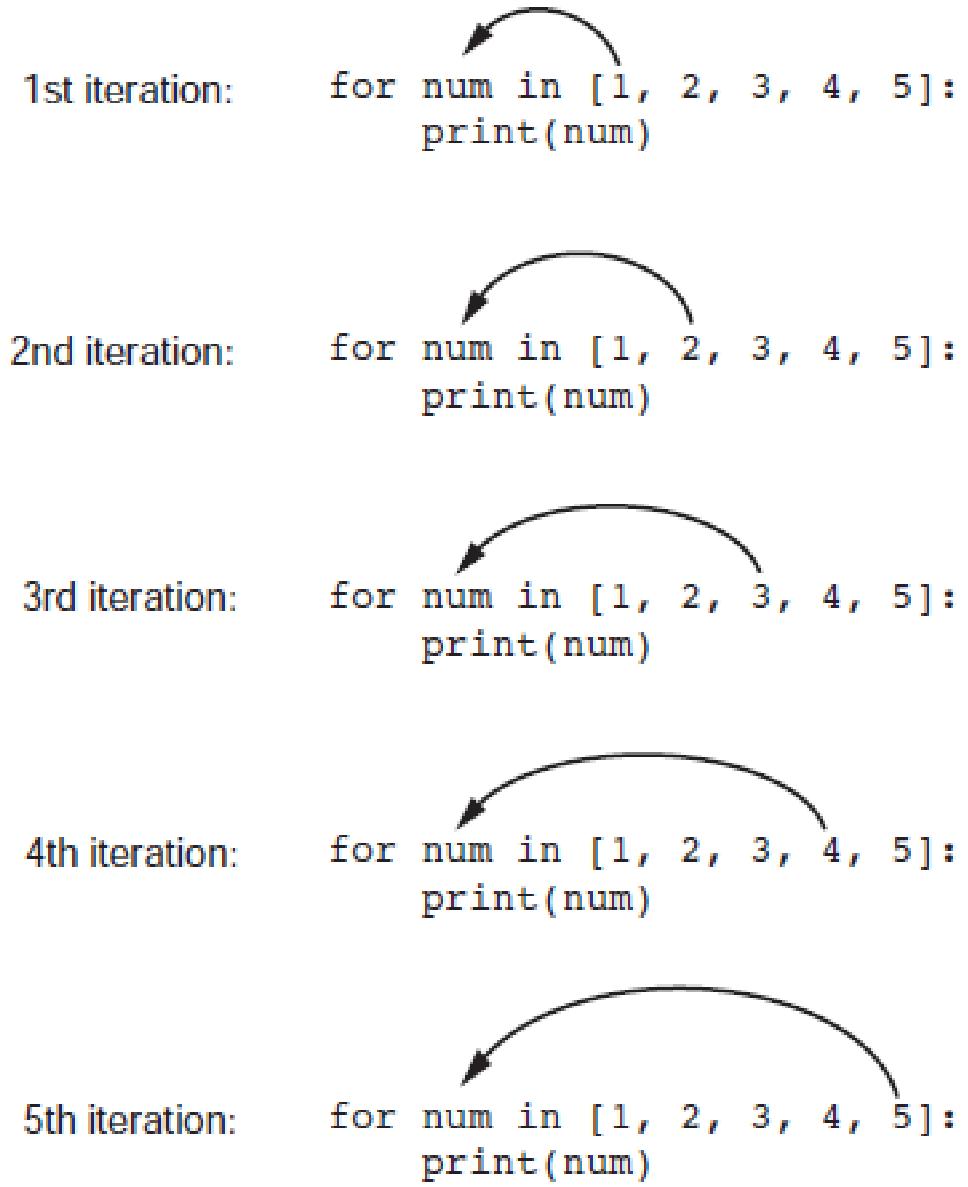
```python
# This program demonstrates a simple for loop
# that uses a list of numbers.

print('I will display the numbers 1 through 5.')
for num in [1, 2, 3, 4, 5]:
    print(num)
```

Program Output

```
I will display the numbers 1 through 5.
1
2
3
4
5
```

- The for statement executes in the following manner:
- The *variable is assigned the first* value in the list, and then the statements that appear in the block are executed.
- Then, *variable* is assigned the next value in the list, and the statements in the block are executed again.
- This continues until *variable has been assigned the last value in the list.*

The `for` loop



1st iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

2nd iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

3rd iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

4th iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

5th iteration:

```
for num in [1, 2, 3, 4, 5]:
    print(num)
```

Example: for loop

Program 4-6 (simple_loop3.py)

```
# This program also demonstrates a simple for
# loop that uses a list of strings.

for name in ['Winken', 'Blinken', 'Nod']:
    print(name)
```

