CN101

Lecture 8-10

Functions

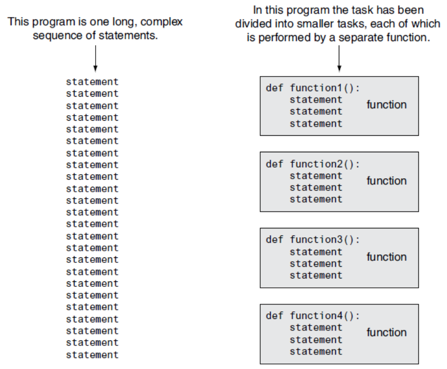
Topics

- 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
- Introduction to Value-Returning Functions: Generating Random Numbers
- Writing Your Own Value-Returning Functions
- The `math` Module

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

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
```

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

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

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

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

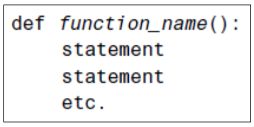
- A void function:
  - Simply executes the statements it contains and then terminates.
- A value-returning function:
  - Executes the statements it contains, and then it returns a value back to the statement that called it.
    - The `input`, `int`, and `float` functions are examples of value-returning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

Defining and Calling a Function

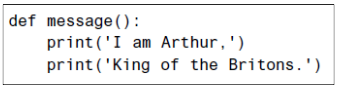
- Functions are given names
  - Function naming rules:
    - Cannot use key words as a function name
    - Cannot contain spaces
    - First character must be a letter or underscore
    - All other characters must be a letter, number or underscore
    - Uppercase and lowercase characters are distinct

Defining and Calling a Function (cont’d.)

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



```
def function_name():
    statement
    statement
    etc.
```



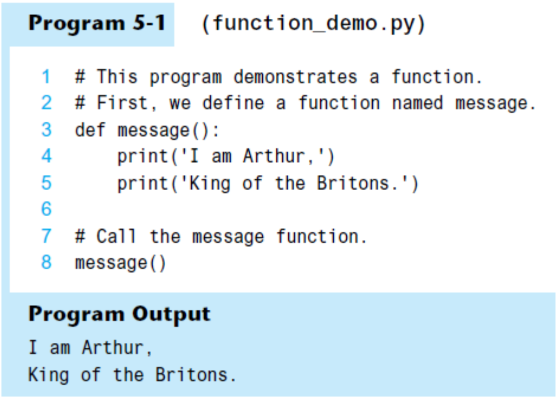
```
def message():
    print('I am Arthur,')
    print('King of the Britons.')
```

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



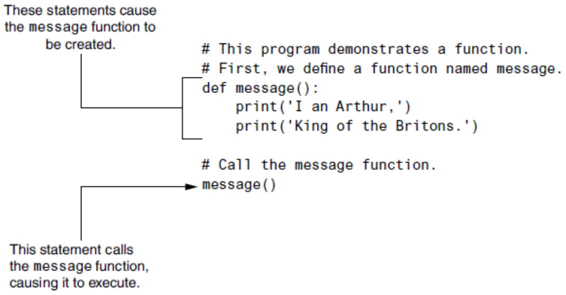
Program 5-1 (function_demo.py)

```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

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

Program Output

```
I am Arthur,
King of the Britons.
```

The function definition and the function call



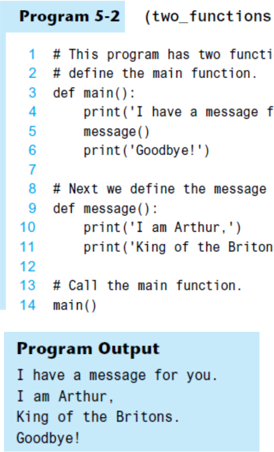
These statements cause the message function to be created.

```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I an Arthur,')
    print('King of the Britons.')

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

This statement calls the message function, causing it to execute.

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



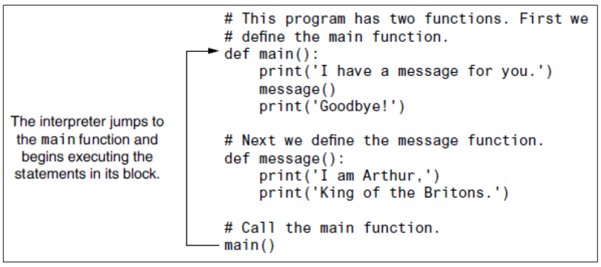
Program 5-2 (two_functions.py)

