CHAPTER 5

Functions

Topics

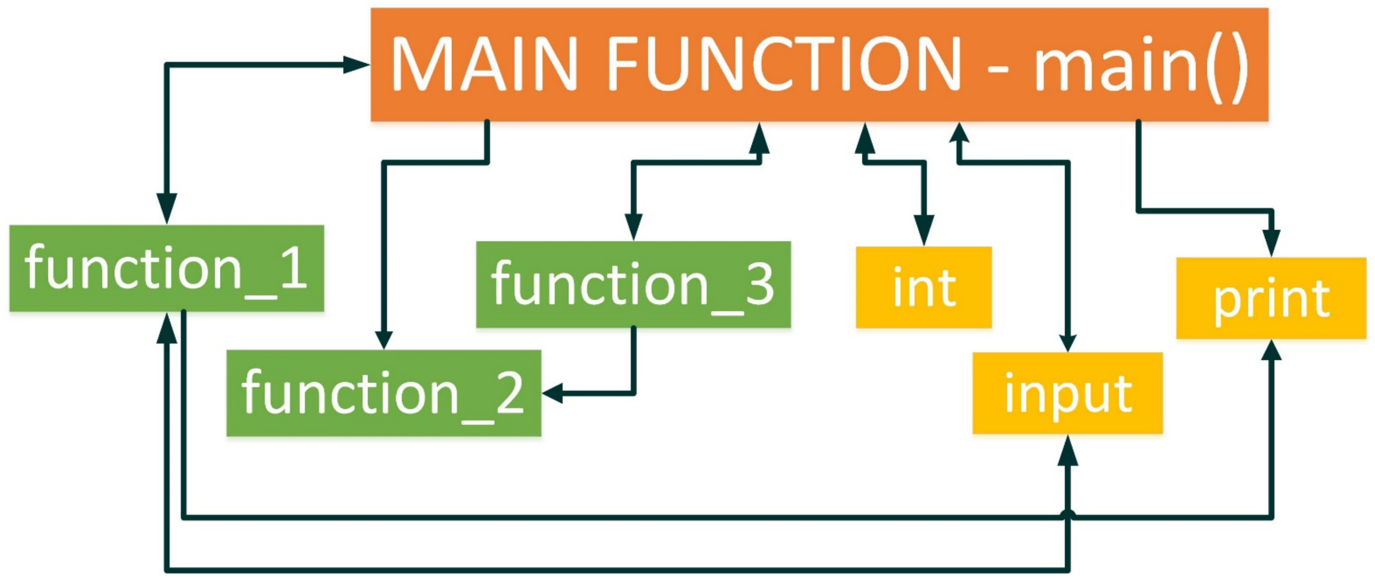
- Introduction to Functions
- Defining and Calling a Void Function
- Designing a Program to Use Functions
- Local Variables
- Passing Arguments to Functions
- Global Variables and Global Constants
- Turtle Graphics: Modularizing Code with Functions
- Introduction to Value-Returning Functions: Generating Random Numbers
- Writing Your Own Value-Returning Functions
- The `math` Module
- Storing Functions in Modules

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 is performed in its own function within the program**
  - This way program becomes more organized and clear.

Graphical Representation of Using Functions

- <u>Main Program</u> is like boss controlling everything
- Functions are like workers getting instructions from caller usually main also can be called by another function.
- <u>Each function</u> performs a specific task.
- Writing program with functions is called <u>modularization</u>.



CONCEPT: A function is a group of statements that exist within a program for the purpose of performing a specific task.

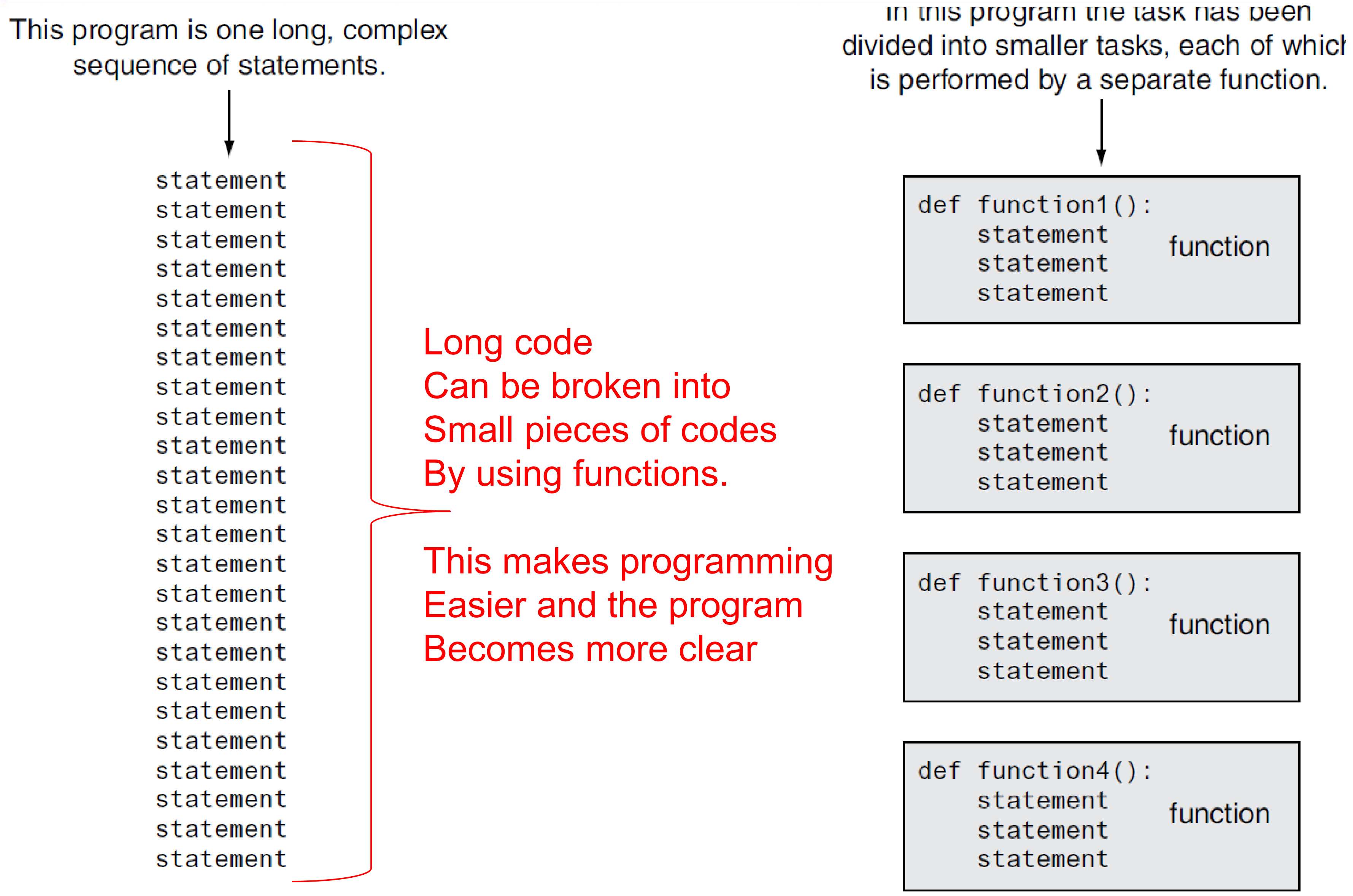
Example: In a real-world application, the overall task of calculating <u>an employee's pay</u> should consist of several subtasks, such as the following:

- Getting the employee's hourly pay rate
- Getting the number of hours worked
- Calculating the employee's gross pay
- Calculating overtime pay
- Calculating withholdings for taxes and benefits
- Calculating the net pay
- Printing the paycheck

A modularized program can be written with each task in its own function:

- A function that gets the employee's hourly pay rate
- A function that gets the number of hours worked
- A function that calculates the employee's gross pay
- A function that calculates the overtime pay
- A function that calculates the withholdings for taxes and benefits
- A function that calculates the net pay
- A function that prints the paycheck

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
```

Long code
Can be broken into
Small pieces of codes
By using functions.

This makes programming
Easier and the program
Becomes more clear

In this program the task has been divided into smaller tasks, each of whic 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
```

