\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC121 PYTHON Programming**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lesson 07 Functions [Part 1]

# **Objectives**

In this lesson, students will learn:

- How to define functions

- How to invoke functions

- How to pass data to a function when it is invoked

- How to write programs with multiple functions

# **7.1 Overview**

We have learned the most fundamental features of Python. We know how to store data in variables, get input, display output, perform calculations, select a block of statements to execute, repeat a block multiple times, and store multiple data items in a list. In theory, we are able to write large programs with these techniques. However, from a practical point of view, these are not enough.

When programmers begin to write longer and more complex programs, they feel like they need a better way to organize the program code. All complex products in this world are made up of parts. For example, to build a car, you first build each part and then put these parts together. We can do the same when we build complex programs. When we design a new program, we think about how to break it up into parts. Then we write code to build each part and put them together properly to build the whole program. This software design technique is called **modular programming**. We are going to discuss how to build programs with this technique in the next several lessons.

# **7.2 Modular Programming**

Parts are generally called routines in programming. How do we design routines? Typically each part of a sophisticated product has a special functionality. For example, the steering wheel of a car is designed to control the direction of the car. The gas pedal is designed to control speed, and the brake is designed to stop the car. We can follow this idea when we design routines for a program. We analyze the program and identify different tasks the program will perform. For example, suppose you are writing a program to manage a class roster. The program probably needs to perform three tasks: adding students, dropping students, and displaying student names. We can create a routine for each task, and then combine them properly to form the complete program.

Each routine in a program should be as independent as possible. That means it should have as little dependency on other routines as possible. However, routines do not exist in a vacuum. Sometimes a routine needs data from another routine in order to perform its task. Python has mechanisms for a routine to pass data to another routine. We will discuss these mechanisms later.

Let’s start with a program in which routines do not need to pass data to other routines. All routines in this program are highly independent.

We are going to write a program to create a midterm exam for a programming course. To make the example simple, there are only three questions in the exam: one multiple choice, one true/false and one ordering. The program will display the questions one by one. After the user has answered a question, the program will show whether the answer is correct or not. The following is an example:

Question 1:

What does an assignment statement do?

a. Display output

b. Read input

c. Store value in a variable

d. Repeat the same task multiple times

Answer: c

Correct!

Question 2:

2ndTest is a valid Python variable name. True or false?

Answer [t/f]: t

Incorrect!

Question 3:

Put the following steps of an algorithm in the correct order:

a. Calculate average = sum / 2

b. get two numbers x and y

c. display average

d. Calculate sum = x + y

Answer [e.g. cbad]: bdac

Correct!

Let’s think about how to write this program with modular programming. We will create three routines, with each routine handling one question. We also need statements in the program to tell the computer to execute these routines. In the next section, we will show you how to do these in Python.

# **7.3 Defining and Invoking Functions**

In Python, routines are called **functions**. A function is a named group of program statements performing some tasks. The code that makes up a function, which is called **function definition**, includes two parts: **function header** and **function body**.

The function header is the first line in function definition. It starts with the keyword def, followed by the function’s name, a pair of parentheses and a colon. There may be something inside the parentheses in some functions. We will talk about that later.

The rest of the function definition is the function body. It is a block of Python statements we want the computer to execute when it executes this function. All statements in the function body must be indented to indicate that they belong to the function.

The following shows the general syntax of Python functions:

*def function\_name(optional parameters here):*

*A block of Python statements here*

Let’s write a function to handle question1:

**def** question1():  
 print(**"Question 1:"**)  
 print(**"What does an assignment statement do?"**)  
 print(**"a. Display output"**)  
 print(**"b. Read input"**)  
 print(**"c. Store value in a variable"**)  
 print(**"d. Repeat the same task multiple times"**)  
 ans = input(**"Answer: "**)  
 **if** ans == **"c"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line*

The name of this function is question1. The function body consists of statements that display the question, read user’s answer, and display feedback.