```
# 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 define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

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

Program Output

```
I have a message for you.
I am Arthur,
King of the Britons.
Goodbye!
```

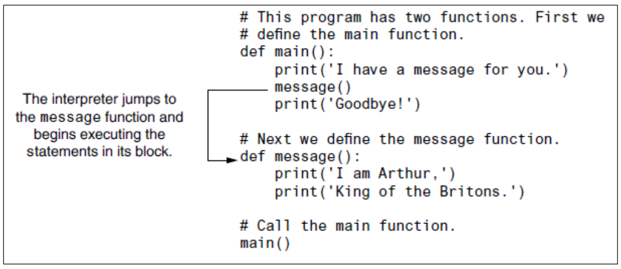
Calling the main function



The interpreter jumps to the main function and begins executing the statements in its block.

```
# 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 define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

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

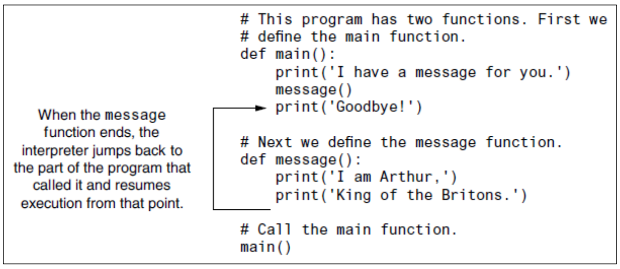
Calling the message function



The interpreter jumps to the message function and begins executing the statements in its block.

```
# 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 define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

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

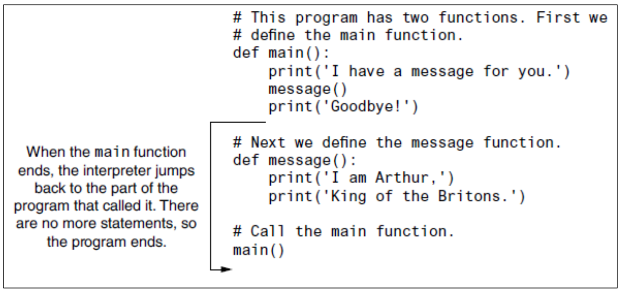
The message function returns



When the message function ends, the interpreter jumps back to the part of the program that called it and resumes execution from that point.

```
# 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 define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

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

The main function returns

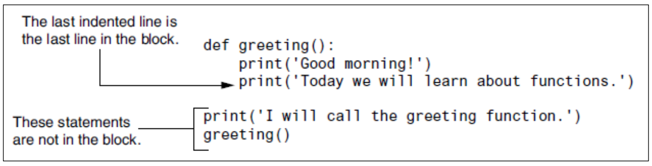


When the main function ends, the interpreter jumps back to the part of the program that called it. There are no more statements, so the program ends.

```
# 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 define the message function.
def message():
    print('I am Arthur,')
    print('King of the Britons.')

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

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



The last indented line is the last line in the block.

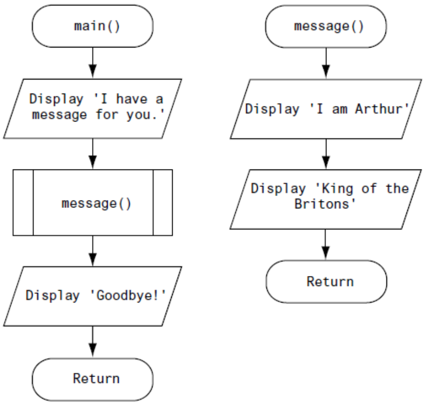
```
def greeting():
    print('Good morning!')
    print('Today we will learn about functions.')

print('I will call the greeting function.')
greeting()
```

These statements are not in the block.

