Computer Programming in Python

Chapter 6
Functions

Chapter 6 Functions

- Functions
  - As programs become longer and execute more tasks
    - The main function grows
    - Code may be repeated in order to repeat functionality
  - The design process includes dividing the program into logical sections of distinct functionality which will be developed individually
    - Referred to as ***modularization***

- Modularization
  - Separating the program into distinct parts provides many benefits
    - The ability to reuse portions of the code
    - The ability to divide the program development among multiple programmers
    - Simplify the task
  - Sections can be developed in ***functions***
  - The functions can be ***called*** when needed, and as many times as needed

Chapter 6 Functions

- Functions
  - Functions are blocks of code that perform specific tasks
  - Functions can be executed as many times and whenever needed
  - Functions can perform a task or compute a value for the program

Functions are blocks of code that perform a specific task

Chapter 6 Functions

- Functions – Two Types
  - *Void functions* that just perform a task
    - The ***print*** function is an example of a void function
      - Simply displays whatever is passed to it
  - *Value-returning functions* that return a value
    - The ***input*** function is an example of a value-returning function
      - Returns something that is then assigned to a variable

```
print('A void function')

value = input('Enter a number:  ')
```

Chapter 6 Functions

- Functions
  - The code for a function is called the ***function definition***
    - Begins with the keyword ***def*** which is followed by a name for the function, a pair of parentheses, and a colon
    - This first line of the function definition is referred to as the ***function header***

function header

```
def function_name():
    statement1
    statement2
    etc.
```

Chapter 6 Functions

- Functions
  - The ***function definition***
    - The statements that will execute when the function is called are indented and form a block of code and are referred to as the function ***body***

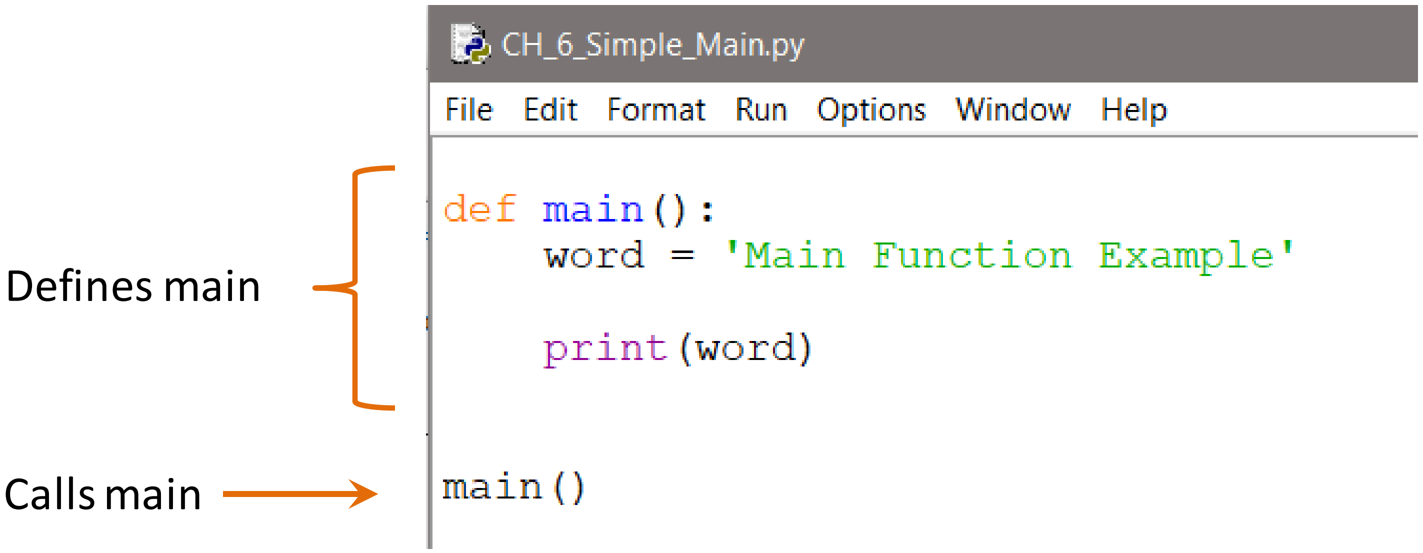
```
def function_name():
    statement1
    statement2
    etc.
```

function body

Chapter 6 Functions

- The Main Function
  - Program examples covered previously were run by the IDLE interpreter without a main function
  - The **Main function** will now be included
    - Every program has a main function where execution begins when the program is launched
    - Main can execute code or call other functions
      - Other functions can call other functions as well

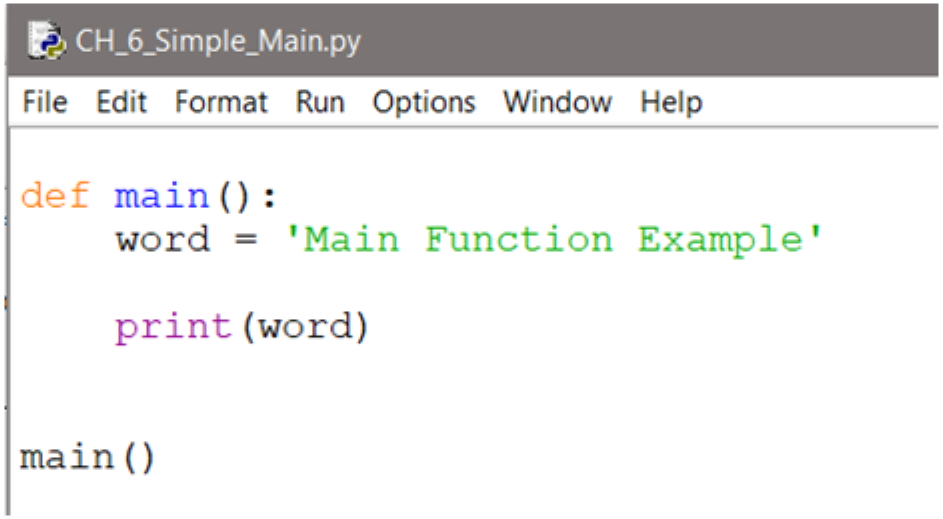
- The Main Function - simple example



```
def main():
    word = 'Main Function Example'

    print(word)


main()
```

Chapter 6 Functions

- The interpreter reads the file top-down
- It understands that def is the keyword for defining a function
- When it reaches the call to main, it executes the function



```python
def main():
    word = 'Main Function Example'

    print(word)


main()
```

```
Main Function Example
>>>
```

