**Python Essentials 1:  
Module 2**

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

In this module, you will 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.

**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.

**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.

**The print() function**

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 print() function

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 print() function**

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.

**LAB**

## Estimated time

5-10 minutes

## Level of difficulty

Very easy

## Objectives

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

## Scenario

The print() command, which is one of the easiest directives in Python, simply prints out a line to the screen.

In your first lab:

* use the print() function to print the line Hello, Python! to the screen. Use double quotes around the string;
* having done that, use the print() function again, but this time print your first name;
* remove the double quotes and run your code. Watch Python's reaction. What kind of error is thrown?
* then, remove the parentheses, put back the double quotes, and run your code again. What kind of error is thrown this time?
* experiment as much as you can. Change double quotes to single quotes, use multiple print() functions on the same line, and then on different lines. See what happens.

**The print() function**

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 return?**

None. Its effect is enough.

# 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.

**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)

**The print() function - instructions**

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.

**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.

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.

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

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 editor window. 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 the code in the editor.

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.

In the editor window you can see a very simple example of using a keyword argument.

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".

**The print() function - the keyword arguments**

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.

**The print() function - the keyword arguments**

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 in the editor, and run it.

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.

# The print() function - the keyword arguments

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.

Can you predict the output?

print("My", "name", "is", sep="\_", end="\*")

print("Monty", "Python.", sep="\*", end="\*\n")

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.

**LAB**

## Estimated time

5-10 minutes

## Level of difficulty

Very Easy

## Objectives

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

## Scenario

Modify the first line of code in the editor, 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.

## Expected output

Programming\*\*\*Essentials\*\*\*in...Python

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

print("Python")

**LAB**

## Estimated time

5-15 minutes

## Level of difficulty

Easy

## Objectives

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

## 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.

print(" \*")

print(" \* \*")

print(" \* \*")

print(" \* \*")

print("\*\*\* \*\*\*")

print(" \* \*")

print(" \* \*")

print(" \*\*\*\*\*")

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

print("Python")

# 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 3.8 comes with 69 built-in functions. You can find their full list provided in alphabetical order in the [Python Standard Library](https://docs.python.org/3/library/functions.html).

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.

**Literals - the data in itself**

Now that you have a little knowledge of some of the powerful features offered by the print() function, 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 it 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.

# Literals - the data in itself

Let's start with a simple experiment - take a look at the snippet in the editor.

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,
* and 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.

We're now going to be spending some time discussing numeric literals and their internal life.

print("2")

print(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.\*

Therefore, you can write this number either like this: 11111111, or like that: 11\_111\_111.

**NOTE**   \*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.

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 and 11111111.

**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)

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)

**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

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

For example: the value of 4.0 could be written as:

4.

This will change neither its type nor its value.

# Ints vs. floats

The decimal point is essentially important in recognizing floating-point numbers in Python.

Look at these two numbers:

4

4.0

You may think that they are exactly the same, but Python sees them in a completely different way.

4 is an **integer** number, whereas 4.0 is a **floating-point** number.

The point is what makes a float.

On the other hand, it's not only points that make a float. You can also use the letter e.

When you want to use any numbers that are very large or very small, you can use **scientific notation**.

Take, for example, the speed of light, expressed in *meters per second*. Written directly it would look like this: 300000000.

To avoid writing out so many zeros, physics textbooks use an abbreviated form, which you have probably already seen: 3 x 108.

It reads: three times ten to the power of eight.

In Python, the same effect is achieved in a slightly different way - take a look:

3E8

The letter E (you can also use the lower-case letter e - it comes from the word **exponent**) is a concise record of the phrase *times ten to the power of*.

Note:

* the **exponent** (the value after the *E*) has to be an integer;
* the **base** (the value in front of the *E*) may be an integer.

**Coding floats**

Let's see how this convention is used to record numbers that are very small (in the sense of their absolute value, which is close to zero).

A physical constant called *Planck's constant* (and denoted as *h*), according to the textbooks, has the value of: **6.62607 x 10-34**.

If you would like to use it in a program, you should write it this way:

6.62607E-34

Note: the fact that you've chosen one of the possible forms of coding float values doesn't mean that Python will present it the same way.