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

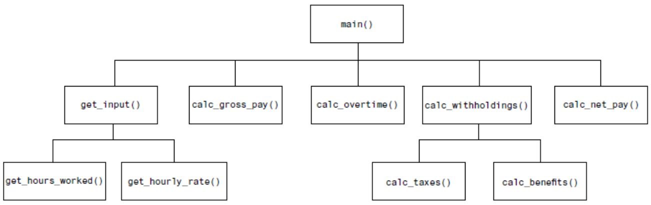
  - Typically draw separate flow chart for each function in the program
    - End terminal symbol usually reads `Return`



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

- Top-down design: technique for breaking algorithm into functions
- 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

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

Program 5-4 (bad_local.py)

```
# Definition of the main function.
def main():
    get_name()
    print('Hello', name)      # This causes an error!

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

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

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

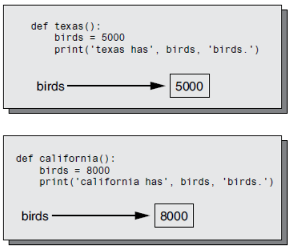
Program 5-5 (birds.py)

```
# 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()
```

Program Output

```
texas has 5000 birds.
california has 8000 birds.
```

Each function has its own birds variable



```
def texas():
    birds = 5000
    print('texas has', birds, 'birds.')
```

```
def california():
    birds = 8000
    print('california has', birds, 'birds.')
```

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
- 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

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

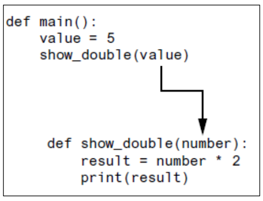
Program 5-6 (pass_arg.py)

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

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

# The show_double function accepts an argument
# and displays double its value.
def show_double(number):
    result = number * 2
    print(result)

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

Program Output

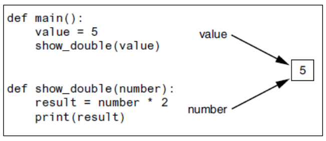
```
10
```

The value variable is passed as an argument



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

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

The value variable and the number parameter reference the same value



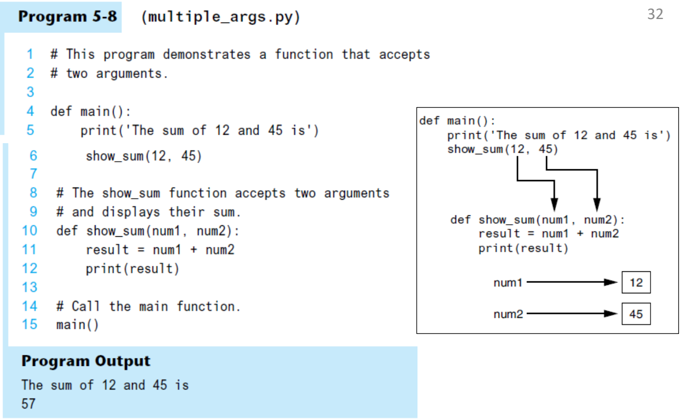
```
def main():
    value = 5
    show_double(value)

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

Passing 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 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.



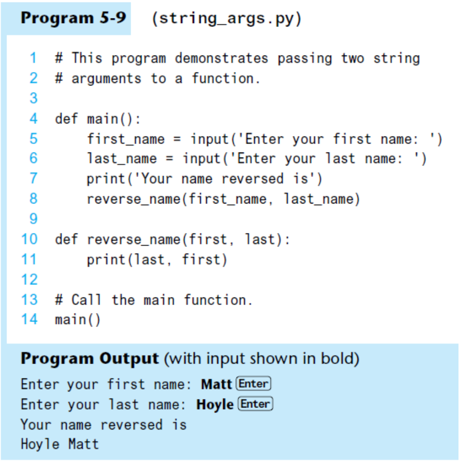
Program 5-8 (multiple_args.py)

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

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

# The show_sum function accepts two arguments
# and displays their sum.
def show_sum(num1, num2):
    result = num1 + num2
    print(result)

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

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


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

Program Output

```
The sum of 12 and 45 is
57
```



Program 5-9 (string_args.py)

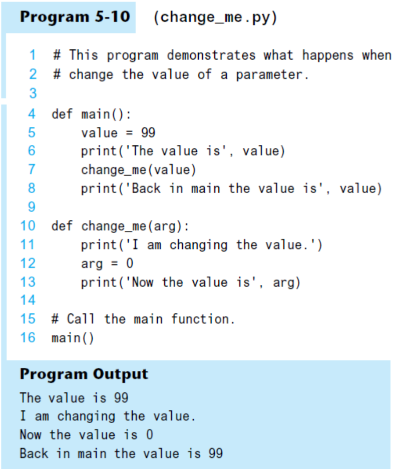
```
# This program demonstrates passing two string
# arguments to a function.

def main():
    first_name = input('Enter your first name: ')
    last_name = input('Enter your last name: ')
    print('Your name reversed is')
    reverse_name(first_name, last_name)

def reverse_name(first, last):
    print(last, first)

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

Program Output (with input shown in bold)