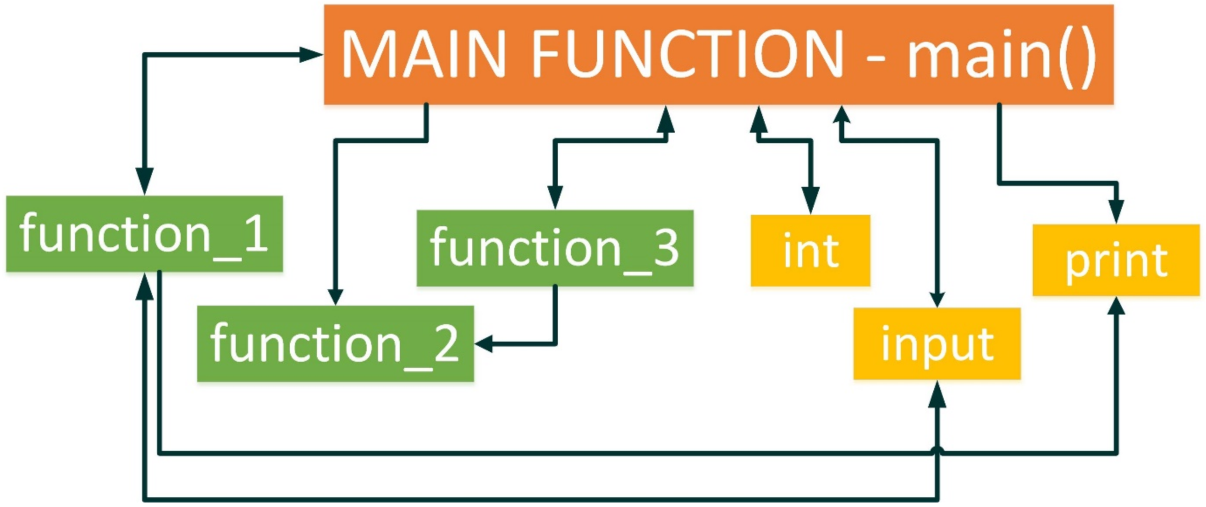
function

Advantages and 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
  - Easier testing and debugging
    - Can test and debug each function individually
  - Faster development
  - Better facilitation of teamwork
    - Different team members can write different functions

Void and Value-Returning Functions

- **A <u>void function</u>: - has no return value.**
  - Simply executes the statements it contains and then terminates. – `function_2` and `print` in the figure below
- **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`, `float`, and `format` functions are examples of value-returning functions.
    - Other than `function_2` and `print` in the figure below



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
- **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**
  - It is good to write # this function …. definition

Defining a Void Function

Python Syntax

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

- **Function header**: first line of function
  - Includes keyword `def` and function name, followed by parentheses and colon

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

$\left.\right\}$ Block belongs to function.
These statements are executed every time when the function is called by a caller.

- **Block**: set of statements that belong together as a group – same as the block idea in if and loops.

Defining and Calling a Function

- **Call a function to execute it**
  - Functions are called with it’s names.
  - If we need to send some arguments to the function then they are placed in the parenthesis separated with comma

    `print('Gaziantep')` is a function call example.
- **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
    - It continues from that point afterwards.

Example_1: Void Functions

```python
# This program demonstrates a function.
# First, we define functions
# Two functions are named message and print_line
def message():
    print('I am Ali,')
    print('I am a student at HKU.')
    print('I live in Gaziantep.')

def print_line():
    print('**********************')
```

Function `message`

Function `print_line`

```python
# Calling the functions
print_line()
message()
print_line()
```

`print_line()` → Calling `print_line` function.

`message()` → Calling `message` function.

`print_line()` → Calling `print_line` function again.

Program Output

```
**********************
I am Ali,
I am a student at HKU.
I live in Gaziantep.
**********************
```

Example_2: Using `main()` Function

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

```
# This program demonstrates a function.
# First, we define main function and other functions
# Main function is the mainline logic of the program
def main():
    print_line()
    message()
    print_line()

def message():
    print('I am Ali,')
    print('I am a student at HKU.')
    print('I live in Gaziantep.')

def print_line():
    print('**********************')

# Calling Main
main()
```

Function `main` **The program logic reside under main function.**

Function `message`

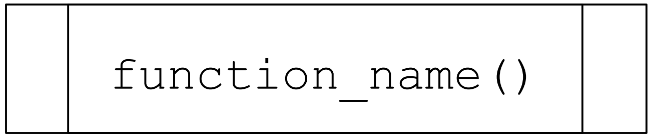
Function `print_line`

Calling `main` function – This is where the program starts.

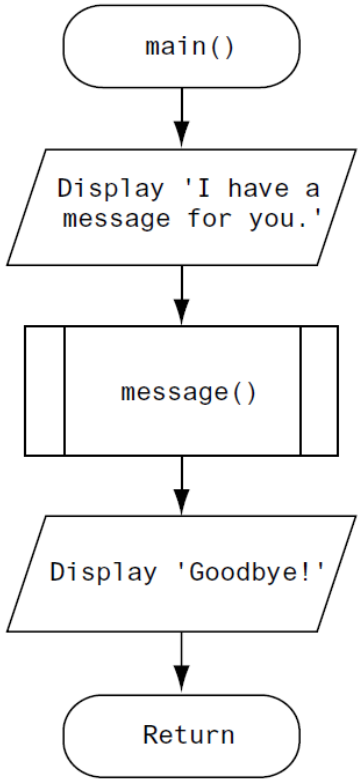
When the main function ends, the interpreter jumps back to the part of the program that called it.
There are no more statements, so the program ends.

Example_3: Functions in a Flowchart

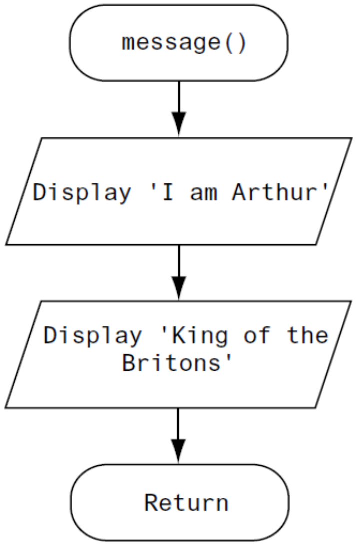
- **In a flowchart, function call shown as rectangle with vertical bars at each side**
  - Function name written in the symbol
  - Typically draw separate flow chart for each function in the program



Main Program



Message Function



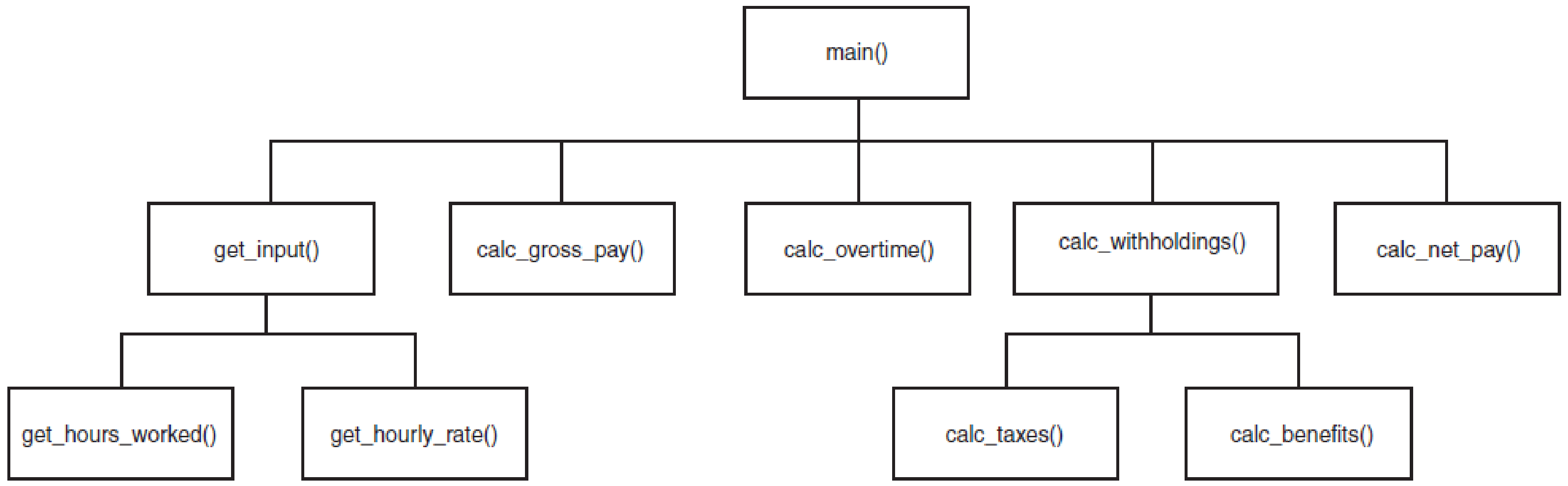
Python Code

```
# This program has two functions.
# First we define the main function.
def main():
    print('I have a message for you.')
    message()
    print('Goodbye!')

# Next we dene the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

# Call the main function.
main()
```

Designing a Program to Use Functions

- Hierarchy Chart: depicts relationship between functions
- Flowcharts are good tools for graphically depicting the flow of logic inside a function.
- Programmers commonly use hierarchy charts to represent the relationships between functions.
- Hierarchy chart are also known as structure charts.

