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

**CSC121 PYTHON Programming**

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

Lesson 08 Functions [Part 2]

# **Objectives**

In this learning unit, students will learn:

- What value-returning functions are and how to write them

- What return values are and how to send and receive them

- How to send multiple return values

- How to receive multiple return values

# **8.1 Overview**

Functions sometimes need to send and receive data. In the last lesson we learned how one function can pass data to another function when it invokes that function. For example, suppose we have a function calc\_tests\_average that calculates the average of two test scores. We can pass midterm and final scores to this function.

*# Example 1: calculate average of midterm end final scores***def** main():  
 midterm = float(input(**"Enter midterm score: "**))  
 final = float(input(**"Enter final score: "**))  
 *# call calc\_test\_avg and pass midterm and final to it* calc\_test\_avg(midterm, final)   
  
**def** calc\_test\_avg(test1, test2):  
 average = (test1 + test2)/2  
 print(**"The average score is"**, average)  
  
main()

Suppose the user enters 85 and 95. The average will be 90.

Enter midterm score: 85

Enter final score: 95

The average score is 90.0

Process finished with exit code 0

To make this program successful, the main function must be able to pass two arguments to the calc\_test\_avg function. This is shown in the following diagram:

main()

midterm

final

calc\_test\_avg()

test1

test2

average

Now suppose we want to modify the program a little bit. In the main function, we add statements to test the average score and determine whether the student passes or fails.

*# Example 1A: calculate average of midterm end final scores***def** main():  
 midterm = float(input(**"Enter midterm score: "**))  
 final = float(input(**"Enter final score: "**))  
   
 *# need a statement here to do three things:   
 # call calc\_test\_avg, pass midterm and final to it   
 # and receive the calculated average score* **if avg\_score >= 60:  
 print("pass")  
 else:  
 print("fail")**  
**def** calc\_test\_avg(test1, test2):  
 average = (test1 + test2)/2  
 print(**"The average score is"**, average)  
   
 *# need a statement here to send average to main*main()

In the main function, a variable named avg\_score is created to store the average score. This variable is used in the if else statement to determine whether the student has passed or failed. To make this program work, we must find a way for the calc\_test\_avg function to send the value of the variable average to the variable avg\_score in main. Otherwise, the if else statement in main will not have data to determine whether the student has passed or not.

main()

midterm

final

avg\_score

calc\_test\_avg()

test1

test2

average

In programming, the value sent from the called function (i.e. calc\_test\_avg in this example) to the calling function (i.e. main in this example) is called a **return value**. A function that has a return value is called a **value-returning function**. In this example, we say that average is the **return value** of the **value-returning function** calc\_test\_avg. The next section will show you how to write statements in value-returning function to send return values.

# **8.2 Writing Value-Returning Functions**

Value-returning functions are used in many programs. They usually perform calculations or other types of processing, and then return the result to the calling function just like the average score example we saw earlier. The code in that example is incomplete. We need code in the calc\_test\_avg function to return average. We do that with a **return** statement. The syntax of a return statement is quite simple. It starts with the keyword **return**, followed by the value needed to be sent. Example:

return average

The following is the complete code of the calc\_test\_avg function:

**def** calc\_test\_avg(test1, test2):  
 average = (test1 + test2)/2  
 print(**"The average score is"**, average)  
   
 *# return average to the calling function* **return** average

Suppose test1 and test2 receive 85 and 95 from the calling function. The average of these two scores, which is 90, will be stored in the variable average and then displayed by the print statement. After that, the return statement returns 90, which is the value stored in average, to the calling function main.