```
Enter your first name: Matt Enter
Enter your last name: Hoyle Enter
Your name reversed is
Hoyle Matt
```

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

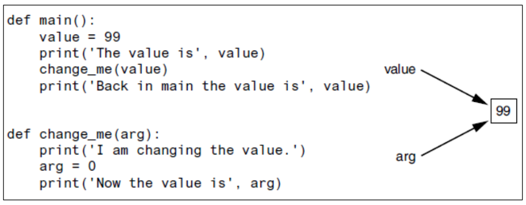


Program 5-10 (change_me.py)

```
# 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()
```

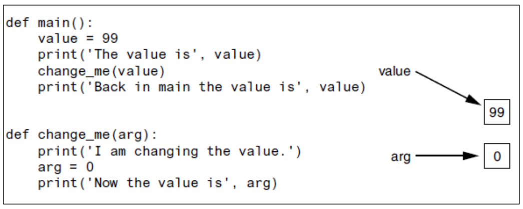
Program Output

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



```
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)
```

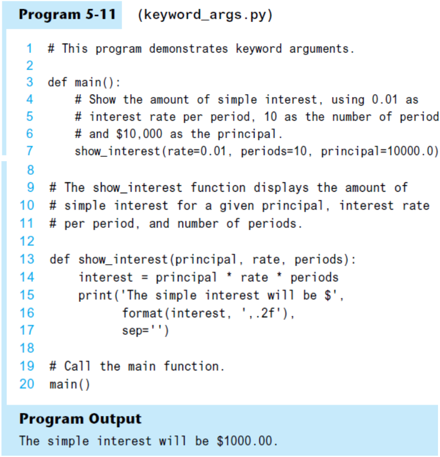
The `value` variable passed to the `change_me` function cannot be changed by it



```
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

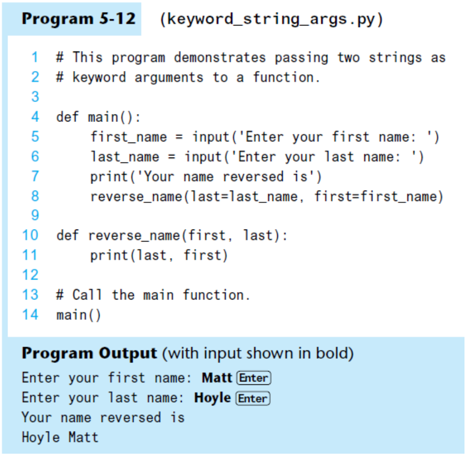


Program 5-11 (keyword_args.py)

```
# 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()
```

Program Output

```
The simple interest will be $1000.00.
```



Program 5-12 (keyword_string_args.py)

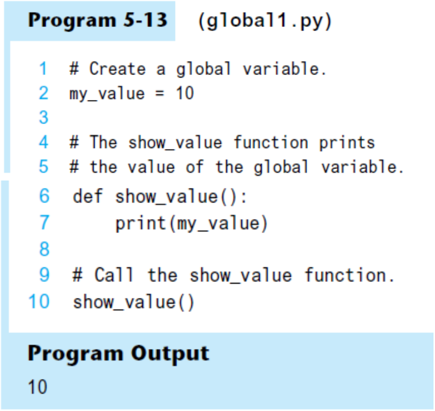
```
# This program demonstrates passing two strings as
# keyword arguments to a function.

def main():
    first_name = input('Enter your first name: ')
    last_name = input('Enter your last name: ')
    print('Your name reversed is')
    reverse_name(last=last_name, first=first_name)

def reverse_name(first, last):
    print(last, first)

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

Program Output (with input shown in bold)

```
Enter your first name: Matt Enter
Enter your last name: Hoyle Enter
Your name reversed is
Hoyle Matt
```

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`*

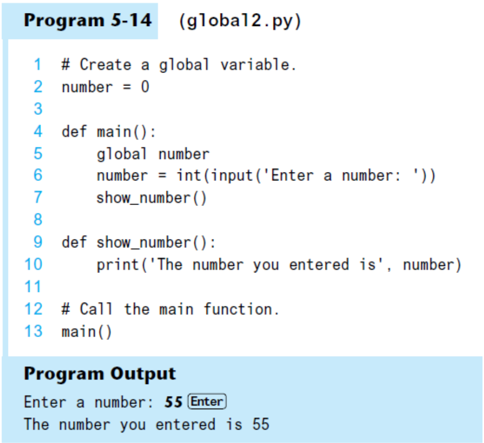


Program 5-13 (global1.py)

