Python Essentials - Module 1

Cisco Networking Academy

July 10, 2022

Abstract

In this module you'll learn:

- how to write and run simple Python programs;
- what Python literals, operators, and expressions are;
- what variables are and what are the rules that govern them;
- how to perform basic input and output operations.

1 Hello, World!

It's time to start writing some real, working Python code. It'll be very simple for the time being.

As we're going to show you some fundamental concepts and terms, these snippets of code won't be serious or complex.

Run the code in the editor window on the right. If everything goes okay here, you'll see the line of text in the console window.

Alternatively, launch IDLE, create a new Python source file, fill it with this code, name the file and save it. Now run it. If everything goes okay, you'll see the rhyme's line in the IDLE console window. The code you have run should look familiar. You saw something very similar when we led you through the setting up of the IDLE environment.

Now we'll spend some time showing and explaining to you what you're actually seeing, and why it looks like this.

As you can see, the first program consists of the following parts:

- the word

  ```
  print
  ```

- an opening parenthesis;
- a quotation mark;
- a line of text:

```
    Hello, World!
```

- another quotation mark;
- a closing parenthesis.

Each of the above plays a very important role in the code.

```
print("Hello, World")
```

2 The print() Function

Look at the line of code below:

```
print("Hello, World")
```

The word **print** that you can see here is a **function name**. That doesn't mean that wherever the word appears it is always a function name. The meaning of the word comes from the context in which the word has been used.

You've probably encountered the term function many times before, during math classes. You can probably also list several names of mathematical functions, like sine or log.

Python functions, however, are more flexible, and can contain more content than their mathematical siblings.

A function (in this context) is a separate part of the computer code able to:

- **cause some effect** (e.g., send text to the terminal, create a file, draw an image, play a sound, etc.); this is something completely unheard of in the world of mathematics;
- **evaluate a value** (e.g., the square root of a value or the length of a given text) and **return it as the function's result**; this is what makes Python functions the relatives of mathematical concepts.

Moreover, many of Python functions can do the above two things together.

Where do the functions come from?

- They may come from Python itself; the print function is one of this kind; such a function is an added value received together with Python and its environment (it is built-in); you don't have to do anything special (e.g., ask anyone for anything) if you want to make use of it;
- they may come from one or more of Python's add-ons named modules; some of the modules come with Python, others may require separate installation - whatever the case, they all need to be explicitly connected with your code (we'll show you how to do that soon);

- you can write them yourself, placing as many functions as you want and need inside your program to make it simpler, clearer and more elegant.

The name of the function should be significant (the name of the print function is self-evident).

Of course, if you're going to make use of any already existing function, you have no influence on its name, but when you start writing your own functions, you should consider carefully your choice of names.

As we said before, a function may have:

- an **effect**;
- a **result**.

There's also a third, very important, function component - **the argument(s)**.

Mathematical functions usually take one argument, e.g., sin(x) takes an x, which is the measure of an angle.

Python functions, on the other hand, are more versatile. Depending on the individual needs, they may accept any number of arguments - as many as necessary to perform their tasks. Note: any number includes zero - some Python functions don't need any argument.

In spite of the number of needed/provided arguments, Python functions strongly demand the presence of a pair of parentheses - opening and closing ones, respectively.

If you want to deliver one or more arguments to a function, you place them inside the parentheses. If you're going to use a function which doesn't take any argument, you still have to have the parentheses.

Note: to distinguish ordinary words from function names, place a pair of empty parentheses after their names, even if the corresponding function wants one or more arguments. This is a standard convention.

The function we're talking about here is `print()` .

Does the `print()` function in our example have any arguments?

Of course it does, but what are they?

The only argument delivered to the `print()` function in this example is a string:

```
print("Hello, World!")
```

As you can see, the **string is delimited with quotes** - in fact, the quotes make the string - they cut out a part of the code and assign a different meaning to it.

You can imagine that the quotes say something like: the text between us is not code. It isn't intended to be executed, and you should take it as is.

Almost anything you put inside the quotes will be taken literally, not as code, but as data. Try to play with this particular string - modify it, enter some new content, delete some of the existing content.

There's more than one way to specify a string inside Python's code, but for now, though, this one is enough.

So far, you have learned about two important parts of the code: **the function** and **the string**. We've talked about them in terms of syntax, but now it's time to discuss them in terms of semantics.

The function name (print in this case) along with the parentheses and argument(s), forms the function invocation.