# **8.3 Receiving Return Values**

We still have a missing piece in the average score example. We need to store the return value of calc\_test\_avg in the variable avg\_score so that the if else statement in main can determine whether the student has passed the course or not. To do that, we modify the original statement that calls the calc\_test\_avg function a little bit. We store the return value by making this statement an assignment statement:

avg\_score = calc\_test\_avg(midterm, final)

The statement tells the computer to store the return value of calc\_test\_avg in avg\_score. The following is the complete code of the average score example:

*# Example 1B: calculate average of midterm end final scores***def** main():  
 midterm = float(input(**"Enter midterm score: "**))  
 final = float(input(**"Enter final score: "**))  
   
 avg\_score = calc\_test\_avg(midterm, final)  
  
 **if** avg\_score >= 60:  
 print(**"pass"**)  
 **else**:  
 print(**"fail"**)  
  
**def** calc\_test\_avg(test1, test2):  
 average = (test1 + test2)/2  
 print(**"The average score is"**, average)  
  
 *# return average to the calling function* **return** average  
  
main()

The following shows a test run of the program:

Enter midterm score: 85

Enter final score: 95

The average score is 90.0

pass

Process finished with exit code 0

There is one thing you need to be very careful when you write functions with return statements. Execution of a function terminates immediately right after a return statement is finished. If you have additional statements after the return statement in the function’s definition, those statements will be ignored. Suppose we write the calc\_test\_avg function like this:

**def** calc\_test\_avg(test1, test2):  
 average = (test1 + test2)/2

**return** average  
 print(**"The average score is"**, average) #statement ignored  
  
Execution of the calc\_test\_avg function terminates immediately after average is returned. The average score will not be displayed because the print statement is ignored.

# **8.4 More Examples of Value-Returning Functions**

Let’s look at a few more examples of value-returning functions.

## Example 2

*Write a program to compare two numbers and find the larger one. Input two numbers in the main function. Write a find\_larger function to compare the numbers and return the larger one. Write a statement in main to display the larger number.*

*# Example 2: Compare two numbers and find the larger one***def** main():  
 num1 = float(input(**'Enter a number: '**))  
 num2 = float(input(**'Enter another number: '**))  
 larger = find\_larger(num1, num2)  
 print(**'The larger number is'**, larger)  
  
**def** find\_larger(x, y):  
 **if** x > y:  
 larger = x  
 **else**:  
 larger = y  
 **return** larger  
  
main()

Make up some data to test this program yourself.

Many programs need to get and validate user input. Value-returning function is a good tool for this type of work because you can move the input and validation statements out of the main function to make it simple and clean. Let’s look at an example.

## Example 3

*In this program, write a function get\_full\_time\_status to ask the user whether he is a full time student. Tell the user to answer with “y” or “n”. Validate user’s answer with a validation loop. Return “y” or “n” as a return value.*

*# Example 3: ask user whether he is full time student***def** main():  
 full\_time = get\_full\_time\_status()  
 **if** full\_time == **'y'**:  
 print(**"Full-time student."**)  
 **else**:  
 print(**"Not a full-time student"**)  
  
**def** get\_full\_time\_status ():  
 ft\_status = input(**"Are you a full-time student? [y/n] "**)  
 **while** ft\_status != **'y' and** ft\_status != **'n'**:  
 print(**"Please answer with y or n"**)  
 ft\_status = input(**"Are you a full-time student? [y/n] "**)  
 **return** ft\_status  
  
main()

Make up some data to test this program yourself.

## Example 4

Suppose we want to modify Example 3 to accept a few more possible answers from the user. We will consider the answers “Y”, “yes” and “YES” the same as “y”, and the answers “N”, “no” and “NO” the same as “n”.

Analysis: Since we need to accept a bigger set of answers from the user, we need to expand the condition of the validation loop to test those new possibilities. It will make the loop condition very long and cumbersome if we simply add more tests to the condition. A better solution is to write a new function to test whether the input answer is valid or not. We will name this function “is\_invalid “, which return True if the input is invalid and False if the input is valid.