```
# Create a global variable.
my_value = 10

# The show_value function prints
# the value of the global variable.
def show_value():
    print(my_value)

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

Program Output

```
10
```



Program 5-14 (global2.py)

```
# Create a global variable.
number = 0

def main():
    global number
    number = int(input('Enter a number: '))
    show_number()

def show_number():
    print('The number you entered is', number)

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

Program Output

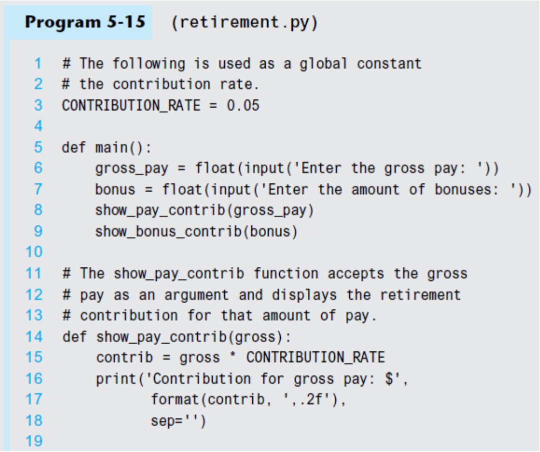
```
Enter a number: 55 Enter
The number you entered is 55
```

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

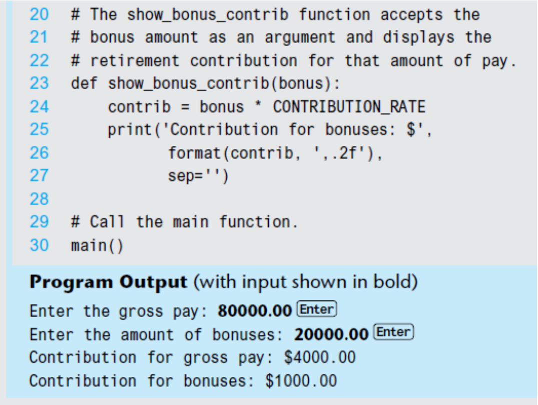
Global Constants

- 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



Program 5-15 (retirement.py)

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

def main():
    gross_pay = float(input('Enter the gross pay: '))
    bonus = float(input('Enter the amount of bonuses: '))
    show_pay_contrib(gross_pay)
    show_bonus_contrib(bonus)

# The show_pay_contrib function accepts the gross
# pay as an argument and displays the retirement
# contribution for that amount of pay.
def show_pay_contrib(gross):
    contrib = gross * CONTRIBUTION_RATE
    print('Contribution for gross pay: $',
          format(contrib, ',.2f'),
          sep='')

```



```
# The show_bonus_contrib function accepts the
# bonus amount as an argument and displays the
# retirement contribution for that amount of pay.
def show_bonus_contrib(bonus):
    contrib = bonus * CONTRIBUTION_RATE
    print('Contribution for bonuses: $',
          format(contrib, ',.2f'),
          sep='')

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

Program Output (with input shown in bold)

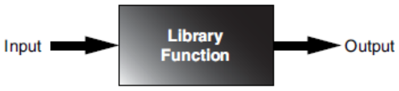
```
Enter the gross pay: 80000.00 Enter
Enter the amount of bonuses: 20000.00 Enter
Contribution for gross pay: $4000.00
Contribution for bonuses: $1000.00
```

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`*

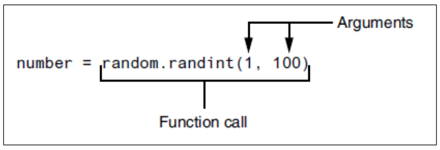
Generating Random Numbers

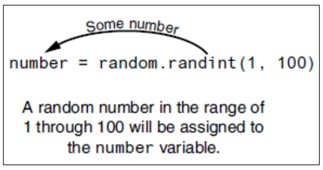
- Random number are useful in a lot of programming tasks
- `random` module: includes library functions for working with random numbers
- Dot notation: notation for calling a function belonging to a module
  - Format: `module_name.function_name()`

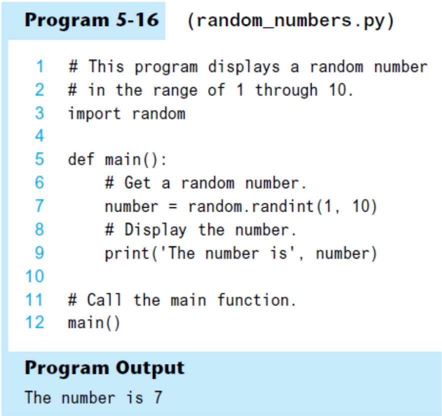
Generating Random Numbers (cont'd.)

- `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

A statement that calls the random function




Some number
number = random.randint(1, 100)
A random number in the range of 1 through 100 will be assigned to the number variable.

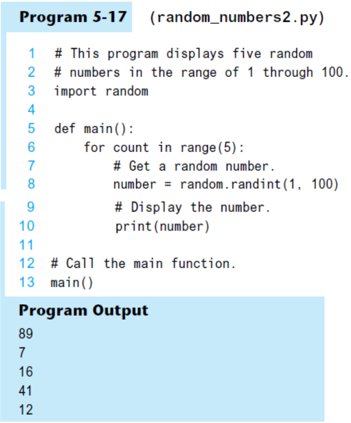


Program 5-16 (random_numbers.py)