We can also write functions to handle questions 2 and 3:

**def** question2():  
 print(**"Question 2:"**)  
 print(**"2ndTest is a valid Python variable name. True or false?"**)  
 ans = input(**"Answer [t/f]: "**)  
 **if** ans == **"f"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line***def** question3():  
 print(**"Question 3:"**)  
 print(**"Put the following steps of an algorithm in the correct order:"**)  
 print(**"a. Calculate average = sum / 2"**)  
 print(**"b. get two numbers x and y"**)  
 print(**"c. display average"**)  
 print(**"d. Calculate sum = x + y"**)  
 ans = input(**"Answer [e.g. cbad]: "**)  
 **if** ans == **"bdac"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)

Let’s put all three function definitions in a program:

**def** question1():  
 print(**"Question 1:"**)  
 print(**"What does an assignment statement do?"**)  
 print(**"a. Display output"**)  
 print(**"b. Read input"**)  
 print(**"c. Store value in a variable"**)  
 print(**"d. Repeat the same task multiple times"**)  
 ans = input(**"Answer: "**)  
 **if** ans == **"c"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line***def** question2():  
 print(**"Question 2:"**)  
 print(**"2ndTest is a valid Python variable name. True or false?"**)  
 ans = input(**"Answer [t/f]: "**)  
 **if** ans == **"f"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line***def** question3():  
 print(**"Question 3:"**)  
 print(**"Put the following steps of an algorithm in the correct order:"**)  
 print(**"a. Calculate average = sum / 2"**)  
 print(**"b. get two numbers x and y"**)  
 print(**"c. display average"**)  
 print(**"d. Calculate sum = x + y"**)  
 ans = input(**"Answer [e.g. cbad]: "**)  
 **if** ans == **"bdac"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)

This program is incomplete because it is still missing something. If you run the program above, the execution will end immediately without doing anything. The reason is that functions are not executed unless there are statements in the program to tell the computer to execute them. Let’s add these three statements to the program:

question1() *# invoke the function question1*question2() *# invoke the function question2*question3() *# invoke the function question3*

These statements tell the computer to execute those three functions. In programming, we say that these statements invoke or call the functions. Now the program is complete:

**def** question1():  
 print(**"Question 1:"**)  
 print(**"What does an assignment statement do?"**)  
 print(**"a. Display output"**)  
 print(**"b. Read input"**)  
 print(**"c. Store value in a variable"**)  
 print(**"d. Repeat the same task multiple times"**)  
 ans = input(**"Answer: "**)  
 **if** ans == **"c"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line***def** question2():  
 print(**"Question 2:"**)  
 print(**"2ndTest is a valid Python variable name. True or false?"**)  
 ans = input(**"Answer [t/f]: "**)  
 **if** ans == **"f"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line***def** question3():  
 print(**"Question 3:"**)  
 print(**"Put the following steps of an algorithm in the correct order:"**)  
 print(**"a. Calculate average = sum / 2"**)  
 print(**"b. get two numbers x and y"**)  
 print(**"c. display average"**)  
 print(**"d. Calculate sum = x + y"**)  
 ans = input(**"Answer [e.g. cbad]: "**)  
 **if** ans == **"bdac"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
   
question1() *# invoke the function question1*question2() *# invoke the function question2*question3() *# invoke the function question3*

The following is a sample test run of the program:

Question 1:

What does an assignment statement do?

a. Display output

b. Read input

c. Store value in a variable

d. Repeat the same task multiple times

Answer: c

Correct!

Question 2:

2ndTest is a valid Python variable name. True or false?

Answer [t/f]: t

Incorrect.

Question 3:

Put the following steps of an algorithm in the correct order:

a. Calculate average = sum / 2

b. get two numbers x and y

c. display average

d. Calculate sum = x + y

Answer [e.g. cbad]: bdac

Correct!

