

# Programming Essentials in Python: Module 2

## Data types, variables, basic input-output operations, basic operators

In this module, you will learn about:

- data types and the basic methods of formatting, converting, inputting and outputting data;
- operators;
- variables.

### Your very first program

## 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.

Launch IDLE, create a new Python source file, fill it with this code: `print("Hello, World!")`, 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.

## 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's 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 or some values** (e.g., the square root of a value or the length of a given text); this is what makes Python's functions the relatives of mathematical concepts.

Moreover, many of Python's 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.

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

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:

- First, 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);
- second, 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);
- third, 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;
- fourth, the function **executes its code**, causes the desired effect (if any), evaluates the desired result(s) (if any) and finishes its task;
- finally, 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 the `print()` function evaluate?

None. Its effect is enough - `print()` does not evaluate anything.

## The `print()` function - instructions

You already know that this program contains one function invocation. In turn, the function invocation is one of the possible kinds of Python **instruction**. Ergo, this program consists of just one instruction.

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 below:

```
1 print("The itsy bitsy spider climbed up the waterspout.")
2 print("Down came the rain and washed the spider out.")
```

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.
```

output

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.

```
1 print("The itsy bitsy spider climbed up the waterspout.")
2 print()
3 print("Down came the rain and washed the spider out.")
```

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.
```

output

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.

## The `print()` function - the escape and newline characters

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

```
1 print("The itsy bitsy spider\nclimbed up the waterspout.")
2 print()
3 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 backslash comes from the word *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.
```

output

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("\\")
```

2. 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.

## 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.

Look at the code. This is what we're going to test now:

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

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.
```

output

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.

## 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 this code in the editor.

```
1 print("My name is", "Python.")
2 print("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).

Run the code and check if the output matches your predictions.

## 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:

```
1 print("My name is", "Python.", end=" ")
2 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 argument, 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.
```

output

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"`.

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

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.
```

output

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*).

Look at the code below, and run it.

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

The `sep` argument delivers the following results:

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

output

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.

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

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

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

Can you predict the output?

Run the code and see if it matches your predictions.

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.

## Key takeaways

1. The `print()` function is a **built-in** function. It prints/outputs a specified message to the screen/console window.
2. Built-in functions, contrary to user-defined functions, are always available and don't have to be imported. Python 3.7.1 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 (**function invocation**), 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"`, or `'I am a string, too'`.

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.