```
# This program displays a random number
# in the range of 1 through 10.
import random

def main():
    # Get a random number.
    number = random.randint(1, 10)
    # Display the number.
    print('The number is', number)

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

Program Output

```
The number is 7
```

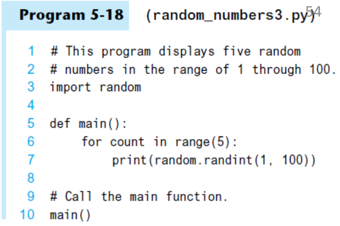


Program 5-17 (random_numbers2.py)

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

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

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

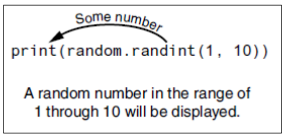
Program Output

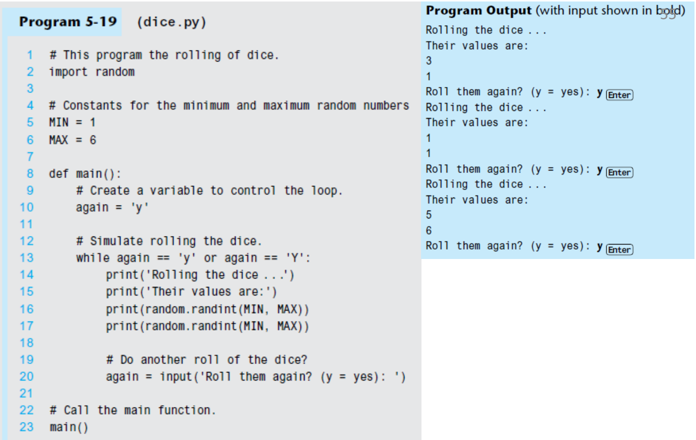
```
89
7
16
41
12
```



Program 5-18 (random_numbers3.py)

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

def main():
    for count in range(5):
        print(random.randint(1, 100))

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


Some number
print(random.randint(1, 10))
A random number in the range of 1 through 10 will be displayed.

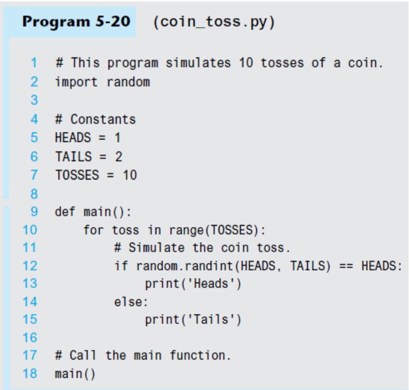


Program 5-19 (dice.py)

```
# This program the rolling of dice.
import random

# Constants for the minimum and maximum random numbers
MIN = 1
MAX = 6

def main():
    # Create a variable to control the loop.
    again = 'y'

    # Simulate rolling the dice.
    while again == 'y' or again == 'Y':
        print('Rolling the dice ...')
        print('Their values are:')
        print(random.randint(MIN, MAX))
        print(random.randint(MIN, MAX))

        # Do another roll of the dice?
        again = input('Roll them again? (y = yes): ')

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

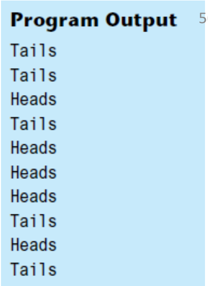
Program Output (with input shown in bold)