The function definition must be placed before the statement that invokes that function. Otherwise, you will get an error message saying that the function is not defined. Suppose we move the statements that invoke question1, question2, and question3 to the beginning of the program:

*# try to invoke functions before they are defined*question1() *# invoke the function question1*question2() *# invoke the function question2*question3() *# invoke the function question3***def** question1():  
 print(**"Question 1:"**)  
 print(**"What does an assignment statement do?"**)  
 print(**"a. Display output"**)  
 print(**"b. Read input"**)  
 print(**"c. Store value in a variable"**)  
 print(**"d. Repeat the same task multiple times"**)  
 ans = input(**"Answer: "**)  
 **if** ans == **"c"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line***def** question2():  
 print(**"Question 2:"**)  
 print(**"2ndTest is a valid Python variable name. True or false?"**)  
 ans = input(**"Answer [t/f]: "**)  
 **if** ans == **"f"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)  
 print() *# display a blank line***def** question3():  
 print(**"Question 3:"**)  
 print(**"Put the following steps of an algorithm in the correct order:"**)  
 print(**"a. Calculate average = sum / 2"**)  
 print(**"b. get two numbers x and y"**)  
 print(**"c. display average"**)  
 print(**"d. Calculate sum = x + y"**)  
 ans = input(**"Answer [e.g. cbad]: "**)  
 **if** ans == **"bdac"**:  
 print(**"Correct!"**)  
 **else**:  
 print(**"Incorrect."**)

You will get the following error message when you run the program:

Traceback (most recent call last):

File "C:/Python Projects/untitled/p.py", line 1, in <module>

question1() # invoke the function question1

NameError: name 'question1' is not defined

Process finished with exit code 1

# **7.4 Another Example: Lunch Combo Program**

We learned how to define and invoke functions. Let’s look at another example.

*Lunch combos served in the cafeteria of a community college include a sandwich, a side item and a drink. For sandwich, customer can choose hamburger, cheeseburger or chicken sandwich. For side item, customer can choose one of the following: French fries, mashed potato, green bean or garden salad. For drink we can have soda, iced tea or coffee. The cafeteria wants a program for customers to enter their choices. The program will display what items are ordered.*

Before we start writing the code, let’s develop a plan:

1. Display some instruction on the computer screen
2. Ask customer to choose a sandwich, display customer’s choice
3. Ask customer to choose a side item, display customer’s choice
4. Ask customer to choose a drink, display customer’s choice

We are going to write three functions, with one function handling one choice. At the end of the program, we write a statement to display some general instruction. Then write three statements to invoke the three functions. The following is the Python code:

**def** choose\_sandwich():  
 print(**"Sandwich -- Please choose one: "**)  
 print(**"Enter 1 for hamburger"**)  
 print(**"Enter 2 for cheeseburger"**)  
 print(**"Enter 3 for chicken sandwich"**)  
 sandwich = int(input(**"Please enter 1, 2 or 3: "**))  
 **if** sandwich == 1:  
 print(**"Sandwich choice: hamburger"**)  
 **elif** sandwich == 2:  
 print(**"Sandwich choice: cheeseburger"**)  
 **elif** sandwich == 3:  
 print(**"Sandwich choice: chicken sandwich"**)  
 print()  
  
**def** choose\_sideItem():  
 print(**"Side item -- Please choose one:"**)  
 print(**"Enter 1 for French fries"**)  
 print(**"Enter 2 for mashed potato"**)  
 print(**"Enter 3 for green bean"**)  
 print(**"Enter 4 for garden salad"**)  
 side\_item = int(input(**"Please enter 1, 2, 3 or 4: "**))  
 **if** side\_item == 1:  
 print(**"Side item choice: French fries"**)  
 **elif** side\_item == 2:  
 print(**"Side item choice: mashed potato"**)  
 **elif** side\_item == 3:  
 print(**"Side item choice: green bean"**)  
 **elif** side\_item == 4:  
 print(**"Side item choice: garden salad"**)  
 print()  
  
**def** choose\_drink():  
 print(**"Drink -- Please choose one: "**)  
 print(**"Enter 1 for soda"**)  
 print(**"Enter 2 for iced tea"**)  
 print(**"Enter 3 for coffee"**)  
 drink = int(input(**"Please enter 1, 2, or 3: "**))  
 **if** drink == 1:  
 print(**"Drink choice: soda"**)  
 **elif** drink == 2:  
 print(**"Drink choice: iced tea"**)  
 **elif** drink == 3:  
 print(**"Drink choice: coffee"**)  
 print()

print(**"This program asks you to choose sandwich, side item and drink."**)  
choose\_sandwich()  
choose\_sideItem()  
choose\_drink()

Sample test run of the program:

This program asks you to choose sandwich, side item and drink.