Python may sometimes choose **different notation** than you.

For example, let's say you've decided to use the following float literal:

0.0000000000000000000001

When you run this literal through Python:

print(0.0000000000000000000001)

this is the result:

1e-22

**output**

Python always chooses **the more economical form of the number's presentation**, and you should take this into consideration when creating literals.

**Strings**

Strings are used when you need to process text (like names of all kinds, addresses, novels, etc.), not numbers.

You already know a bit about them, e.g., that **strings need quotes** the way floats need points.

This is a very typical string: "I am a string."

However, there is a catch. The catch is how to encode a quote inside a string which is already delimited by quotes.

Let's assume that we want to print a very simple message saying:

I like "Monty Python"

How do we do it without generating an error? There are two possible solutions.

The first is based on the concept we already know of the **escape character**, which you should remember is played by the **backslash**. The backslash can escape quotes too. A quote preceded by a backslash changes its meaning - it's not a delimiter, but just a quote. This will work as intended:

print("I like \"Monty Python\"")

Note: there are two escaped quotes inside the string - can you see them both?

The second solution may be a bit surprising. Python can use **an apostrophe instead of a quote**. Either of these characters may delimit strings, but you must be **consistent**.

If you open a string with a quote, you have to close it with a quote.

If you start a string with an apostrophe, you have to end it with an apostrophe.

This example will work too:

print('I like "Monty Python"')

Note: you don't need to do any escaping here.

**Coding strings**

Now, the next question is: how do you embed an apostrophe into a string placed between apostrophes?

You should already know the answer, or to be precise, two possible answers.

Try to print out a string containing the following message:

I'm Monty Python.

Do you know how to do it? Click *Check* below to see if you were right:

Check

As you can see, the backslash is a very powerful tool - it can escape not only quotes, but also apostrophes.

We've shown it already, but we want to emphasize this phenomenon once more - **a string can be empty** - it may contain no characters at all.

An empty string still remains a string:

''

""

# Boolean values

To conclude with Python's literals, there are two additional ones.

They're not as obvious as any of the previous ones, as they're used to represent a very abstract value - **truthfulness**.

Each time you ask Python if one number is greater than another, the question results in the creation of some specific data - a **Boolean** value.

The name comes from George Boole (1815-1864), the author of the fundamental work, *The Laws of Thought*, which contains the definition of **Boolean algebra** - a part of algebra which makes use of only two distinct values: True and False, denoted as 1 and 0.

A programmer writes a program, and the program asks questions. Python executes the program, and provides the answers. The program must be able to react according to the received answers.

Fortunately, computers know only two kinds of answers:

* Yes, this is true;
* No, this is false.

You'll never get a response like: *I don't know* or *Probably yes, but I don't know for sure*.

Python, then, is a **binary** reptile.

These two Boolean values have strict denotations in Python:

True

False

You cannot change anything - you have to take these symbols as they are, including **case-sensitivity**.

Challenge: What will be the output of the following snippet of code?

print(True > False)

print(True < False)

Run the code in the Sandbox to check. Can you explain the result?

**LAB**

## Estimated time

5-10 minutes

## Level of difficulty

Easy

## Objectives

* becoming familiar with the print() function and its formatting capabilities;
* practicing coding strings;
* experimenting with Python code.

## Scenario

Write a one-line piece of code, using the print() function, as well as the newline and escape characters, to match the expected result outputted on three lines.

## Expected output

"I'm"

""learning""

"""Python"""

**output**

**Key takeaways**

1. **Literals** are notations for representing some fixed values in code. Python has various types of literals - for example, a literal can be a number (numeric literals, e.g., 123), or a string (string literals, e.g., "I am a literal.").

2. The **binary system** is a system of numbers that employs *2* as the base. Therefore, a binary number is made up of 0s and 1s only, e.g., 1010 is *10* in decimal.

Octal and hexadecimal numeration systems, similarly, employ *8* and *16* as their bases respectively. The hexadecimal system uses the decimal numbers and six extra letters.

