Intro to Python

- **Comments: notes of explanation within a program**
  - Ignored by Python interpreter
    - Intended for a person reading the program's code
  - Begin with a # character
- **End-line comment: appears at the end of a line of code**
  - Typically explains the purpose of that line

Variables

- **Variable: name that represents a value stored in the computer memory**
  - Used to access and manipulate data stored in memory
  - A variable references the value it represents
- **Assignment statement: used to create a variable and make it reference data**
  - General format is `variable = expression`
    - Example: `age = 29`
    - Assignment operator: the equal sign (=)

Variable Naming Rules

- **Rules for naming variables in Python:**
    - Variable name cannot be a Python key word
    - Variable name cannot contain spaces
    - First character must be a letter or an underscore
    - After first character may use letters, digits, or underscores
    - Variable names are case sensitive
- **Variable name should reflect its use**

Numeric Data Types, Literals, and the `str` Data Type

- **Data types: categorize value in memory**
  - e.g., int for integer, float for real number, str used for storing strings in memory
- **Numeric literal: number written in a program**
  - No decimal point considered int, otherwise, considered float
- **Some operations behave differently depending on data type**

Reassigning a Variable to a Different Type

- A variable in Python can refer to items of any type

Figure 2-7 The variable x references an integer

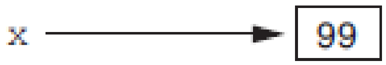
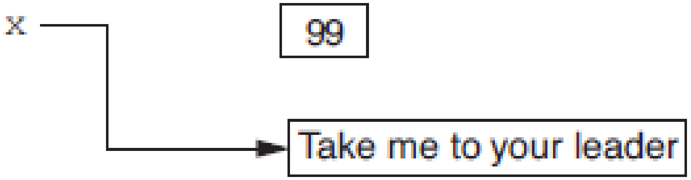


Figure 2-8 The variable x references a string



Reading Input from the Keyboard

- **Most programs need to read input from the user**
- **Built-in `input` function reads input from keyboard**
  - Returns the data as a string
  - Format: `*variable* = input(*prompt*)`
    - `prompt` is typically a string instructing user to enter a value
  - Does not automatically display a space after the prompt

Reading Numbers with the `input` Function

- **`input` function always returns a string**
- **Built-in functions convert between data types**
  - `int(`*`item`*`)` converts *`item`* to an `int`
  - `float(`*`item`*`)` converts *`item`* to a `float`
  - Nested function call: general format:
    `function1(function2(argument))`
    - value returned by function2 is passed to function1
  - Type conversion only works if item is valid numeric value, otherwise, throws exception

Performing Calculations

- **Math expression: performs calculation and gives a value**
  - Math operator: tool for performing calculation
  - Operands: values surrounding operator
    - Variables can be used as operands
  - Resulting value typically assigned to variable
- **Two types of division:**
  - `/` operator performs floating point division
  - `//` operator performs integer division
    - Positive results truncated, negative rounded away from zero

Operator Precedence and Grouping with Parentheses

- **Python operator precedence:**
  1. Operations enclosed in parentheses
     - Forces operations to be performed before others
  2. Exponentiation (**)
  3. Multiplication (*), division (/ and //), and remainder (%)
  4. Addition (+) and subtraction (-)
- **Higher precedence performed first**
  - Same precedence operators execute from left to right

Breaking Long Statements into Multiple Lines

- Long statements cannot be viewed on screen without scrolling and cannot be printed without cutting off
- Multiline continuation character (\): Allows to break a statement into multiple lines

```
result = var1 * 2 + var2 * 3 + \
         var3 * 4 + var4 * 5
```

Breaking Long Statements into Multiple Lines

- **Any part of a statement that is enclosed in parentheses can be broken without the line continuation character.**

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

total = (value1 + value2 +
         value3 + value4 +
         value5 + value6)
```

More About Data Output

- **`print` function displays line of output**
  - Newline character at end of printed data
  - Special argument `end='`*`delimiter`*`'` causes `print` to place *`delimiter`* at end of data instead of newline character
- **`print` function uses space as item separator**
  - Special argument `sep='`*`delimiter`*`'` causes `print` to use *`delimiter`* as item separator

More About Data Output (cont’d.)

- **Special characters appearing in string literal**
  - Preceded by backslash (`\`)
    - Examples: newline (`\n`), horizontal tab (`\t`)
  - Treated as commands embedded in string
- **When + operator used on two strings in performs string concatenation**
  - Useful for breaking up a long string literal

Formatting Numbers

- **Can format display of numbers on screen using built-in `format` function**
  - Two arguments:
    - Numeric value to be formatted
    - Format specifier
  - Returns string containing formatted number
  - Format specifier typically includes precision and data type
    - Can be used to indicate scientific notation, comma separators, and the minimum field width used to display the value

Formatting Numbers (cont'd.)

- **The `%` symbol can be used in the format string of `format` function to format number as percentage**
- **To format an integer using `format` function:**
  - Use `d` as the type designator
  - Do not specify precision
  - Can still use `format` function to set field width or comma separator

Magic Numbers

- **A magic number is an unexplained numeric value that appears in a program's code. Example:**

```
amount = balance * 0.069
```

- **What is the value 0.069? An interest rate? A fee percentage? Only the person who wrote the code knows for sure.**

The Problem with Magic Numbers

- **It can be difficult to determine the purpose of the number.**
- **If the magic number is used in multiple places in the program, it can take a lot of effort to change the number in each location, should the need arise.**
- **You take the risk of making a mistake each time you type the magic number in the program’s code.**
  - For example, suppose you intend to type 0.069, but you accidentally type .0069. This mistake will cause mathematical errors that can be difficult to find.

Named Constants

- You should use named constants instead of magic numbers.
- A named constant is a name that represents a value that does not change during the program's execution.
- Example:

```
INTEREST_RATE = 0.069
```

- This creates a named constant named **`INTEREST_RATE`**, assigned the value 0.069. It can be used instead of the magic number:

```
amount = balance * INTEREST_RATE
```

Pearson

Advantages of Using Named Constants

- **Named constants make code self-explanatory (self-documenting)**
- **Named constants make code easier to maintain (change the value assigned to the constant, and the new value takes effect everywhere the constant is used)**
- **Named constants help prevent typographical errors that are common when using magic numbers**

The `if` Statement

- Control structure: logical design that controls order in which set of statements execute
- Sequence structure: set of statements that execute in the order they appear
- Decision structure: specific action(s) performed only if a condition exists
  - Also known as selection structure

The `if` Statement (cont'd.)

- **Python syntax:**