Sandwich -- Please choose one:

Enter 1 for hamburger

Enter 2 for cheeseburger

Enter 3 for chicken sandwich

Please enter 1, 2 or 3: 2

Sandwich choice: cheeseburger

Side item -- Please choose one:

Enter 1 for French fries

Enter 2 for mashed potato

Enter 3 for green bean

Enter 4 for garden salad

Please enter 1, 2, 3 or 4: 1

Side item choice: French fries

Drink -- Please choose one:

Enter 1 for soda

Enter 2 for iced tea

Enter 3 for coffee

Please enter 1, 2, or 3: 3

Drink choice: coffee

# **7.5 Main Function**

Earlier we mentioned that the function definition had to be placed before the statement that invoked that function. Therefore, in the two examples we have seen so far, statements that invoke functions are placed right at the end of the program. However, sometimes programmers like to put statements that invoke functions and some other statements that do not belong to any function right at the beginning of the program, because this makes it easier to see the overall logic of the whole program. To make this possible, programmers like to define an extra function right at the beginning of the program and put all those statements in that function. Usually this function is named main because the mainline logic of the program is implemented there. At the end of the program, we write one statement to invoke the main function. In other words, many Python programs are organized like this:

*main():*

*# write statements here to implement the main logic of the whole program*

*# write definitions of other functions here*

*main() # invoke the main function*

Let’s rewrite the lunch combo program with this structure:

**def** main():  
 print(**"This program asks you to choose sandwich, side item and drink."**)  
 chooseSandwich()  
 chooseSideItem()  
 chooseDrink()  
  
**def** chooseSandwich():  
 print(**"Sandwich -- Please choose one: "**)  
 print(**"Enter 1 for hamburger"**)  
 print(**"Enter 2 for cheeseburger"**)  
 print(**"Enter 3 for chicken sandwich"**)  
 sandwich = int(input(**"Please enter 1, 2 or 3: "**))  
 **if** sandwich == 1:  
 print(**"Sandwich choice: hamburger"**)  
 **elif** sandwich == 2:  
 print(**"Sandwich choice: cheeseburger"**)  
 **elif** sandwich == 3:  
 print(**"Sandwich choice: chicken sandwich"**)  
 print()  
  
**def** chooseSideItem():  
 print(**"Side item -- Please choose one:"**)  
 print(**"Enter 1 for French fries"**)  
 print(**"Enter 2 for mashed potato"**)  
 print(**"Enter 3 for green bean"**)  
 print(**"Enter 4 for garden salad"**)  
 side\_item = int(input(**"Please enter 1, 2, 3 or 4: "**))  
 **if** side\_item == 1:  
 print(**"Side item choice: French fries"**)  
 **elif** side\_item == 2:  
 print(**"Side item choice: mashed potato"**)  
 **elif** side\_item == 3:  
 print(**"Side item choice: green bean"**)  
 **elif** side\_item == 4:  
 print(**"Side item choice: garden salad"**)  
 print()  
  
**def** chooseDrink():  
 print(**"Drink -- Please choose one: "**)  
 print(**"Enter 1 for soda"**)  
 print(**"Enter 2 for iced tea"**)  
 print(**"Enter 3 for coffee"**)  
 drink = int(input(**"Please enter 1, 2, or 3: "**))  
 **if** drink == 1:  
 print(**"Drink choice: soda"**)  
 **elif** drink == 2:  
 print(**"Drink choice: iced tea"**)  
 **elif** drink == 3:  
 print(**"Drink choice: coffee"**)  
 print()  
  
main() *# invoke main function*

This program is easier to understand because right at the beginning we see the statements that invoke the food item choosing functions. People who read the program code will get a general idea of what this program is doing just by reading the main function.