We'll discuss this in more depth soon, but we should just shed a little light on it right now.

```
print("Hello, World!")
```

What happens when Python encounters an invocation like this one below?

```
function_name(argument)
```

Let's see:

1. Python checks if the name specified is legal (it browses its internal data in order to find an existing function of the name; if this search fails, Python aborts the code);
2. Python checks if the function's requirements for the number of arguments allows you to invoke the function in this way (e.g., if a specific function demands exactly two arguments, any invocation delivering only one argument will be considered erroneous, and will abort the code's execution);
3. Python leaves your code for a moment and jumps into the function you want to invoke; of course, it takes your argument(s) too and passes it/them to the function;
4. The function executes its code, causes the desired effect (if any), evaluates the desired result(s) (if any) and finishes its task;
5. Python returns to your code (to the place just after the invocation) and resumes its execution.

Three important questions have to be answered as soon as possible:

1. What is the effect the `print()` function causes?

The effect is very useful and very spectacular. The function:

- takes its arguments (it may accept more than one argument and may also accept less than one argument)
- converts them into human-readable form if needed (as you may suspect, strings don't require this action, as the string is already readable)
- and sends the resulting data to the output device (usually the console); in other words, anything you put into the print() function will appear on your screen.

No wonder then, that from now on, you'll utilize `print()` very intensively to see the results of your operations and evaluations.

2. What arguments does `print()` expect?

Any. We'll show you soon that `print()` is able to operate with virtually all types of data offered by Python. Strings, numbers, characters, logical values, objects - any of these may be successfully passed to `print()`.

3. What value does `print()` function return?

None. Its effect is enough.

2.1 The print() function - instructions

You have already seen a computer program that contains one function invocation. A function invocation is one of many possible kinds of Python instructions.

Of course, any complex program usually contains many more instructions than one. The question is: how do you couple more than one instruction into the Python code?

Python's syntax is quite specific in this area. Unlike most programming languages, **Python requires that there cannot be more than one instruction in a line**.

A line can be empty (i.e., it may contain no instruction at all) but it must not contain two, three or more instructions. This is strictly prohibited.

Note: Python makes one exception to this rule - it allows one instruction to spread across more than one line (which may be helpful

when your code contains complex constructions).

Let's expand the code a bit, you can see it in the editor. Run it and note what you see in the console.

Your Python console should now look like this:

```
The itsy bitsy spider climbed up the waterspout.
Down came the rain and washed the spider out.
```

This is a good opportunity to make some observations:

- the program invokes the `print()` **function twice**, and you can see two separate lines in the console - this means that `print()` begins its output from a new line each time it starts its execution; you can change this behavior, but you can also use it to your advantage;
- each `print()` invocation contains a different string, as its argument and the console content reflects it - this means **that the instructions in the code are executed in the same order** in which they have been placed in the source file; no next instruction is executed until the previous one is completed (there are some exceptions to this rule, but you can ignore them for now)

We've changed the example a bit - we've added one **empty** `print()` function invocation. We call it empty because we haven't delivered any arguments to the function.

You can see it in the editor window. Run the code.

What happens?

If everything goes right, you should see something like this:

```
The itsy bitsy spider climbed up the waterspout.

Down came the rain and washed the spider out.
```

As you can see, the empty `print()` invocation is not as empty as you may have expected - it does output an empty line, or (this interpretation is also correct) its output is just a newline.

This is not the only way to produce a **newline** in the output console. We're now going to show you another way.

2.2 The print() function - the escape and newline characters

We've modified the code again. Look at it carefully.

```
print("The itsy bitsy spider\nclimbed up the waterspout.")
print()
print("Down came the rain\nand washed the spider out.")
```

There are two very subtle changes - we've inserted a strange pair of characters inside the rhyme. They look like this: `\n`.

Interestingly, while **you can see two characters, Python sees one.**

The backslash (\) has a very special meaning when used inside strings - this is called **the escape character**.

The word escape should be understood specifically - it means that the series of characters in the string escapes for the moment (a very short moment) to introduce a special inclusion.

In other words, the backslash doesn't mean anything in itself, but is only a kind of announcement, that the next character after the backslash has a different meaning too.

The letter `n` placed after the backshlash comes from the work *newline*.

Both the backslash and the `n` form a special symbol named **a newline character**, which urges the console to start a **new output line**.

Run the code. Your console should now look like this:

```
The itsy bitsy spider
climbed up the waterspout.

Down came the rain
and washed the spider out.
```

As you can see, two newlines appear in the nursery rhyme, in the places where the `\n` have been used.

This convention has two important consequences:

1. If you want to put just one backslash inside a string, don't forget its escaping nature - you have to double it, e.g., such an invocation will cause an error:

```
print("\")
```

while this one won't:

```
print("\\")
```

Not all escape pairs (the backslash coupled with another character) mean something.

Experiment with your code in the editor, run it, and see what happens.

2.3 The print() function - using multiple arguments