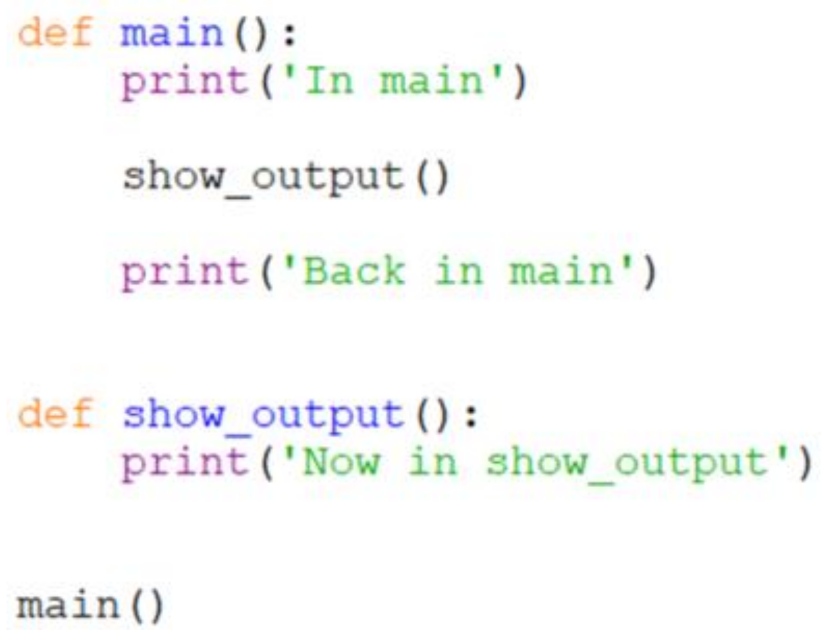
Chapter 6 Functions

- Main and another Function

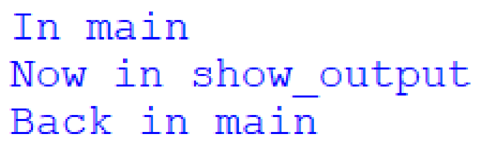
```
def main():                   # main function header
    print('In main')

    show_output()             # call to show_output

    print('Back in main')


def show_output():            # function definition
    print('Now in show_output')


main()                        # call the main function
                              # and begin execution
```

Chapter 6 Functions

- The interpreter reads through the lines of code, and when it reaches the last line it executes the *main* function
- The main function executes the print statement, then calls show_output
  - Control transfers to show_output
- Show_output executes the print statement and ends, so control returns to main
- Main then executes the second print statement and the program ends



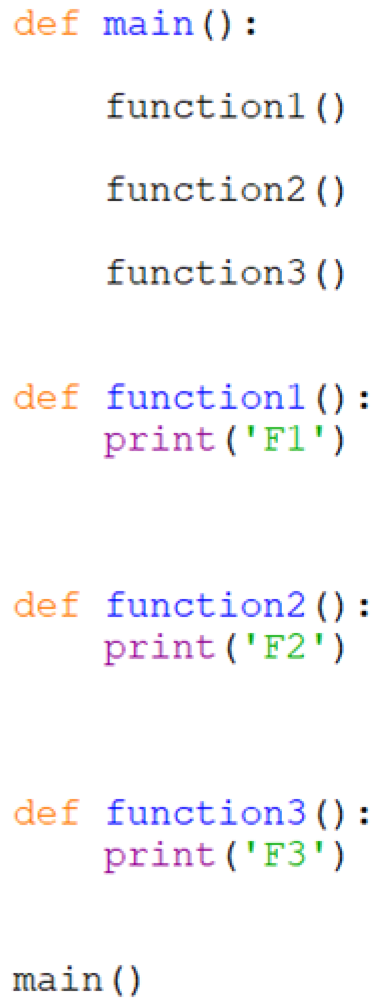
```
def main():
    print('In main')

    show_output()

    print('Back in main')


def show_output():
    print('Now in show_output')


main()
```



```
In main
Now in show_output
Back in main
```

Chapter 6 Functions

- Indentation reminder
  - Indentation forms a block of code
    - It is much easier to use the tab key for indentation than to count spaces to be sure they are always the same
  - Function names, including main begin at the margin
  - Function bodies are indented forming a block of code for the function
  - The IDE highlights items by color-coding the text to help



```
def main():

    function1()

    function2()

    function3()


def function1():
    print('F1')


def function2():
    print('F2')


def function3():
    print('F3')


main()
```

Chapter 6 Functions

- Variable ***Scope***
  - The part of a program where a variable is accessible
  - A variable defined within a function is a ***local variable***
    - The variable's scope is the function where it is defined
      - This includes the main function
  - A variable defined inside a function is not accessible outside that function

Variables defined in a function are local to that function

Chapter 6 Functions

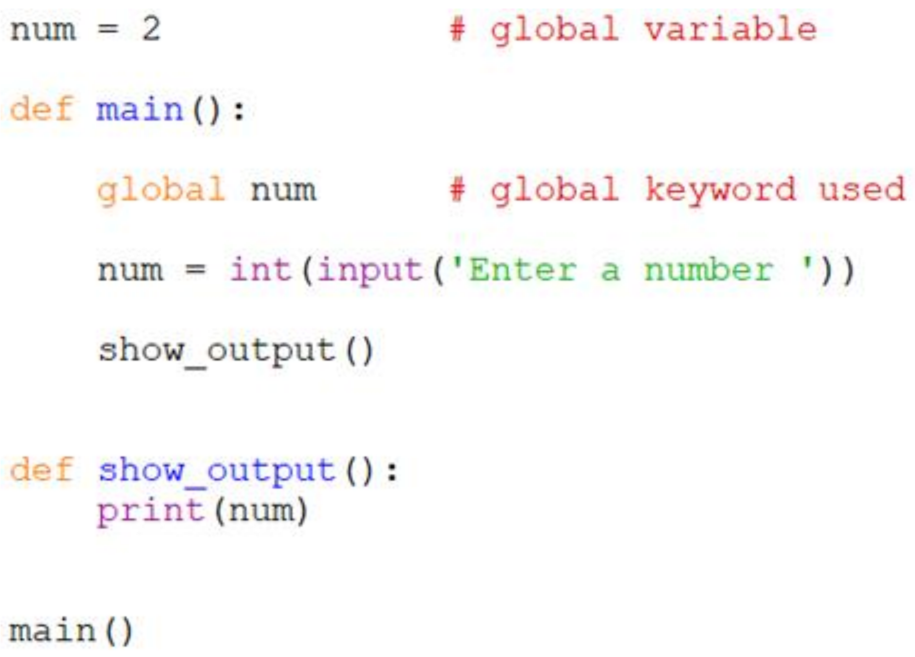
- Variable ***Scope***
  - Different functions could have variables with the same name without causing any conflict
  - Each of the variables would have its particular function as its scope, and would not be accessible by another function
  - If several engineers are working on the same program, but they are working on different functions, they may name a local variable using the same name