We will write the code of the is\_invalid function in a moment. With the is\_invalid function, we can modify the input validation loop like this:

**while** is\_invalid(ft\_status) == **True**:  
 print(**"Please answer with y or n"**)  
 ft\_status = input(**"Are you a full-time student?[y/n] "**)

Now we need to write the code for the is\_invalid function. We use a Boolean variable named invalid to store whether the user’s answer is invalid. We write a series of if-elif-else statement to test whether the input answer is one of those acceptable ones. If the input answer is acceptable, we set invalid to False because the answer is actually valid. If the input answer does not match any one in the acceptable group, we set invalid to True. We return invalid as a return value. The code may look a little long but the logic is clear:

**def** is\_invalid(full\_time):  
 **if** full\_time == **'y'**:  
 invalid = **False  
 elif** full\_time == **'Y'**:  
 invalid = **False  
 elif** full\_time == **'yes'**:  
 invalid = **False  
 elif** full\_time == **'YES'**:  
 invalid = **False  
 elif** full\_time == **'n'**:  
 invalid = **False  
 elif** full\_time == **'N'**:  
 invalid = **False  
 elif** full\_time == **'no'**:  
 invalid = **False  
 elif** full\_time == **'NO'**:  
 invalid = **False  
 else**:  
 invalid = **True  
   
 return** invalid

Let’s make the modifications described above, plus we need to modify the if else statement in main function to accept the new answers. The following is the complete code:

*# Example 4: ask user whether he is full time student***def** main():  
 full\_time = get\_full\_time\_status()  
 **if** full\_time == **'y' or** full\_time == **'Y' or** full\_time == **"yes" or** full\_time == **"YES"**:  
 print(**"Full-time student."**)  
 **else**:  
 print(**"Not a full-time student"**)  
  
**def** get\_full\_time\_status ():  
 ft\_status = input(**"Are you a full-time student? [y/n] "**)  
 **while** is\_invalid(ft\_status) == **True**:  
 print(**"Please answer with y or n"**)  
 ft\_status = input(**"Are you a full-time student? [y/n] "**)  
 **return** ft\_status  
  
**def** is\_invalid(full\_time):  
 **if** full\_time == **'y'**:  
 invalid = **False  
 elif** full\_time == **'Y'**:  
 invalid = **False  
 elif** full\_time == **'yes'**:  
 invalid = **False  
 elif** full\_time == **'YES'**:  
 invalid = **False  
 elif** full\_time == **'n'**:  
 invalid = **False  
 elif** full\_time == **'N'**:  
 invalid = **False  
 elif** full\_time == **'no'**:  
 invalid = **False  
 elif** full\_time == **'NO'**:  
 invalid = **False  
 else**:  
 invalid = **True  
  
 return** invalid  
  
main()

Study the code carefully. Make up some data to test this program yourself.

## Example 5

*A company sells three models of printers:*

*Model 100: $499*

*Model 200: $549*

*Model 300: $609*

*Write a program that asks the customer to choose a model. Display the choice and the price. Write a function to validate user’s choice of model.*

*# Example 5: Validate printer model***def** main():  
 model = int(input(**'Enter model number [100/200/300]: '**))  
 **while** is\_invalid(model):  
 print(**'The valid model numbers are 100, 200 and 300.'**)  
 model = int(input(**'Enter model number [100/200/300]: '**))  
 **if** model == 100:  
 print(**'Model 100 is chosen. Please pay $499'**)  
 **elif** model == 200:  
 print(**'Model 200 is chosen. Please pay $549'**)  
 **elif** model == 300:  
 print(**'Model 300 is chosen. Please pay $609'**)  
  
**def** is\_invalid(mod\_num):  
 **if** mod\_num == 100:  
 status = **False  
 elif** mod\_num == 200:  
 status = **False  
 elif** mod\_num == 300:  
 status = **False  
 else**:  
 status = **True  
 return** status  
  
main()

Since there are not too many valid printer models, you can combine the three tests into one to shorten the code of the is\_invalid function if you want to:

*# Example 5A: Validate printer model***def** main():  
 model = int(input(**'Enter model number [100/200/300]: '**))  
 **while** is\_invalid(model):  
 print(**'The valid model numbers are 100, 200 and 300.'**)  
 model = int(input(**'Enter model number [100/200/300]: '**))  
 **if** model == 100:  
 print(**'Model 100 is chosen. Please pay $499'**)  
 **elif** model == 200:  
 print(**'Model 200 is chosen. Please pay $549'**)  
 **elif** model == 300:  
 print(**'Model 300 is chosen. Please pay $609'**)  
  
**def** is\_invalid(mod\_num):  
 **if** mod\_num == 100 **or** mod\_num == 200 **or** mod\_num == 300:  
 status = **False  
 else**:  
 status = **True  
 return** status  
  
main()

Make up some data to test this program yourself.