Example: The hierarchy chart for a hypothetical pay calculating program.



Notice the hierarchy chart does not show the steps that are taken inside a function. Because they do not reveal any details about how functions work, they do not replace flowcharts or pseudocode.

Local Variables

- **<u>Local variable</u>: variable that is assigned a value inside a function and cannot be accessed outside of the 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
- **<u>Scope</u>: scope of a variable is the part of a program in which a variable may be accessed**
  - For local variable: scope is the function in which created

Example:

```
# Definition of the main function.
def main():
get_name()
print('Hello', name)
```

This causes an error!
name is not defined – Interpreter Gives Error!

```
# Definition of the get_name function.
def get_name():
name = input('Enter your name: ')
```

`name` variable is declared in `get_name function` so it is only accessible within this function

```
# Call the main function.
main()
```

Local Variables (cont'd.)

- **Local variable cannot be accessed by statements outside the function in which it is created**
- **Different functions may have local variables with the same name**
  - Each function does not see the other function's local variables, so no confusion

Example:

```
# This program demonstrates two functions that
# have local variables with the same name.

def main():
    # Call the texas function.
    texas()
    # Call the california function.
    california()

# Definition of the texas function. It creates
# a local variable named birds.
def texas():
    birds = 5000
    print('Texas has', birds, 'birds.')

# Definition of the california function. It also
# creates a local variable named birds.
def california():
    birds = 8000
    print('California has', birds, 'birds.')

# Call the main function.
main()
```

$\longrightarrow$ `birds` is a local variable to `texas` function.

$\longrightarrow$ `birds` is a local variable to `california` function.

Program Output

```
Texas has 5000 birds.
California has 8000 birds
```

Checkpoint

5.10 What is a local variable? How is access to a local variable restricted?

Any variable created inside a function is a local variable. A local variable is created inside a function and cannot be accessed by any statement that are outside the function.

5.11 What is a variable's scope?

Scope of a variable is the part of a program in which a variable may be accessed. A local variable's scope is the function in which the variable is created.

5.12 Is it permissible for a local variable in one function to have the same name as a local variable in a different function?

Yes, it is. Different functions can have local variables with the same names because the functions cannot see each other's local variables.

Passing Arguments to Functions

- **<u>Argument</u>: piece of data that is sent into a function by the caller.**
  - Arguments are used to pass information/data between functions.
  - Function can process the argument/s sent by caller.
  - Multiple arguments can be sent to a function (separated by comma).
  - When calling the function, the argument is placed in parentheses following the function name.

<u>Example:</u> Below statements demonstrate no or different number of arguments are passed to the function by the caller.

`print('Ali is',x,'years old.')` ->Passing three arguments.

`y = int(x)` -> Passing only one argument.

`b = int(input('Enter a number:'))` -> Passing only one argument.

`name = input('Enter your name:')` -> Passing only one argument.

`s = input( )` -> Passing no argument.

`message( )` -> Passing no argument.

`sum( x , y , z)` -> Passing three arguments.

`minimum( a , b)` -> Passing two argument.

Example: Passing Arguments to Functions

```
# This program demonstrates an argument being
# passed to a function.

def main():
    x = 5
    show_double(x)

# The show_double function accepts an argument
# and displays double its value.

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

# Calling the main function.
main()
```

Here `x` is the argument sent to `show_double` function.
Value of `x` variable is sent to the `show_double` function.

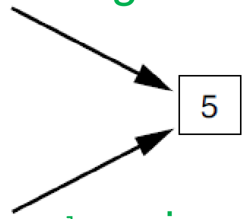
Program Output

```
10
```

Passing Arguments to Functions (Parameter?)

- **<u>Parameter variable</u>: variable that is assigned the value of an argument sent caller**
  - The parameter and the argument reference the same value
  - General format: `def` *`function_name`*`(`*`parameter`*`):`
  - <u>Scope of a parameter</u>: the function in which the parameter is used/created. They are local variables.

```
def main():
    x = 5
    show_double(x)    -> x is argument.

def show_double(number):    -> number is parameter.
    result = number * 2
    print(result)
```



`x` argument and `number` parameter variables references to the same value.

Passing Multiple Arguments

- **Python allows writing a function that accepts multiple arguments**
  - 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.

```
# This program demonstrates a function that accepts
# two arguments.

def main():
        a = 10
        print('Sum of',a ,'and',5,'is',end='  ')
        show_sum(a, 5)
                      Here values of a and 5 arguments  passed.
# The show_sum function accepts two arguments
# and displays their sum.
def show_sum(num1, num2): Two values 10 and 5 are accepted by num1 and num2
        result = num1 + num2
        print(result,'.')

# Call the main function.
main()
```

Making Changes to Parameters

- **Changes made to a parameter value within the function do not affect the value of the argument in the caller.**
  - Known as *pass by value*

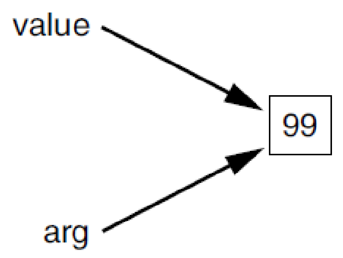
Example:

```
# This program demonstrates what happens when you
# change the value of a parameter
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)

# Call the main function.
main()
```

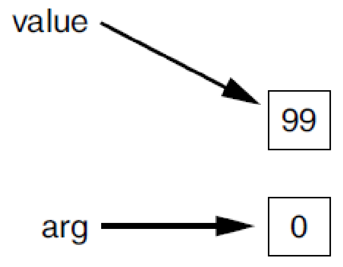
Program Output

```
The value is 99
I am changing the value.
Now the value is 0
Back in main the value is 99
```

Right at Calling Function



After `arg = 0`



Keyword Arguments

- **<u>Keyword argument:</u> argument that specifies which parameter the value should be passed to a function**
  - Normally parameters are assigned according to order
  - Position when calling function is irrelevant/unimportant

```
# This program demonstrates keyword arguments.

def main():
    # Show the amount of simple interest using 0.01 as
    # interest rate per period, 10 as the number of periods,
    # and $10,000 as the principal.
    show_interest(rate=0.01, periods=10, principal=10000.0)

# The show_interest function displays the amount of
# simple interest for a given principal, interest rate
# per period, and number of periods.

def show_interest(principal, rate, periods):
    interest = principal * rate * periods
    print('The simple interest will be $', \
          format(interest, ',.2f'), \
          sep='')

# Call the main function.
main()
```

Mixed Order!
Possible!

Checkpoint

Checkpoint
5.13 What are the pieces of data that are passed into a function called?
Arguments

5.14 What are the variables that receive pieces of data in a function called?
Parameter variables

5.15 What is a parameter variable's scope?
Parameter variables have a function scope. They only can be accessed within the function. They are local variables in the function they are created.

5.16 When a parameter is changed, does this affect the argument that was passed into the parameter?
No, it doesn't. Because parameter variables are local variables and have a function scope.

5.17 The following statements call a function named show_data. Which of the statements passes arguments by position, and which passes keyword arguments?
a. show_data(name='Kathryn', age=25) **->Passing arguments by keywords**
b. show_data('Kathryn', 25) **-> Passing arguments by position**

Global Variables 1/3

- **Global variable: created by assignment statement written outside all the functions – usually at the beginning**
  - Can be accessed by any statement in the program file, including from within a function – can not be modified.