```
Rolling the dice ...
Their values are:
3
1
Roll them again? (y = yes): y Enter
Rolling the dice ...
Their values are:
1
1
Roll them again? (y = yes): y Enter
Rolling the dice ...
Their values are:
5
6
Roll them again? (y = yes): y Enter
```



Program 5-20 (coin_toss.py)

```
# This program simulates 10 tosses of a coin.
import random

# Constants
HEADS = 1
TAILS = 2
TOSSES = 10

def main():
    for toss in range(TOSSES):
        # Simulate the coin toss.
        if random.randint(HEADS, TAILS) == HEADS:
            print('Heads')
        else:
            print('Tails')

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



Program Output

```
Tails
Tails
Heads
Tails
Heads
Heads
Heads
Tails
Heads
Tails
```

Generating Random Numbers (cont’d.)

- `randrange` function: similar to `range` function, but returns randomly selected integer from the resulting sequence

```
number = random.randrange(0, 101, 10)
```

  - Same arguments as for the `range` function
- `random` function: returns a random float in the range of 0.0 and 1.0

```
number = random.random()
```

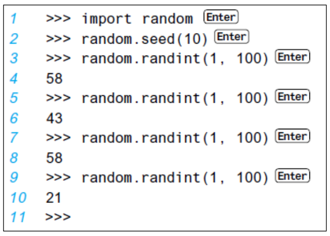
  - Does not receive arguments
- `uniform` function: returns a random float but allows user to specify range

```
number = random.uniform(1.0, 10.0)
```

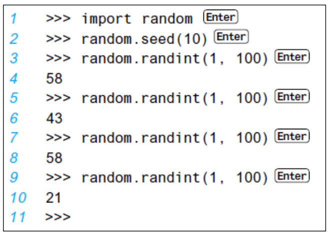
Random Number Seeds

- Random number created by functions in random module are actually pseudo-random numbers
- Seed value: 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

If we start a new interactive session and repeat these statements, we get the same sequence of pseudorandom numbers, as shown here:



```
>>> import random Enter
>>> random.seed(10) Enter
>>> random.randint(1, 100) Enter
58
>>> random.randint(1, 100) Enter
43
>>> random.randint(1, 100) Enter
58
>>> random.randint(1, 100) Enter
21
>>>
```



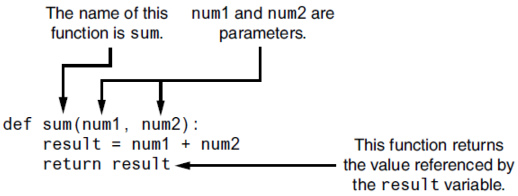
```
>>> import random Enter
>>> random.seed(10) Enter
>>> random.randint(1, 100) Enter
58
>>> random.randint(1, 100) Enter
43
>>> random.randint(1, 100) Enter
58
>>> random.randint(1, 100) Enter
21
>>>
```

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

```
def function_name():
    statement
    statement
    etc.
    return expression
```

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



The name of this function is sum.

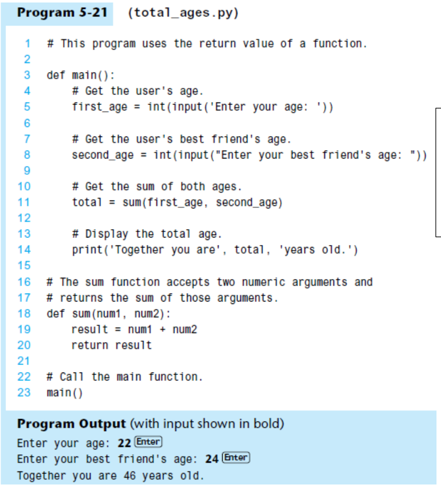
num1 and num2 are parameters.

```
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

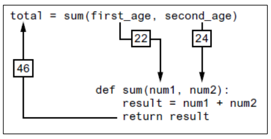


Program 5-21 (total_ages.py)

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

def main():
    # Get the user's age.
    first_age = int(input('Enter your age: '))

    # Get the user's best friend's age.
    second_age = int(input("Enter your best friend's age: "))

    # Get the sum of both ages.
    total = sum(first_age, second_age)

    # Display the total age.
    print('Together you are', total, 'years old.')

# The sum function accepts two numeric arguments and
# returns the sum of those arguments.
def sum(num1, num2):
    result = num1 + num2
    return result

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

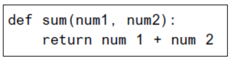
Program Output (with input shown in bold)