# **8.5 Multiple Return Statements in a Value-Returning Function**

You can put multiple return statements in a value-returning function. This is particularly useful when a function will return one of a few possible values.

## Example 6

*Modify the is\_invalid function in Example 5A to use two return statements.*

*# Example 6: Validate printer model***def** main():  
 model = int(input(**'Enter model number [100/200/300]: '**))  
 **while** is\_invalid(model):  
 print(**'The valid model numbers are 100, 200 and 300.'**)  
 model = int(input(**'Enter model number [100/200/300]: '**))  
 **if** model == 100:  
 print(**'Model 100 is chosen. Please pay $499'**)  
 **elif** model == 200:  
 print(**'Model 200 is chosen. Please pay $549'**)  
 **elif** model == 300:  
 print(**'Model 300 is chosen. Please pay $609'**)  
  
**def** is\_invalid(mod\_num):  
 **if** mod\_num == 100 **or** mod\_num == 200 **or** mod\_num == 300:  
 **return False  
 else**:  
 **return True**main()

The is\_invalid function will return either true or false, depending on which return statement executes.

# **8.6 Return Multiple Values**

You can use one return statement to return multiple values simultaneously. For example, the following get\_name\_and\_age function returns both the name and age of the user.

**def** get\_name\_and\_age():  
 name = input(**"Enter name: "**)  
 age = int(input(**"Enter age: "**))  
 **return** name, age

How do we write the statement to call get\_name\_and\_age and receive both name and age? We need to put two variables on the left hand side of the assignment statement to receive the two return values:

user\_name, user\_age = get\_name\_and\_age()

The value of name will be received and stored in user\_name, while the value of age will be received and stored in user\_age. The order of variables on the left hand side is important. If we reverse the order, return values will be received by wrong variables.

user\_age, user\_name = get\_name\_and\_age() # wrong order!

## Example 7

*Write a program that uses a function to get name and age of the user.*

*# Example 7: get user's name and age***def** main():  
 user\_name, user\_age = get\_name\_and\_age()  
 print(user\_name, **"is"**, user\_age, **"years old."**)  
  
**def** get\_name\_and\_age():  
 name = input(**"Enter name: "**)  
 age = int(input(**"Enter age: "**))  
 **return** name, age  
  
main()

Make up some data to test this program yourself.

# **8.7 Generator Expressions**

In Python, generators provide a convenient way to implement **iterators**. **Generators** allow you to declare a function that behaves like an **iterator**. An iterator is an entity that enables a program to traverse a sequence, that is, an **iterable**. A **generator** is an iterator constructor, that is, a function that creates an iterator.

The main feature of a **generator** is the evaluation of the elements of an iterable on demand. When you call a normal function with a **return** statement the function is terminated whenever it encounters a return statement. A generator function is an iterator created using a function with a **yield** statement. In a function with a **yield** statement the state of the function is “saved” from the last call and can be picked up the next time you call a generator function.

**Generator expressions** allow the creation of a generator on-the-fly without a **yield** keyword. A generator expression only yields (returns) one item at a time, it can lead to big savings in memory usage, over creating a list. **Generator expressions** make the most sense in scenarios where you need to take one item at a time, do a lot of calculations based on that item, and then move on to the next item in a sequence.

To illustrate **generators**, we will compare two different implementations of a function, **first\_n**, that represents the first ‘n’ non-negative integers:

First, consider the simple example of building a list and returning it.