3. **Integers** (or simply **int**s) are one of the numerical types supported by Python. They are numbers written without a fractional component, e.g., 256, or -1 (negative integers).

4. **Floating-point** numbers (or simply **float**s) are another one of the numerical types supported by Python. They are numbers that contain (or are able to contain) a fractional component, e.g., 1.27.

5. To encode an apostrophe or a quote inside a string you can either use the escape character, e.g., 'I\'m happy.', or open and close the string using an opposite set of symbols to the ones you wish to encode, e.g., "I'm happy." to encode an apostrophe, and 'He said "Python", not "typhoon"' to encode a (double) quote.

6. **Boolean values** are the two constant objects True and False used to represent truth values (in numeric contexts 1 is True, while 0 is False.

**EXTRA**

There is one more, special literal that is used in Python: the None literal. This literal is a so-called NoneType object, and it is used to represent **the absence of a value**. We'll tell you more about it soon.

**Exercise 1**

What types of literals are the following two examples?

"Hello ", "007"  
Check

**Exercise 2**

What types of literals are the following four examples?

"1.5", 2.0, 528, False  
Check

**Exercise 3**

What is the decimal value of the following binary number?

1011  
Check

**Python as a calculator**

Now, we're going to show you a completely new side of the print() function. You already know that the function is able to show you the values of the literals passed to it by arguments.

In fact, it can do something more. Take a look at the snippet:

print(2+2)

Retype the code in the editor and run it. Can you guess the output?

You should see the number four. Feel free to experiment with other operators.

Without taking this too seriously, you've just discovered that Python can be used as a calculator. Not a very handy one, and definitely not a pocket one, but a calculator nonetheless.

Taking it more seriously, we are now entering the province of **operators** and **expressions**.

**Basic operators**

An **operator** is a symbol of the programming language, which is able to operate on the values.

For example, just as in arithmetic, the + (plus) sign is the operator which is able to **add** two numbers, giving the result of the addition.

Not all Python operators are as obvious as the plus sign, though, so let's go through some of the operators available in Python, and we'll explain which rules govern their use, and how to interpret the operations they perform.

We'll begin with the operators which are associated with the most widely recognizable arithmetic operations:

+, -, \*, /, //, %, \*\*

The order of their appearance is not accidental. We'll talk more about it once we've gone through them all.

**Remember**: Data and operators when connected together form **expressions**. The simplest expression is a literal itself.

# Arithmetic operators: exponentiation

A \*\* (double asterisk) sign is an **exponentiation** (power) operator. Its left argument is the **base**, its right, the **exponent**.

Classical mathematics prefers notation with superscripts, just like this: **23**. Pure text editors don't accept that, so Python uses \*\* instead, e.g., 2 \*\* 3.

Take a look at our examples in the editor window.

Note: we've surrounded the double asterisks with spaces in our examples. It's not compulsory, but it improves the **readability** of the code.

The examples show a very important feature of virtually all Python **numerical operators**.

Run the code and look carefully at the results it produces. Can you see any regularity here?

**Remember**: It's possible to formulate the following rules based on this result:

* when **both** \*\* arguments are integers, the result is an integer, too;
* when **at least one** \*\* argument is a float, the result is a float, too.

This is an important distinction to remember.

print(2 \*\* 3)

print(2 \*\* 3.)

print(2. \*\* 3)

print(2. \*\* 3.)

**Arithmetic operators: multiplication**

An \* (asterisk) sign is a **multiplication** operator.

Run the code below and check if our *integer vs. float* rule is still working.

print(2 \* 3)

print(2 \* 3.)

print(2. \* 3)

print(2. \* 3.)

**Arithmetic operators: division**

A / (slash) sign is a **divisional** operator.

The value in front of the slash is a **dividend**, the value behind the slash, a **divisor**.

Run the code below and analyze the results.

print(6 / 3)

print(6 / 3.)

print(6. / 3)

print(6. / 3.)

You should see that there is an exception to the rule.

**The result produced by the division operator is always a float**, regardless of whether or not the result seems to be a float at first glance: 1 / 2, or if it looks like a pure integer: 2 / 1.

Is this a problem? Yes, it is. It happens sometimes that you really need a division that provides an integer value, not a float.