Example:

```
# Defining a global variable
# This is done outside of all functions
number = 10

def main():
    function()
    print('In the Main:', number)


def function():
    print('In the Function:',number)

#Calling the main function
main()
```

$\rightarrow$ `number` is a global variable.
It is declared outside of all the functions.
it can be accessed from everywhere in the program.

Program Output

```
In the Function: 10
In the Main: 10
```

Global Variables 2/3

- **If a function needs to assign a value to the <u>global variable</u>, the global variable must be re-declared within the function**
  - General format: `global` *`variable_name`*

Example:

```
# Defining a global variable
# This is done outside of all functions
number = 10

def main():
    global number
    number *= 2
    function()
    number += 4
    print('In the Main:', number)


def function():
    global number
    number -= 3
    print('In the Function:',number)

#Calling the main function
main()
```

→ `number` is declared as a global variable.

→ `number` is re-declared in the main function as global variable. So main function can modify it / assign a value to it.

→ `number` is re-declared in the function as global variable. So the function can modify it / assign a value to it.

Program Output

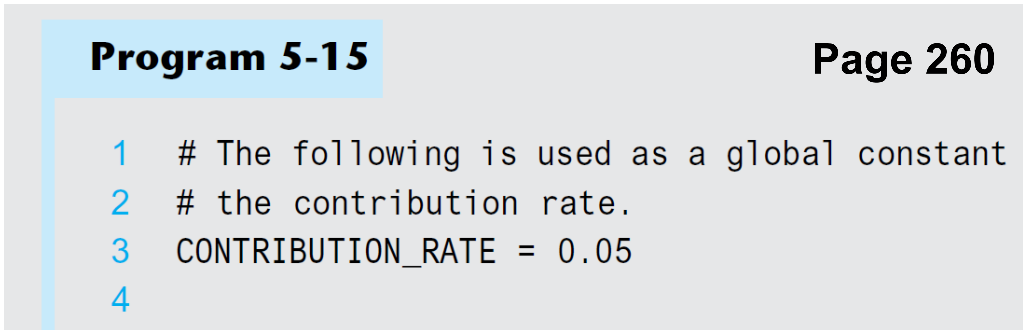
```
In the Function: 17
In the Main: 21
```

Global Variables 3/3

- **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

- **<u>Global constant</u>: global name that references a value that cannot be changed**
  - PI number or any other constants that should remain the same through out the program can be considered as global constant.
  - Although it is not advised to used global variables in a program, it is permissible to use global constants in a program
  - Normally Python doesn't allow creating true global constants.
  - <u>To simulate global constant in Python, create global variable and do not re-declare it within functions</u>



Program 5-15 **Page 260**

```
# The following is used as a global constant
# the contribution rate.
CONTRIBUTION_RATE = 0.05

```

`CONTRIBUTION_RATE` is only declared one time as a global variable outside all the functions, so it can not be change anywhere by any functions and it will remain constant all throughout the program.
It is a common practice to write a constant's name in all uppercase letters (similar to named constants). This serves as a reminder that the value referenced by the name is not to be changed in the program.

Introduction to Value-Returning Functions

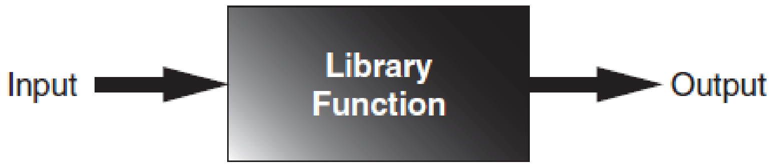
- **Void function: group of statements within a program for performing a specific task**
  - Call function when you need to perform the task
  - Do not return any value back to the caller
  - All the functions that we have written so far are void functions.
- **Value-returning function: written similar to void function, but returns a value back to the caller**
  - Value returned to part of program that called the function when function finishes executing
  - usually do some calculations and sent the result to caller.
- **We will first study the random module then we will study how to write a function with return value.**
  - **random module**
  - **math module**

  These two functions includes many functions with returning value. We will study math module later in this chapter.

Standard Library Functions

- **<u>Standard library</u>: library of pre-written functions that comes with Python**
  - *Library functions* perform tasks that programmers commonly need
    - https://docs.python.org/3/library/
    - Viewed by programmers as a “black box”

A library function viewed as a black box



- **Some library functions built into Python interpreter**
  - To use, just call the function
  - Example: `print`, `input`, `range`, `int`, `float`, `not`, etc.

The `import` Statement

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

<u>Example:</u>

```
import turtle
import math
import random
```

Generating Random Numbers

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

Example:

`turtle.circle() ->` **calling circle function under turtle module.**

`turtle.left() ->` **calling left function under turtle module.**

`xxx.yyy() ->` **calling yyy function under xxx module.**

Do not forget to import a module before calling the functions under it.

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

`grade = random.randint(1, 100)` – > Assigning a random number

The values 1 and 100 are included in the range.

`number = random.randint(1, 1000)` – > Assigning a random number

`dice = random.randint(1, 6)` – > Assigning a random number

`print(random.randint(1, 100))` – > Displaying a random number

  - You can experiment with the function in interactive mode

```
>>> import random Enter
>>> random.randint(1, 10) Enter
5
>>> random.randint(1, 100) Enter
98
>>> random.randint(100, 200) Enter
181
>>>
```

Example: Random Numbers–`randint` – 1/3

Write a program that displays 5 random numbers in the range from 1 to 100.

```
# This program displays five random
# numbers in the range of 1 through 100.
import random

def main():
    for count in range(5):        ⟶ Repeating 5 times.
        # Get a random number.
        number = random.randint(1, 100)
        # Display the number.
        print(number)

# Call the main function.
main()
```

could be done as `print(random.randint(1, 100))`

Program Output

```
90
79
2
82
88
```

Example: Random Numbers—`randint` – 2/3

Write a program that generates a desired number of numbers in the range of 1 to 10. The program displays the numbers and also shows the average of the numbers. You should have a function named `gen_num` receiving one argument from the caller which indicates the number of numbers to be processed.

```
# This program generates n numbers
# Displays the average of the numbers
import random

def main():
    non=int(input('How many numbers?'))
    gen_num(non) # Calling Function

def gen_num( n ):
    tot=0  # Accomolation variable
    for count in range(n):
        # Get a random number.
        number = random.randint(1, 10)
        tot+=number
        #Display the number
        print(number, end=' ')
    # Display the average.
    print('\nAverage is', format(tot/n,'.2f'))

# Call the main function.
main()
```

Main determines how many numbers to be processed

Calling `gen_num` number by passing `non`

Program Output

```
How many numbers?11
5 6 10 6 5 5 5 2 6 5 5
Average is 5.45
```

Example: Random Numbers–`randint` – 3/3

Write a game program in which user throws two dice 10 times. The program calculates the score of the user as the sum of the dice came out. The program should display the score at the end of the program.
Hint: Dice have 6 faces from 1-6 and dice throwing is random process.

Program Output

```
 1 -)   2 5
 2 -)   2 4
 3 -)   4 6
 4 -)   3 5
 5 -)   2 6
 6 -)   6 1
 7 -)   6 2
 8 -)   3 1
 9 -)   4 1
10 -)   1 5
Your Score is 69
```

```
# This program throw two dice 10 times
import random

def main():
    score = 0
    for count in range(10):
        # Throw the dice
        die_1 = random.randint(1, 6)
        die_2 = random.randint(1, 6)
        score = score + die_1 + die_2
        # Display the dice in tabular form.
        print(format(count+1,'2d'),'-) ',die_1,die_2)
    # Display the score.
    print('Your Score is',score)

# Call the main function.
main()
```

Repeating process 10 times

Generating two numbers – dice in the game adding them to the `score` accumulator variable

Displaying two dice results repeatedly.

Displaying score at the end – not repeated

Generating Random Numbers (cont'd.)

- **`randrange` function: used also generate integer numbers. It acts similar to `range` function, but returns randomly selected integer from the resulting sequence**
  - Arguments meanings are the same as for the `range` function
  - But the result is random among these values.