# **7.6 Passing Data to a Function when it is Invoked**

In the examples we have seen so far, there is no need for any function to pass/receive data to/from another function. However, some programs are different. Sometimes the calling function (i.e. the function that invokes another function) needs to send data to the called function (i.e. the function invoked) because the called function needs the data to perform its task. Sometimes the called function also has some data to send back to the calling function, typically the result of its processing (e.g. result of calculation).

Before we show how data communications between functions are done in Python, we need to talk about the **scope** of a variable. Typically a variable is accessible only in part of a program. The area in which a variable is accessible in a program is called the scope of that variable. It is primarily determined by where the variable is created. A variable created inside a function is accessible only in that function. Other functions cannot access that variable directly. The following example illustrates this point.

**def** main():  
 x = 17  
 function1()  
  
**def** function1():  
 print(x)  
  
main()

The main function creates a variable x and assigns the value 17 to it. It then invokes function1. In function1, a statement tries to display the value of x but it is unsuccessful. We get the following error message when this program is run:

File "C:/Python Projects/untitled/p3.py", line 3, in main

function1()

File "C:/Python Projects/untitled/p3.py", line 6, in function1

print(x)

NameError: name 'x' is not defined

The error message says the variable x is not defined in function1. This variable is inaccessible to function1 because it is defined in main, not in function1.

If we want function1 to display the value of x, the main function must send x to function1. Python allows a function to send data to the called function while that function is invoked. Data sent to a function when it is invoked are called **arguments**. They are listed inside the parentheses after the name of the function in the statement where the function is invoked. For example, if we change the statement in main that invokes function1 to this:

function1(x)

The value of variable x will be passed to function1 when function1 is invoked.

We also need to modify the function header of function1 to receive the value of x:

**def** function1(x):

The x in function1’s header is called a **parameter**. A parameter is a variable that receives an initial value from the calling function.

The following is the complete Python code:

**def** main():  
 x = 17  
 function1(x)  
  
**def** function1(x):  
 print(x)  
  
main()

17 will be displayed when the program above is run.

When we pass data from one function to another, the argument and the corresponding parameter need not have the same name. In the example above, there is no need to use the name x in both functions. We can name the parameter y, or any name we want.

**def** main():  
 x = 17  
 function1(x)  
  
**def** function1(y):  
 print(y)  
  
main()

17 will be passed and displayed correctly if you run the program above.

Let’s look at another example.

*Let’s write a program to covert temperature from Fahrenheit to Celsius. The program gets the temperature in Fahrenheit in the main function. It then sends the temperature to another function to calculate the temperature in Celsius.*

Python code:

**def** main():  
 f = float(input(**"Enter temperature in Fahrenheit: "**))  
 to\_celsius(f)  
  
**def** to\_celsius(temp\_f):  
 c = (temp\_f - 32)/9 \* 5  
 print(**"Equivalent temperature in Celsius: "**, c)  
  
main()

The variable f is passed as an argument to the function to\_celsius, which has a parameter temp\_f to receive it. temp\_f is then used to calculate c. Sample test run:

Enter temperature in Fahrenheit: 95

Equivalent temperature in Celsius: 35.0

# **7.7 Passing Multiple Arguments to a Function**

So far we have seen two examples for sending data to a function at the time the function is invoked. In both examples the calling function sends only one value to the called function. In fact, the calling function can send multiple values. We just need more arguments and parameters.

Let’s look at an example. We are going to write a program to calculate BMI. The program will get the weight and height from the user in the main function. Then these two numbers are passed to the function bmi\_calculator to calculate BMI. The following is the Python code:

**def** main():  
 weight = float(input(**"What is your weight [in pounds]? "**))  
 height = float(input(**"What is your height [in inches]? "**))  
 bmi\_calculator(weight, height)  
   
**def** bmi\_calculator(w, h):  
 bmi = 703 \* w / (h\*h)  
 print(**"BMI: "**, bmi)  
  
main()

The statement

bmi\_calculator(weight, height)

invokes the function bmi\_calculator and passes weight and height as arguments to it.