Fortunately, Python can help you with that.

# Arithmetic operators: integer division

A // (double slash) sign is an **integer divisional** operator. It differs from the standard / operator in two details:

* its result lacks the fractional part - it's absent (for integers), or is always equal to zero (for floats); this means that **the results are always rounded**;
* it conforms to the *integer vs. float rule*.

Run the example below and see the results:

print(6 // 3)

print(6 // 3.)

print(6. // 3)

print(6. // 3.)

As you can see, *integer by integer division* gives an **integer result**. All other cases produce floats.

Let's do some more advanced tests.

Look at the following snippet:

print(6 // 4)

print(6. // 4)

Imagine that we used / instead of // - could you predict the results?

Yes, it would be 1.5 in both cases. That's clear.

But what results should we expect with // division?

Run the code and see for yourself.

What we get is two ones - one integer and one float.

The result of integer division is always rounded to the nearest integer value that is less than the real (not rounded) result.

This is very important: **rounding always goes to the lesser integer**.

Look at the code below and try to predict the results once again:

print(-6 // 4)

print(6. // -4)

Note: some of the values are negative. This will obviously affect the result. But how?

The result is two negative twos. The real (not rounded) result is -1.5 in both cases. However, the results are the subjects of rounding. The **rounding goes toward the lesser integer value**, and the lesser integer value is -2, hence: -2 and -2.0.

**NOTE**

Integer division can also be called **floor division**. You will definitely come across this term in the future.

# Operators: remainder (modulo)

The next operator is quite a peculiar one, because it has no equivalent among traditional arithmetic operators.

Its graphical representation in Python is the % (percent) sign, which may look a bit confusing.

Try to think of it as of a slash (division operator) accompanied by two funny little circles.

The result of the operator is a **remainder left after the integer division**.

In other words, it's the value left over after dividing one value by another to produce an integer quotient.

Note: the operator is sometimes called **modulo** in other programming languages.

Take a look at the snippet - try to predict its result and then run it:

print(14 % 4)

As you can see, the result is two. This is why:

* 14 // 4 gives 3 → this is the integer **quotient**;
* 3 \* 4 gives 12 → as a result of **quotient and divisor multiplication**;
* 14 - 12 gives 2 → this is the **remainder**.

This example is somewhat more complicated:

print(12 % 4.5)

What is the result?

Check

# Operators: how not to divide

As you probably know, **division by zero doesn't work**.

Do **not** try to:

* perform a division by zero;
* perform an integer division by zero;
* find a remainder of a division by zero.

**Operators: addition**

The **addition** operator is the + (plus) sign, which is fully in line with mathematical standards.

Again, take a look at the snippet of the program below:

print(-4 + 4)

print(-4. + 8)

The result should be nothing surprising. Run the code to check it.

**The subtraction operator, unary and binary operators**

The **subtraction** operator is obviously the - (minus) sign, although you should note that this operator also has another meaning - **it can change the sign of a number**.

This is a great opportunity to present a very important distinction between **unary** and **binary** operators.

In subtracting applications, the **minus operator expects two arguments**: the left (a **minuend** in arithmetical terms) and right (a **subtrahend**).

For this reason, the subtraction operator is considered to be one of the binary operators, just like the addition, multiplication and division operators.

But the minus operator may be used in a different (unary) way - take a look at the last line of the snippet below:

print(-4 - 4)

print(4. - 8)

print(-1.1)

By the way: there is also a unary + operator. You can use it like this:

print(+2)

The operator preserves the sign of its only argument - the right one.

Although such a construction is syntactically correct, using it doesn't make much sense, and it would be hard to find a good rationale for doing so.

Take a look at the snippet above - can you guess its output?

# Operators and their priorities

So far, we've treated each operator as if it had no connection with the others. Obviously, such an ideal and simple situation is a rarity in real programming.

Also, you will very often find more than one operator in one expression, and then this presumption is no longer so obvious.

Consider the following expression:

2 + 3 \* 5

You probably remember from school that **multiplications precede additions**.

You surely remember that you should first multiply 3 by 5 and, keeping the 15 in your memory, then add it to 2, thus getting the result of 17.

