Python Essentials - Module 2

Cisco Networking Academy

July 12, 2022

Contents

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 Your Very First Program

1.1 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")
```

1.2 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.

1.2.1 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.

1.2.2 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.

1.2.3 The print() function - using multiple arguments

So far we have tested the `print()` function behavior with no arguments, and with one argument. It's also worth trying to feed the `print()` function with more than one argument.

This is what we're going to test now:

```
print("The itsy bitsy spider","climbed up","the water spout.")
```

There is one `print()` function invocation, but it contains **three arguments**. All of them are strings.

The arguments are **separated by commas**. We've surrounded them with spaces to make them more visible, but it's not really necessary,

and we won't be doing it anymore.

In this case, the commas separating the arguments play a completely different role than the comma inside the string. The former is a part of Python's syntax, the latter is intended to be shown in the console.

If you look at the code again, you'll see that there are no spaces inside the strings.

Run the code and see what happens.

The console should now be showing the following text:
`The itsy bitsy spider climbed up the waterspout.`

The spaces, removed from the strings, have appeared again. Can you explain why?

Two conclusions emerge from this example:

- a `print()` function invoked with more than one argument **outputs them all on one line**;
- the `print()` function **puts a space between the outputted arguments** on its own initiative.

1.2.4 The print() function - the positional way of passing the arguments

Now that you know a bit about `print()` function customs, we're going to show you how to change them.

You should be able to predict the output without running the code in the editor.

```
print("My name is", "Python.")
print("Monty Python.")
```

Output will be:

```
My name is Python.
Monty Python.
```

The way in which we are passing the arguments into the `print()` function is the most common in Python, and is called **the positional way** (this name comes from the fact that the meaning of the argument is dictated by its position, e.g., the second argument will be outputted after the first, not the other way round).

1.3 The print() function - the keyword arguments

Python offers another mechanism for the passing of arguments, which can be helpful when you want to convince the `print()` function to change its behavior a bit.

We aren't going to explain it in depth right now. We plan to do this when we talk about functions. For now, we simply want to show you how it works. Feel free to use it in your own programs.

The mechanism is called keyword arguments. The name stems from the fact that the meaning of these arguments is taken not from its location (position) but from the special word (keyword) used to identify them.

The `print()` function has two keyword arguments that you can use for your purposes. The first of them is named `end`.

You can see a very simple example of using a keyword argument:

```
print("My name is", "Python.", end=" ")
print("Monty Python.")
```

In order to use it, it is necessary to know some rules:

- a keyword argument consists of three elements: a **keyword** identifying the argument (`end` here); an **equal sign** (`=`); and a **value** assigned to that argument;
- any keyword arguments have to be put **after the last positional argument** (this is very important)

In our example, we have made use of the `end` keyword rgument, and set it to a string containing one space.

Run the code to see how it works.

The console should now be showing the following text:

```
My name is Python.  Monty Python.
```

As you can see, the `end` keyword argument determines the characters the `print()` function sends to the output once it reaches the end of its positional arguments.

The default behavior reflects the situation where the `end` keyword argument is **implicitly** used in the following way: `end="\n"`. What this states is that instead of the default behavior of the end of that `print()` function call - which is to hit the 'return' and move to the next line - it is replaced with simply a space (the space within the two double quotes). This is why `Monty Python` is on the same line as `My name is Python`, not

the next line.

And now it's time to try something more difficult.

```
print("My name is ", end="")
print("Monty Python.")
```

If you look carefully, you'll see that we've used the `end` argument, but the string assigned to it is empty (it contains no characters at all).

What will happen now? Run the program in the editor to find out.

As the `end` argument has been set to nothing, the `print()` function outputs nothing too, once its positional arguments have been exhausted.

The console should now be showing the following text:

```
My name is Monty Python.
```

Note: **no newlines have been sent to the output.**

The string assigned to the end keyword argument can be of any length. Experiment with it if you want.

We've said previously that the `print()` function separates its outputted arguments with spaces. This behavior can be changed, too.

The **keyword argument** that can do this is named `sep` (like *separator*). Here is the code:

```
print("My", "name", "is", "Monty", "Python.", sep="-")
```

The `sep` argument delivers the following results:

```
My-name-is-Monty-Python.
```

What we gather is the default separator for the `print()` function is a space, but in this instance the keyword argument `sep` was used to print a '-' as the separator.

The `print()` function now uses a dash, instead of a space, to separate the outputted arguments.

Note: the `sep` argument's value may be an empty string, too. Try it for yourself.

```
print("My", "name", "is", "Monty", "Python.", sep="-")
```

Output is: `MynameisMontyPython.`

Both keyword arguments **may be mixed in one invocation**, just like here in the editor window.

The example doesn't make much sense, but it visibly presents the interactions between `end` and `sep`.

```
print("My", "name", "is", sep="_", end="*")
print("Monty", "Python.", sep="*", end="*\n")
```

What would the output be?

