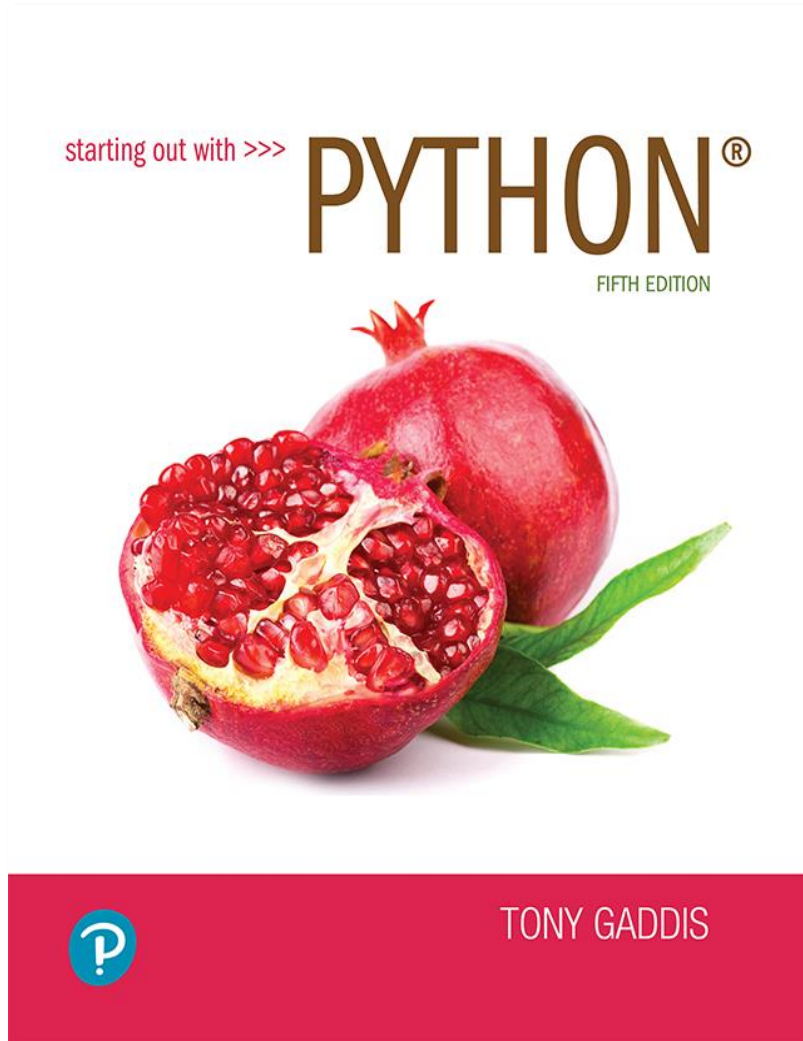


# Starting out with Python

Fifth Edition



## Chapter 5

### Functions

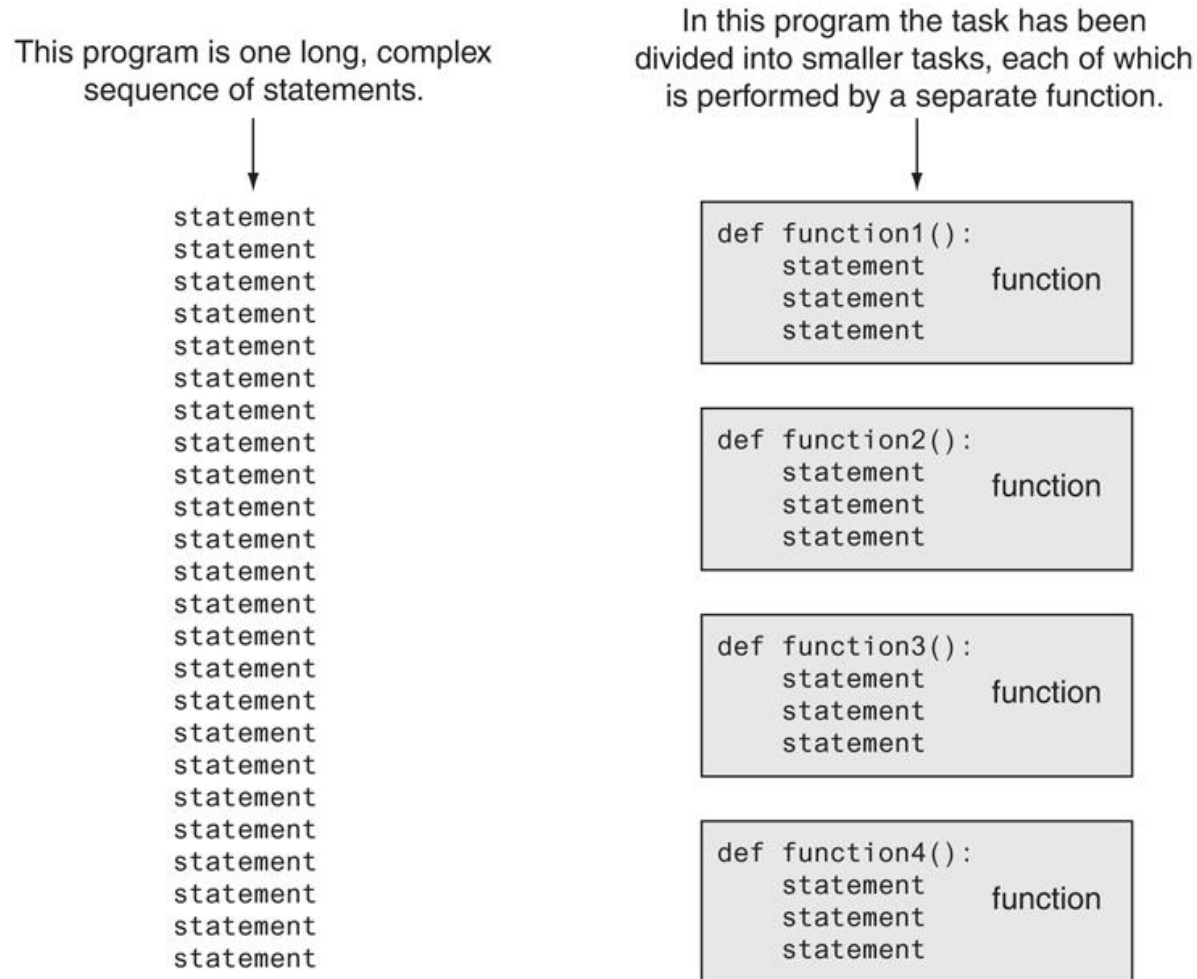
# Topics (1 of 2)

- 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
- The `math` Module

# Introduction to Functions (1 of 2)

- 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

# Introduction to Functions (2 of 2)



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

# 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 (1 of 5)

- Functions are given names
  - Function naming rules:
    - Cannot use keywords 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 (2 of 5)

- 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
```



# Defining and Calling a Function (3 of 5)

- 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 (4 of 5)

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

# Defining and Calling a Function (5 of 5)

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

# Indentation in Python

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

# Designing a Program to Use Functions

(1 of 3)

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

# Using the `pass` Keyword

- You can use the `pass` keyword to create empty functions
- The `pass` keyword is ignored by the Python interpreter
- This can be helpful when designing a program

```
def step1():  
    pass
```

```
def step2():  
    pass
```

# Local Variables (1 of 2)

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