The phenomenon that causes some operators to act before others is known as **the hierarchy of priorities**.

Python precisely defines the priorities of all operators, and assumes that operators of a larger (higher) priority perform their operations before the operators of a lower priority.

So, if you know that \* has a higher priority than +, the computation of the final result should be obvious.

# Operators and their bindings

The **binding** of the operator determines the order of computations performed by some operators with equal priority, put side by side in one expression.

Most of Python's operators have left-sided binding, which means that the calculation of the expression is conducted from left to right.

This simple example will show you how it works. Take a look:

print(9 % 6 % 2)

There are two possible ways of evaluating this expression:

* from left to right: first 9 % 6 gives 3, and then 3 % 2 gives 1;
* from right to left: first 6 % 2 gives 0, and then 9 % 0 causes **a fatal error**.

Run the example and see what you get.

The result should be 1. This operator has **left-sided binding**. But there's one interesting exception.

# Operators and their bindings: exponentiation

Repeat the experiment, but now with exponentiation.

Use this snippet of code:

print(2 \*\* 2 \*\* 3)

The two possible results are:

* 2 \*\* 2 → 4; 4 \*\* 3 → 64
* 2 \*\* 3 → 8; 2 \*\* 8 → 256

Run the code. What do you see?

The result clearly shows that **the exponentiation operator uses right-sided binding**.

**List of priorities**

Since you're new to Python operators, we don't want to present the complete list of operator priorities right now.

Instead, we'll show you a truncated form, and we'll expand it consistently as we introduce new operators.

Look at the table below:

|  |  |  |
| --- | --- | --- |
| **Priority** | **Operator** |  |
| 1 | \*\* |  |
| 2 | +, - (note: unary operators located next to the right of the power operator bind more strongly) | unary |
| 3 | \*, /, //, % |  |
| 4 | +, - | binary |

Note: we've enumerated the operators in order **from the highest (1) to the lowest (4) priorities**.

Try to work through the following expression:

print(2 \* 3 % 5)

Both operators (\* and %) have the same priority, so the result can be guessed only when you know the binding direction. What do you think? What is the result?

Check

**Operators and parentheses**

Of course, you're always allowed to use **parentheses**, which can change the natural order of a calculation.

In accordance with the arithmetic rules, **subexpressions in parentheses are always calculated first**.

You can use as many parentheses as you need, and they're often used to **improve the readability** of an expression, even if they don't change the order of the operations.

An example of an expression with multiple parentheses is here:

print((5 \* ((25 % 13) + 100) / (2 \* 13)) // 2)

Try to compute the value that's printed to the console. What's the result of the print() function?

Check

# Key takeaways

1. An **expression** is a combination of values (or variables, operators, calls to functions ‒ you will learn about them soon) which evaluates to a certain value, e.g., 1 + 2.

2. **Operators** are special symbols or keywords which are able to operate on the values and perform (mathematical) operations, e.g., the \* operator multiplies two values: x \* y.

3. Arithmetic operators in Python: + (addition), - (subtraction), \* (multiplication), / (classic division ‒ always returns a float), % (modulus ‒ divides left operand by right operand and returns the remainder of the operation, e.g., 5 % 2 = 1), \*\* (exponentiation ‒ left operand raised to the power of right operand, e.g., 2 \*\* 3 = 2 \* 2 \* 2 = 8), // (floor/integer division ‒ returns a number resulting from division, but rounded down to the nearest whole number, e.g., 3 // 2.0 = 1.0)

4. A **unary** operator is an operator with only one operand, e.g., -1, or +3.

5. A **binary** operator is an operator with two operands, e.g., 4 + 5, or 12 % 5.

6. Some operators act before others – **the hierarchy of priorities**:

* the \*\* operator (exponentiation) has the highest priority;
* then the unary + and - (note: a unary operator to the right of the exponentiation operator binds more strongly, for example: 4 \*\* -1 equals 0.25)
* then \*, /, //, and %;
* and, finally, the lowest priority: the binary + and -.