Program Output

```
Winken
Blinken
Nod
```

Using the range Function with the for Loop

- Python provides a built-in function named range that simplifies the process of writing a count-controlled for loop.
- The range function creates a type of object known as an iterable.
- An *iterable is an object that is similar to a list.*
- *It contains a sequence of values that can* be iterated over with something like a loop.
- Here is an example of a for loop that uses the range function:

```
for num in range(5):
        print(num)
```

for example...contd

- Notice that instead of using a list of values, we call to the range function passing 5 as an argument.
- In this statement the range function will generate an iterable sequence of integers in the range of 0 up to (but not including) 5.
- This code works the same as the following:

```
for num in [0, 1, 2, 3, 4]:
        print(num)
```

range()

- If you pass one argument to the range function, is used as the ending limit of the sequence of numbers.
- If you pass two arguments to the range function, the first argument is used as the starting value of the sequence and the second argument is used as the ending limit.
- Here is an example:

```
for num in range(1, 5):
        print(num)
```

- This code will display the following:

```
1
2
3
4
```

range()

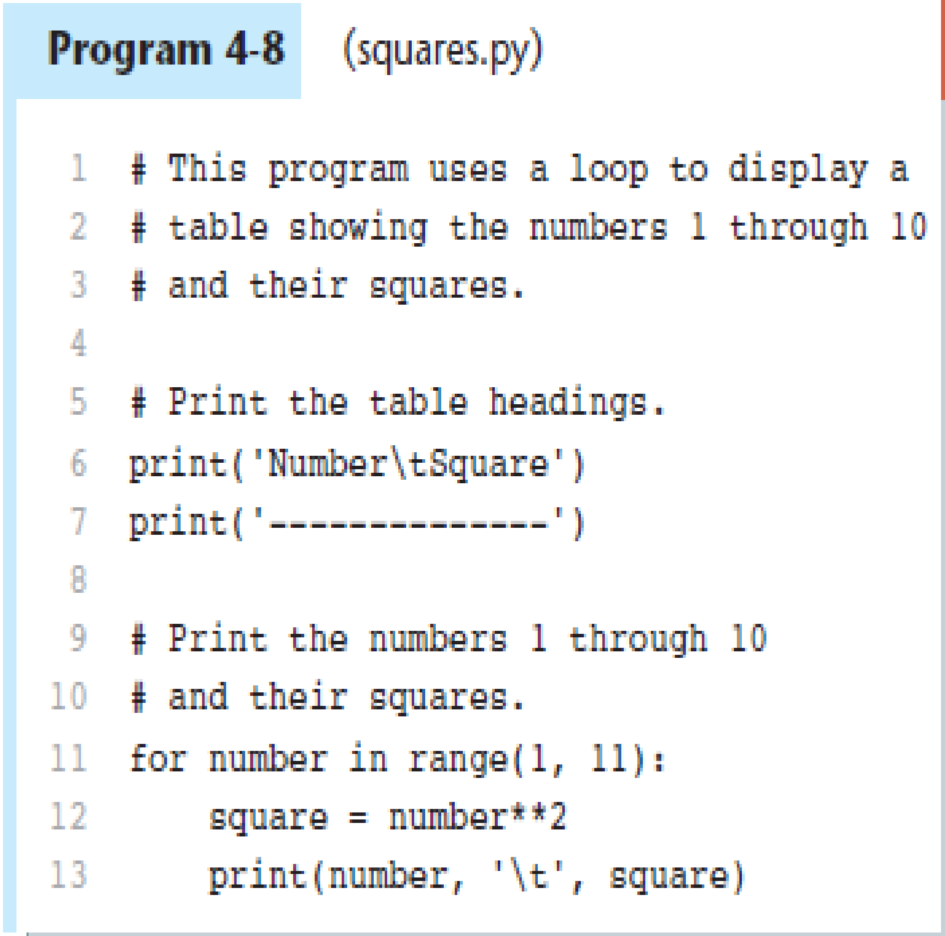
- By default, the range function produces a sequence of numbers that increase by 1 for each successive number in the list.
- If you pass a third argument to the range function, that argument is used as *step value. Instead of increasing by 1, each successive number in the* sequence will increase by the step value.
- Here is an example:

```
for num in range(1, 10, 2):
        print(num)
```

- In this for statement, three arguments are passed to the range function:
  - The first argument, 1, is the starting value for the sequence.
  - The second argument, 10, is the ending limit of the list.
  - This means that the last number in the sequence will be 9.
  - The third argument, 2, is the step value.
  - This means that 2 will be added to each successive number in the sequence.
- This code will display the following:

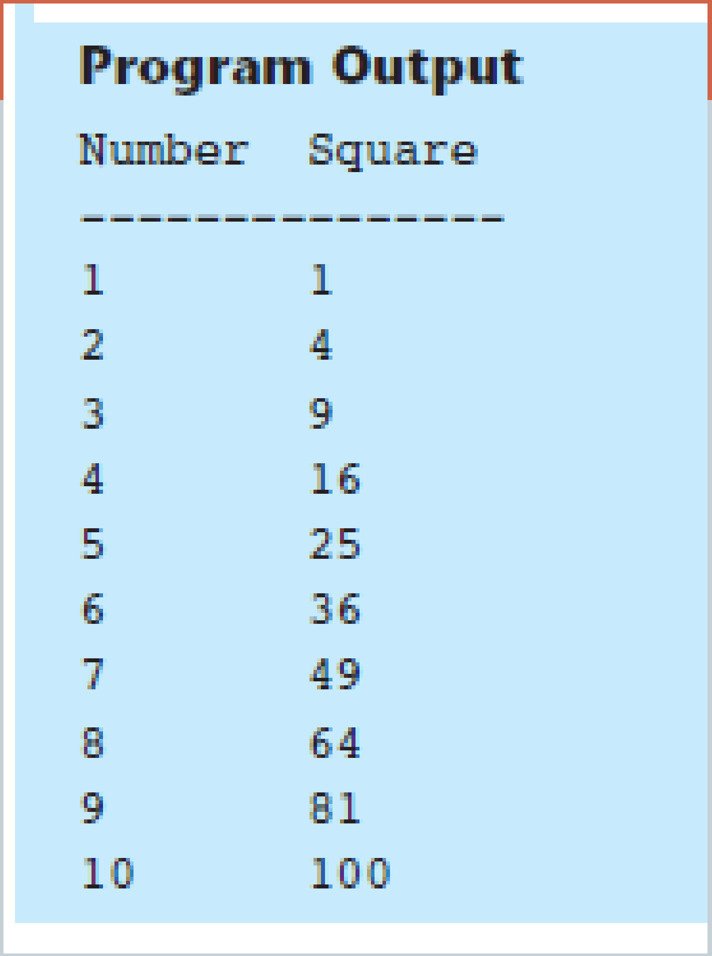
```
1
3
5
7
9
```

Using the Target Variable Inside the Loop



Program 4-8 (squares.py)

```
# This program uses a loop to display a
# table showing the numbers 1 through 10
# and their squares.

# Print the table headings.
print('Number\tSquare')
print('--------------')

# Print the numbers 1 through 10
# and their squares.
for number in range(1, 11):
    square = number**2
    print(number, '\t', square)
```



Program Output

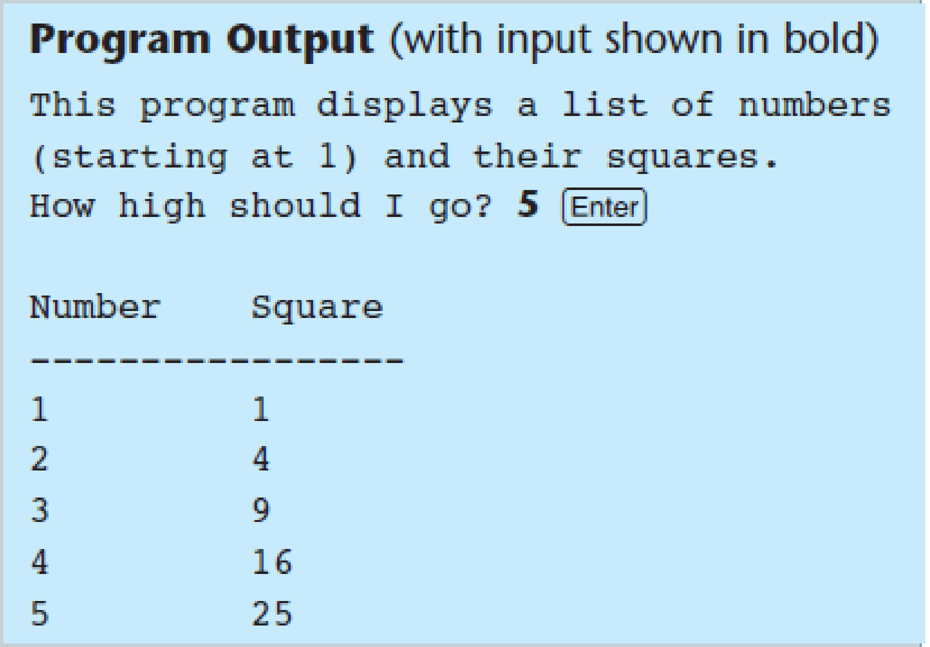
```
Number   Square
-----------------
1          1
2          4
3          9
4          16
5          25
6          36
7          49
8          64
9          81
10         100
```

Letting the User Control the Loop Iterations

Program

```
# This program uses a loop to display a
# table of numbers and their squares.

# Get the ending limit.
print('This program displays a list of numbers')
print('(starting at 1) and their squares.')
end = int(input('How high should I go?'))

# Print the table headings.
 print()
 print('Number\tSquare')
 print('--------------')

 # Print the numbers and their squares.
 for number in range(1, end + 1):
 square = number**2
 print(number, '\t', square)
```

Output



Program Output (with input shown in bold)