# Local Variables (2 of 2)

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




# Passing Arguments to Functions (1 of 4)

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

# Passing Arguments to Functions (2 of 4)

```
def main():  
    value = 5  
    show_double(value)  
  
def show_double(number):  
    result = number * 2  
    print(result)
```

A diagram consisting of a horizontal line segment extending to the right from the `show_double(value)` line, followed by a vertical line segment extending downwards to an arrowhead pointing at the `show_double(number):` line, illustrating the flow of the argument `value` to the function `show_double`.

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

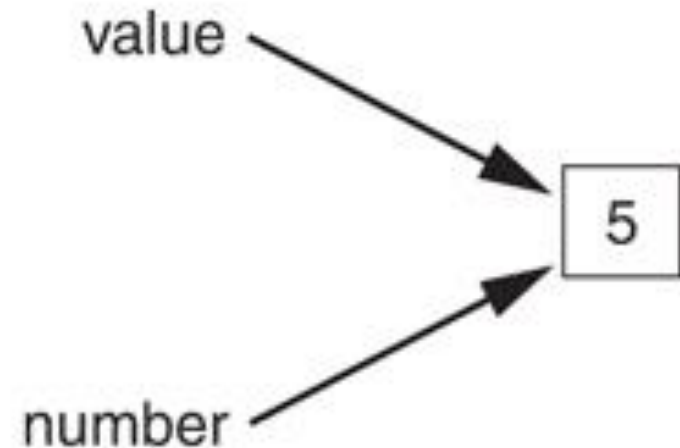
# Passing Arguments to Functions (3 of 4)

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

# Passing Arguments to Functions (4 of 4)

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

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



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

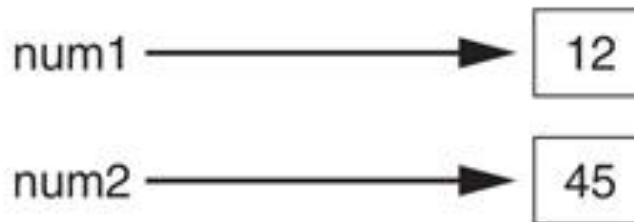
# Passing Multiple Arguments (1 of 2)

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

# Passing Multiple Arguments (2 of 2)

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

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



**Figure 5-16** Two arguments passed to two parameters

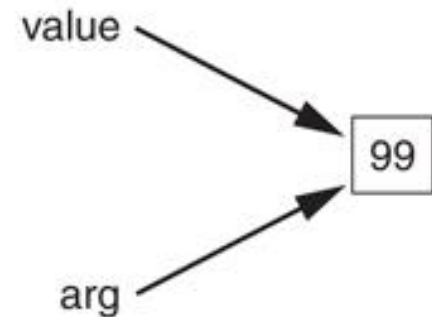
# Making Changes to Parameters (1 of 3)

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

# Making Changes to Parameters (2 of 3)