7. Subexpressions in **parentheses** are always calculated first, e.g., 15 - 1 \* (5 \* (1 + 2)) = 0.

8. The **exponentiation** operator uses **right-sided binding**, e.g., 2 \*\* 2 \*\* 3 = 256.

**Exercise 1**

What is the output of the following snippet?

print((2 \*\* 4), (2 \* 4.), (2 \* 4))

Check

**Exercise 2**

What is the output of the following snippet?

print((-2 / 4), (2 / 4), (2 // 4), (-2 // 4))

Check

**Exercise 3**

What is the output of the following snippet?

print((2 % -4), (2 % 4), (2 \*\* 3 \*\* 2))

Check

# What are variables?

It seems fairly obvious that Python should allow you to encode literals carrying number and text values.

You already know that you can do some arithmetic operations with these numbers: add, subtract, etc. You'll be doing that many times.

But it's quite a normal question to ask how to **store the results** of these operations, in order to use them in other operations, and so on.

How do you save the intermediate results, and use them again to produce subsequent ones?

Python will help you with that. It offers special "boxes" (containers) for that purpose, and these boxes are called **variables** - the name itself suggests that the content of these containers can be varied in (almost) any way.

What does every Python variable have?

* a name;
* a value (the content of the container)

Let's start with the issues related to a variable's name.

Variables do not appear in a program automatically. As developer, you must decide how many and which variables to use in your programs.

You must also name them.

If you want to **give a name to a variable**, you must follow some strict rules:

* the name of the variable must be composed of upper-case or lower-case letters, digits, and the character \_ (underscore)
* the name of the variable must begin with a letter;
* the underscore character is a letter;
* upper- and lower-case letters are treated as different (a little differently than in the real world - *Alice* and *ALICE* are the same first names, but in Python they are two different variable names, and consequently, two different variables);
* the name of the variable must not be any of Python's reserved words (the keywords - we'll explain more about this soon).



# Correct and incorrect variable names

Note that the same restrictions apply to function names.

Python does not impose restrictions on the length of variable names, but that doesn't mean that a long variable name is always better than a short one.

Here are some correct, but not always convenient variable names:

MyVariable, i, t34, Exchange\_Rate, counter, days\_to\_christmas, TheNameIsSoLongThatYouWillMakeMistakesWithIt, \_.

Moreover, Python lets you use not only Latin letters but also characters specific to languages that use other alphabets.

These variable names are also correct:

Adiós\_Señora, sûr\_la\_mer, Einbahnstraße, переменная.

And now for some **incorrect names**:

10t (does not begin with a letter), Exchange Rate (contains a space)

**NOTE**

The [PEP 8 -- Style Guide for Python Code](https://www.python.org/dev/peps/pep-0008/) recommends the following naming convention for variables and functions in Python:

* variable names should be lowercase, with words separated by underscores to improve readability (e.g., var, my\_variable)
* function names follow the same convention as variable names (e.g., fun, my\_function)
* it's also possible to use mixed case (e.g., myVariable), but only in contexts where that's already the prevailing style, to retain backwards compatibility with the adopted convention.

# Keywords

Take a look at the list of words that play a very special role in every Python program.

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

They are called **keywords** or (more precisely) **reserved keywords**. They are reserved because **you mustn't use them as names**: neither for your variables, nor functions, nor any other named entities you want to create.

The meaning of the reserved word is **predefined**, and mustn't be changed in any way.

Fortunately, due to the fact that Python is case-sensitive, you can modify any of these words by changing the case of any letter, thus creating a new word, which is not reserved anymore.

For example - **you can't name** your variable like this:

import

You mustn't have a variable named in such a way - it is prohibited. But you can do this instead:

Import

These words might be a mystery to you now, but you'll soon learn the meaning of them.

# Creating variables

What can you put inside a variable?

Anything.

You can use a variable to store any value of any of the already presented kinds, and many more of the ones we haven't shown you yet.

The value of a variable is what you have put into it. It can vary as often as you need or want. It can be an integer one moment, and a float a moment later, eventually becoming a string.