In the header line of the bmi\_calculator function, there are two parameters w and h to receive the arguments. The number of arguments and the number of parameters must be the same. Arguments and parameters are paired by their positions in the argument list and parameter list. The first argument in the argument list is passed to the first parameter in the parameter list, while the second argument is passed to the second parameter in the list, and so on.

The following is a sample test run of the program:

What is your weight [in pounds]? 145

What is your height [in inches]? 66

BMI: 23.40105601469238

If we change the order of the arguments or parameters in their lists, data may be sent to the wrong variables. For example, if we switch the order of w and f in the parameter list, weight will be mistakenly passed to f, while height will be mistakenly passed to h.

**def** main():  
 weight = float(input(**"What is your weight [in pounds]? "**))  
 height = float(input(**"What is your height [in inches]? "**))  
 bmi\_calculator(weight, height)  
  
**def** bmi\_calculator(h, w): *# wrong order of parameters* bmi = 703 \* w / (h\*h)  
 print(**"BMI: "**, bmi)  
  
main()

BMI calculation is wrong.

What is your weight [in pounds]? 145

What is your height [in inches]? 66

BMI: 2.2068014268727705

Let’s look at one more example about passing multiple arguments.

*Students in a Chinese course are required to take two written tests and also make two verbal presentations. Each verbal presentation is a 3-minute talk on a self-chosen topic. When determining course grade, only the higher of the two written test scores and the higher of the two presentation scores are used. Test is 60% of the course grade, while presentation is 40%. Write a program to calculate a student’s course grade. In addition to main, use one more function in this program. Read written test scores and verbal presentation scores in main, pass them to the function grade\_calculator to calculate course grade.*

The following is the Python code:

**def** main():  
 test1 = float(input(**"Enter first test score: "**))  
 test2 = float(input(**"Enter second test score: "**))  
 present1 = float(input(**"Enter first presentation score: "**))  
 present2 = float(input(**"Enter second presentation score: "**))  
 grade\_calculator(test1, test2, present1, present2)  
  
**def** grade\_calculator(t1, t2, p1, p2):  
 higher\_test = t1  
 **if** t2 > t1:  
 higher\_test = t2  
 higher\_present = p1  
 **if** p2 > p1:  
 higher\_present = p2  
 grade = higher\_test \* 0.6 + higher\_present \* 0.4  
 print(**"Your grade:"**, grade)  
  
main()

Four arguments, test1, test2, present1 and present2, are passed to four parameters, t1, t2, p1, and p2, when grade\_calculator is invoked. Once the data are passed properly, the higher test score and higher presentation score are selected to calculate course grade.

Sample test run of the program:

Enter first test score: 91

Enter second test score: 93

Enter first presentation score: 88

Enter second presentation score: 92

Your grade: 92.6

Arguments that are paired with particular parameters based on their positions in the argument list are called **positional arguments**. The arguments test1, test2, present1 and present2 in the program above are all positional arguments. Python provides another way to pair arguments with parameters. For example, we can rewrite the main function of the program above like this:

**def** main():  
 test1 = float(input(**"Enter first test score: "**))  
 test2 = float(input(**"Enter second test score: "**))  
 present1 = float(input(**"Enter first presentation score: "**))  
 present2 = float(input(**"Enter second presentation score: "**))  
 grade\_calculator(p1=present1, p2=present2, t1=test1, t2=test2)

In the statement that invokes grade\_calculator and passes arguments to it, we explicitly specify which parameter each argument is sent to. These arguments are called **keyword arguments**. A keyword argument is an argument that is paired with a parameter by specifying the parameter name. The following is the complete Python code of the modified program:

**def** main():  
 test1 = float(input(**"Enter first test score: "**))  
 test2 = float(input(**"Enter second test score: "**))  
 present1 = float(input(**"Enter first presentation score: "**))  
 present2 = float(input(**"Enter second presentation score: "**))  
 grade\_calculator(p1=present1, p2=present2, t1=test1, t2=test2)  
  