```
My_name_is*Monty*Python*
```

Now that you understand the `print()` function, you're ready to consider how to store and process data in Python.

Without `print()`, you wouldn't be able to see any results.

1.4 LAB - The print() Function

1.4.1 Objectives

- becoming familiar with the `print()` function and its formatting capabilities;
- experimenting with Python code.

1.4.2 Scenario

Modify the first line of code, using the `sep` and `end` keywords, to match the expected output. Use the two `print()` functions in the editor.

Don't change anything in the second `print()` invocation.

1.4.3 Provided Code

```
print("Programming","Essentials","in")
print("Python")
```

1.4.4 Expected Code

```
Programming***Essentials***in...Python
```

1.4.5 Modified Code

```
print("Programming","Essentials","in",sep="***",end="...")
print("Python")
```

1.5 LAB: Formatting the Output

1.5.1 Objectives

- experimenting with existing Python code;
- discovering and fixing basic syntax errors;
- becoming familiar with the `print()` function and its formatting capabilities.

1.5.2 Scenario

We strongly encourage you to **play with the code** we've written for you, and make some (maybe even destructive) amendments. Feel free to modify any part of the code, but there is one condition - learn from your mistakes and draw your own conclusions.

Try to:

- minimize the number of `print()` function invocations by inserting the `\n` sequence into the strings
- make the arrow twice as large (but keep the proportions)
- duplicate the arrow, placing both arrows side by side; note: a string may be multiplied by using the following trick: `"string" * 2` will produce `"stringstring"` (we'll tell you more about it soon)
- remove any of the quotes, and look carefully at Python's response; pay attention to where Python sees an error - is this the place where the error really exists?
- do the same with some of the parentheses;
- change any of the `print` words into something else, differing only in case (e.g., `Print`) - what happens now?
- replace some of the quotes with apostrophes; watch what happens carefully.

1.5.3 Provided Code

```
print("    *")
print("   * *")
print("  *   *")
print(" *     *")
print("***   ***")
print("  *   *")
print("  *   *")
print("  *****")
```

1.5.4 Modified Code

Challenge 1 Minimize the number of `print()` function invocations by inserting the `\n` sequence into the strings.

```
print("    *","   * *","  *   *","***   ***","  *   *","  *   *","  *****",sep="\n")
```

Challenge 2 Duplicate the arrow, placing both arrows side by side; note: a string may be multiplied by using the following trick: `"string" * 2` will produce `"stringstring"`.

```
print("    *" *2)
print("   * *" *2)
print("  *   *" *2)
print(" *     *" *2)
print("***   ***" *2)
print("  *   *" *2)
print("  *   *" *2)
print("  *****" *2)
```

1.6 Section Summary - Key Takeaways

1. The `print()` function is a **built-in** function. It prints/outputs a specified message to the screen/consol window.
2. Built-in functions, contrary to user-defined functions, are always available and don't have to be imported. Python 3.8 comes with 69 built-in functions. You can find their full list provided in alphabetical order in the Python Standard Library.
3. To call a function (this process is known as **function invocation** or **function call**), you need to use the function name followed by parentheses. You can pass arguments into a function by placing them inside the parentheses. You must separate arguments with a comma, e.g., `print("Hello,", "world!")`. An "empty" `print()` function outputs an empty line to the screen.
4. Python strings are delimited with **quotes**, e.g., `"I am a string"` (double quotes), or `'I am a string, too'` (single quotes).
5. Computer programs are collections of **instructions**. An instruction is a command to perform a specific task when executed, e.g., to print a certain message to the screen.
6. In Python strings the **backslash** (\) is a special character which announces that the next character has a different meaning, e.g., \n (the **newline character**) starts a new output line.
7. **Positional arguments** are the ones whose meaning is dictated by their position, e.g., the second argument is outputted after the first, the third is outputted after the second, etc.

8. **Keyword arguments** are the ones whose meaning is not dictated by their location, but by a special word (keyword) used to identify them.

9. The `end` and `sep` parameters can be used for formatting the output of the `print()` function. The sep parameter specifies the separator between the outputted arguments (e.g., `print("H", "E", "L", "L", "O", sep="-")`, whereas the `end` parameter specifies what to print at the end of the print statement.

2 Python Literals

2.1 Literals - the Data in Itself

Now that you have a little knowledge of some of the powerful features offered by the `print()` unction, it's time to learn about some new issues, and one important new term - the **literal**.

A literal is data whose values are determined by the literal itself.

As this is a difficult concept to understand, a good example may be helpful. Take a look at the following set of digits:
`123`

Can you guess what value it represents? Of course you can - it's one hundred twenty three. But what about this:
`c`

Does `c` represent any value? Maybe. It can be the symbol of the speed of light, for example. It also can be the constant of integration. Or even the length of a hypotenuse in the sense of a Pythagorean theorem. There are many possibilities. You cannot choose the right one without some additional knowledge.

And this is the clue: `123` is a literal, and `c` is not.

You use literals to **encode data and to put them into your code**. We're now going to show you some conventions you have to obey when using Python.

Let's start with a simple experiment:

```
print("2")
print(2)
```

The first line looks familiar. The second seems to be erroneous due to the visible lack of quotes. Try to run it. If everything went okay, you should now see two identical lines. What happened? What does it mean? Through this example, you encounter two different types of literals:

- a **string**, which you already know
- an **integer** number, something completely new

The `print()` function presents them in exactly the same way - this example is obvious, as their human-readable representation is also the same. Internally, in the computer's memory, these two values are stored in completely different ways - the string exists as just a string - a series of letters. The number is converted into machine representation (a set of bits). The `print()` function is able to show them both in a form readable to humans.

2.2 Integers

You may already know a little about how computers perform calculations on numbers. Perhaps you've heard of the **binary system**, and know that it's the system computers use for storing numbers, and that they can perform any operation upon them. We won't explore the intricacies of positional numeral systems here, but we'll say that the numbers handled by modern computers are of two types:

- **integers**, that is, those which are devoid of the fractional part;
- and **floating-point** numbers (or simply **floats**), that contain (or are able to contain) the fractional part.

This definition is not entirely accurate, but quite sufficient for now. The distinction is very important, and the boundary between these two types of numbers is very strict. Both of these kinds of numbers differ significantly in how they're stored in a computer memory and in the range of acceptable values. The characteristic of the numeric value which determines its kind, range, and application, is called the **type**.

If you encode a literal and place it inside Python code, the form of the literal determines the representation (**type**) Python will use to **store it in the memory**. For now, let's leave the floating-point numbers aside (we'll come back to them soon) and consider the question of how Python recognizes integers.

The process is almost like how you would write them with a pencil on paper - it's simply a string of digits that make up the number. But there's a reservation - you must not interject any characters that are not digits inside the number. Take, for example, the number eleven million one hundred and eleven thousand one hundred and eleven. If you took a pencil in your hand right now, you would write the number like this:`11,111,111`, or like this: `11.111.111`, or even like this: `11 111 111`.

It's clear that this provision makes it easier to read, especially when the number consists of many digits. However, Python doesn't accept

things like these. It's **prohibited**. What Python does allow, though, is the use of **underscores** in numeric literals.[1] Therefore, you can write this number either like this: `11111111` or like that: `11_111_111`.

And how do we code negative numbers in Python? As usual - by adding a minus. You can write: `-11111111` or `-11_111_111`. Positive numbers do not need to be preceded by the plus sign, but it's permissible, if you wish to do it. The following lines describe the same number: `+11111111` or `11111111`.

2.2.1 Integers: Octal and Hexadecimal Numbers

There are two additional conventions in Python that are unknown to the world of mathematics. The first allows us to use numbers in an **octal** representation. If an integer number is preceded by an `0O` or `0o` prefix (zero-o), it will be treated as an octal value. This means that the number must contain digits taken from the [0..7] range only.

`0o123` is an **octal** number with a (decimal) value equal to `83`. The `print()` function does the conversion automatically. Try this:

```
print(0o123)
```

Output is: `83`.

The second convention allows us to use **hexadecimal** numbers. Such numbers should be preceded by the prefix `0x` or `0X` (zero-x). `0x123` is a **hexadecimal** number with a (decimal) value equal to `291`. The `print()` function can manage these values too. Try this:

```
print(0x123)
```

Output is: `291`.

2.3 Floats

Now it's time to talk about another type, which is designed to represent and to store the numbers that (as a mathematician would say) have a **non-empty decimal fraction**. They are the numbers that have (or may have) a fractional part after the decimal point, and although such a definition is very poor, it's certainly sufficient for what we wish to discuss. Whenever we use a term like *two and a half or minus zero point four*, we think of numbers which the computer considers **floating-point** numbers:

```
2.5
-0.4
```

[1] Python 3.6 has introduced underscores in numeric literals, allowing for placing single underscores between digits and after base specifiers for improved readability. This feature is not available in older versions of Python.

Note: *two and a half* looks normal when you write it in a program, although if your native language prefers to use a comma instead of a point in the number, you should ensure that your **number doesn't contain any commas** at all.

Python will not accept that, or (in very rare but possible cases) may misunderstand your intentions, as the comma itself has its own reserved meaning in Python. If you want to use just a value of two and a half, you should write it as shown above. Note once again - there is a point between *2* and *5* - not a comma.

As you can probably imagine, the value of zero point four could be written in Python as `0.4`. But don't forget this simple rule - you can omit zero when it is the only digit in front of or after the decimal point. In essence, you can write the value `0.4` as `.4` in Python. For example: the value of `4.0` could be written as `4`. This will change neither its type nor its value.

2.4 Ints vs. Floats