```
if condition:
    Statement
    Statement
```

- **First line known as the `if` clause**
  - Includes the keyword `if` followed by condition
    - The condition can be true or false
    - When the `if` statement executes, the condition is tested, and if it is true the block statements are executed. otherwise, block statements are skipped

Boolean Expressions and Relational Operators (cont’d.)

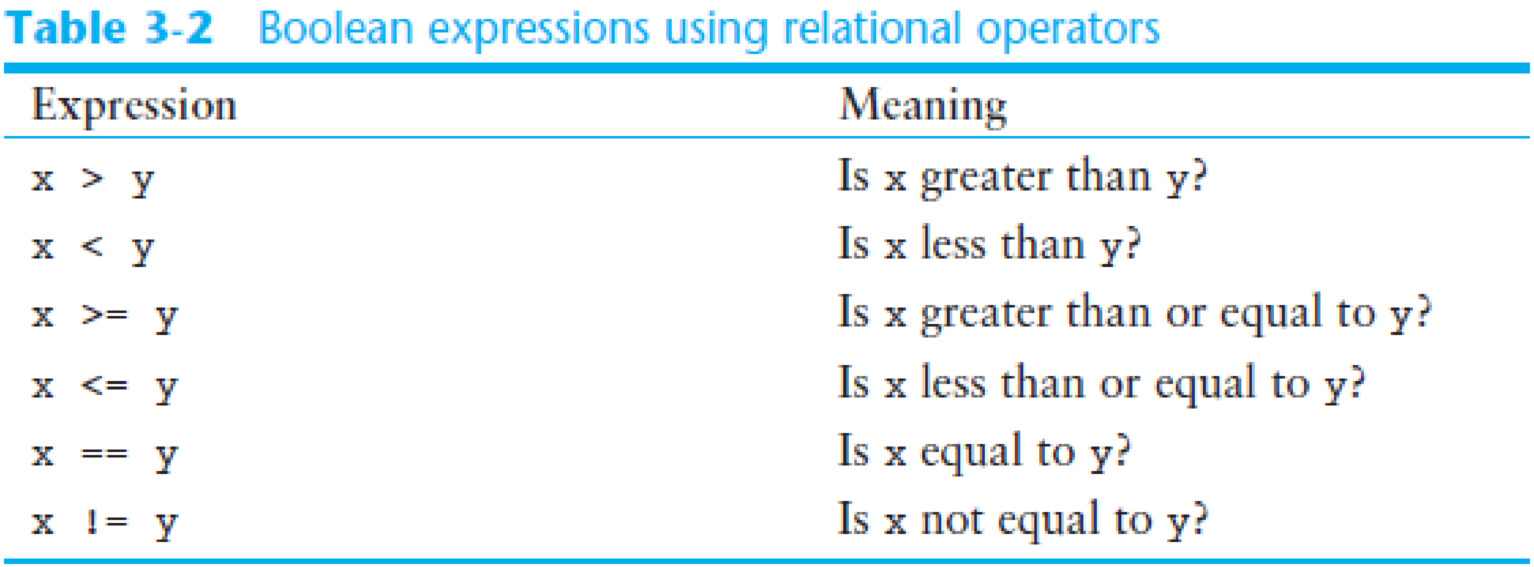


Table 3-2 Boolean expressions using relational operators

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?

The `if-else` Statement

- **Dual alternative decision structure: two possible paths of execution**
  - One is taken if the condition is true, and the other if the condition is false
  - Syntax:

    ```
    if condition:
        statements
    else:
        other statements
    ```

  - `if` clause and `else` clause must be aligned
  - Statements must be consistently indented

Nested Decision Structures and the `if-elif-else` Statement

- **A decision structure can be nested inside another decision structure**
  - Commonly needed in programs
  - Example:
    - Determine if someone qualifies for a loan, they must meet two conditions:
      - Must earn at least $30,000/year
      - Must have been employed for at least two years
    - Check first condition, and if it is true, check second condition

The `if-elif-else` Statement

- **`if-elif-else` statement: special version of a decision structure**
  - Makes logic of nested decision structures simpler to write
    - Can include multiple `elif` statements
  - Syntax:

```
if condition_1:
     statement(s)
elif condition_2:
     statement(s)
elif condition_3:
     statement(s)
else
     statement(s)
```

Insert as many `elif` clauses as necessary.

Short-Circuit Evaluation

- **Short circuit evaluation: deciding the value of a compound Boolean expression after evaluating only one sub expression**
  - Performed by the `or` and `and` operators
    - For `or` operator: If left operand is true, compound expression is true. Otherwise, evaluate right operand
    - For `and` operator: If left operand is false, compound expression is false. Otherwise, evaluate right operand

Intro to Repetition Structures

- **Often have to write code that performs the same task multiple times**
  - Disadvantages to duplicating code
    - Makes program large
    - Time consuming
    - May need to be corrected in many places
- **<u>Repetition structure</u>: makes computer repeat included code as necessary**
  - Includes condition-controlled loops and count-controlled loops

The `while` Loop: a Condition-Controlled Loop

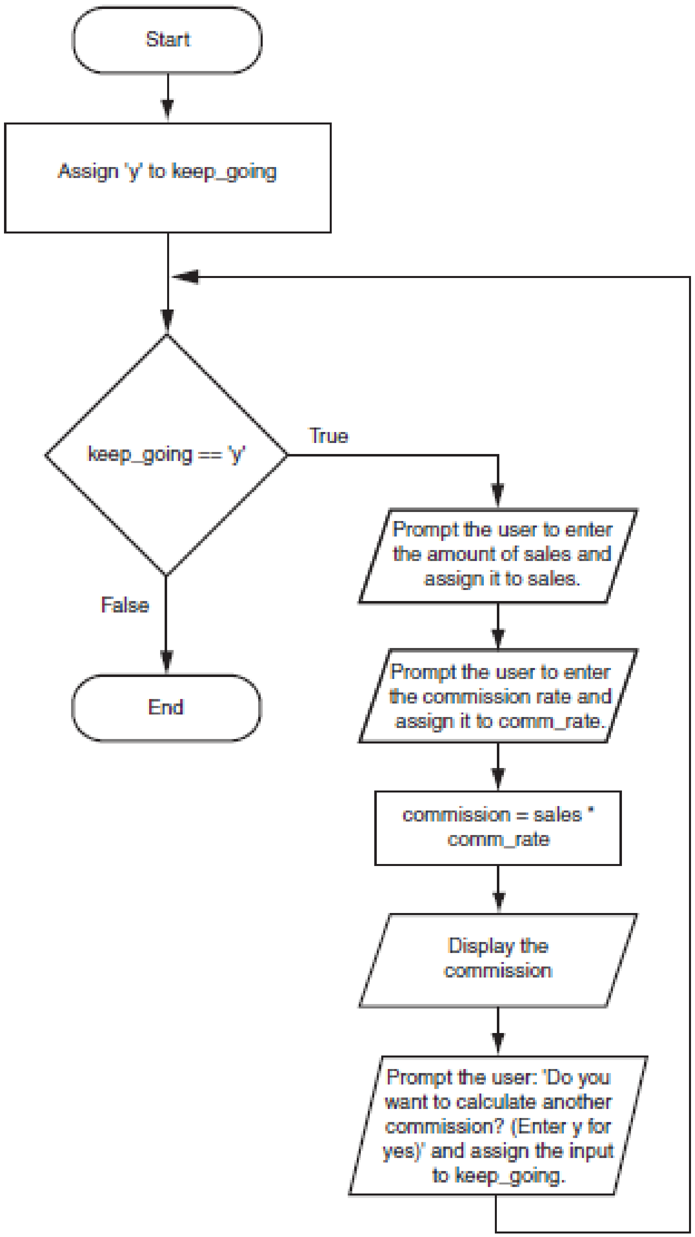
- **`while` loop: while condition is true, do something**
  - Two parts:
    - Condition tested for true or false value
    - Statements repeated as long as condition is true
  - In flow chart, line goes back to previous part
  - General format:

```
while condition:
    statements
```

The `while` Loop: a Condition-Controlled Loop (cont'd.)

- **In order for a loop to stop executing, something has to happen inside the loop to make the condition false**
- **Iteration: one execution of the body of a loop**
- **`while` loop is known as a *pretest* loop**
  - Tests condition before performing an iteration
    - Will never execute if condition is false to start with
    - Requires performing some steps prior to the loop

Figure 4-3 Flowchart for Program 4-1


Start
Assign 'y' to keep_going
keep_going == 'y'
True
False
End
Prompt the user to enter the amount of sales and assign it to sales.
Prompt the user to enter the commission rate and assign it to comm_rate.
commission = sales * comm_rate
Display the commission
Prompt the user: 'Do you want to calculate another commission? (Enter y for yes)' and assign the input to keep_going.

Infinite Loops

- **Loops must contain within themselves a way to terminate**
  - Something inside a `while` loop must eventually make the condition false
- **Infinite loop: loop that does not have a way of stopping**
  - Repeats until program is interrupted
  - Occurs when programmer forgets to include stopping code in the loop

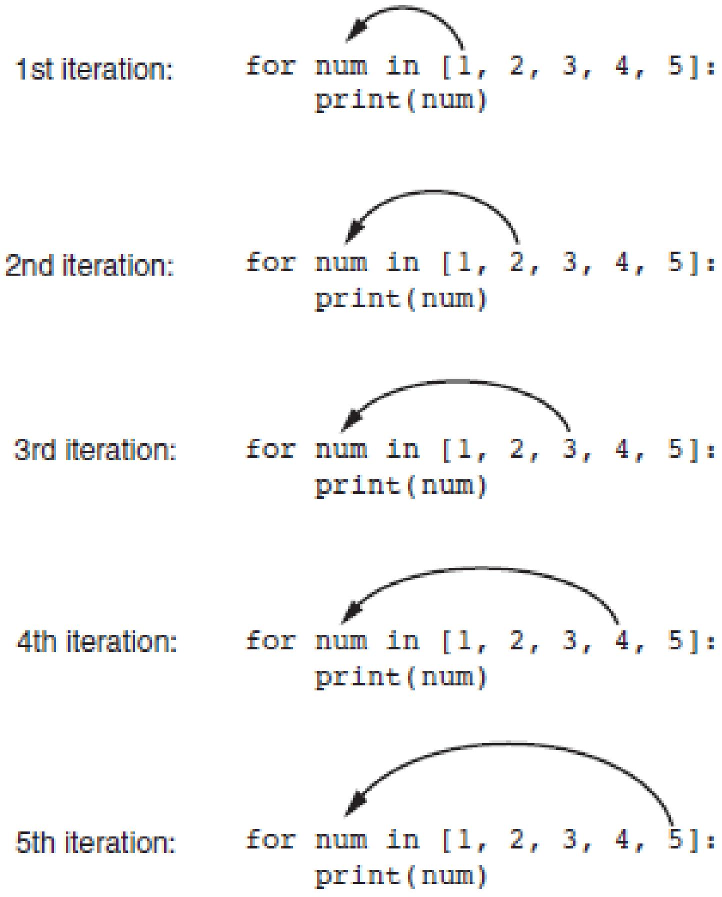
The `for` Loop: a Count-Controlled Loop

- **Count-Controlled loop: iterates a specific number of times**
  - Use a `for` statement to write count-controlled loop
    - Designed to work with sequence of data items
      - Iterates once for each item in the sequence
    - General format:

```
for variable in [val1, val2, etc]:
    statements
```

    - Target variable: the variable which is the target of the assignment at the beginning of each iteration

Figure 4-4 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)
```

Using the `range` Function with the `for` Loop

- **The `range` function simplifies the process of writing a `for` loop**
  - `range` returns an iterable object
    - Iterable: contains a sequence of values that can be iterated over
- **`range` characteristics:**
  - One argument: used as ending limit
  - Two arguments: starting value and ending limit
  - Three arguments: third argument is step value

Using the Target Variable Inside the Loop

- **Purpose of target variable is to reference each item in a sequence as the loop iterates**
- **Target variable can be used in calculations or tasks in the body of the loop**
  - Example: calculate square root of each number in a range

Letting the User Control the Loop Iterations

- **Sometimes the programmer does not know exactly how many times the loop will execute**
- **Can receive range inputs from the user, place them in variables, and call the `range` function in the for clause using these variables**
  - Be sure to consider the end cases: `range` does not include the ending limit

Generating an Iterable Sequence that Ranges from Highest to Lowest

- **The `range` function can be used to generate a sequence with numbers in descending order**
  - Make sure starting number is larger than end limit, and step value is negative
  - Example: `range (10, 0, -1)`

The Augmented Assignment Operators (cont’d.)

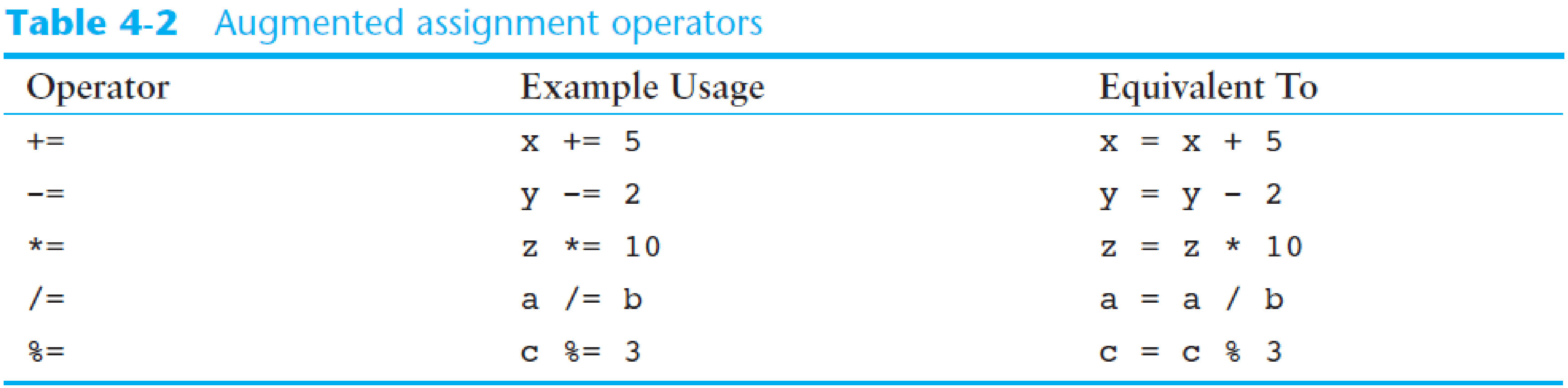


Table 4-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
`+=`	`x += 5`	`x = x + 5`
`-=`	`y -= 2`	`y = y - 2`
`*=`	`z *= 10`	`z = z * 10`
`/=`	`a /= b`	`a = a / b`
`%=`	`c %= 3`	`c = c % 3`

