Exploration: Functions

Function calls

We've already been using function calls to print values, read input, get the type of a value or variable, cast from one type to another, and to create ranges. A function call consists of the name of a function followed by a pair of parentheses, which contain any **arguments** for the function. A function carries out some task and may or may not **return** a value when it's done.

int(-12.5)

In the above function call, the name of the function is *int*. It's being passed one argument, the number -12.5. In this example, the int function would return the integer -12 as its **return value**.

print("num orders =", num\_orders)

In the above function call, the name of the function is *print*. It's being passed two arguments, the string "num orders =" and the variable *num\_orders*. The print function doesn't have a return value (technically it does, but it's a special value called *None*).

print(float(input()))

As we've already seen, you can chain function calls together. The above line would read input from the user, cast that input to a float, and then print out the result.

When you make a function call that returns a value, make sure you don't accidentally "drop" it. Most of the time you would either assign the return value to a variable or have the function call be part of a larger expression that uses the return value (for example chaining function calls).

Function definitions

To define a function, you use the "def" keyword. For example, we could define a function that adds together three numbers as in the example below.  Try clicking "run" and then in the bottom window type "sum\_three(-5, 189, 12)" and hit <enter>.

In the above definition, the name of the function is *sum\_three*. The rules for function names are the same as for variable names. The **parameters** of the function are *num1*, *num2*, and *num3*. When you call a function, the arguments in the function call are matched up (in order) with the parameters given in the function definition, so if we called the function like this:

> result = sum\_three(-5, 189, 12)

Then -5 is assigned to num1, 189 is assigned to num2, and 12 is assigned to num3. Just to reiterate, the *arguments* are the values that are passed in the function call. The *parameters* are the names in parentheses in the first line of the function definition.

When the computer is executing the program and it reaches a function call, then it jumps to the function definition and executes the code there. Once the function is done, the computer returns back to the function call and continues from there. A function is done when the computer either reaches the end of the code in the definition, or it reaches a return statement, whichever happens first. A return statement doesn't have to go at the end of the function, and you can even have multiple return statements in a function. As soon as the code executes **any** return statement, it immediately ends the function and returns execution (and any return value) back to the function call. If your function doesn't have a return value, you can still use the return keyword if you want to end the function before reaching the end of the definition for some reason. You can see how this works in the first function below.

However the more normal way to write that particular example would be the second version. You can see that they both give the same output for the same input.

The sum\_three function could also be defined like this:

This definition is more concise and probably preferable for this function, but in cases where you have a long and/or complicated expression, splitting it up into separate steps can make it easier to read and debug.

If a function doesn't have any parameters, you still need the parentheses following the function name:

def print\_greeting():  
 print("howdy")

We would call this function like so:

print\_greeting()

A function must be defined somewhere above the place where it's called. That is, you cannot have the function call appear earlier in the file than the function definition. Otherwise when the computer is executing your program it will see the function call before it knows what that function does.

Function scope

Variables in a function definition only exist while that function is executing. When the function is done, those variables go away. Such variables have **function scope**. If a function uses a variable name that is already being used in some other part of the program, it is a separate variable - changing the one in the function does not change any other variables of that name. If you run the following code:

Then the program will start at the line "num = 5", which assigns the value 5 to a variable called *num*. Next is the function call "func()". At this point execution jumps to the function definition, which assigns the value 12 to a variable called *num*, but this is a separate variable, which has function scope. Assigning 12 to it does not affect the value of the other variable named num. Next, the function prints out the value of the local *num* variable, which is 12. Having reached the end of the function definition, execution returns back to the function call. The next line prints out the value of the original *num* variable, which is still 5.

Default arguments

It's possible to give a parameter a default value that it will have if no argument is passed for it in the function call. The syntax looks like you're assigning a value to that parameter, but if the function call does pass an argument for that parameter, then the default argument is not used. Here's a simple example:

def sum(num\_1, num\_2, num\_3=0):  
 return num\_1 + num\_2 + num\_3

In this example, num\_1, num\_2, and num\_3 are **parameters**, and zero is a default **argument**. The default argument allows us to call *sum* with either two or three arguments.

Parameters with default arguments must all go at the end of the parameter list. In the *sum* function, we can't give num\_1 a default argument and then not give num\_2 and num\_3 default arguments.

Remember that when you use a default argument, the value on the right side of the '=' is the default **argument**, while the value on the left side is the **parameter** you're assigning it to.

Changing the value of a parameter

If we pass a variable to a function and the function changes that parameter, what happens to the original variable that was passed? Does it change because the parameter in the function was changed? Let's try it and see:

Here the program starts by assigning 8 to *num*. Next it calls the function *square\_val*, passing *num* as an argument. Now execution jumps to the function definition, and the argument *num* is matched with the parameter *val*, so now *val* and *num* refer to the same value. Next, *val* is assigned the value of *val* times itself. Then the function prints out the value of *val*, which is 64. Since the function has finished, execution returns to the function call. Next the value of *num* is printed out, which is 8. We can see that even though *val* and *num* referred to the same value, changing *val* did not change *num*. This is not because of scope. We'll discuss the reason behind it in a later module. What if we want *num* to change? We can accomplish that like so:

In this version we changed two things:

1. We have the function return the altered value.
2. When we call the function, we store the result of the function call (its return value) in *num*.

Now when we run the code, we see that *num* is updated to the square of its previous value.

Docstrings

A docstring is a special type of string you should use to help document your functions. A docstring gives a description of the function's purpose, with a triple quotation mark at the beginning and end. It goes on the first indented line of the function.

Python has a built-in help function that will print out the description you provided in the docstring:

> help(area\_of\_circle)

There also exist third-party tools that will read and display docstrings in various formats.

Stack traces

When a Python program crashes, the error messages will include a stack trace that tells you which functions were active at the time. If function A calls function B, which calls function C, and if there is an error in C that crashes the program, then the stack trace that is printed out will include which line of code we were on in functions A, B, and C. Here is an example:

When we try to call func\_A(), it calls func\_B(), which calls func\_C(). Then because func\_C tries to use a nonexistent variable, the program crashes and gives us the stack trace.

This can be helpful when trying to figure out exactly what sequence of events caused a program to crash.

Exercises

(See the module overview for a link to example solutions.)

1. Write a function that takes a numeric parameter and returns twice that number. Write a print statement that calls your function with the number 5 and prints the function's return value.

Sample input: 5  
Expected output: 10

2. Write a function that takes two string parameters and returns the string formed by concatenating those two strings together. Write a print statement that calls your function with the values "hello " and "world" and prints the function's return value.

Sample input: "hello ", "world"  
Expected output: "hello world"

3. Write a function named is\_even that takes an integer parameter and returns a bool value - True if the integer is even, and False if the integer is odd.

Sample input: 7  
Expected output: False