  - **One argument:** used as ending limit

    `random.randrange(10)` $\longrightarrow$ **[0,1,2,3,4,5,6,7,8,9]**
  - **Two arguments:** starting value and ending limit – not including ending value.

    `random.randrange(5,10)` $\longrightarrow$ **[5,6,7,8,9]**
  - **Three arguments:** third argument is step value

    `random.randrange(10,101,10)` $\longrightarrow$ **[10,20,30,……..,80,90,100]**

Generating Random Numbers (cont'd.)

- **`random` function: returns a random float in the range of 0.0 and 1.0 – 0.0 included but 1.0 not included.**
  - Does not receive arguments

$\texttt{random.random()} \longrightarrow$ **[0.0 , …, 0.99999999999999]**

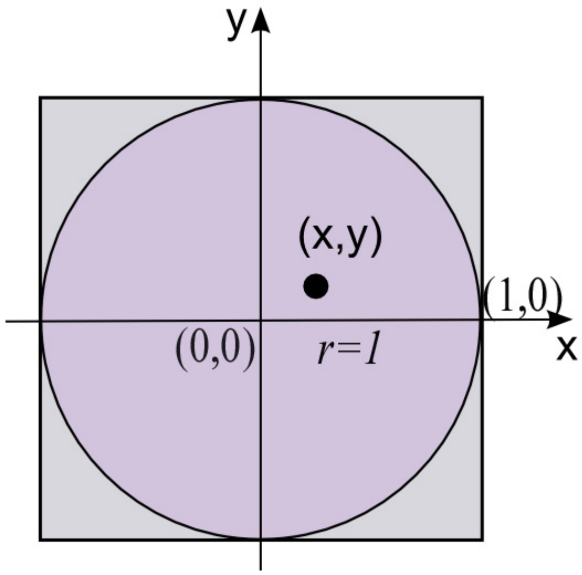
- **`uniform` function: returns a random float but allows user to specify range**

$\texttt{random.uniform(-10.0,10.0)} \longrightarrow$ **[-10.0 , … 0.0, … 10.0]**

$\texttt{random.uniform(-1.0,1.0)} \longrightarrow$ **[-1.0 , … 0.0, … 1.0]**

Example: Random Numbers–`randuniform`

Consider that you throw darts at a dart board randomly, size of 2 units each side. Then the probability of getting the darts into circular area shown in the figure is equal to ratio of circle to the circle of the square. The ratio of the area of the circle to the area of the square is π / 4. Write a python program that estimates the π number simulating this dart throw idea.



```
# This estimates the Pi number by using probability.
import random
NOT = 100000  # Number of Throws to be done
def main():
    print('    Throw        Pi   ')
    print(' --------  --------')
    in_circle=0 # Counting the ones falling inside
    for count in range(NOT+1):
        # Throw dart random number.
        x = random.uniform(-1.0, 1.0)
        y = random.uniform(-1.0, 1.0)
        # Check if within the circle.
        if (x**2 + y**2) <= 1.0:
            in_circle += 1
        if count % 10000 == 0 and count != 0:
                #Calculate Pi and print every 10000s
                #Pi = 4* Probabilty
                pi = 4.0 * in_circle/count
                print(format(count,'8d') , format(pi,'.7f'))
# Call the main function.
main()
```

Repeating NOT times

Generating two numbers
Checking if it is on the circle
Adding 1 to in_circle

Printing only …

Program Output

```
   Throw        Pi
 --------   -------
   10000    3.1500000
   20000    3.1520000
   30000    3.1574667
   40000    3.1475000
   50000    3.1452800
   60000    3.1454000
   70000    3.1426286
   80000    3.1438500
   90000    3.1456000
  100000    3.1460400
```

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

```python
# This program demonstrates the usage of random.seed() function
import random
random.seed(10) # Using Seed 10

def main():
    #Generating 10 numbers
    for i in range(10):
        print(random.randint(1, 100))

# Call the main function.
main()
```

Program Output

```
74
5
55
62
74
2
27
60
63
36
```

Run program again
Gives the same result
Every time you run.

Checkpoint

5.23 Why are library functions like “black boxes”?

Because we use them without looking into it in detail.

5.24 What does the following statement do?

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

Generates an integer number from 1 to 99 inclusively and assign it to x variable.

5.25 What does the following statement do?

```
print(random.randint(1, 20))
```

Generates an integer number from 1 to 19 inclusively and displays the result.

5.26 What does the following statement do?

```
print(random.randrange(10, 20))
```

Generates an integer number from 10 to 19 inclusively and displays the result.

5.27 What does the following statement do?

```
print(random.random())
```

Generates a float number 0 to 1 (excluding 1.0)

5.28 What does the following statement do?

```
print(random.uniform(0.1, 0.5))
```

Generates a float number 0.1 to 0.5

5.29 When the random module is imported, what does it use as a seed value for random number generation?

By default it uses the time of the computer

5.30 What happens if the same seed value is always used for generating random numbers?

Same seed value gives the same results at any run

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
  - The return expression could be one or multiple items of any type.
  - Usually return is done at the end of the function
  - Return can be done multiple places under certain circumstances.

Value-Returning Function Syntax in Python

```
def function_name ( parameter_list ):
        statement
        statement
        …
        return …
```

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 for displaying for further processing purpose

Example: Value Returning Function – 1/5

Write a program that sums the two numbers. In your program there should be a function named `sum` which sums the numbers and return the total to the caller. You program should display the result.

```
# This program uses the return value of a function.

def main():
    # Get the numbers.
    x = float(input('Enter the first number: '))   } Main inputs the numbers
    y = float(input('Enter the second number: '))  }

    # Get the sum of numbers.
    total = sum(x, y) → Calling sum by passing x and y then assign the result to total

    # Display the total.
    print('Summation is', total) → Displaying the result

# The sum function accepts two numeric arguments and
# returns the sum of those arguments.
def sum(num1, num2): → Receiving the values and storing them in num2 and num2
    result = num1 + num2 → Adding the numbers
    return result → Sending the result to the caller.

# Call the main function.
main()
```

Program Output

```
Enter the first number: 17.5
Enter the second number: 34
Summation is 51.5
```

Example: Value Returning Function – 2/5

Write a program that takes an integer from the user then displays if the entered number is EVEN or ODD. The program should include a function named `even_odd` which is called by main and it returns EVEN or ODD if the number is even or odd.

```python
# This program uses the return value of a function.
# Determines if an integer number is even or odd.

def main():
    # Get the number.
    x = int(input('Enter an integer number: '))

    # Display the result.
    print('Entered number',x,'is', even_odd(x))

# The even_odd function accepts one integer number
# It returns EVEN or ODD to the caller.
def even_odd(num):
    if num % 2 == 0:
        return 'EVEN'
    return 'ODD'

# Call the main function.
main()
```

Main Sending x to function
Displaying the result

Returning EVEN string literal if the condition is true

Returning ODD string literal if the condition is false

Program Output

```
Enter an integer number: 13
Entered number 13 is ODD
```

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

EXAMPLE:

IPO Chart for the `sum` function		
Input	**Processing**	**Output**
Two Numbers	Calculate the sum of the numbers	Total

IPO Chart for the `odd_even` function		
Input	**Processing**	**Output**
An integer Number	Check the condition number%2 == 0	EVEN or ODD

See Page 275 and 276 for more comprehensive Sale Price Program

Returning Strings

- `input` built-in function is a string returning function.

- You can write functions that return strings

EXAMPLE: `get_name` function takes the name from user with prompt and send it back to the caller.

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

Example: Value Returning Function – 3/5

Write a program that takes the name and last name of the user and display them in one line. Design two function `get_name` and `get_last` to get the inputs from the user.

```
# This program uses two functions returning string.

def main():
    name=get_name()
    last=get_last()
    print('Name and Last Name is',name,last)

def get_name():
    name=input('Enter the name:')
    return name

def get_last():
    return input('Enter the last name:')

# Call the main function.
main()
```

Main is calling functions then displaying the name information

Getting name as string from the user.

Returning it back to the caller.

Getting name as string from the user. Returning it to caller without assigning to any variables.