Chapter 6 Functions

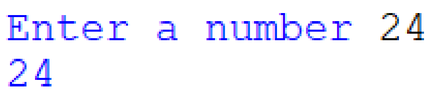
- Global Scope
  - A variable defined outside of all functions (including main) is a ***global variable***
    - It would be accessible by all parts of the program
      - It has the whole program as its scope
    - A function that needs to change it, precedes it with the keyword global

Chapter 6 Functions

- Global Scope Example
  - The variable *num* is defined outside of any function
  - Main changes it by preceding it with ‘global’
  - The function displays the value proving that it has access to the global variable and that it has been changed



```
num = 2                     # global variable

def main():

    global num              # global keyword used

    num = int(input('Enter a number '))

    show_output()


def show_output():
    print(num)


main()
```



```
Enter a number 24
24
```

Chapter 6 Functions

- Global Variables
  - Use them sparingly if at all
    - Their use makes debugging very difficult since any part of the program can change a global variable
      - Difficult to determine which part of the program is causing the problem
    - Any function that accesses and uses a global variable is dependent on that variable and cannot easily be used in another program

Chapter 6 Functions

- Global Variables
  - Two occasions for their use
    - Consistency
      - In a collaborative programming environment when multiple engineers are working on the same project and a consistent value is required.
    - Efficiency
      - A project that uses a value or set of values in many places, and the value tends to change.

Global Constants are typically used in these instances

Chapter 6 Functions

- Global Constant
  - A ***constant*** is a value that cannot (should not) be changed by the program
  - Python does not formally have constants, but they can be implied by following the standard for naming constants with all uppercase letters and underscores between words

```
EARTH_DIAMETER = 3963
```

Global Constants are often used in large programs

Chapter 6 Functions

- Global Constant
  - Without using the keyword global, the value cannot be changed by a function, but it will appear that it does even though a new variable has actually been declared
    - This will be very hard to debug
  - Follow the rules for Global Constants
    - Use all uppercase letters with underscores between words
    - Don't write any code to change them

Chapter 6 Functions

- Passing Values to Functions
  - When a function needs to use a variable defined somewhere else in the program, the variable is passed to the function as an ***argument***
  - To the function receiving the argument, it is technically referred to as a ***parameter***
  - Technically speaking, arguments are passed to functions and parameters are received by them

Chapter 6 Functions

- Passing Values (arguments) to Functions
  - When a variable is passed to a function as an ***argument***, what is actually passed to the function is the value of the variable (a copy)
    - What is being stored in the variable
  - This is pass-by-value

Pass-by-value passes a copy of the value to the function

Chapter 6 Functions

- Passing Values to Functions
  - It doesn’t matter what the receiving function calls the value it receives, except that it must use that name internally

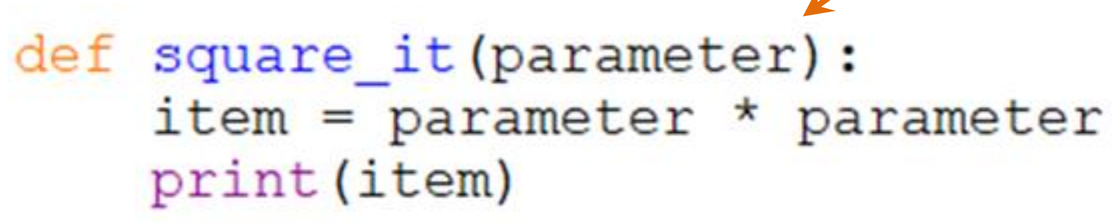
```
def main():
    argument = 4

    square_it(argument)


def square_it(parameter):
    item = parameter * parameter
    print(item)


main()
```

the value 4 is passed

the value 4 is received

- Passing Values to Functions
  - The parameter variable is actually a local variable to the function and has the function as its scope
    - It is assigned the value passed in

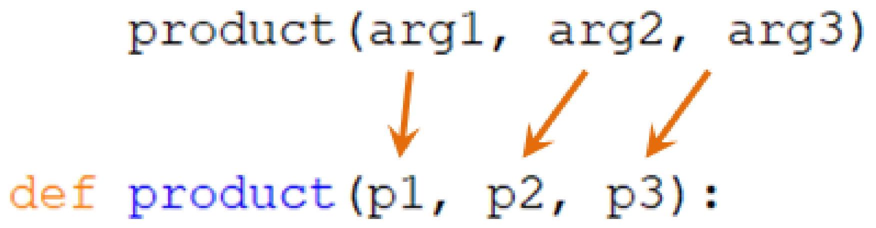
local variable



```
def square_it(parameter):
    item = parameter * parameter
    print(item)
```

When the function completes, the local variables are destroyed

- Passing Multiple Values to Functions
  - Multiple arguments can be passed to functions as long as the function has parameters to receive them
  - They can be different data types
  - They are received in the order that they are passed



```
product(arg1, arg2, arg3)

def product(p1, p2, p3):
```

Chapter 6 Functions

- Passing Multiple Values to Functions

```python
def main():
    arg1 = 10
    arg2 = 'Product'
    arg3 = 27

    product(arg1, arg2, arg3)


def product(p1, p2, p3):
    value = p1 * p3
    print(p2 + ' is: ', value)


main()
```

```
Product is:  270
```

Chapter 6 Functions

- Value-returning functions
  - Return a value to the calling function
  - Have a return statement

```
def function_name():
    statement1
    statement2
    etc.
    return something
```

Chapter 6 Functions

- Value-returning Functions - Example
  - The function returns the value of *num* to main which assigns it to *usernum*

```
def main():

    usernum = get_input()
    print('User entered ', usernum)


def get_input():
    num = int(input('Enter a number '))
    return(num)


main()
```

Modular Programming

Chapter 6 Functions

- Functions - Modularization
  - As more functions are written to perform functionality, main becomes a series of function calls
  - Most of the functionality that a program executes can be placed in a function
  - As requirements are decomposed and the Design Phase begins, areas of the program that lend themselves to being functions will surface
  - Once, the functionality is determined, the functions can be defined

Chapter 6 Functions

- Defining and Naming Functions
  - When designing and creating functions
    - Determine what the function will do
      - Each function should accomplish one task
    - Name the function what it does
      - Follow the same naming conventions for variables with all lowercase letter and words separated by underscores
      - Since functions perform an action, verbs are usually used to name functions

Chapter 6 Functions

- Defining and Naming Functions
  - When designing and creating functions
    - Determine what parameters the function needs in order to accomplish the task
    - Determine if the function will return a value and if so, what data type will it return

Chapter 6 Functions

- Defining and Naming Functions - Example
  - Write a function that receives an hourly rate and number of hours worked and returns the gross pay.
    - The function will compute gross pay
      - Input: the function will need the hourly rate and number of hours worked
      - Processing: the function will compute the gross pay
      - Output: the function will return the gross pay