*# Build and return a list*

def first\_n (n):

num = 0

nums = []

while num < n:

nums.append (num)

num += 1

return nums

sum\_of\_first\_n = sum (first\_n (1000000))

print (sum\_of\_first\_n)

Output:

499999500000

The code is quite simple and straightforward, but its builds the full list of 1 million integers in memory. There is a better way.

Python generators yield items instead of returning a list. Now, rewrite the above first\_n function as a generator function. Note the use of the yield statement, instead of a return.

*# Generate the items, instead of returning a list*

def first\_n (n):

num = 0

while num < n:

yield num

num += 1

sum\_of\_first\_n = sum (first\_n (1000000))

print (sum\_of\_first\_n)

499999500000

Note that the expression of the number generation logic is clear and natural. It is very similar to the implementation that built a list in memory, but it uses much less memory. The **yield** expression returns one number at a time to the calling function and then returns to generate another number.

When we use the **first\_n** generator function, instead of the **first\_n** function that returns a list, we do not incur the cost of building a one million element list in memory. The generator created will generate each number, which the **sum** functionwill consume to accumulate the sum. The **sum** function takes an iterable argument and automatically iterates over the passed iterable argument. The same is true with for loops.

Another example of a Python generator is with the function for generating Fibonacci numbers using Python's **yield** statement:

***# The generator fibonacci function***

**def** **fibonacci (limit):**

**a =** 0

**b** **=** 1

s = a + b

**for** **n** **in** **range(limit):**

**yield** **a**

**a = b**

**b = s**

**s = a + b**

**for** **number** **in** **fibonacci (**100**):**

**print (number)**

Output:

0

1

1

...

135301852344706746049

218922995834555169026

Generator expressions provide an additional shortcut to build generators out of expressions similar to that of list comprehensions. In fact, we can turn a list comprehension into a generator expression by replacing the square brackets "[ ]" with parentheses "( )". Alternately, we can think of list comprehensions as generator expressions wrapped in a list constructor. Consider the following example:

***# list comprehension***

**double\_n = [2 \* n for n in range(1, 11)]**

**print (double\_n)**

**[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]**

***# same as the list comprehension above***

**double\_n = list (2 \* n for n in range(1, 11))**

**print (double\_n)**

**[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]**

Notice how a list comprehension looks essentially like a generator expression passed to a list constructor. If you do not need the list, it is more efficient to use a generator expression. You can use the same syntax for generator expressions, as with list comprehensions.

Now, rewrite the sum of the first **‘n’** numbers as both a list comprehension and a generator expression:

***# List comprehension***

**n = 1000000**

**sum\_of\_first\_n = sum ([num for num in range(n)])**

**print (sum\_of\_first\_n)**

**Output:**

**499999500000**

***# Generator expression***

**n = 1000000**

**sum\_of\_first\_n = sum (num for num in range(n))**

**print (sum\_of\_first\_n)**

**Output:**

**499999500000**

The Python Software Foundation has good documentation that further explains iterators and generators. Please refer to these web pages:

[Iterators and Generators (Opens in a New Window](https://docs.python.org/3/tutorial/classes.html#iterators)).

Plain Text: https://docs.python.org/3/tutorial/classes.html#iterators

[Iterator Type and Generator Type (Opens in a New Window)](https://docs.python.org/dev/library/stdtypes.html#iterator-types).

Plain Text: https://docs.python.org/dev/library/stdtypes.html#iterator-types

# **8.8 Zip Function**

The zip function is used to combine iterables. The **zip** function takes a group of iterable objects as arguments. The **zip** function returns an **iterator** of tuples.

* If no parameters are passed, **zip** returns an empty iterator
* If a single iterable is passed, **zip** returns an iterator of 1-tuples - the number of elements in each tuple is 1.
* If multiple iterables are passed, the **i-th** tuple contains **i-th** element from each of the argument iterables. The returned iterator is truncated in length to the length of the shortest argument iterable. Suppose, two iterables are passed; one iterable containing 3 and other containing 5 elements. Then, the returned iterator has 3 tuples.