Program Output

```
Enter the name:Ahmet
Enter the last name:KHAN
Name and Last Name is Ahmet KHAN
```

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

Example:

Function return: Assume name of function is `funct_ex`

`return` *`exp1,exp2`*

Calling this function: It is possible to execute program in two ways

```
print(funct_ex(…))
var1 , var2 = funct_ex(…)
```

Example: Value Returning Function – 4/5

Write a program that takes the name and last name of the user and display them in one line. Design a function named `get_info` which gets the name and last names from the user then returns them to the caller.

```
# This program demonstrates the usage of
# multiple returning function

def main():
    firstname , lastname = get_info()
    print('Name and Lastname is',firstname,lastname)

# The get_info function accepts no arguments
# Returns two strings - name and last name.
def get_info():
    name=input('Enter the name:')
    last=input('Enter the last name:')
    return name,last

# Call the main function.
main()
```

Main is calling `get_info` function then displaying the name information

Getting names as strings from the user.
Returning both of them to caller.

Program Output

```
Enter the name:Ahmet
Enter the last name:KHAN
Name and Last Name is Ahmet KHAN
```

Returning Boolean Values

- **Boolean function: returns either `True` or `False`**
  - `isdown` and `isvisible` under `turtle` module are Boolean expression returning functions.
  - 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

Example:

Boolean Function returns True or False: One of the followings

```
return True
return False
```

Example: Value Returning Function

Write a program that takes an integer from the user then displays if the entered number is EVEN or ODD. The program should include a function named `is_even` which is called by main and it returns True or False if the number is even or odd, respectively.

```
# This program uses the return value of a function.
# is_even function returns Boolean value: True or False

def main():
    # Get the number.
    x = int(input('Enter an integer number: '))

    # Call function
    # Check the result - if it is True or False
    # Display even or odd
    if is_even( x ):
        print('Entered number',x,'is even.')
    else:
        print('Entered number',x,'is odd.')

# The even_odd function accepts one integer number
# It returns EVEN or ODD to the caller.
def is_even(num):
    if num % 2 == 0:
        return True
    return False

# Call the main function.
main()
```

Main sending x to function
Then checking
if it is true then displays Even
if it is False then displays Odd

Returning True

Returning False

Program Outputs

```
Enter an integer number: 87
Entered number 87 is odd.


Enter an integer number: -22
Entered number -22 is even.
```

Checkpoint

5.31 What is the purpose of the `return` statement in a function?

To return data to the caller. It can be used for many purposes.

5.32 Look at the following function definition:

```
def do_something(number):
      return number * 2
```

a. What is the name of the function?

`do_something`

b. What does the function do?

It returns the double of the number sent by the caller back to the caller.

c. Given the function definition, what will the following statement display?

```
      print(do_something(10))
```

20

5.33 What is a Boolean function?

A Function returns True or False – Boolean variable to the caller.

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 to the caller
  - Use of module requires an `import math` statement

The `math` Module (cont'd.)

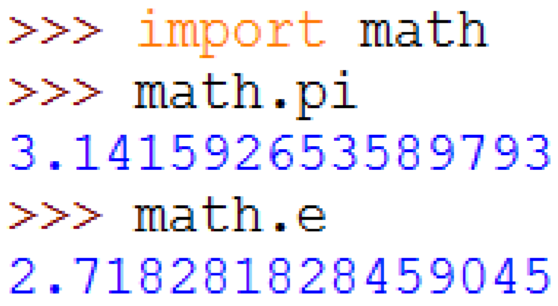
Table 5-2 Many of the functions in the `math` module

`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 Variables/Constants

- **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
```



```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

Storing Functions in Modules

- **In large, complex programs, it is important to keep code organized by modularizing the whole program.**
- **<u>Modularization</u>: 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
- **Module is a file that contains Python code**
  - Contains function definition but does not contain calls to the functions
  - After importing we can 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

- **<u>Menu-driven program</u>: 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

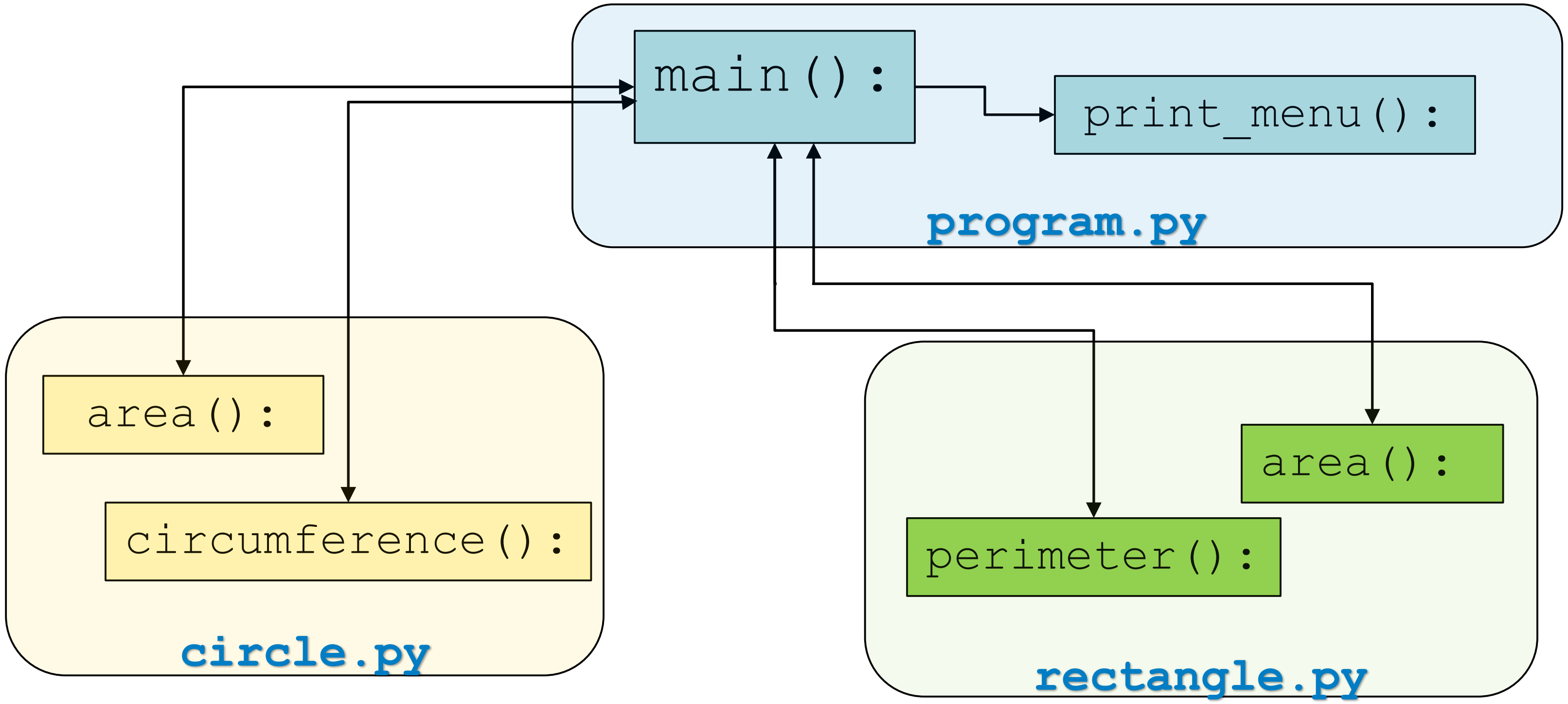
<u>Example:</u> Below is the menu section for multipurpose menu-driven program. The following menu repeated other than 1-5 is entered.

```
Program Menu
1.Draw a Circle
2.Draw a Square
3.Draw a Triangle
4.Draw a Line
5.Put a Dot
Please Select (1 - 5 something else to EXIT) :
```

Example: Modularized Menu-Driven Program 1/3

Write a program that calculates the areas and perimeters of a rectangle and circle. Modularize the program by including the functions which does the calculations for rectangle and circle by using the functions under the modules named `rectangle.py` and `circles.py`, respectively.