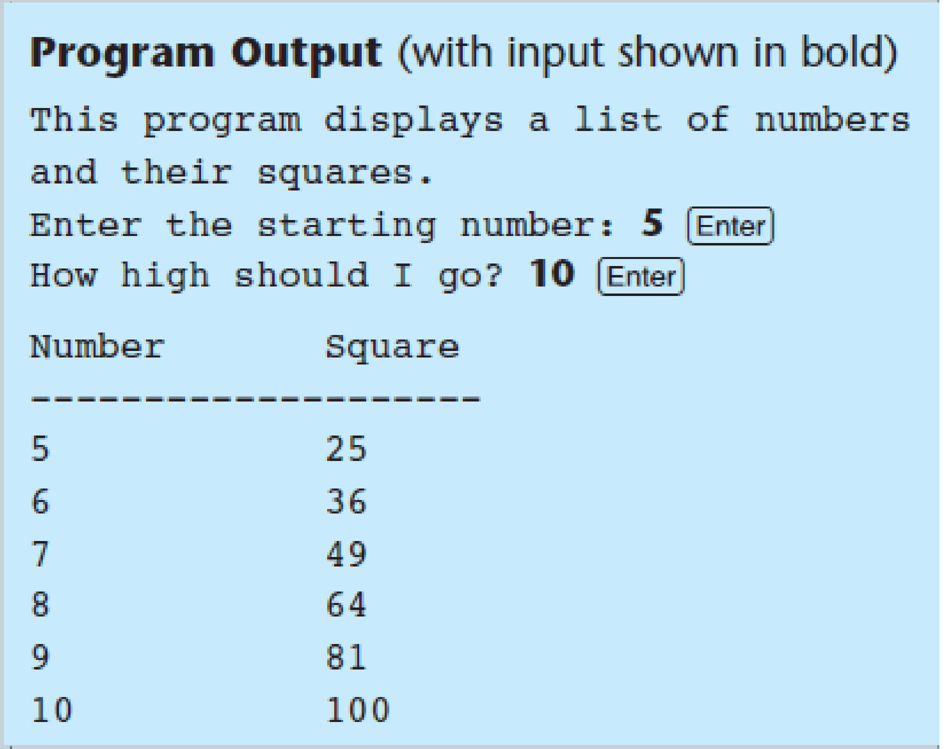
```
This program displays a list of numbers
(starting at 1) and their squares.
How high should I go? 5 Enter

Number    Square
-----------------
1         1
2         4
3         9
4         16
5         25
```

Example

Program

```
# This program uses a loop to display a
# table of numbers and their squares.

# Get the starting value.
print('This program displays a list of numbers')
print('and their squares.')
start = int(input('Enter the starting number: '))

# Get the ending limit.
end = int(input('How high should I go?'))

# Print the table headings.
print()
print('Number\tSquare')
print('--------------')

# Print the numbers and their squares.
for number in range(start, end + 1):
square = number**2
print(number, '\t', square)
```

Output



Program Output (with input shown in bold)

```
This program displays a list of numbers
and their squares.
Enter the starting number: 5 Enter
How high should I go? 10 Enter

Number        Square
--------------------
5             25
6             36
7             49
8             64
9             81
10            100
```

Generating an Iterable Sequence that Ranges from Highest to Lowest

- The range function was used to generate a sequence with numbers that go from lowest to highest.
- Alternatively, you can use the range function to generate sequences of numbers that go from highest to lowest

```
range(10, 0, –1)
```

- In this function call, the starting value is 10, the sequence's ending limit is 0, and the step value is 21.
- This expression will produce the following sequence:

  10, 9, 8, 7, 6, 5, 4, 3, 2, 1

- Here is an example of a for loop that prints the numbers 5 down to 1:

```
for num in range(5, 0, –1):
        print(num)
```

Sentinels

- A sentinel is a special value that marks the end of a sequence of values.
- When processing a long sequence of values with a loop, perhaps a better technique is to use a sentinel.
- A *sentinel* is a special value that marks the end of a sequence of items.
- When a program reads the sentinel value, it knows it has reached the end of the sequence, so the loop terminates.
- A sentinel value must be distinctive enough that it will not be mistaken as a regular value in the sequence
- Example:
- For example, suppose a doctor wants a program to calculate the average weight of all her patients. The program might work like this: A loop prompts the user to enter either a patient's weight, or 0 if there are no more weights. When the program reads 0 as a weight, it interprets this as a signal that there are no more weights