Chapter 6 Functions

- Defining and Naming Functions - Example
  - Gross pay function

parameter
s

```
def compute_gross_pay(hourly_rate, hours_worked):
    gross_pay = hourly_rate * hours_worked
    return(gross_pay)
```

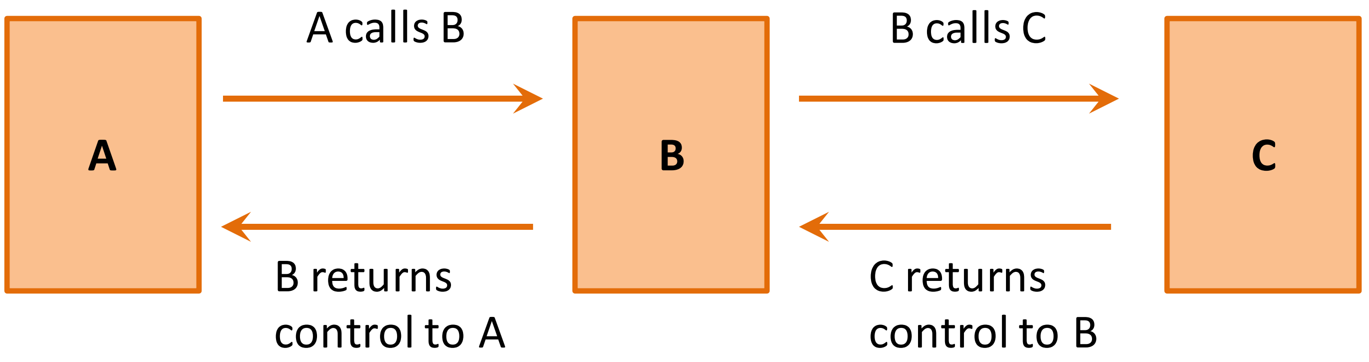
return value

Chapter 6 Functions

- Functions calling other functions
  - When a function calls another function, program control transfers to that function.
  - When the function completes, control transfers back (returns) to the calling function
    - Even if there is no returned value
    - Even if there are no more lines to execute

Chapter 6 Functions

- Functions calling other functions
  - Consider an example:
    - Function “A” calls function “B” which calls function “C”



Chapter 6 Functions

- Recursion
  - A function can even call itself
  - The calls will end when the base case is reached
  - Similar to winding up a spring that unwinds when the base case is reached
    - Like a loop, there must be change that eventually ends the recursion…this is the base case

Chapter 6 Functions

- Recursion
  - The recursion ends when number reaches zero

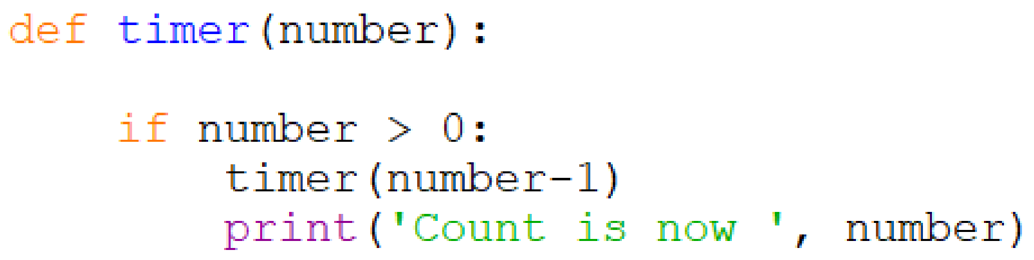
```
def main():

    timer(10)


def timer(number):

    if number > 0:
        print('Count is now ', number)
        timer(number-1)

main()
```

the function calls itself

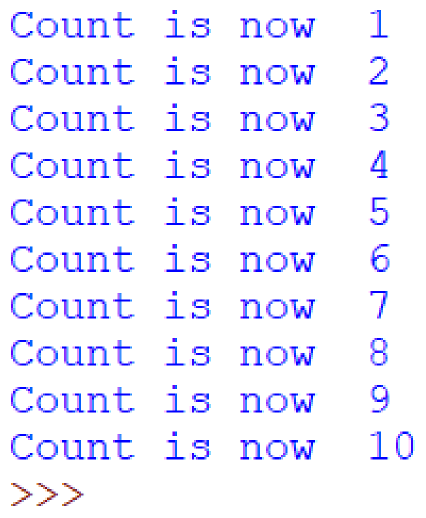
```
Count is now  10
Count is now  9
Count is now  8
Count is now  7
Count is now  6
Count is now  5
Count is now  4
Count is now  3
Count is now  2
Count is now  1
>>>
```

Chapter 6 Functions

- Recursion
  - Moving the print statements changes the output
  - The print statement doesn't execute until the return from the from the calls



```python
def timer(number):

    if number > 0:
        timer(number-1)
        print('Count is now ', number)
```



```
Count is now  1
Count is now  2
Count is now  3
Count is now  4
Count is now  5
Count is now  6
Count is now  7
Count is now  8
Count is now  9
Count is now  10
>>>
```

Chapter 6 Functions

- Recursion
  - Direct Recursion
    - A function calls itself
  - Indirect Recursion
    - Function ‘A’ calls function ‘B’ which then calls function ‘A’

A loop can always be implemented in place of Recursion

Chapter 6 Functions

- Tools for Function (and program) Design
  - An Input, Processing, Output document (***IPO***) is a helpful tool for function design as well program design
    - May be in the form of a chart or document
    - Includes the name of the function, a brief description of what it does, the input needed, the processing it will accomplish, and the output or return value
    - An IPO can also be used for the overall program

Sometimes referred to as a Design Document

Chapter 6 Functions

- IPO General Format
  - <u>Function IPOs:</u>

```
get_input()
    Description: Obtains user input
    Input: number from user
    Processing: none
    Output: returns the number

compute_square(number)
    Description: Computes the square of the number
    Input: a number
    Processing: square the number
    Output: return the value
```

Chapter 6 Functions

- IPO General Format
  - <u>Program IPO:</u>

Description: the program calls three functions to obtain user input of a number, square the number, and display the square of the number.