```
Enter your age: 22 Enter
Enter your best friend's age: 24 Enter
Together you are 46 years old.
```

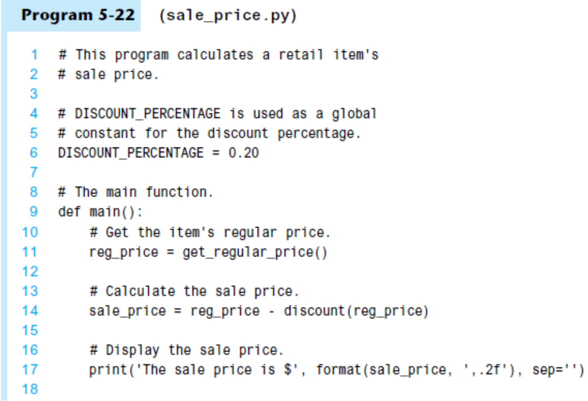


```
total = sum(first_age, second_age)

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

Because the return statement can return the value of an expression, you can eliminate the result variable and rewrite the function as:

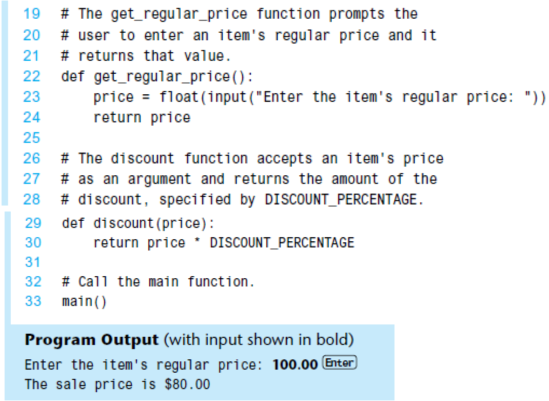


```
def sum(num1, num2):
    return num 1 + num 2
```



Program 5-22 (sale_price.py)

```
# This program calculates a retail item's
# sale price.

# DISCOUNT_PERCENTAGE is used as a global
# constant for the discount percentage.
DISCOUNT_PERCENTAGE = 0.20

# The main function.
def main():
    # Get the item's regular price.
    reg_price = get_regular_price()

    # Calculate the sale price.
    sale_price = reg_price - discount(reg_price)

    # Display the sale price.
    print('The sale price is $', format(sale_price, ',.2f'), sep='')

```



```
# The get_regular_price function prompts the
# user to enter an item's regular price and it
# returns that value.
def get_regular_price():
    price = float(input("Enter the item's regular price: "))
    return price

# The discount function accepts an item's price
# as an argument and returns the amount of the
# discount, specified by DISCOUNT_PERCENTAGE.
def discount(price):
    return price * DISCOUNT_PERCENTAGE

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

Program Output (with input shown in bold)

```
Enter the item's regular price: 100.00 Enter
The sale price is $80.00
```

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

```
def is_even(number):
    # Determine whether number is even. If it is,
    # set status to true. Otherwise, set status
    # to false.
    if (number % 2) == 0:
        status = True
    else:
        status = False
    # Return the value of the status variable.
    return status
```

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

```
def get_name():
    # Get the user's first and last names.
    first = input('Enter your first name: ')
    last = input('Enter your last name: ')

    # Return both names.
    return first, last
```

```
first_name, last_name = get_name()
```

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

`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
```

Program 5-25 (hypotenuse.py)

```
# This program calculates the length of a right
# triangle's hypotenuse.
import math

def main():
    # Get the length of the triangle's two sides.
    a = float(input('Enter the length of side A: '))
    b = float(input('Enter the length of side B: '))

    # Calculate the length of the hypotenuse.
    c = math.hypot(a, b)

    # Display the length of the hypotenuse.
    print('The length of the hypotenuse is', c)

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

Program Output (with input shown in bold)

```
Enter the length of side A: 5.0 Enter
Enter the length of side B: 12.0 Enter
The length of the hypotenuse is 13.0
```

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

Summary (cont'd.)

- 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