We can visualize the structure chart of the program as the followings



Example: Modularized Menu-Driven Program 2/3

Modules: circle and rectangle

```
# The circle module has functions that perform
# area and circumference for circles.

# The functions accepts
# a circle's radius as arguments.
# returns the calculated results to the caller.

import math

def area(radius):
    return math.pi * radius**2

def circumference(radius):
    return 2 * math.pi * radius
```

circle.py

`area` and `circumference` functions

```
# The rectangle module has functions that perform
# area and perimeter for rectangles.

# The functions accepts
# a rectangle's width and length as arguments.
# returns the calculated results to the caller.

def area(width, length):
    return width * length

def perimeter(width, length):
    return 2 * (width + length)
```

rectangle.py

`area` and `perimeter` functions

Example: Modularized Menu-Driven Program 3/3

```
import circle
import rectangle
# The main function.
def main():
    choice = 1 # Initially Choice is a valid option
    while choice >= 1 and choice <=4:
        display_menu() # display the menu.
        choice = int(input('Enter your choice: ')) # Get the user's choice
        # Perform the selected action accoring to the user's choice
        if choice == 1:
            radius = float(input("Enter the circle's radius: "))
            print('The area is', circle.area(radius))
        elif choice == 2:
            radius = float(input("Enter the circle's radius: "))
            print('The circumference is',circle.circumference(radius))
        elif choice == 3:
            width = float(input("Enter the rectangle's width: "))
            length = float(input("Enter the rectangle's length: "))
            print('The area is', rectangle.area(width, length))
        elif choice == 4:
            width = float(input("Enter the rectangle's width: "))
            length = float(input("Enter the rectangle's length: "))
            print('The perimeter is',rectangle.perimeter(width, length))
        else:
            print('Exiting the program. . .')

# The display_menu function displays a menu.
def display_menu():
    print('***** PROGRAM MENU *****')
    print('1) Area of a circle')
    print('2) Circumference of a circle')
    print('3) Area of a rectangle')
    print('4) Perimeter of a rectangle')
    print('Something Else to Quit')

# Call the main function.
main()
```

Importing the modules to access the functions included in these modules.

Repeating while $1 \leq \text{choice} \leq 4$

Displaying Menu
Getting New Choice
Performing Specific Task according to the choice of user

Exiting program if the user choice is not in the range.

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.**

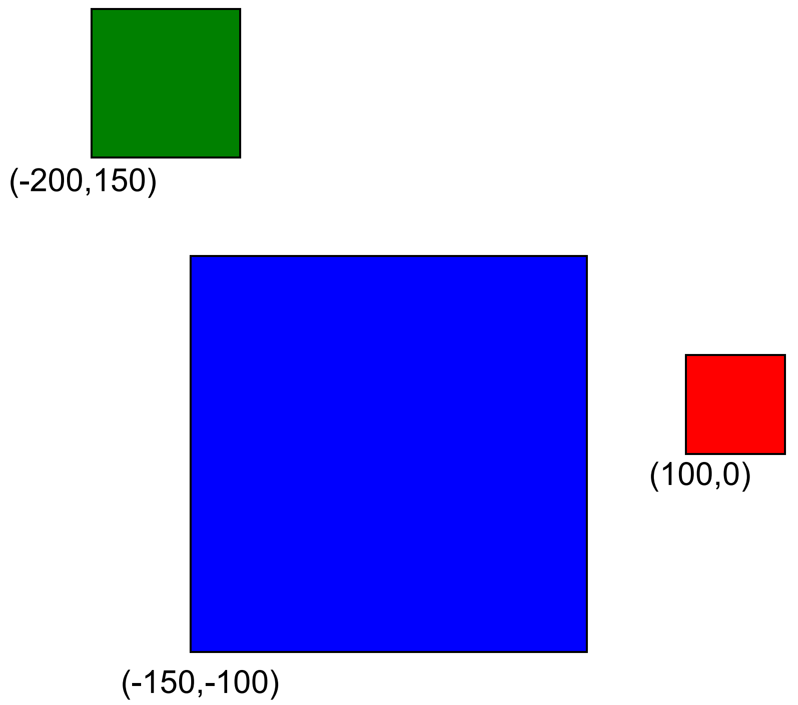
```
def square(x, y, width, color):
    turtle.penup()             # Raise the pen
    turtle.goto(x, y)          # Move to (X,Y)
    turtle.fillcolor(color)    # Set the fill color
    turtle.pendown()           # Lower the pen
    turtle.begin_fill()        # Start filling
    for count in range(4):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

Note: `turtle.setheading(0)` starts drawing square starting from the left bottom corner. So the x and y are the coordinates of left lower corner.
You may try to make your function such that x and y would be the center of the square.

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 (center of the circle), radius, and color.**

```
def circle(x, y, radius, color):
    turtle.setheading(0)         # Direction is set to EAST
    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
```

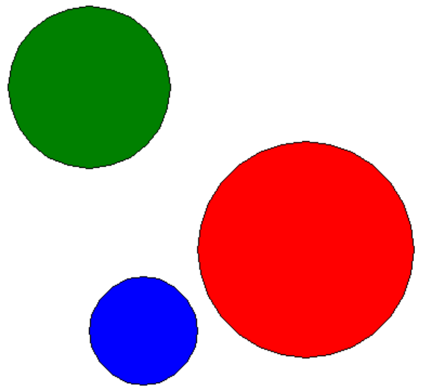
Turtle Graphics: Modularizing Code with Functions

- **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')
```

$\underbrace{\texttt{-200, 150,}}$

Fist two arguments are
the x and y coordinates of center of the circles



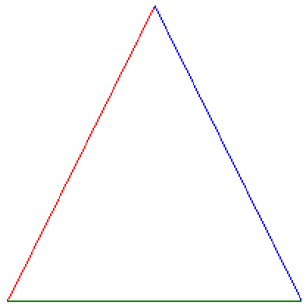
Turtle Graphics: Modularizing Code with Functions

- **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')
```

Review Questions

Page 297-298

Multiple Choice

3. The first line of a function definition is known as the __________.
   a. body
   b. introduction
   c. initialization
   [✓ d.] header
4. You __________ a function to execute it.
   a. define
   [✓ b.] call
   c. import
   d. export

6. A __________ is a diagram that gives a visual representation of the relationships between functions in a program.
   a. flowchart
   b. function relationship chart
   c. symbol chart
   [✓ d.] hierarchy chart
10. A(n) __________ is a special variable that receives a piece of data when a function is called.
    a. argument
    [✓ b.] parameter
    c. header
    d. packet
11. A variable that is visible to every function in a program file is a __________.
    a. local variable
    b. universal variable
    c. program-wide variable
    [✓ d.] global variable

Review Questions

Multiple Choice

12. When possible, you should avoid using __________ variables in a program.
    a. local
    [✓ b.] global
    c. reference
    d. parameter

14. This standard library function returns a random integer within a specified range of values.
    a. `random`
    [✓ b.] `randint`
    c. `random_integer`
    d. `uniform`

15. This standard library function returns a random floating-point number in the range of 0.0 up to 1.0 (but not including 1.0).
    [✓ a.] `random`
    b. `randint`
    c. `random_integer`
    d. `uniform`

17. This statement causes a function to end and sends a value back to the part of the program that called the function.
    a. `end`
    b. `send`
    c. `exit`
    [✓ d.] `return`

18. This is a design tool that describes the input, processing, and output of a function.
    a. hierarchy chart
    [✓ b.] IPO chart
    c. datagram chart
    d. data processing chart

Algorithm Workbench

9. Write a function named `times_ten` that accepts a number as an argument. When the function is called, it should return the value of its argument multiplied times 10.

```
def times_ten(number):
    return number * 10
```

10. Write a function named `is_valid_length` that accepts a string and an integer as arguments. When the function is called, it should return False if the length of the string is greater than the integer, otherwise it should return True.

```
def is_valid_length(text, limit):
    if len(text) > limit:
        return False
    else:
        return True
```

Programming Exercises

8. Paint Job Estimator

A painting company has determined that for every 112 square feet of wall space, one gallon of paint and eight hours of labor will be required. The company charges $35.00 per hour for labor. Write a program that asks the user to enter the square feet of wall space to be painted and the price of the paint per gallon.

The program should display the following data:

- The number of gallons of paint required
- The hours of labor required
- The cost of the paint
- The labor charges
- The total cost of the paint job

Given that every 112 square feet of wall space, one gallon of paint and eight hours of labor will be required.

=> 1 Gallon = 112 square feet = 8 hour. labor

The company charges $35.00 per hour for labor.

8. Paint Job Estimator – Program-

1 Gallon = 112 square feet = 8 hour Labor. 1 hour labor = $35.00.