Input: number from user

Processing: square the number

Output: display the result

IPO format and content differ, but the concept is consistent

Chapter 6 Functions

- Functions and Methods
  - Different programming languages use different terminology with respect to functions
    - Java uses both method and function
    - C/C++ use the term function
    - Python uses both function and method

  - Python Terminology
    - Function – a named block of executable statements
    - Method - a function that exists inside of an object

Modular Programming

and

Python Modules

Chapter 6 Functions

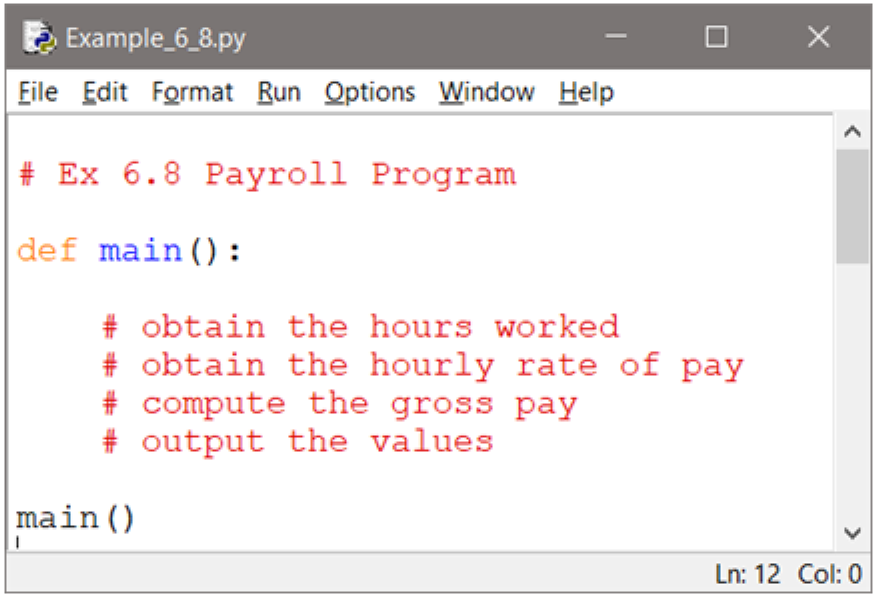
- Modular Programming using Files
  - Using functions separates operations into manageable chunks and enhances development and maintenance
  - But multiple engineers cannot easily work on the same program because it is in a single program file
  - Adding files (modules) to a program allows multiple engineers to work on the same program at the same time, permitting ***collaborative development***

Chapter 6 Functions

- Modular Programming using Files
  - Large and complex program requirements are decomposed during design into manageable sections
    - Then further refined into functions
    - Functions that are related are developed in their own files (modules)
    - The files are then ***imported*** into the main program

Chapter 6 Functions

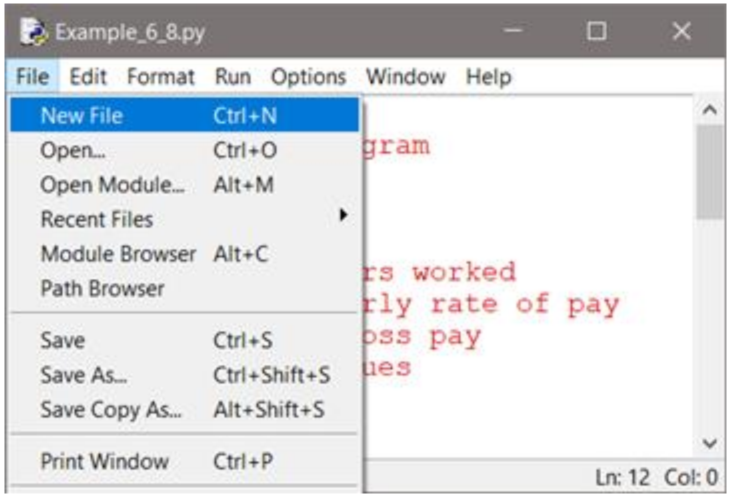
- Modular Programming using Files - Example
  - The Payroll Program main file is shown with pseudocode comments that map out the program



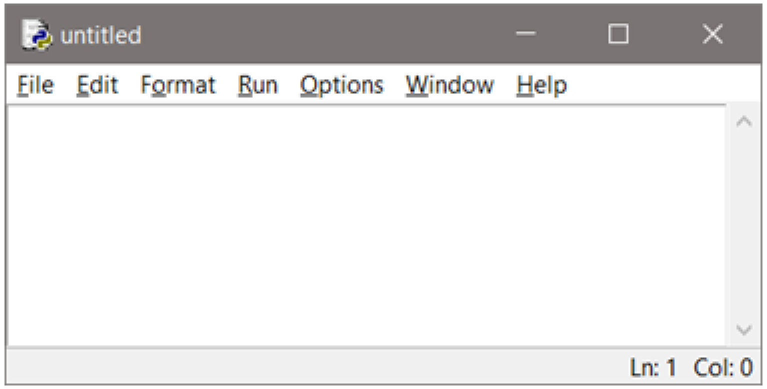
```
# Ex 6.8 Payroll Program

def main():

    # obtain the hours worked
    # obtain the hourly rate of pay
    # compute the gross pay
    # output the values

main()
```

Modular Programming using Files - Example

- Creating the second file for the functions
  - Select File | New File from the menu of the main program file
  - A new, unnamed file will appear



```
gram


rs worked
rly rate of pay
oss pay
ues
```



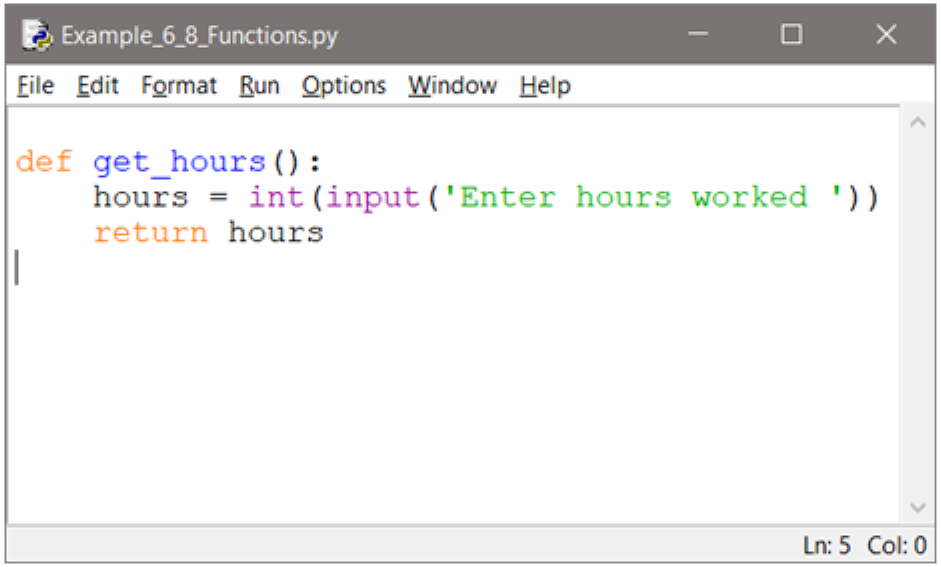
Chapter 6 Functions

- Modular Programming using Files - Example
  - In software development, it is always best to use the "***Build a little, test a little***" concept aka. ***Iterative Enhancement***
    - Develop a small part of the program and test and debug that part until it is working correctly
    - Then, develop another small part and test and debug the program with the additional part