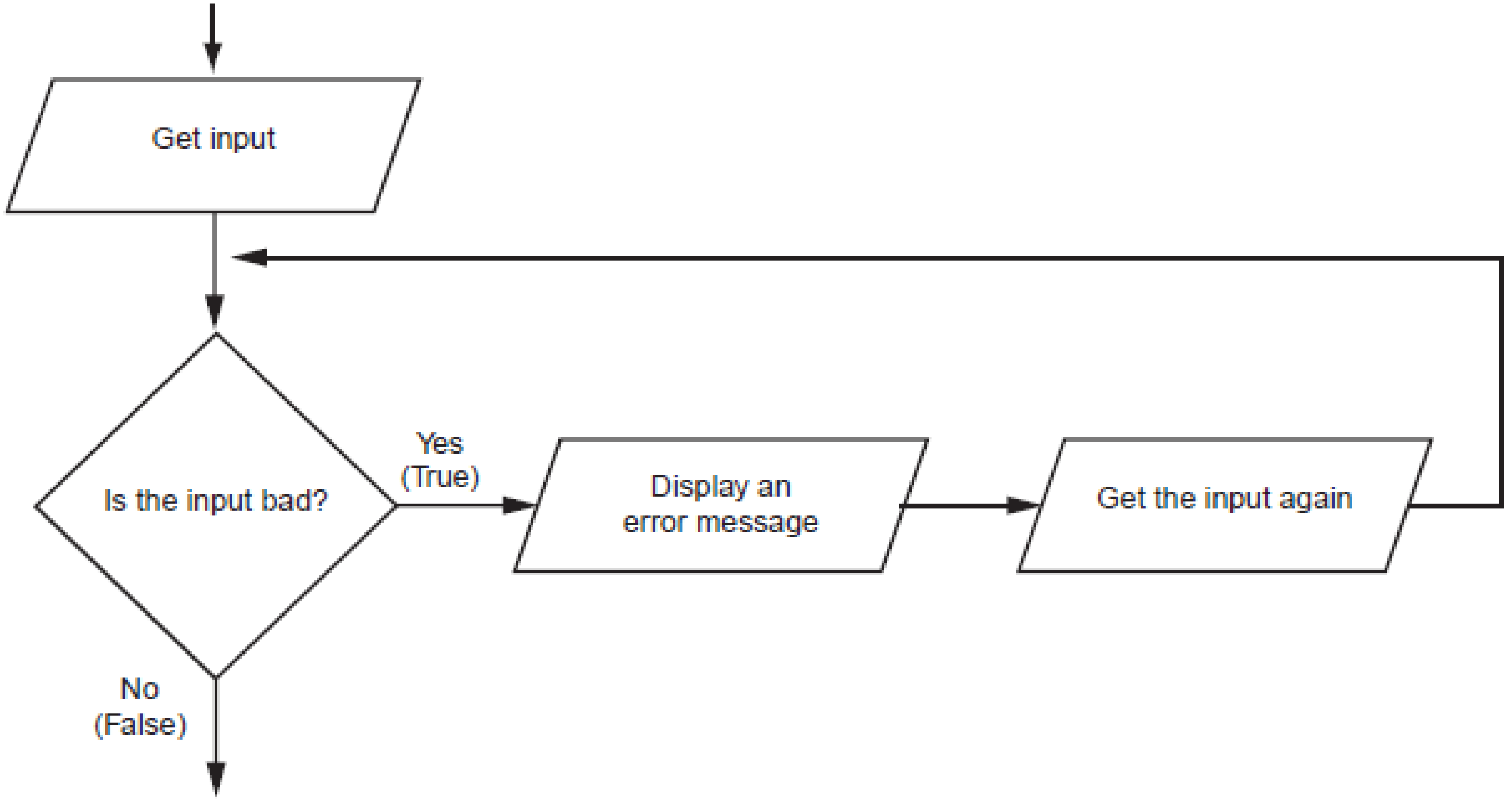
Input Validation Loops

- Input validation is the process of inspecting data that has been input to a program, to make sure it is valid before it is used in a computation.
- Input validation is commonly done with a loop that iterates as long as an input variable references bad data.
- One of the most famous sayings among computer programmers is “garbage in, garbage out.” This saying, sometimes abbreviated as *GIGO*, refers to the fact that computers cannot tell the difference between good data and bad data.
- If a user provides bad data as input to a program, the program will process that bad data and, as a result, will produce bad data as output.
- Examples:
  - One example is a negative number entered for the hours worked;
  - another is an invalid hourly pay rate.

Input validation

- The integrity of a program’s output is only as good as the integrity of its input.
- For this reason, you should design your programs in such a way that bad input is never accepted.
- When input is given to a program, it should be inspected before it is processed.
- If the input is invalid, the program should discard it and prompt the user to enter the correct data.
- This process is known as *input validation*.