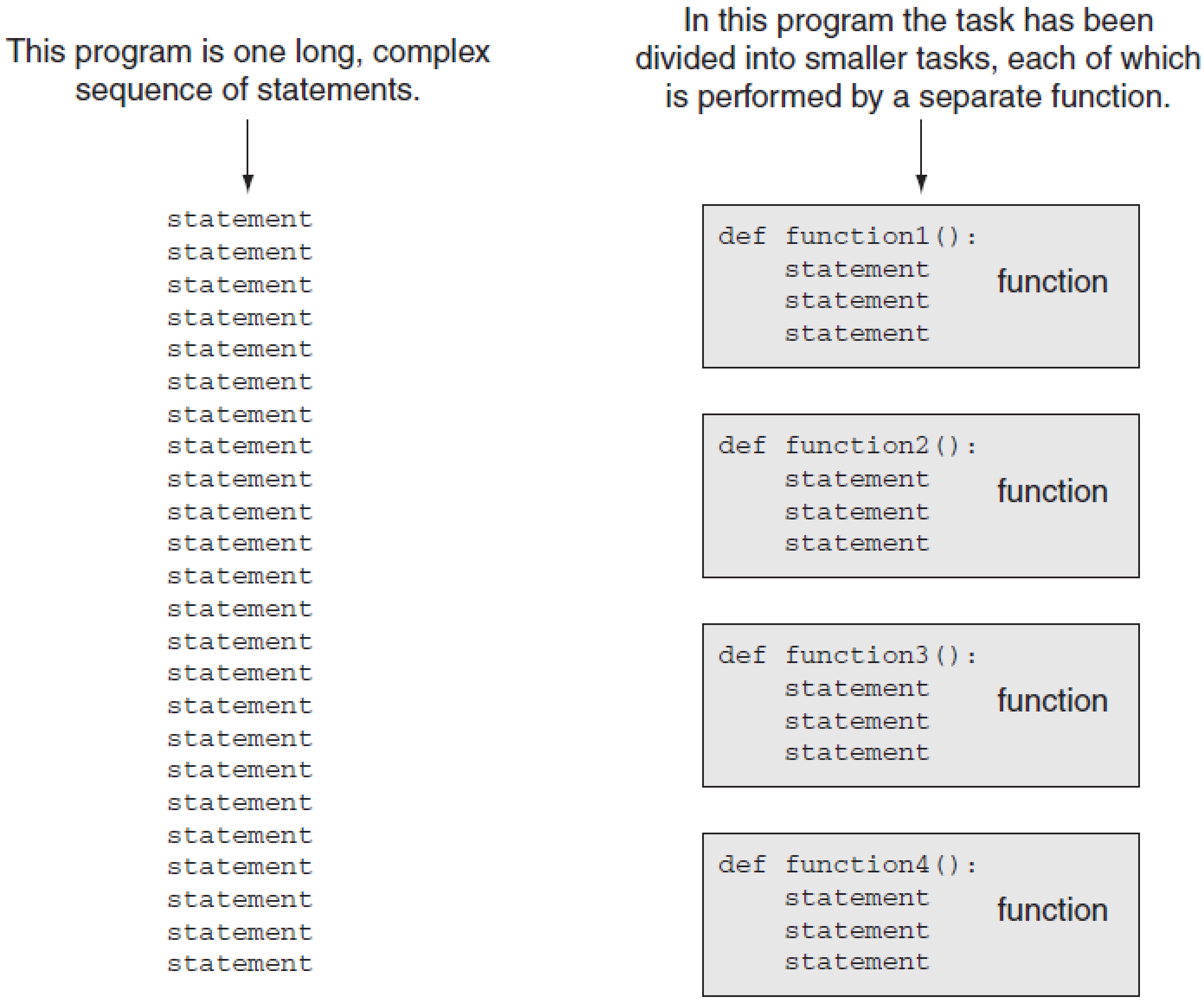
Sentinels

- **<u>Sentinel</u>: special value that marks the end of a sequence of items**
  - When program reaches a sentinel, it knows that the end of the sequence of items was reached, and the loop terminates
  - Must be distinctive enough so as not to be mistaken for a regular value in the sequence
  - Example: when reading an input file, empty line can be used as a sentinel

Introduction to Functions

- **Function: group of statements within a program that perform as specific task**
  - Usually one task of a large program
    - Functions can be executed in order to perform overall program task
  - Known as *divide and conquer* approach
- **Modularized program: program wherein each task within the program is in its own function**

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



This program is one long, complex sequence of statements.

```
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
```

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

```
def function1():
    statement
    statement
    statement
```

function

```
def function2():
    statement
    statement
    statement
```

function

```
def function3():
    statement
    statement
    statement
```

function

```
def function4():
    statement
    statement
    statement
```

function

Benefits of Modularizing a Program with Functions

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

Void Functions and Value-Returning Functions

- A <u>void function</u>:
  - Simply executes the statements it contains and then terminates.
- A <u>value-returning function</u>:
  - Executes the statements it contains, and then it returns a value back to the statement that called it.
    - The `input`, `int`, and `float` functions are examples of value-returning functions.

Defining and Calling a Function

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

Defining and Calling a Function (cont'd.)

- **Function name should be descriptive of the task carried out by the function**
  - Often includes a verb
- **<u>Function definition</u>: specifies what function does**

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

Defining and Calling a Function (cont'd.)

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

Defining and Calling a Function (cont’d.)

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

Defining and Calling a Function (cont'd.)

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

Indentation in Python

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

Designing a Program to Use Functions

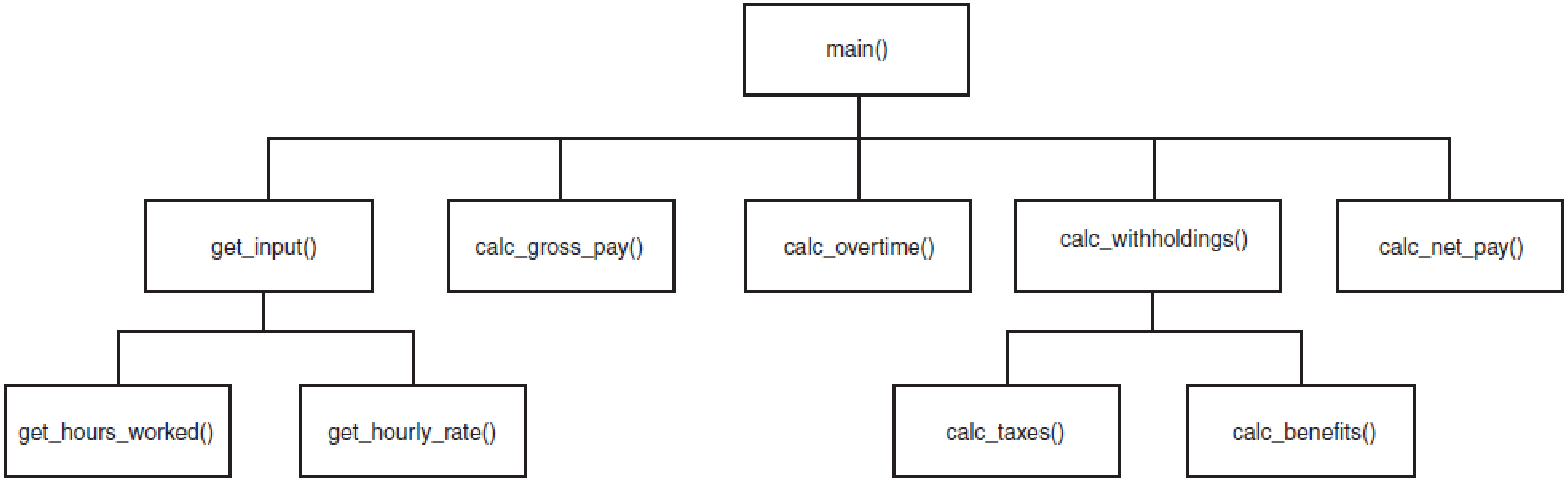
- **In a flowchart, function call shown as rectangle with vertical bars at each side**
  - Function name written in the symbol
  - Typically draw separate flow chart for each function in the program
    - End terminal symbol usually reads `Return`
- **<u>Top-down design</u>: technique for breaking algorithm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

Designing a Program to Use Functions (cont'd.)

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

Designing a Program to Use Functions (cont'd.)

Figure 5-10 A hierarchy chart



Local Variables

- **Local variable: variable that is assigned a value inside a function**
  - Belongs to the function in which it was created
    - Only statements inside that function can access it, error will occur if another function tries to access the variable
- **Scope: the part of a program in which a variable may be accessed**
  - For local variable: function in which created

Local Variables (cont'd.)