Now, the returned iterator from the zip function is not an iterable. In order to use the combined iterables, the returned iterator must be passed as an argument to the list or tuple function:

For example:

***# prints returned iterator info***

**x = [1, 2, 3]**

**y = [4, 5, 6]**

**z\_list = zip (x, y)**

**print (z\_list)**

**Output:**

**<zip object at 0x00000261F76C8A48>**

***# prints zipped lists***

**x = [1, 2, 3]**

**y = [4, 5, 6]**

**z\_list = list (zip (x, y))**

**print (z\_list)**

**Output:**

**[(1, 4), (2, 5), (3, 6)]**

The **zip** function in conjunction with the \* operator can be used to unzip a list.

***# prints unzipped lists***

**x2, y2 = zip (\*z\_list)**

**print (list (x2))**

**print (list (y2))**

Output:

**[1, 2, 3]**

**[4, 5, 6]**

# **8.9 Anonymous Functions: Lambda**

Python supports the creation of anonymous functions, using a construct called **"lambda"**. Anonymous functions are inline functions which are defined without a name. Lambda functions can have any number of arguments, but only one expression. The expression is evaluated and returned. Lambda is a tool for building functions.

While normal functions are defined using the **def** keyword, in Python anonymous functions are defined using the **lambda** keyword.

Consider the following normal function that returns its parameter multiplied by 2:

**def double (x):**

**return x \* 2**

The corresponding lambda function would be written as

**double = lambda x: x \* 2**

In the above program,  **x** is the argument and **x \* 2** is the expression that gets evaluated and returned. Note that the lambda function does not include the **return** keyword, but it is implied.

This function has no name. It returns a function object which is assigned to the identifier **double**. We can now call it as a normal function as shown in the following code snippets:

***# regular function***

**def double (x):**

**return x \* 2**

**print (double (10))**

**Output:**

**20**

***# lambda function***

**double = lambda x: x \* 2**

**print (double (10))**

**Output:**

**20**

If this is confusing, just think of **lambda** as being just a fancy way of saying **function**. Other than its name, there is nothing obscure, intimidating or cryptic about it. When you read the following line, replace **lambda** by **function** in your mind and add return before the last expression. Notice that this syntax is very similar to the function syntax used in mathematics.

***# In Python***

**f = lambda x: x + 1**

**print (f(3))**

**Output:**

**4**

***# In Math***

**f(x): x + 1**

**f(3) is 4**

You can put a lambda definition anywhere a function is expected, and you don't have to assign it to a variable at all. Like nested function definitions, lambda functions can reference variables from the containing scope:

***# lambda function returned by a function***

**def increment\_by\_n (n):**

**return lambda x: x + n**

**inc = increment\_by\_n (3)**

**print (inc (1))**

**Output:**

**4**

The above example uses a **lambda** expression to return a function. The variable n is 3 and the variable x is 1. Thus, x is incremented by 3.

Another use of lambda is to pass a small function as an argument. The **sort** method of the list class can take a function argument that specifies the index on which to sort a list of lists.

***# sort the inner lists on their second element***

***# - the one with index 1***

**pairs = [(1, 'one'), (2, 'two'), (3, 'three')]**

**pairs.sort (key=lambda item: item[1])**

**print (pairs)**

**[(1, 'one'), (3, 'three'), (2, 'two')]**

# **8.10 Sorted Function**

A list can be sorted simply in a couple of ways. The first way is shown above using the sort method. Python also has a sorted function that takes a list or some other iterable object as an argument and sorts it. The **sorted** function can also take a function as an argument in the same way as the **sort** method.

**pairs = [(1, 'one'), (2, 'two'), (3, 'three')]**

**# sort the inner lists on their second element in reverse order,**

**# the one with index 1.**

**print (sorted (pairs, key=lambda p: p[1], reverse=True))**

**Output:**

**[(2, 'two'), (3, 'three'), (1, 'one')]**

**# without using lambda, this would be**

**def sort2 (elem):**

**return (elem[1])**

**print (sorted (pairs, key=sort2, reverse=True))**

**Output:**

**[(2, 'two'), (3, 'three'), (1, 'one')]**