Logic containing an input validation loop



Example

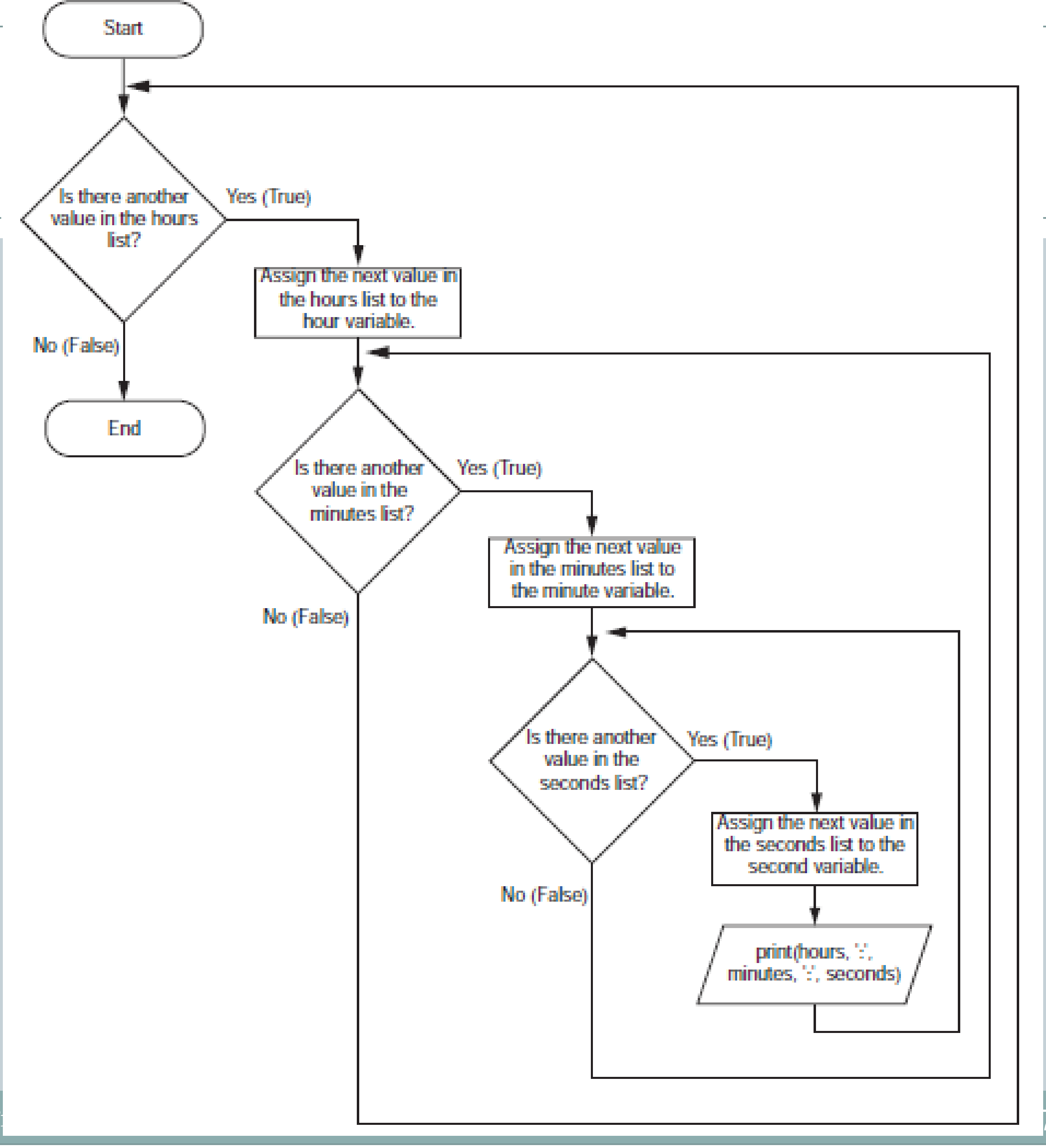
- An input validation loop is sometimes called an *error trap* or an *error handler*.
- What if you also want to reject any test scores that are greater than 100?
- You can modify the input validation loop so it uses a compound Boolean expression, as shown next.

```
# Get a test score.
score = int(input('Enter a test score: '))
# Make sure it is not less than 0 or greater than 100.
while score < 0 or score > 100:
        print('ERROR: The score cannot be negative')
        print('or greater than 100.')
        score = int(input('Enter the correct score: '))
```

- The loop in this code determines whether score is less than 0 or greater than 100.
- If either is true, an error message is displayed and the user is prompted to enter a correct score

Nested Loops

- A loop that is inside another loop is called a nested loop.
- A clock is a good example of something that works like a nested loop.
- The second hand, minute hand, and hour hand all spin around the face of the clock.
- The hour hand, however, only makes 1 revolution for every 12 of the minute hand's revolutions. And it takes 60 revolutions of the second hand for the minute hand to make 1 revolution.
- This means that for every complete revolution of the hour hand, the second hand has revolved 720 times.
- Here is a loop that partially simulates a digital clock. It displays the seconds from 0 to 59:


Start
Is there another value in the hours list?
Yes (True)
Assign the next value in the hours list to the hour variable.
No (False)
End
Is there another value in the minutes list?
Yes (True)
Assign the next value in the minutes list to the minute variable.
No (False)
Is there another value in the seconds list?
Yes (True)
Assign the next value in the seconds list to the second variable.
No (False)
print(hours, ':', minutes, ':', seconds)

```
for seconds in range(60):
    print(seconds)
```