```
def main():  
    value = 99  
    print(f'The value is {value}.')  
    change_me(value)  
    print(f'Back in main the value is {value}.')
```

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



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

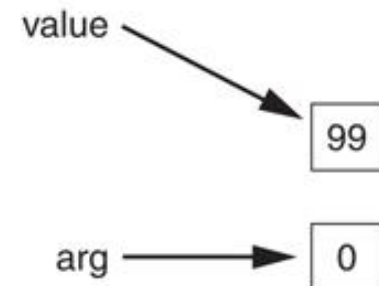


# Making Changes to Parameters (3 of 3)

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

```
def main():  
    value = 99  
    print(f'The value is {value}.')  
    change_me(value)  
    print(f'Back in main the value is {value}.')
```

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



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

# Keyword Arguments

- Keyword argument: argument that specifies which parameter the value should be passed to
  - Position when calling function is irrelevant
  - General Format:
    - `function_name(parameter=value)`
- Possible to mix keyword and positional arguments when calling a function
  - Positional arguments must appear first

# Global Variables and Global Constants

(1 of 2)

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

# Global Variables and Global Constants

(2 of 2)

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

# Global Constants

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

# Introduction to Value-Returning Functions: Generating Random Numbers

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

# Standard Library Functions and the `import` Statement (1 of 3)

- 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 (2 of 3)

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



# Standard Library Functions and the `import` Statement (3 of 3)



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

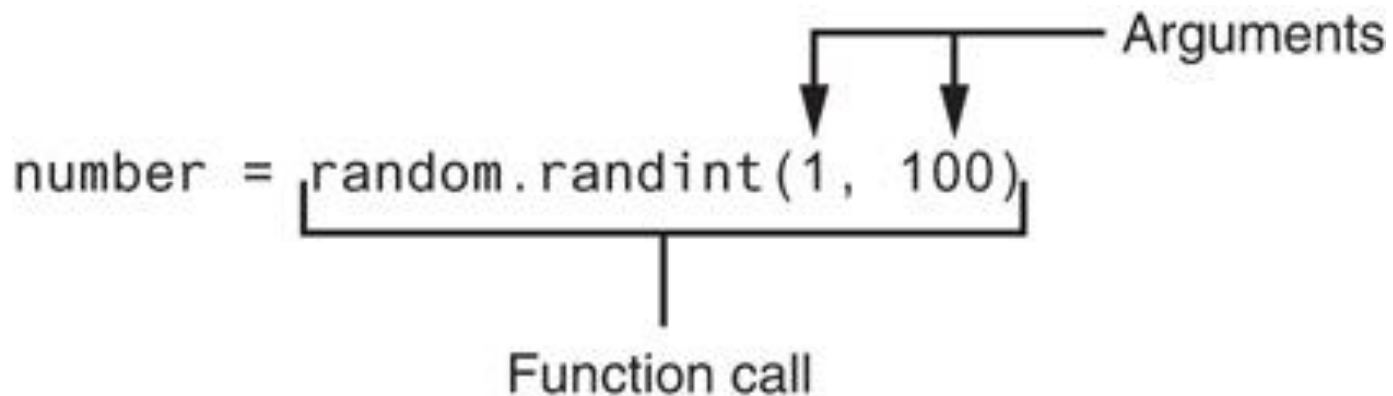
# Generating Random Numbers (1 of 5)

- 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 (2 of 5)

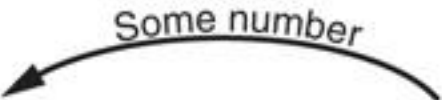
- randint function: generates a random number in the range provided by the arguments
  - Returns the random number to part of program that called the function
  - Returned integer can be used anywhere that an integer would be used
  - You can experiment with the function in interactive mode

# Generating Random Numbers (3 of 5)



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

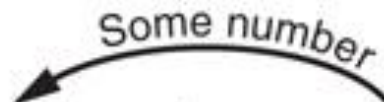
# Generating Random Numbers (4 of 5)



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

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