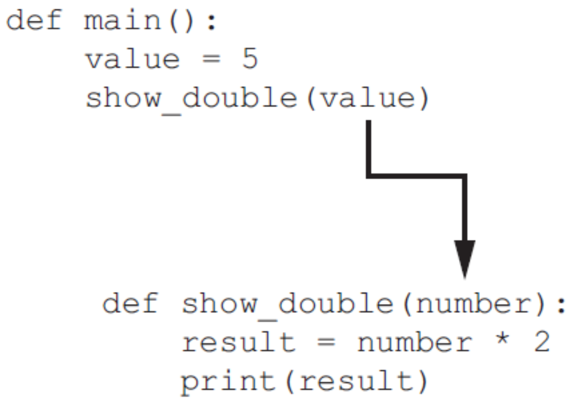
- **Local variable cannot be accessed by statements inside its function which precede its creation**
- **Different functions may have local variables with the same name**
  - Each function does not see the other function's local variables, so no confusion

Passing Arguments to Functions

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

Passing Arguments to Functions (cont'd.)

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



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

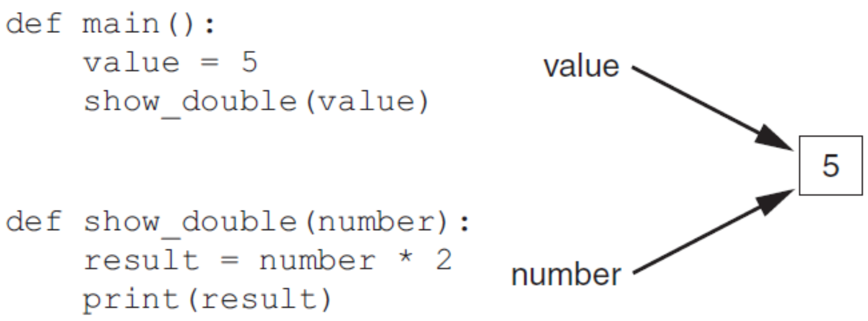
```
def show_double(number):
    result = number * 2
    print(result)
```

Passing Arguments to Functions (cont’d.)

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

Passing Arguments to Functions (cont’d.)

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



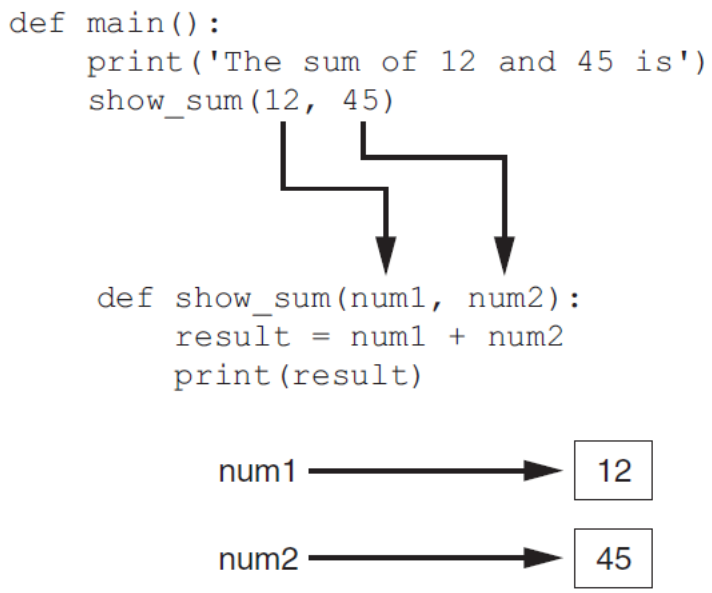
```
def main():
    value = 5
    show_double(value)

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

Passing Multiple Arguments

- **Python allows writing a function that accepts multiple arguments**
  - Parameter list replaces single parameter
    - Parameter list items separated by comma
- **Arguments are passed *by position* to corresponding parameters**
  - First parameter receives value of first argument, second parameter receives value of second argument, etc.

Passing Multiple Arguments (cont’d.)

Figure 5-16 Two arguments passed to two parameters



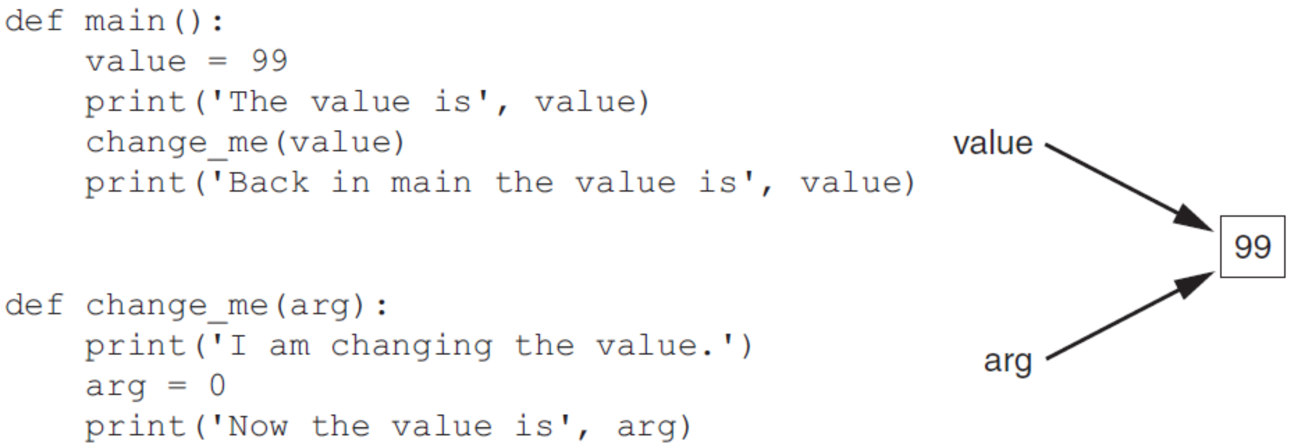
```
def main():
    print('The sum of 12 and 45 is')
    show_sum(12, 45)

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

Making Changes to Parameters

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

Making Changes to Parameters (cont'd.)

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

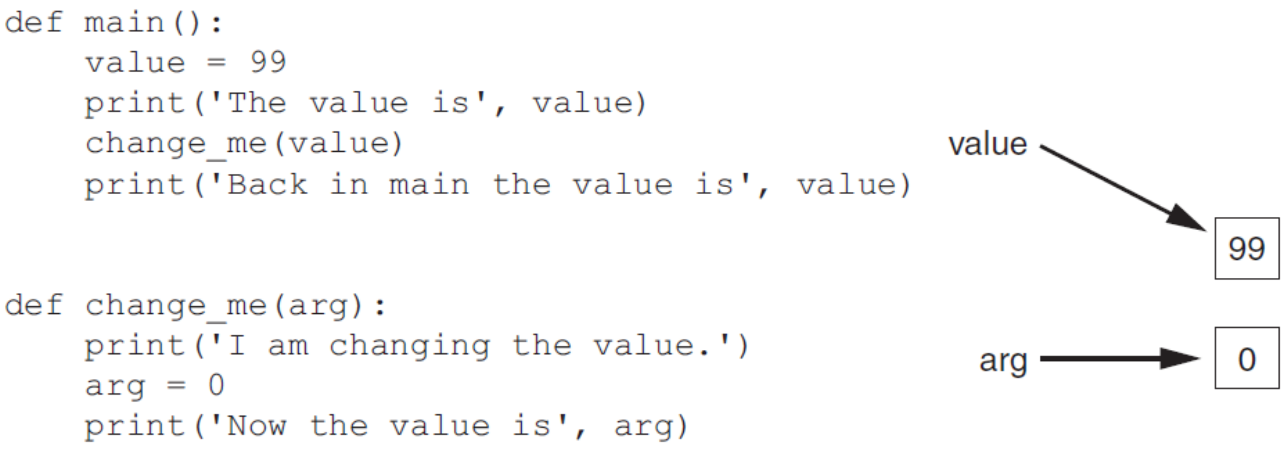


```
def main():
    value = 99
    print('The value is', value)
    change_me(value)
    print('Back in main the value is', value)


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

Making Changes to Parameters (cont'd.)

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

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



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

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

Keyword Arguments

- **Keyword argument: argument that specifies which parameter the value should be passed to**
  - Position when calling function is irrelevant
  - General Format:
  -