We can add a minutes variable and nest the loop above inside another loop that cycles through 60 minutes:

```
for minutes in range(60):
    for seconds in range(60):
        print(minutes, ':', seconds)
```

To make the simulated clock complete, another variable and loop can be added to count the hours:

```
for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print(hours, ':', minutes, ':', seconds)
```

This code's output would be:

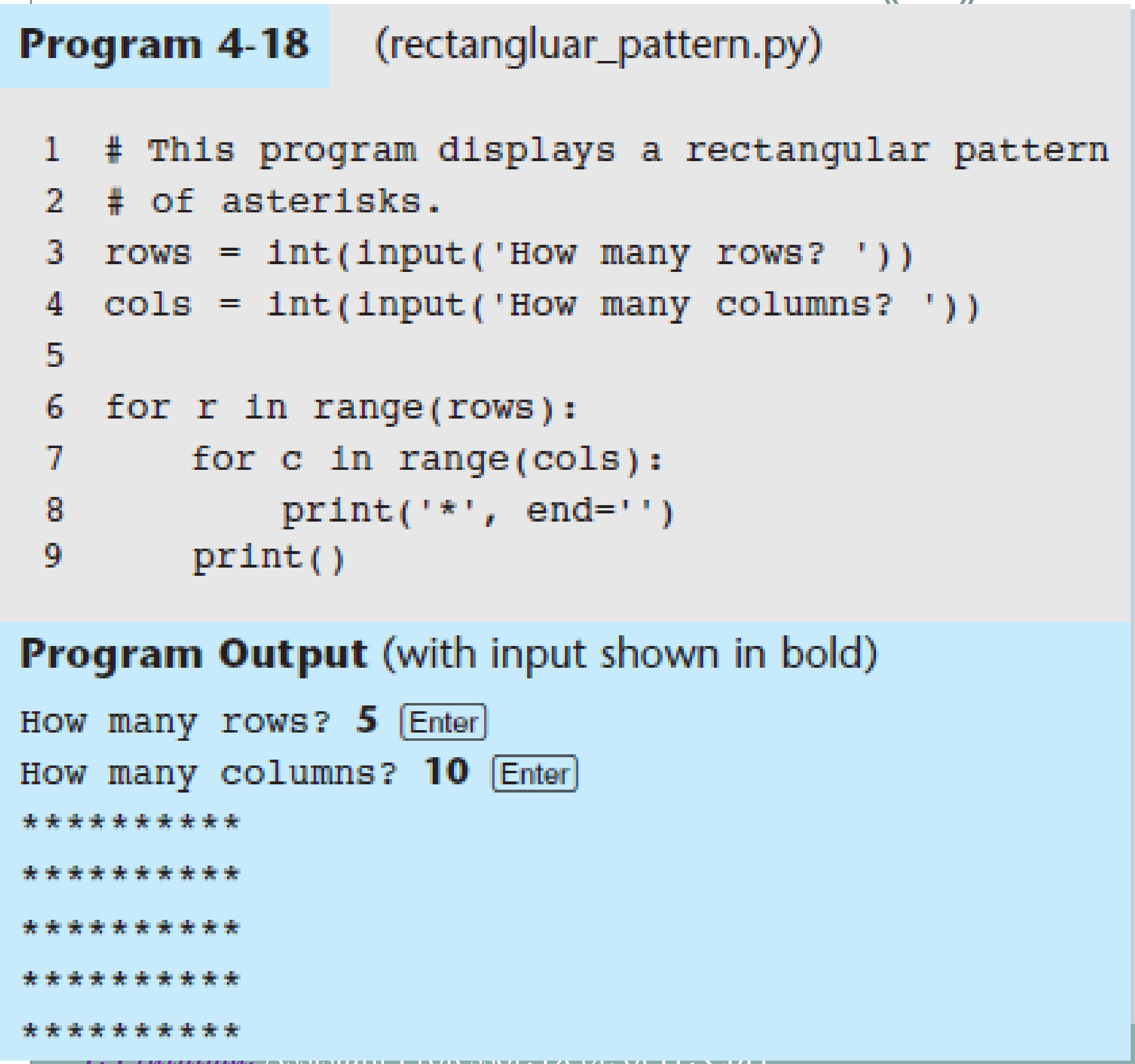
```
0:0:0
0:0:1
0:0:2
```

(The program will count through each second of 24 hours.)

```
23:59:59
```

The innermost loop will iterate 60 times for each iteration of the middle loop. The middle loop will iterate 60 times for each iteration of the outermost loop. When the outermost loop has iterated 24 times, the middle loop will have iterated 1,440 times and the innermost loop will have iterated 86,400 times! Figure 4-8 shows a flowchart for the complete clock

