The Return Statement

by Sophia

WHAT'S COVERED

In this lesson, you will learn about the return statement in functions. Specifically, this lesson covers:

1. Implicit Return
2. Explicit Return
3. Return Multiple Values
4. Return With Conditionals

1. Implicit Return

KEY CONCEPT

The **return statement** is an important part of functions and methods, as it allows the functions and methods to return Python objects back to the code that called them. These objects that are being returned are called the function or method’s **return value**.

Here is an example.

EXAMPLE

```
def return_one_hundred():
    return 100
```

In this example, `return 100` is the return statement whereas the return value is the integer 100. So, the **return statement** is the actual line that starts with "return...", while the return value is the value that's returned from the return statement. With the return value, we can then use those objects to perform other processes in our programs. Although it may seem obvious how to use return statements, it is an important skill to learn in order to ensure that our code is robust.

In the prior lesson, we discussed our ability to create functions that can be reused. These functions return any Python object to the code that called the function. As we’ve seen, a function can take any number of arguments (even none) and perform some operation(s). Lastly, it can also return a value or object. All Python functions have a return value, either implicitly or explicitly.

BRAINSTORM

Before we continue, do you know the difference between implicit and explicit? Implicit means that something is understood, though it is not clearly defined. And explicit means that something is stated directly, with no room for confusion. In Python, any time we see the word “return”, the return value is explicitly called, whereas without the word “return”, it is implicitly called.

In the prior lesson, we also defined and used a function called `print_stuff()`. Let’s rerun that function and see some return values. We will set a variable called something to `print_stuff()` and output that to the screen. We will also double check that variable’s type with the `type()` function.

Here is our code.

EXAMPLE

```
def print_stuff():
    print("#######################")
    print('Using for comment block')
    print("#######################")

something = print_stuff();
print(something)
print(type(something))
```

And here is the output . Notice that for `print(something)`, because no return type was returned from the function explicitly, None was returned. The value None is of the type NoneType.

```
#######################
Using for comment block
#######################
None
<class 'NoneType'>
```

It is important to note that a Python function will always have a return value. If we don’t specifically return a value in Python, Python will implicitly return a default value for us. The default value will always be None. In our example above, this makes sense because the purpose of our `print_stuff()` function was to output the three lines to the screen. So, the `print_stuff()` function doesn’t really need to return anything useful or meaningful. Any function that does not explicitly return a value will automatically return None. If you see None being returned from a function, but are expecting a value, you will want to double-check if there’s an error. In this example, None is expected since our function did not return any values.

TRY IT

Directions: Go ahead and add the example above into the IDE and see if you get the same implicit `type()` in the output.

TERMS TO KNOW

Return Statement

The return statement is an important part of functions and methods as it allows the functions and methods to return Python objects back to the code that called them. The return statement is the actual line that starts with the word "return".

Return Value

These objects that are being returned from the return statement are called the function or method’s return value.

2. Explicit Return

Anytime that we have an explicit return statement that is executed in a function, the function executed is terminated and sends the return value back to the code that called it. Typically, the return statement is the last line in a function block, but it does not have to be. To add the explicit return statement, we would use the reserved keyword `return` followed by the optional return value. If no return value is passed, None is returned instead. So, the reserved keyword **return** is used to exit a function and return a value with either an optional return value we specify, or None if no value is specified.

Let’s look at our previous example again.

↪ EXAMPLE

```
def return_one_hundred():
    return 100
```

In the header of the function, we defined the function with the name `return_one_hundred()` with no parameters. Then in the function’s body, we added the return statement at the end as `return 100`. The value 100 is the function’s return value. This means that anytime we make a call to the `return_one_hundred()` function, the function returns back the value 100 to the code calling the function.

↪ EXAMPLE

```
def return_one_hundred():
    return 100

something = return_one_hundred();
print(something)
print(type(something))
```

We see the following on the output.

```
100
<class 'int'>
```

We set the variable something to be the function’s return value. Then output the value and the type of the return value.

TRY IT

Directions: Now try this last example using the explicit return statement. Did you get the same value output? Try changing the return value to something other than an integer and see if you change the type. Try “$100” and 100.00.