```
function_name(parameter=value)
```

- **Possible to mix keyword and positional arguments when calling a function**
  - Positional arguments must appear first

Global Variables and Global Constants

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

Global Variables and Global Constants (cont'd.)

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

Global Constants

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

Introduction to Value-Returning Functions: Generating Random Numbers

- **void function: group of statements within a program for performing a specific task**
  - Call function when you need to perform the task
- **Value-returning function: similar to void function, returns a value**
  - Value returned to part of program that called the function when function finishes executing

Standard Library Functions and the `import` Statement

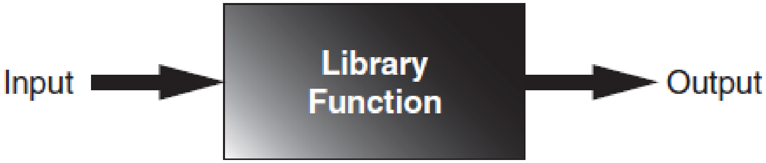
- **Standard library: library of pre-written functions that comes with Python**
  - *Library functions* perform tasks that programmers commonly need
    - Example: `print`, `input`, `range`
    - Viewed by programmers as a “black box”
- **Some library functions built into Python interpreter**
  - To use, just call the function

Standard Library Functions and the `import` Statement (cont'd.)

- **Modules: files that stores functions of the standard library**
  - Help organize library functions not built into the interpreter
  - Copied to computer when you install Python
- **To call a function stored in a module, need to write an `import` statement**
  - Written at the top of the program
  - Format: `import` *`module_name`*

Standard Library Functions and the `import` Statement (cont'd.)

Figure 5-19 A library function viewed as a black box



Generating Random Numbers

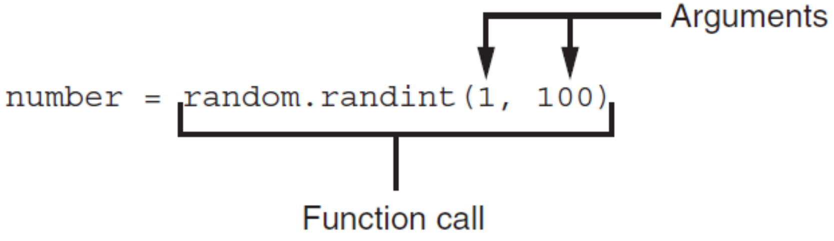
- **Random number are useful in a lot of programming tasks**
- **`random` module: includes library functions for working with random numbers**
- **Dot notation: notation for calling a function belonging to a module**
  - Format: `module_name.function_name()`

Generating Random Numbers (cont’d.)

- **`randint` function: generates a random number in the range provided by the arguments**
  - Returns the random number to part of program that called the function
  - Returned integer can be used anywhere that an integer would be used
  - You can experiment with the function in interactive mode

Generating Random Numbers (cont’d.)

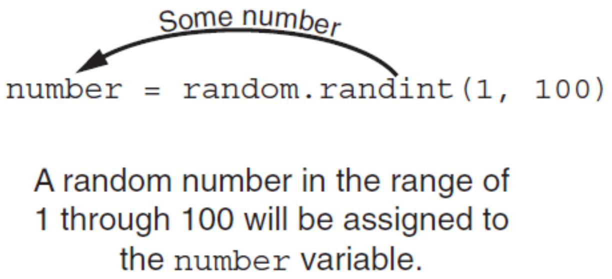
Figure 5-20 A statement that calls the `random` function



```
number = random.randint(1, 100)
```

Generating Random Numbers (cont’d.)

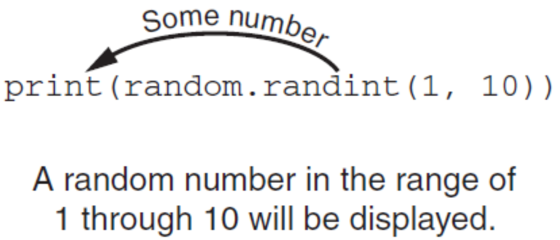
Figure 5-21 The `random` function returns a value



```
number = random.randint(1, 100)
```

A random number in the range of 1 through 100 will be assigned to the `number` variable.

Figure 5-22 Displaying a random number



```
print(random.randint(1, 10))
```

A random number in the range of 1 through 10 will be displayed.

Generating Random Numbers (cont'd.)

- **`randrange` function: similar to `range` function, but returns randomly selected integer from the resulting sequence**
  - Same arguments as for the `range` function
- **`random` function: returns a random float in the range of 0.0 and 1.0**
  - Does not receive arguments
- **`uniform` function: returns a random float but allows user to specify range**

Random Number Seeds

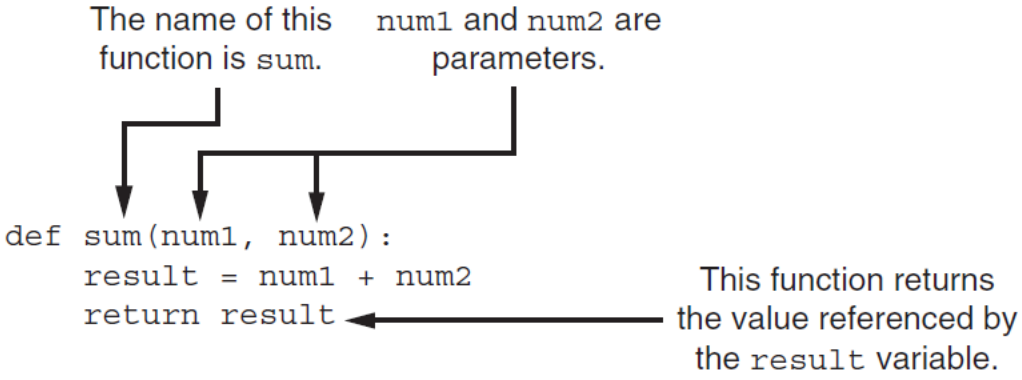
- **Random number created by functions in random module are actually pseudo-random numbers**
- **<u>Seed value</u>: initializes the formula that generates random numbers**
  - Need to use different seeds in order to get different series of random numbers
    - By default uses system time for seed
    - Can use `random.seed()` function to specify desired seed value

Writing Your Own Value-Returning Functions

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

Writing Your Own Value-Returning Functions (cont'd.)

Figure 5-23 Parts of the function



The name of this function is sum.

num1 and num2 are parameters.