```python
# Global constants for paint job estimator
FEET_PER_GALLON = 112
LABOR_HOURS = 8
LABOR_CHARGE = 35
```

Company dependent values are defined as Global Constants.

```python
def main(): # main function
    # Get wall space and paint price
    feetWall = float(input('Enter wall space in square feet: '))
    pricePaint = float(input('Enter paint price per gallon: '))
```

Inputting Wall Space and Paint Price / Gallon

```python
    # Calculate gallons of paint
    gallonPaint = int(feetWall / FEET_PER_GALLON) + 1
    # Calculate labor hours
    hourLabor = gallonPaint * LABOR_HOURS
    # Calculate labor charge
    costLabor = hourLabor * LABOR_CHARGE
    # Calculate paint cost
    costPaint = gallonPaint * pricePaint
```

Processing
Calculating gallon of paint needed.
Calculating Hour of Labor needed.
Calculating Labor Cost.
Calculating Paint Cost.

```python
    # Show cost estimate by calling the function showCostEstimate
    showCostEstimate(gallonPaint, hourLabor, costPaint, costLabor)
```

Sending processed info to function to display.

```python
# The showCostEstimate function accepts costs data as arguments
def showCostEstimate(gallonPaint, hourLabor, costPaint, costLabor):
    totalCost = 0.0
    # Calculate total cost
    totalCost = costPaint + costLabor
    # Display results
    print ('Gallons of paint: ', gallonPaint)
    print ('Hours of labor: ', hourLabor)
    print ('Paint charges: $' , format(costPaint, ',.2f'), sep='' )
    print ('Labor charges: $' , format(costLabor, ',.2f'), sep='')
    print ('Total cost: $' ,  format(totalCost, ',.2f'), sep='')
```

Processing info
Calculating Total
Displaying the results.

```python
 # Call the main function.
 main()
```

Programming Exercises

12. Maximum of Two Values

Write a function named `max2` that accepts two integer values as arguments and returns the value that is the greater of the two. For example, if 7 and 12 are passed as arguments to the function, the function should return 12. Use the function in a program that prompts the user to enter two integer values. The program should display the value that is the greater of the two.

Program

```
def main(): # Main Function
    # Get numbers from the user
    num1 = int(input('Enter number 1: '))
    num2 = int(input('Enter number 2: '))

    # Display result by sending numbers to max2 Function
    print ('The maximum number is:', max2(num1, num2))

# The max2 function returns the maximum
# of the two numbers received from the caller
def max2(x, y):
        if x > y:
            return x
        else:
            return y

# Call the main function.
main()
```

Program Output

```
Enter number 1: 12
Enter number 2: -6
The maximum number is: 12
```

Programming Exercises

When an object is falling because of gravity, the following formula can be used to determine the distance the object falls in a specific time period:

$$d = {}^{1}/_{2}\, gt^2$$

The variables in the formula are as follows: *d* is the distance in meters, *g* is 9.8, and *t* is the amount of time, in seconds, that the object has been falling.

Write a function named `falling_distance` that accepts an object's falling time (in seconds) as an argument. The function should return the distance, in meters, that the object has fallen during that time interval. Write a program that calls the function in a loop that passes the values 1 through 10 as arguments and displays the return value.

Program

```
# This Program Tabulates Falling distances 1-10s
def main(): # main function

    #Set up results chart Header
    print ('Time  Falling Distance')
    print ('----------------------')

    # loop on time (in seconds from 1 to 10)
    for t in range(1, 11):
        dist = falling_distance(t)
        print(format(t,'4d'), format(dist,'12.2f'))

# The falling_distance function receives the time
# Returns the distance it has fallen in that time
def falling_distance(time):
    fallDistance = (9.8 * time ** 2) / 2
    return fallDistance

# Call the main function.
main()
```

Program Output

```
Time  Falling Distance
----------------------------
   1        4.90
   2       19.60
   3       44.10
   4       78.40
   5      122.50
   6      176.40
   7      240.10
   8      313.60
   9      396.90
  10      490.00
```

Programming Exercises

16. Odd/Even Counter

In this chapter, you saw an example of how to write an algorithm that determines whether a number is even or odd. Write a program that generates 100 random numbers and keeps a count of how many of those random numbers are even, and how many of them are odd.

Program

```
import random
# main function
def main():
    odd_count = 0  #Accumulator Variable for odds
    even_count = 0 #Accumulator Variable for evens

    for counter in range(100): # Repeating 100 times

        # get random number in the range from 1-1000
        number = random.randint(1, 1000)
        # Check whether number is odd or even
        if isEven(number):
            even_count+=1
        else:
            odd_count+=1

    print ( 'Number of Odds:',odd_count)
    print ( 'Number of Evens:',even_count)

# The isEven function returns True if number is even, False if odd.
def isEven(num):
    if num % 2 == 0:
        return True
    else:
        return False

# Call the main function.
main()
```

Program Output

```
Number of Odds: 47
Number of Evens: 53
```

Programming Exercises

17. Prime Numbers

A prime number is a number that is only evenly divisible by itself and 1.

For example, the number 5 is prime because it can only be evenly divided by 1 and 5. The number 6, however, is not prime because it can be divided evenly by 1, 2, 3, and 6.

Write a Boolean function named `is_prime` which takes an integer as an argument and returns true if the argument is a prime number, or false otherwise. Use the function in a program that prompts the user to enter a number then displays a message indicating whether the number is prime.

TIP: Recall that the % operator divides one number by another and returns the remainder of the division. In an expression such as num1 % num2, the % operator will return 0 if num1 is evenly divisible by num2. This can be used to check if number is divisible to another number.

Program Output

```
Enter an integer: 17
17 is a prime number.
```

17. Prime Numbers – Program –

```python
#This program determines if a given number is a prime number or not.
def main(): # main function

    # Get number from user
    number = int(input('Enter an integer: '))

    # Display information regarding whether the number is prime
    if is_prime(number): # if True - number is prime
        print (number,'is a prime number.')
    else:
        print (number,'is not a prime number.')

# The is_prime function receives a number as an argument,
# and returns True if number is prime, False otherwise.
def is_prime(number):

    # Start from 2 up to Half of the number
    for i in range(2, int(number/2) + 1):
        if number % i == 0:
            return False  #If divisible then it is not prime

    return True  #Now Done with repetition - returning True
# Call the main function.
main()
```

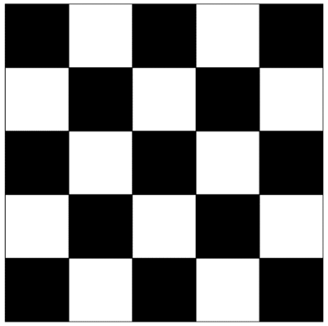
Program Output

```
Enter an integer: 17
17 is a prime number.
```

Programming Exercises

25. Turtle Graphics: Checkerboard

Write a turtle graphics program that uses the square function presented in this chapter, along with a loop (or loops) to draw the checkerboard pattern shown.



```
import turtle
def main():  # Main Function
    turtle.speed(5)
    color = 'black'     #Start color with black
# Draw the figure row by row from bottom to top.
    for y in range(-250, 250, 100):
        for x in range(-250, 250, 100):
            square(x, y, 100, color)
            if color == 'black':  #Convert to white if it is black
                color = 'white'
            else:
                color = 'black'
    turtle.hideturtle() # Hide the turtle

# The square function draws a square.
def square(x, y, width, color):
    turtle.setheading(0)       # Make sure direction is EAST
    turtle.penup()             # Raise the pen
    turtle.goto(x, y)          # Move to the specified location
    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

# Calling Main Function
main()
```

Summary

- **This chapter covered:**
  - The advantages of using functions
  - The syntax for defining and calling a function
  - Methods for designing a program to use functions
  - Use of local variables and their scope
  - Syntax and limitations of passing arguments to functions
  - Global variables, global constants, and their advantages and disadvantages
  - Value-returning functions, including:
    - Writing value-returning functions
    - Using value-returning functions
    - Functions returning multiple values
  - Using library functions and the `import` statement
  - Modules, including:
    - The `random` and `math` modules
    - Grouping your own functions in modules
  - Modularizing Turtle Graphics Code