Incremental programming eliminates wasted time debugging large amounts of code

Chapter 6 Functions

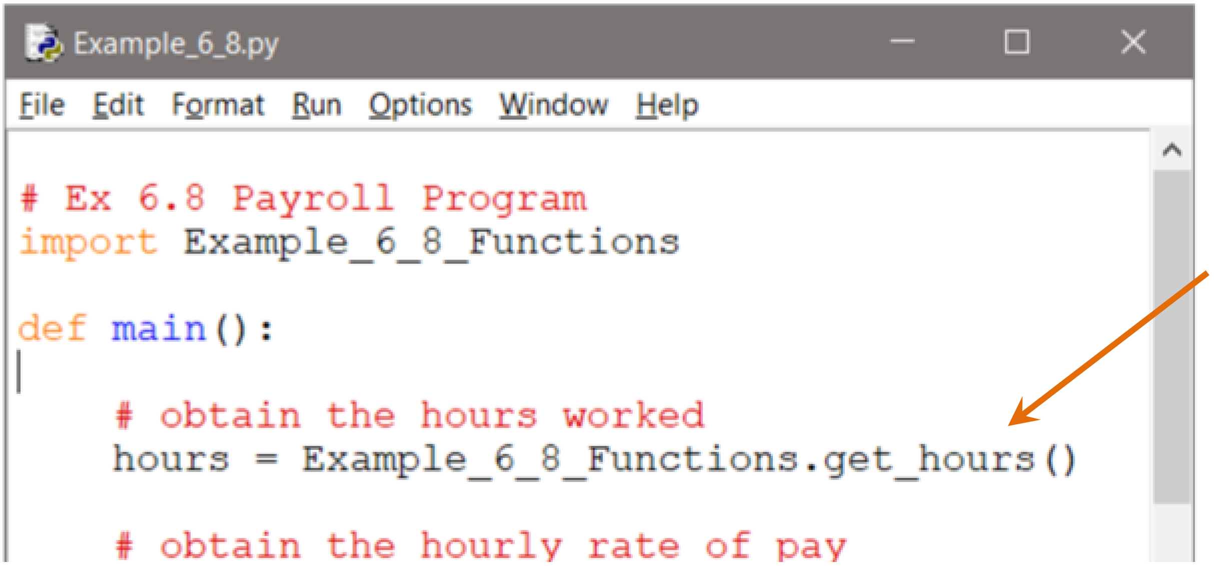
- Modular Programming using Files - Example
  - The first function for the Payroll Program example obtains the number of hours worked and is written in the new file
  - The file is saved using a name that identifies what it contains



```
def get_hours():
    hours = int(input('Enter hours worked '))
    return hours
```

- Modular Programming using Files - Example
  - The file containing the function must be imported into the main file
  - The function is called using the name of the file (omitting the .py file extension), the dot operator, and the name of the function

import



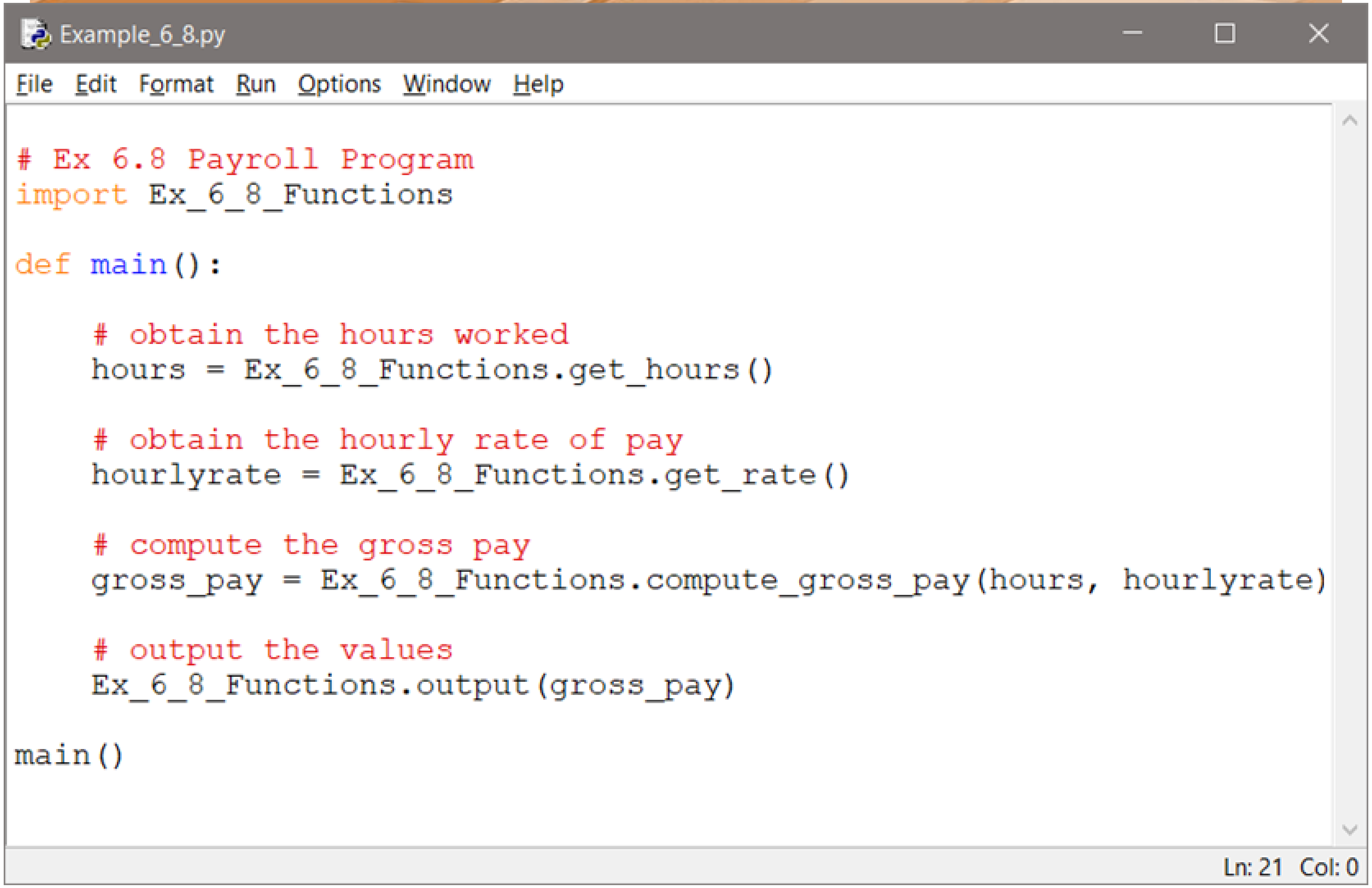
```
# Ex 6.8 Payroll Program
import Example_6_8_Functions

def main():

    # obtain the hours worked
    hours = Example_6_8_Functions.get_hours()

    # obtain the hourly rate of pay
```