```
def sum(num1, num2):
    result = num1 + num2
    return result
```

This function returns the value referenced by the result variable.

How to Use Value-Returning Functions

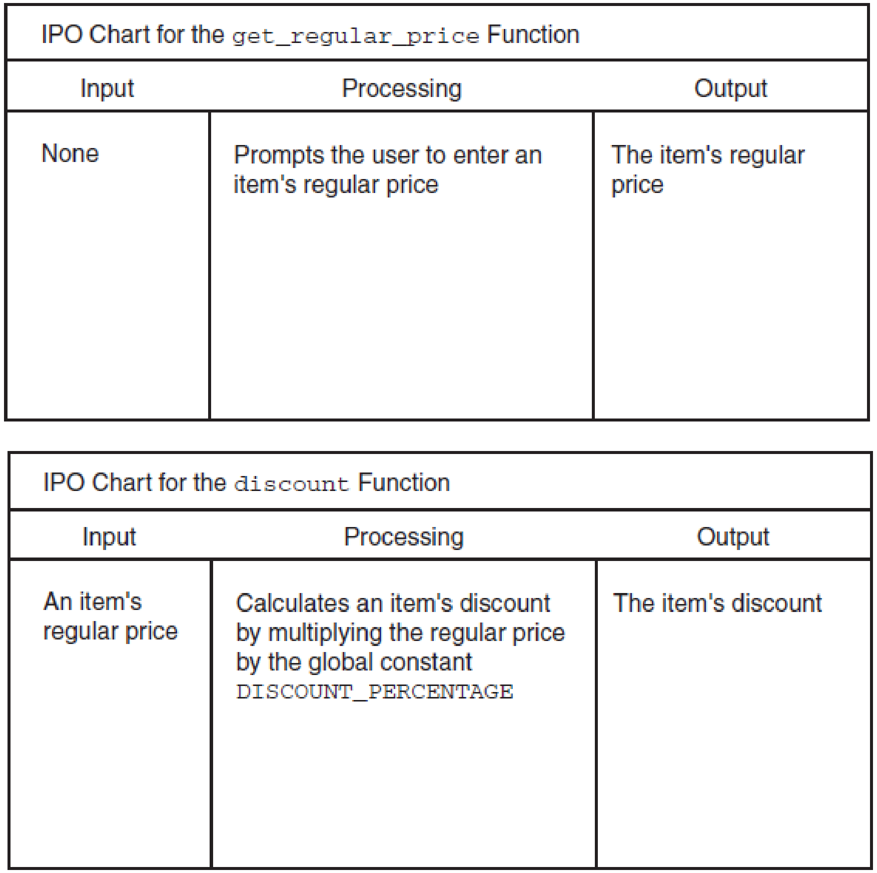
- **Value-returning function can be useful in specific situations**
  - Example: have function prompt user for input and return the user’s input
  - Simplify mathematical expressions
  - Complex calculations that need to be repeated throughout the program
- **Use the returned value**
  - Assign it to a variable or use as an argument in another function

Using IPO Charts

- **IPO chart: describes the input, processing, and output of a function**
  - Tool for designing and documenting functions
  - Typically laid out in columns
  - Usually provide brief descriptions of input, processing, and output, without going into details
    - Often includes enough information to be used instead of a flowchart

Using IPO Charts (cont’d.)

Figure 5-25 IPO charts for the `getRegularPrice` and `discount` functions



IPO Chart for the `get_regular_price` Function

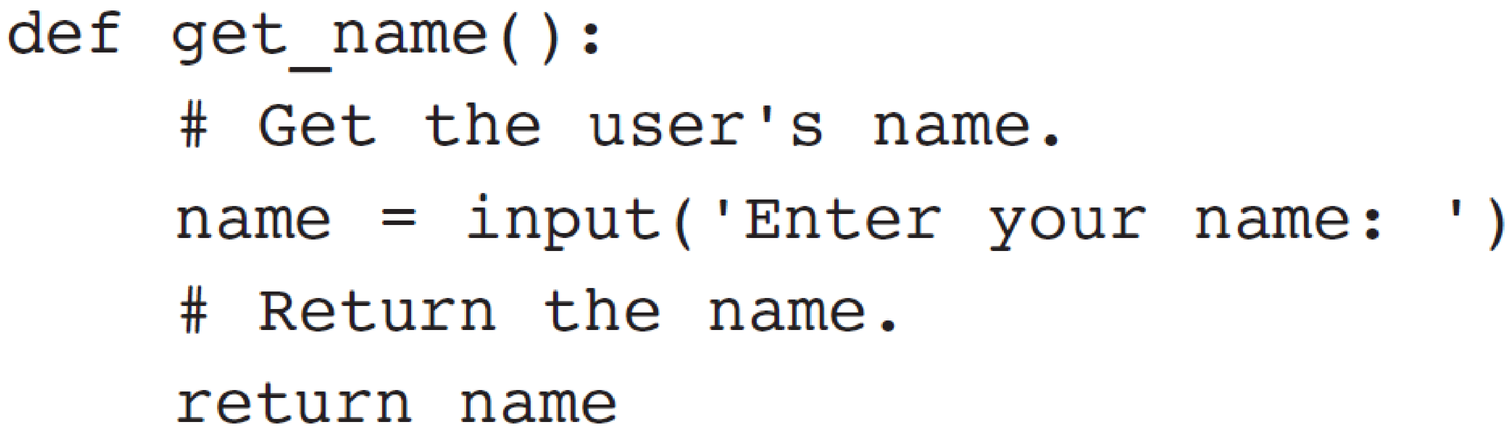
Input	Processing	Output
None	Prompts the user to enter an item's regular price	The item's regular price

IPO Chart for the `discount` Function

Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant `DISCOUNT_PERCENTAGE`	The item's discount

Returning Strings

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



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

Returning Boolean Values

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

Returning Multiple Values

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

The `math` Module

- **`math` module: part of standard library that contains functions that are useful for performing mathematical calculations**
  - Typically accept one or more values as arguments, perform mathematical operation, and return the result
  - Use of module requires an `import math` statement

The `math` Module (cont'd.)

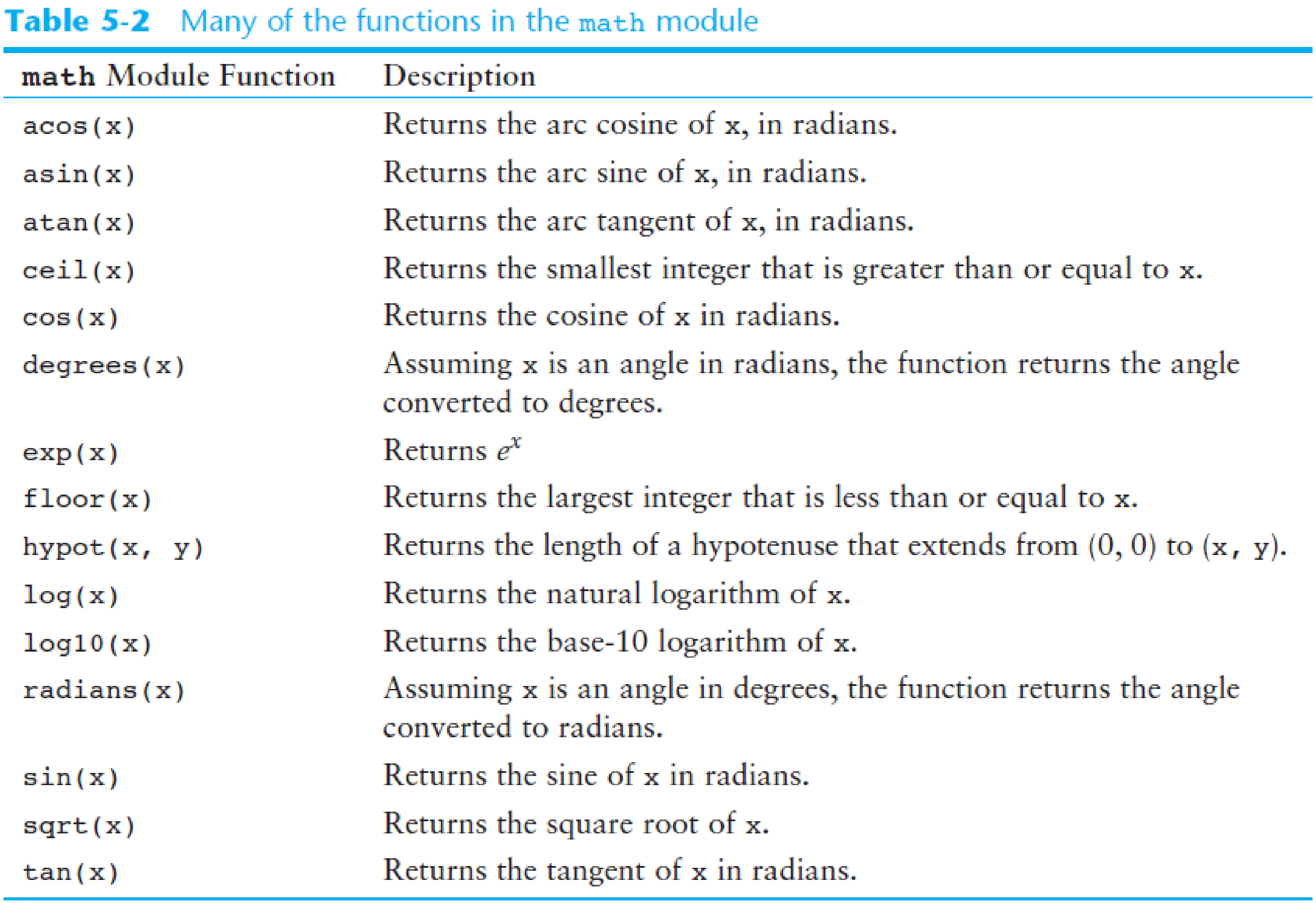


Table 5-2 Many of the functions in the `math` module

`math` Module Function	Description
`acos(x)`	Returns the arc cosine of x, in radians.
`asin(x)`	Returns the arc sine of x, in radians.
`atan(x)`	Returns the arc tangent of x, in radians.
`ceil(x)`	Returns the smallest integer that is greater than or equal to x.
`cos(x)`	Returns the cosine of x in radians.
`degrees(x)`	Assuming x is an angle in radians, the function returns the angle converted to degrees.
`exp(x)`	Returns e^x
`floor(x)`	Returns the largest integer that is less than or equal to x.
`hypot(x, y)`	Returns the length of a hypotenuse that extends from (0, 0) to (x, y).
`log(x)`	Returns the natural logarithm of x.
`log10(x)`	Returns the base-10 logarithm of x.
`radians(x)`	Assuming x is an angle in degrees, the function returns the angle converted to radians.
`sin(x)`	Returns the sine of x in radians.
`sqrt(x)`	Returns the square root of x.
`tan(x)`	Returns the tangent of x in radians.

The `math` Module (cont'd.)

- **The `math` module defines variables `pi` and `e`, which are assigned the mathematical values for *pi* and *e***
  - Can be used in equations that require these values, to get more accurate results
- **Variables must also be called using the dot notation**
  - Example:

```
circle_area = math.pi * radius**2
```

Storing Functions in Modules

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

Storing Functions in Modules (cont’d.)

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

Menu Driven Programs

- **Menu-driven program: displays a list of operations on the screen, allowing user to select the desired operation**
  - List of operations displayed on the screen is called a *menu*
- **Program uses a decision structure to determine the selected menu option and required operation**
  - Typically repeats until the user quits

Turtle Graphics: Modularizing Code with Functions

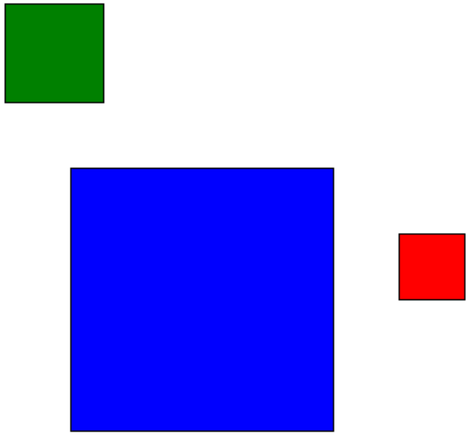
- **Commonly needed turtle graphics operations can be stored in functions and then called whenever needed.**
- **For example, the following function draws a square. The parameters specify the location, width, and color.**

```
def square(x, y, width, color):
    turtle.penup()            # Raise the pen
    turtle.goto(x, y)         # Move to (X,Y)
    turtle.fillcolor(color)   # Set the fill color
    turtle.pendown()          # Lower the pen
    turtle.begin_fill()       # Start filling
    for count in range(4):    # Draw a square
        turtle.forward(width)
        turtle.left(90)
    turtle.end_fill()         # End filling
```

Turtle Graphics: Modularizing Code with Functions

- **The following code calls the previously shown `square` function to draw three squares:**

```
square(100, 0, 50, 'red')
square(-150, -100, 200, 'blue')
square(-200, 150, 75, 'green')
```



Turtle Graphics: Modularizing Code with Functions

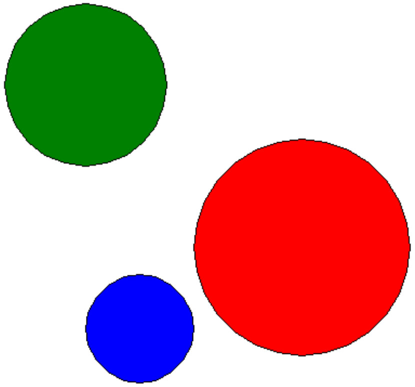
- **The following function draws a circle. The parameters specify the location, radius, and color.**

```
def circle(x, y, radius, color):
    turtle.penup()             # Raise the pen
    turtle.goto(x, y - radius) # Position the turtle
    turtle.fillcolor(color)    # Set the fill color
    turtle.pendown()           # Lower the pen
    turtle.begin_fill()        # Start filling
    turtle.circle(radius)      # Draw a circle
    turtle.end_fill()          # End filling
```

Turtle Graphics: Modularizing Code with Functions

- **The following code calls the previously shown `circle` function to draw three circles:**

```
circle(0, 0, 100, 'red')
circle(-150, -75, 50, 'blue')
circle(-200, 150, 75, 'green')
```



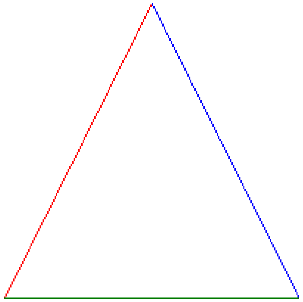
Turtle Graphics: Modularizing Code with Functions

- **The following function draws a line. The parameters specify the starting and ending locations, and color.**

```
def line(startX, startY, endX, endY, color):
    turtle.penup()              # Raise the pen
    turtle.goto(startX, startY) # Move to the starting point
    turtle.pendown()            # Lower the pen
    turtle.pencolor(color)      # Set the pen color
    turtle.goto(endX, endY)     # Draw a square
```

Turtle Graphics: Modularizing Code with Functions

- **The following code calls the previously shown `line` function to draw a triangle:**



```
TOP_X = 0
TOP_Y = 100
BASE_LEFT_X = -100
BASE_LEFT_Y = -100
BASE_RIGHT_X = 100
BASE_RIGHT_Y = -100
line(TOP_X, TOP_Y, BASE_LEFT_X, BASE_LEFT_Y, 'red')
line(TOP_X, TOP_Y, BASE_RIGHT_X, BASE_RIGHT_Y, 'blue')
line(BASE_LEFT_X, BASE_LEFT_Y, BASE_RIGHT_X, BASE_RIGHT_Y, 'green')
```