If we define a function that has an explicit return value, we can use that return value in any additional expressions.

EXAMPLE

```
def return_one_hundred():
    return 100

print(return_one_hundred())
print(return_one_hundred() + return_one_hundred())
print(return_one_hundred() * 5)
```

Here is the output.

```
100
200
500
```

Since we have the `return_one_hundred()` function returning a numeric value, we can use that value in a mathematical operation in the same way as if we simply used variables to set those values. This is how we can make use of the function’s return value.

TRY IT

Directions: Try the additional print lines. Try making some additional output lines of code using the explicit return value.

It’s important to note that the return statement can only be used inside of a function or a method definition. If we try to use it anywhere else, we will get a syntax error:

EXAMPLE

```
def return_one_hundred():
    return 100
return 1
```

Here is that output.

```
  File "/home/main.py", line 3
    return 1
    ^^^^^^^^
SyntaxError: 'return' outside function
```

The return statement `return 1` is the line that created the error, as it’s outside of the function. It is not indented like the `return 100` statement, thus it is not part of the function.

Since we can use any Python object as a return value, we can return anything from strings, lists, tuples, dictionaries, sets, functions, classes or instances (we will be covering instances in Unit 3).

Remember when we had a list and in order to calculate the average, we had to perform the calculation manually?

```
print("Average: ",sum([2, 45, 9, 17, 1, 4])/len([2, 45, 9, 17, 1, 4]))
```

One of the issues with calculating the average is that we would have to pass the list to the sum function and then again to the len function. We could create a list and pass in the variable, but ideally, this should only be done once.

We can build a quick function called `calculate_average` that includes a return statement that utilizes the `sum()` and `len()` (length) functions.

EXAMPLE

```
def calculate_average(numbers):
  return sum(numbers)/len(numbers)

print(calculate_average([10, 20, 30, 40]))
```

Here is that output.

```
25.0
```

Notice that the average of the numbers in the example’s list is the sum divided by the length, or the sum (10+20+30+40 = 100) divided by the length (4)—so, 100/4 = 25.

TRY IT

Directions: Go ahead and try out the previous example and see how much easier it is just to use the return statement.

Now, every time that we need to calculate the average of the list, we can use the function directly. Notice that as part of the function, we didn’t bother to first set a variable to the calculation and then return the value; we just had the calculation as part of the return statement. This is a common practice to simplify the function definition. However, we could have used a variable.

EXAMPLE

```
def calculate_average(numbers):
 my_avg = sum(numbers)/len(numbers)
 return my_avg

print(calculate_average([1,2,3,4]))
```

Here it is with the variable `my_avg` returned this time.

```
2.5
```

TRY IT

Directions: Try this function with the added variable this time.

TERM TO KNOW

return

The reserved keyword `return` is used to exit a function and return a value, either an optional return value we specify or None if no value is specified.

3. Return Multiple Values

We can also use the return statement to return multiple values from a function. To do that, we just need to have several return values separated by commas. This can be useful if we always want certain calculations to be returned. Using our calculation of the average above, perhaps we want to have the values returned for the sum, length, and average all at once. We can do that using multiple values for the return statement and once again using the `sum()` and `len()` functions.

EXAMPLE

```
def calculate_multiple(numbers):
  return sum(numbers), len(numbers), sum(numbers)/len(numbers)

print(calculate_multiple([10, 20, 30, 40]))
```

Here is that output.

```
(100, 4, 25.0)
```

In doing so, we have a tuple that is returned to the user showing the sum, len, and average of the list without having to have separate function calls. Remember, a tuple has round brackets with elements separated by commas.

KEY CONCEPT

So why did I just get a tuple returned? A tuple is the default return when you have multiple parameters, as it's a set number of items being returned. This is a part of Python and not something that we can control.

TRY IT

Directions: Your turn. Create the return statement to return the sum, length, and average of a few numbers in a list.

There is a built-in function `divmod()` which returns multiple values. The **divmod()** function takes in two numbers as arguments and returns two numbers in a tuple. The first number is the quotient of the two input numbers and the second one is the remainder of the division.