**def** grade\_calculator(t1, t2, p1, p2):  
 higher\_test = t1  
 **if** t2 > t1:  
 higher\_test = t2  
 higher\_present = p1  
 **if** p2 > p1:  
 higher\_present = p2  
 grade = higher\_test \* 0.6 + higher\_present \* 0.4  
 print(**"Your grade:"**, grade)  
  
main()

This program generates the same output as before:

Enter first test score: 91

Enter second test score: 93

Enter first presentation score: 88

Enter second presentation score: 92

Your grade: 92.6

Python also allows us to use both positional and keyword arguments in the same function call. When both types are used, we must put all positional arguments before all keyword arguments. The main function of the program above can be written like this:

**def** main():  
 test1 = float(input(**"Enter first test score: "**))  
 test2 = float(input(**"Enter second test score: "**))  
 present1 = float(input(**"Enter first presentation score: "**))  
 present2 = float(input(**"Enter second presentation score: "**))  
 grade\_calculator(test1, p1=present1, p2=present2, t2=test2)

In this version, test1 is a positional argument, which is sent to the first parameter of grade\_calculator. The other three arguments are keyword arguments. The following is the complete program:

**def** main():  
 test1 = float(input(**"Enter first test score: "**))  
 test2 = float(input(**"Enter second test score: "**))  
 present1 = float(input(**"Enter first presentation score: "**))  
 present2 = float(input(**"Enter second presentation score: "**))  
 grade\_calculator(test1, p1=present1, p2=present2, t2=test2)

**def** grade\_calculator(t1, t2, p1, p2):  
 higher\_test = t1  
 **if** t2 > t1:  
 higher\_test = t2  
 higher\_present = p1  
 **if** p2 > p1:  
 higher\_present = p2  
 grade = higher\_test \* 0.6 + higher\_present \* 0.4  
 print(**"Your grade:"**, grade)  
  
main()

If we put any keyword argument before any positional argument, that will be a syntax error. For example, the following statement:

grade\_calculator(p1=present1, test1, p2=present2, t2=test2)

generates this error message:

SyntaxError: positional argument follows keyword argument

Python also provides the ability to assign a default value to a parameter. When no argument is passed to that parameter, the default value will be used. The default value of a parameter is called a **default argument**. The following is an example:

**def** main():  
 price = float(input(**"Enter price: "**))  
 tax\_rate = float(input(**"Enter sales tax rate: "**))  
 print(**"Using tax rate entered by user:"**)  
 sales\_tax\_calculator(price, tax\_rate)  
 print(**"Using default tax rate:"**)  
 sales\_tax\_calculator(price) *# tax rate not passed***def** sales\_tax\_calculator(p, r = 0.075):  
 sales\_tax = p \* r  
 print(**"Sales tax:"**, sales\_tax)  
  
main()

The function sales\_tax\_calculator has two parameters: p and r. r has a default value of 0.075. This function is invoked twice. In the first time, the price and tax rate entered by the user are passed as arguments:

sales\_tax\_calculator(price, tax\_rate)

In the second time, only price is passed:

sales\_tax\_calculator(price) *# tax rate not passed*

The default value of r is used to calculate sales tax. The following is a sample test run:

Enter price: 100

Enter sales tax rate: 0.10

Using tax rate entered by user:

Sales tax: 10.0

Using default tax rate:

Sales tax: 7.5

# **7.8 Passing Lists to a Function**

You can send list elements, a whole list and even multiple lists to a function. First, let’s see an example of sending list elements.