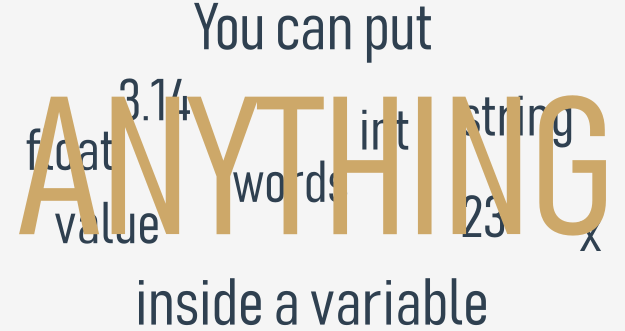
Let's talk now about two important things - **how variables are created**, and **how to put values inside them** (or rather - how to give or **pass values** to them).

**REMEMBER**

**A variable comes into existence as a result of assigning a value to it**. Unlike in other languages, you don't need to declare it in any special way.

If you assign any value to a nonexistent variable, the variable will be **automatically created**. You don't need to do anything else.

The creation (or otherwise - its syntax) is extremely simple: **just use the name of the desired variable, then the equal sign (=) and the value you want to put into the variable.**



Take a look at the snippet:

var = 1

print(var)

It consists of two simple instructions:

* The first of them creates a variable named var, and assigns a literal with an integer value equal to 1.
* The second prints the value of the newly created variable to the console.

Note: print() has yet another side to it - it can handle variables too. Do you know what the output of the snippet will be?

Check

**Using variables**

You're allowed to use as many variable declarations as you need to achieve your goal, like this:

var = 1

account\_balance = 1000.0

client\_name = 'John Doe'

print(var, account\_balance, client\_name)

print(var)

You're **not allowed to use a variable which doesn't exist** (in other words, a variable that was not assigned a value).

This example will **cause an error**:

var = 1

print(Var)

We've tried to use a variable named Var, which doesn't have any value (note: var and Var are different entities, and have nothing in common as far as Python's concerned).

**REMEMBER**

You can use the print() function and combine text and variables using the + operator to output strings and variables, e.g.:

var = "3.8.5"

print("Python version: " + var)

Can you guess the output of the snippet above?

Check

Python version: 3.8.5

**Assigning a new value to an already existing variable**

How do you assign a new value to an already created variable? In the same way. You just need to use the equal sign.

The equal sign is in fact an **assignment operator**. Although this may sound strange, the operator has a simple syntax and unambiguous interpretation.

It assigns the value of its right argument to the left, while the right argument may be an arbitrarily complex expression involving literals, operators and already defined variables.

Look at the code below:

var = 1

print(var)

var = var + 1

print(var)

The code sends two lines to the console:

1

2

**output**

The first line of the snippet **creates a new variable** named var and assigns 1 to it.

The statement reads: assign a value of 1 to a variable named var.

We can say it shorter: assign 1 to var.

Some prefer to read such a statement as: var becomes 1.

The third line **assigns the same variable with the new value** taken from the variable itself, summed with 1. Seeing a record like that, a mathematician would probably protest - no value may be equal to itself plus one. This is a contradiction. But Python treats the sign = not as *equal to*, but as *assign a value*.

So how do you read such a record in the program?

Take the current value of the variable var, add 1 to it and store the result in the variable var.

In effect, the value of variable var has been **incremented** by one, which has nothing to do with comparing the variable with any value.

Do you know what the output of the following snippet will be?

var = 100

var = 200 + 300

print(var)

Check

500 - why? Well, first, the var variable is created and assigned a value of 100. Then, the same variable is assigned a new value: the result of adding 200 to 300, which is 500.

**Solving simple mathematical problems**

Now you should be able to construct a short program solving simple mathematical problems such as the Pythagorean theorem:

*The square of the hypotenuse is equal to the sum of the squares of the other two sides.*

The following code evaluates the length of the hypotenuse (i.e., the longest side of a right-angled triangle, the one opposite of the right angle) using the Pythagorean theorem:

a = 3.0

b = 4.0

c = (a \*\* 2 + b \*\* 2) \*\* 0.5

print("c =", c)

Note: we need to make use of the \*\* operator to evaluate the square root as:

√ (x)  = x(½)

and

c = √ a2 + b2

Can you guess the output of the code?