Using Nested Loops to Print Patterns



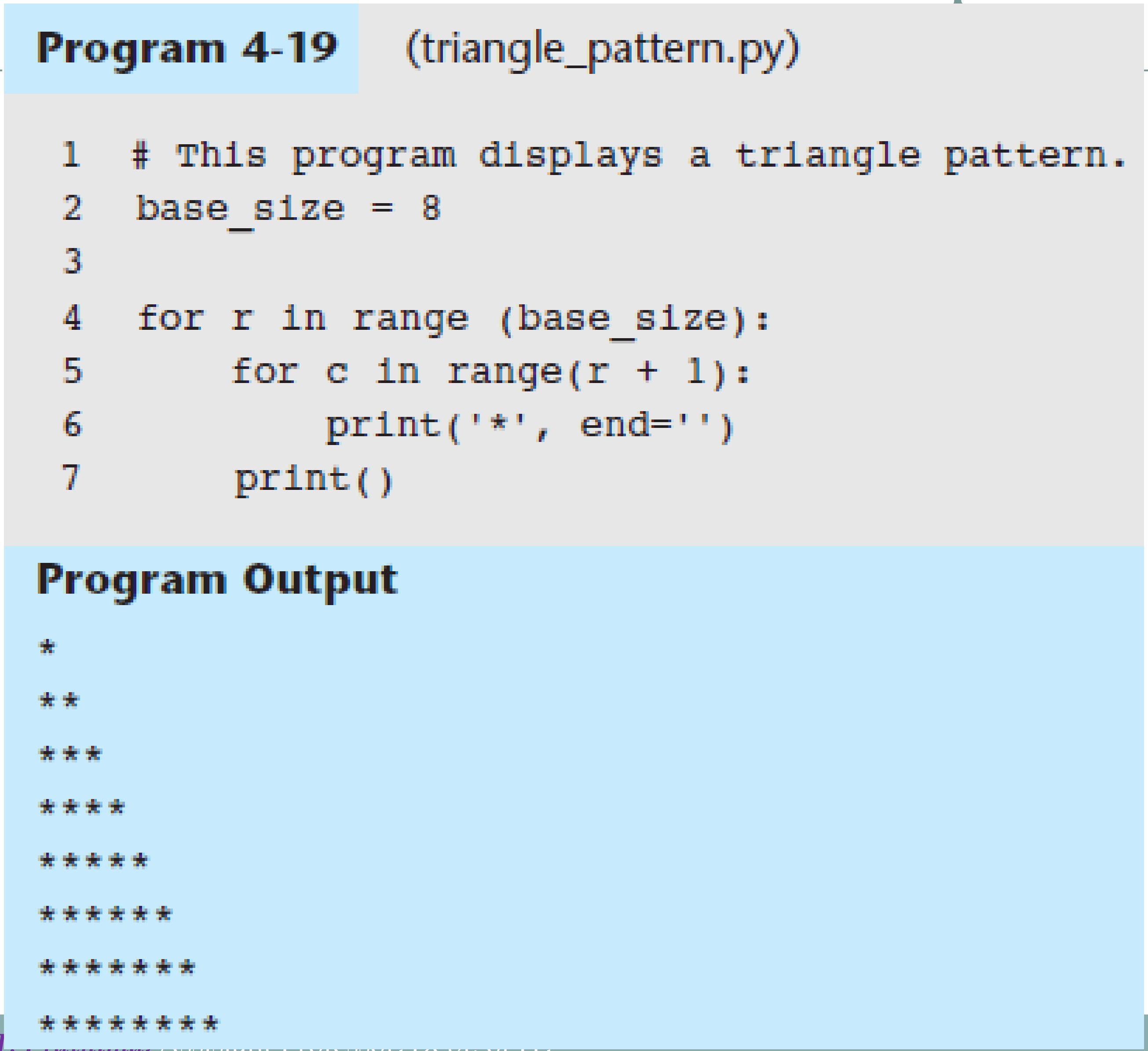
Program 4-18 (rectangluar_pattern.py)

```python
# This program displays a rectangular pattern
# of asterisks.
rows = int(input('How many rows? '))
cols = int(input('How many columns? '))

for r in range(rows):
    for c in range(cols):
        print('*', end='')
    print()
```

Program Output (with input shown in bold)

```
How many rows? 5 Enter
How many columns? 10 Enter
**********
**********
**********
**********
**********
```

Pattern Example



Program 4-19 (triangle_pattern.py)

```python
# This program displays a triangle pattern.
base_size = 8

for r in range (base_size):
    for c in range(r + 1):
        print('*', end='')
    print()
```

Program Output

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

Program 4-20 (stair_step_pattern.py)

```python
# This program displays a stair-step pattern.
num_steps = 6

for r in range(num_steps):
    for c in range(r):
        print(' ', end='')
    print('#')
```

Program Output

```
#
 #
  #
   #
    #
     #
```

References

- Tony Gaddis, “Starting out with Python”, 3rd Edition, Global Edition, Pearson Education