EXAMPLE

```
print(divmod(21, 5))
print(divmod(15, 5))
```

Here is that output.

```
(4, 1)
(3, 0)
```

In the code above, 21/5 has the quotient of 4 with the remainder of 1. The other function call 15/5 has the quotient of 3 and the remainder of 0.

TRY IT

Directions: Try using the `divmod()` function in the previous example.

TERM TO KNOW

divmod()

The `divmod()` function takes in two numbers as arguments and returns two numbers in a tuple.

4. Return With Conditionals

Typically a return statement is located at the end of the body of the function. We are not limited to a single return statement, nor do we have to have them only at the end of the body of the function. If a function has more than one return statement, the first one that is executed will determine the end of a function and the return value.

This is a common approach, as we may need to have different return statements depending on the result of some conditions. For example, if we have a function that takes in a number and is meant to return the absolute value of that number, we will have different criteria. If the number is greater than 0, it returns that number. If the number is less than 0, it returns the non-negative value of that number. If it is 0, then it simply returns 0.

KEY CONCEPT

What is absolute value? The absolute value of a number is the number's distance from 0. This makes any negative number positive, while positive numbers are unaffected. So, the absolute value of 6 is 6 and the absolute value of -6 is also 6.

EXAMPLE

```
def calculate_absolute(number):
  if number > 0:
    return number
  elif number < 0:
    return -number #this is taking a negative number and turning it into a positive number
  else:
    return 0

print(calculate_absolute(-5))
print(calculate_absolute(5))
print(calculate_absolute(0))
```

Here is that output.

```
5
5
0
```

The condition checks for values that are greater than 0, less than 0 or if the number is 0. It's important that we ensure that each possible option gets its own return statement.

TRY IT

Directions: Add the program to the IDE and see if you get the same results.

If we look at this simple program a bit closer, we can also see that there really isn't any difference between returning a 0 or the number itself since we are converting any less than 0 numbers to a positive number anyway.

EXAMPLE

```
def calculate_absolute(number):
  if number >= 0:
    return number
  else:
    return -number

print(calculate_absolute(-5))
print(calculate_absolute(5))
print(calculate_absolute(0))
```

Here is that output.

```
5
5
0
```

TRY IT

Directions: Try it again this time without the need for checking for any numbers less than 0.

In essence, we don’t have to do a separate check for the number being less than 0, as that would be any other number that is being passed in. We can remove that else statement and have the return statement as the final statement in the body.

EXAMPLE

```
def calculate_absolute(number):
  if number >= 0:
    return number
  return -number

print(calculate_absolute(-5))
print(calculate_absolute(5))
print(calculate_absolute(0))
```

Here is that output.

```
5
5
0
```

TRY IT

Directions: Try the final revision of this program with only the single line conditional containing two return statements.

BIG IDEA

One thing to remember when using if statements that utilize several return statements is that we don’t have to use an else clause to cover the last condition. Instead, we can just add the return statement at the end of the function’s code block.

SUMMARY

In this lesson, we learned about the return statement returning values implicitly. With **implicit returns**, generally, no values are being passed back directly. In these cases, None will be returned. We learned about **explicitly returning values** that we identify. We also learned about **returning multiple values** from a single function in cases where we may have multiple items to be passed back. Lastly, we learned about **returning values based on conditional statements**.

Best of luck in your learning!

Source: THIS CONTENT AND SUPPLEMENTAL MATERIAL HAS BEEN ADAPTED FROM “PYTHON FOR EVERYBODY” BY DR. CHARLES R. SEVERANCE ACCESS FOR FREE AT www.py4e.com/html3/ LICENSE: CREATIVE COMMONS ATTRIBUTION 3.0 UNPORTED.

TERMS TO KNOW

Return Statement

The return statement is an important part of functions and methods as it allows the functions and methods to return Python objects back to the code that called them. The return statement is the actual line that starts with the word"return".

Return Value

These objects that are being returned from the return statement are called the function or method’s return value.

divmod()

The divmod() function takes in two numbers as arguments and returns two numbers in a tuple.

return

The reserved keyword return is used to exit a function and return a value, either an optional return value we specify or None if no value is specified.