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



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

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

**Figure 5-22** Displaying a random number

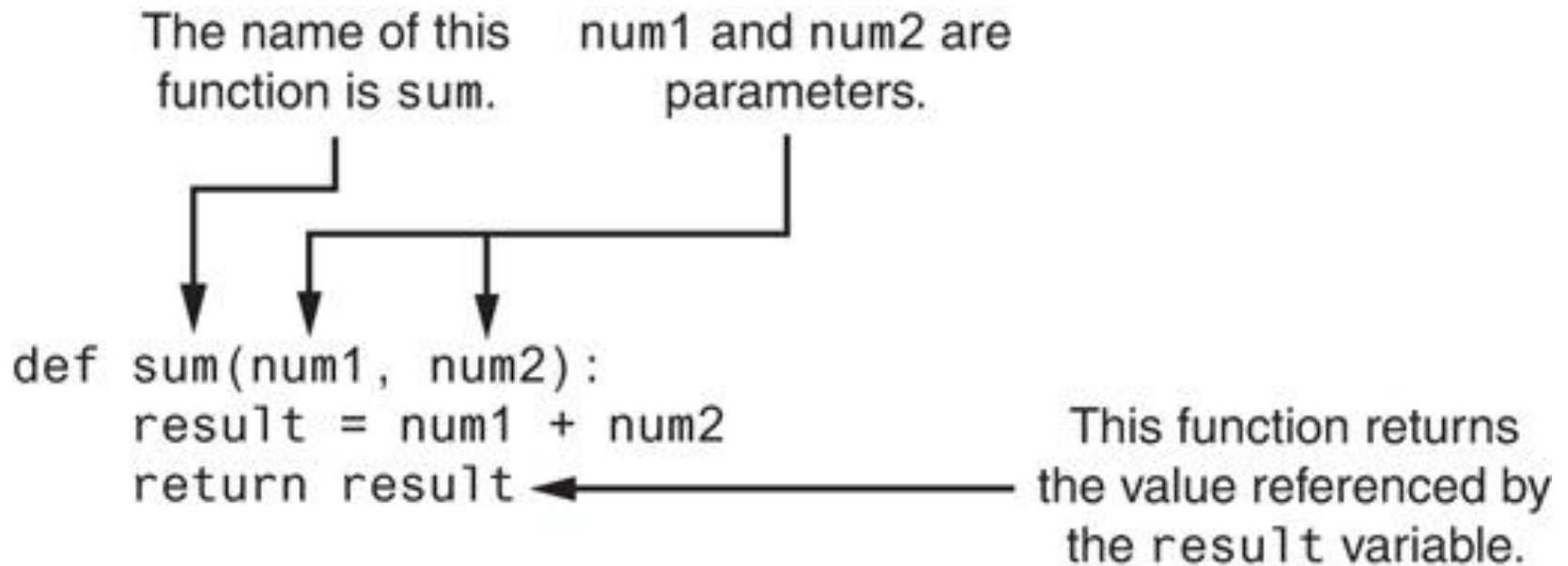
# Generating Random Numbers (5 of 5)

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

# Writing Your Own Value-Returning Functions (1 of 2)

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

# Writing Your Own Value-Returning Functions (2 of 2)



**Figure 5-23** Parts of the function



# 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

# Returning Strings

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

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

# Returning Boolean Values

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

# Returning Multiple Values

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

# Returning None From a Function

- The special value `None` means “no value”
- Sometimes it is useful to return `None` from a function to indicate that an error has occurred

```
def divide(num1, num2):  
    if num2 == 0:  
        result = None  
    else:  
        result = num1 / num2  
    return result
```

# Conditionally Executing the `main` Function (1 of 3)

- It is possible to create a module that can be run as a standalone program or imported into another program
- Suppose *Program A* defines several functions that you want to use in *Program B*
- So, you import *Program A* into *Program B*
- However, you do not want *Program A* to execute its main function when you import it

# Conditionally Executing the `main` Function (2 of 3)

- In the aforementioned scenario, you write each module so it executes its `main` function only when the module is being run as the main program
  - When a source code file is loaded into the Python interpreter, a special variable called `__name__` is created
  - If the source code file has been imported as a module, the `__name__` variable will be set to the name of the module.
  - If the source code file is being executed as the main program, the `__name__` variable will be set to the value `'__main__'`.

# Conditionally Executing the `main` Function (3 of 3)

- To prevent the `main` function from being executed when the file is imported as a module, you can conditionally execute `main`

```
def main():  
    statement  
    statement
```

```
def my_function():  
    statement  
    statement
```

```
if __name__ == '__main__':  
    main()
```