**def** main():  
 list1 = [7, 6, 5, 8]  
 function1(list1[0], list1[1])  
  
**def** function1(x, y):  
 print(**"First two elements of the list:"**, x, y)  
  
main()

The elements list1[0] and list1[1] are passed to function1 and received by x and y. The passed values are then displayed:

First two elements of the list: 7 6

The example above can be modified like this:

**def** main():  
 list1 = [7, 6, 5, 8]  
 function1(list1)  
  
**def** function1(list2):  
 print(**"First two elements of the list:"**, list2[0], list2[1])  
  
main()

The whole list list1 is passed to function1 and received by list2. The first two elements are then displayed. The output is exactly the same as before:

First two elements of the list: 7 6

You can pass multiple lists to a function:

**def** main():  
 list1 = [7, 6, 5, 8]  
 list2 = [9, 1, 2, 5]

function1(list1, list2)  
   
**def** function1(list3, list4):  
 **if** len(list3) == len(list4):  
 print(**"The two lists have the same length"**)  
 **else**:  
 print(**"The two lists have different length"**)  
  
main()

The lists list1 and list2 are passed to function1 and received by list3 and list4. Their lengths are then compared:

The two lists have the same length

# **7.9 More on Arguments and Parameters**

If the value of a parameter is changed in the called function, will be corresponding argument be affected? Suppose an argument x is passed to parameter p and the value of p is changed in the called function. The question is whether the value of x will also change.

Let’s look at the following program.

**def** main():  
 x = 17  
 print(**"Value of x before function call:"**, x)  
 function1(x)  
 print(**"Value of x after function call:"**, x)  
  
**def** function1(p):  
 p = p + 5  
  
main()

In the program above, x is assigned the value of 17. It is then passed to function1 as an argument. p is the corresponding parameter in function1, which receives its initial value from x. Later in function1 p is changed to p + 5. That means the value of p is changed from 17 to 22. The question is whether this statement will make x also change to 22?

To answer this question, let’s run the program and observe the value of x after the function call:

Value of x before function call: 17

Value of x after function call: 17

Here, we can see that the value of x remains 17. That means the change in the value of a parameter has no effect on its corresponding argument.

The same rule applies to many data types in Python including integers, floating point numbers, strings, tuples, etc. However, it is a little tricky for lists. The rule is different in some situations. Let’s look at the following program:

**def** main():  
 list1 = [1, 2]  
 print(**"list1 before function call:"**, list1)  
 function1(list1)  
 print(**"list1 after function call:"**, list1)  
  
**def** function1(list2):  
 list2 = [3, 4]  
  
main()

.

The list argument list1 is passed to function1 and received by the list parameter list2. A new list with elements [3, 4] is assigned to list2 to replace the original list [1, 2]. When list1 is displayed after the function call, we can see that list1 still holds the original elements:

list1 before function call: [1, 2]

list1 after function call: [1, 2]

That means when a new list is assigned to a list parameter, the corresponding list argument has no change. It is still associated to the original list. The rule that applies to other data types also applies here.

However, if we change the value of the elements of the list parameter rather than assigning a new list to it, the effect on the argument is different. Let’s look at the following program:

**def** main():  
 list1 = [1, 2]  
 print(**"list1 before function call:"**, list1)  
 function1(list1)  
 print(**"list1 after function call:"**, list1)  
  
**def** function1(list2):  
 list2[0] = 3  
 list2[1] = 4  
  
main()

This time the statement

list2 = [3, 4]

from the previous program is replaced by two statements that change the values of the list elements:

list2[0] = 3  
 list2[1] = 4

Although they have the same effect on list2, they have very different effect on list1. This time the changes on elements of list2 also apply to list1:

list1 before function call: [1, 2]

list1 after function call: [3, 4]

You may wonder why there is such a big difference between changing values of elements and assigning a new list. When a list is passed, both the argument and parameter variables are associated to that list in the memory:

list1

list2

[1, 2]

If we write statements to change the values of the elements (e.g. list2[0] = 3 and list2[1] = 4), the associations between the variables and the list remain the same even though the elements have new values.

list1

list2

[3, 4]

In this case, both list1 and list2 are associated with the new values. That’s why changes to parameter also apply to argument in this case.

If we write a statement to associate list2 to a new list (e.g. list2 = [3, 4]), list2 is not associated with the old list any more. However, list1 is still associated with the old list. That means there are actually two lists in the memory and they are associated with two different variables:

list1

list2

[1, 2]

[3, 4]

That’s why changes to parameter do not apply to argument in this case.