function call

Chapter 6 Functions

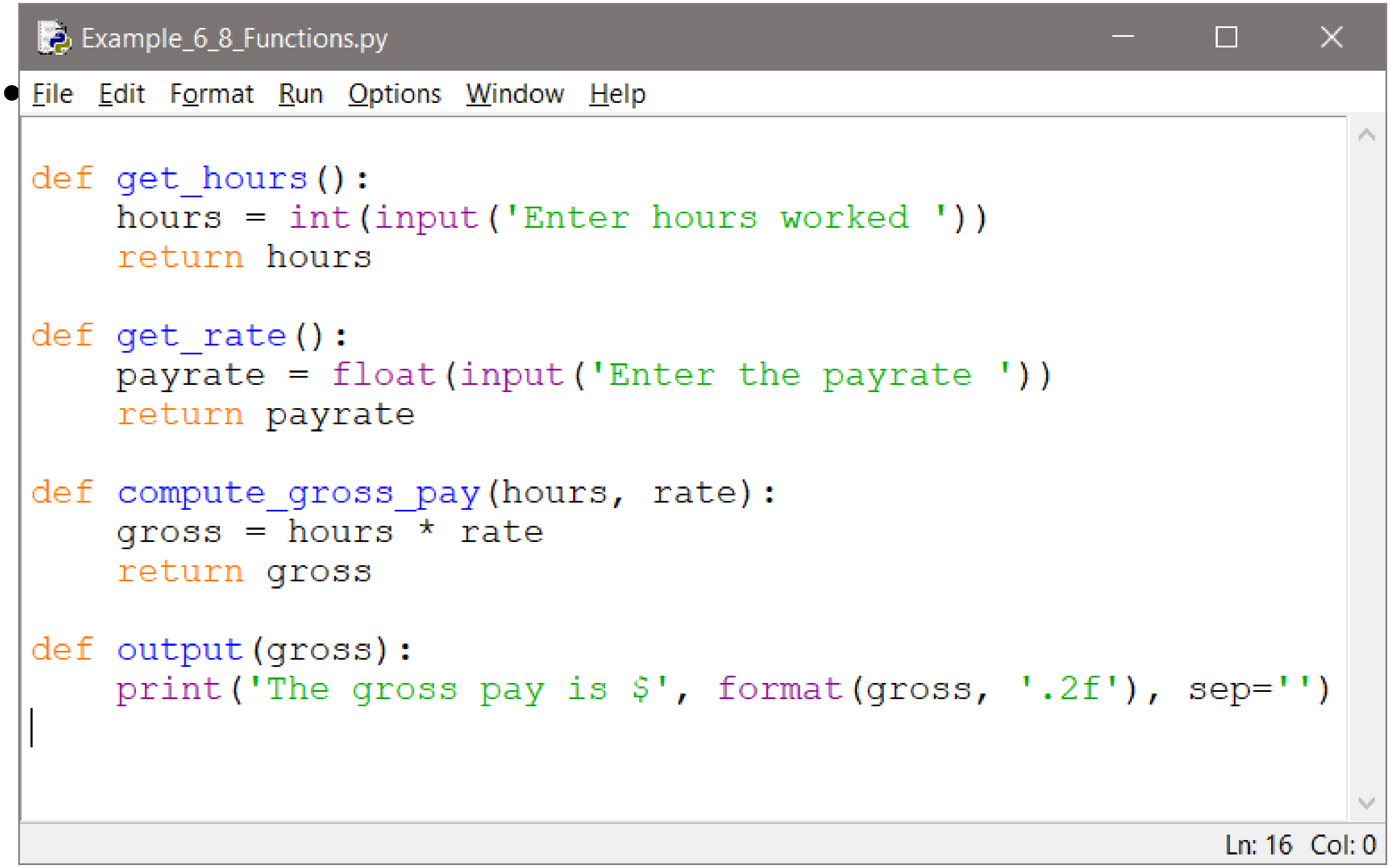
- Modular Programming using Files - Example
  - Other functions are added to the file
  - Calls to the other functions from main are handled the same way
  - Note:
    - Because of variable scope it does not matter if the variables in the main function and another function have the same name
    - They are local to their functions and there is no conflict

Chapter 6 Functions



```python
# Ex 6.8 Payroll Program
import Ex_6_8_Functions

def main():

    # obtain the hours worked
    hours = Ex_6_8_Functions.get_hours()

    # obtain the hourly rate of pay
    hourlyrate = Ex_6_8_Functions.get_rate()

    # compute the gross pay
    gross_pay = Ex_6_8_Functions.compute_gross_pay(hours, hourlyrate)

    # output the values
    Ex_6_8_Functions.output(gross_pay)

main()
```

Chapter 6 Functions



```
def get_hours():
    hours = int(input('Enter hours worked '))
    return hours

def get_rate():
    payrate = float(input('Enter the payrate '))
    return payrate

def compute_gross_pay(hours, rate):
    gross = hours * rate
    return gross

def output(gross):
    print('The gross pay is $', format(gross, '.2f'), sep='')
```

Chapter 6 Functions

- Modular Programming Supports Reuse
  - Python has an extensive set of modules that provide functions for use in programs
    - Often referred to as libraries because they contain groups of related functions
    - Code that has already been written can be used in our programs

Thousands of Python libraries have been developed

The Python Math Module

- The Python Math Module
  - Installed with Python
  - Contains many mathematical functions
    - Must be imported when used in program files
  - The list of math functions includes: *acos(x), asin(x), atan(x), cos(x), hypot(), log(x), sin(x), sqrt(x), tan(x), and others*

Chapter 6 Functions

- The Python Math Module
  - Defines a value for *pi* and *e*
  - Provides conversions for degrees to radians, *radians(x)*, and radians to degrees, *degrees(x)*
  - Contains a function for exponentiation, *pow(x, y)*
    - Can be used instead of x**y

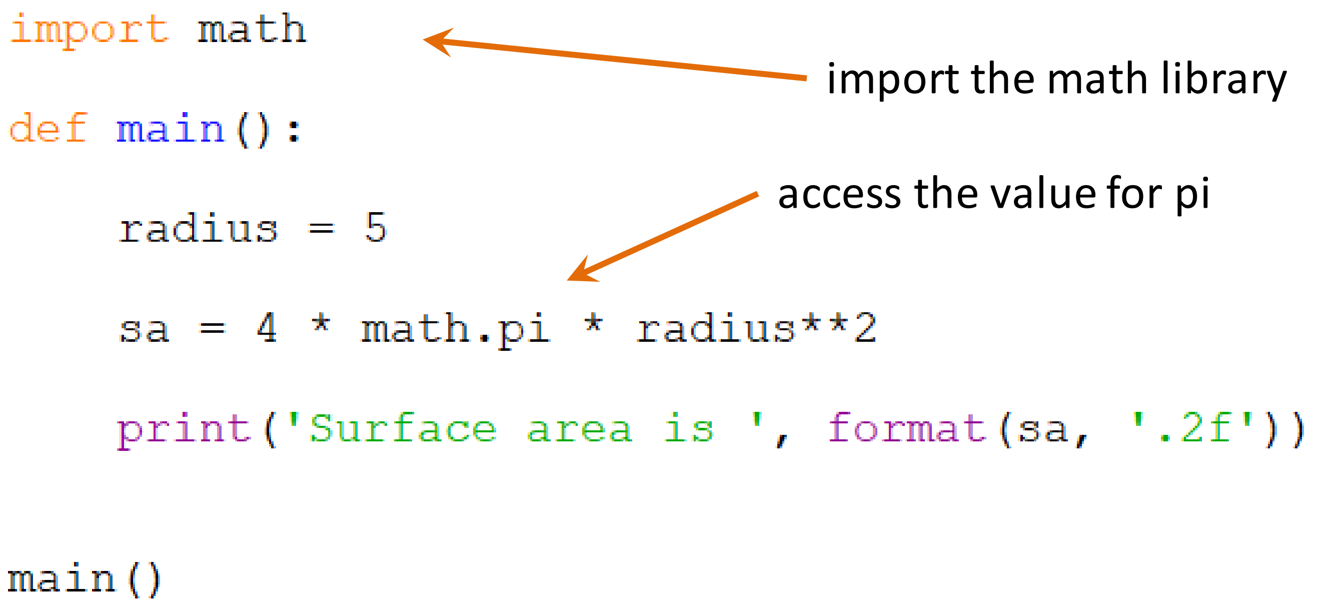
- Python Math Module Square Root
  - Import the module, and precede the function name with *math*, and the dot operator

```
import math

def main():

    num = 3 * 3

    print('Num is ', num)

    sr_num = math.sqrt(num)

    print('Square root of num is', sr_num)


main()
```

import the math library

call square root

- The Python Math Module
  - Math module value for pi



```
import math

def main():

    radius = 5

    sa = 4 * math.pi * radius**2

    print('Surface area is ', format(sa, '.2f'))


main()
```

import the math library

access the value for pi

Chapter 6 Functions

- Random Numbers
  - Many programs generate random numbers including simulations and games
  - They are used to determine random event occurrences, and can be used for encryption
  - Python includes random number generation with library functions that require importing the ***random*** module
    - The random module is installed with Python

Chapter 6 Functions

- Random Numbers
  - Arguments can be passed to the random number functions to determine the range of random numbers that could be generated
  - The four available functions are preceded by the module name *random* and the dot operator

Chapter 6 Functions

- Random Numbers
  - Python has a few different random number functions that provide different benefits and results
    - randint
    - randrange
    - random
    - uniform

Chapter 6 Functions

- Random Numbers
  - The ***randint*** function generates random integers
  - Two arguments determine the range

```
import random

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

Returns a value between 1 and 100 inclusive

Chapter 6 Functions

- Random Numbers
  - The ***randrange*** function can accept one, two, or three arguments
  - When one argument is passed, zero is used as the start of the range and the argument is the limit (which is excluded)

```
num = random.randrange(10)
```

Returns a value between 1 and 9

Chapter 6 Functions

- Random Numbers
  - When two arguments are passed, the first argument is used as the start of the range and the second argument is the limit (which is excluded)

```
num = random.randrange(0, 101)
```

Returns a value between 1 and 100

Chapter 6 Functions

- Random Numbers
  - When three arguments are passed, the first argument is used as the start of the range and the second argument is the limit (which is excluded), and the third is the step

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

Returns a value between 0 and 100 stepping by tens (0,10,20,…)

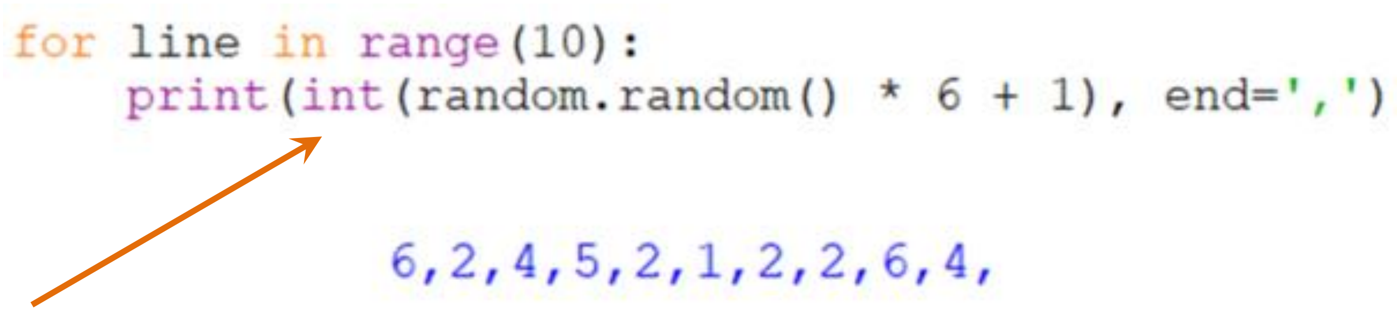
Chapter 6 Functions

- Random Numbers
  - Most random number generators in other programming languages generate a number between 0.0 and 1.0
  - Consider that there are actually many values in that range and the returned value can be modified for any purpose
  - Python provides the ***random*** function which generates floating point random numbers between 0.0 and 1.0

```
num = random.random()
```

Chapter 6 Functions

- Random Numbers
  - Simulate rolling a die
    - Generate random numbers between 0.0 and 1.0
    - Modify the numbers for 1 through 6 inclusive



```
for line in range(10):
    print(int(random.random() * 6 + 1), end=',')
```

```
6,2,4,5,2,1,2,2,6,4,
```

eliminates the decimal

Chapter 6 Functions

- Random Numbers
  - Simulate rolling a die (another way)
    - Use randint with 1 and 7 as the arguments

```
num = random.randint(1, 7)
```

Chapter 6 Functions

- Random Numbers
  - The ***uniform*** function allows setting a range for random floating point numbers

```
num = random.uniform(1.0, 7.5)
```

Chapter 6 Functions

Chapter 6 Functions