Check below and run the code in the editor to confirm your predictions.

Check

c = 5.0

a = 3.0

b = 4.0

c = (a \*\* 2 + b \*\* 2) \*\* 0.5

print("c =", c)

**LAB**

## Estimated time

10 minutes

## Level of difficulty

Easy

## Objectives

* becoming familiar with the concept of storing and working with different data types in Python;
* experimenting with Python code.

## Scenario

Here is a short story:

Once upon a time in Appleland, John had three apples, Mary had five apples, and Adam had six apples. They were all very happy and lived for a long time. End of story.

Your task is to:

* create the variables: john, mary, and adam;
* assign values to the variables. The values must be equal to the numbers of fruit possessed by John, Mary, and Adam respectively;
* having stored the numbers in the variables, print the variables on one line, and separate each of them with a comma;
* now create a new variable named total\_apples equal to addition of the three former variables.
* print the value stored in total\_apples to the console;
* **experiment with your code**: create new variables, assign different values to them, and perform various arithmetic operations on them (e.g., +, -, \*, /, //, etc.). Try to print a string and an integer together on one line, e.g., "Total number of apples:" and total\_apples.

**Shortcut operators**

It's time for the next set of operators that make a developer's life easier.

Very often, we want to use one and the same variable both to the right and left sides of the = operator.

For example, if we need to calculate a series of successive values of powers of 2, we may use a piece like this:

x = x \* 2

You may use an expression like this if you can't fall asleep and you're trying to deal with it using some good, old-fashioned methods:

sheep = sheep + 1

Python offers you a shortened way of writing operations like these, which can be coded as follows:

x \*= 2

sheep += 1

Let's try to present a general description for these operations.

If op is a two-argument operator (this is a very important condition) and the operator is used in the following context:

variable = variable op expression

It can be simplified and shown as follows:

variable op= expression

Take a look at the examples below. Make sure you understand them all.

i = i + 2 \* j ⇒ i += 2 \* j

var = var / 2 ⇒ var /= 2

rem = rem % 10 ⇒ rem %= 10

j = j - (i + var + rem) ⇒ j -= (i + var + rem)

x = x \*\* 2 ⇒ x \*\*= 2

**LAB**

## Estimated time

10 minutes

## Level of difficulty

Easy

## Objectives

* becoming familiar with the concept of, and working with, variables;
* performing basic computations and conversions;
* experimenting with Python code.

## Scenario

Miles and kilometers are units of length or distance.

Bearing in mind that 1 mile is equal to approximately 1.61 kilometers, complete the program in the editor so that it converts:

* miles to kilometers;
* kilometers to miles.

Do not change anything in the existing code. Write your code in the places indicated by ###. Test your program with the data we've provided in the source code.

Pay particular attention to what is going on inside the print() function. Analyze how we provide multiple arguments to the function, and how we output the expected data.

Note that some of the arguments inside the print() function are strings (e.g., "miles is", whereas some other are variables (e.g., miles).

**TIP**

There's one more interesting thing happening there. Can you see another function inside the print() function? It's the round() function. Its job is to round the outputted result to the number of decimal places specified in the parentheses, and return a float (inside the round() function you can find the variable name, a comma, and the number of decimal places we're aiming for). We're going to talk about functions very soon, so don't worry that everything may not be fully clear yet. We just want to spark your curiosity.

After completing the lab, open Sandbox, and experiment more. Try to write different converters, e.g., a USD to EUR converter, a temperature converter, etc. - let your imagination fly! Try to output the results by combining strings and variables. Try to use and experiment with the round() function to round your results to one, two, or three decimal places. Check out what happens if you don't provide any number of digits. Remember to test your programs.

Experiment, draw conclusions, and learn. Be curious.

## Expected output

7.38 miles is 11.88 kilometers

12.25 kilometers is 7.61 miles

**output**

kilometers = 12.25

miles = 7.38

miles\_to\_kilometers = ###

kilometers\_to\_miles = ###

print(miles, "miles is", round(miles\_to\_kilometers, 2), "kilometers")

print(kilometers, "kilometers is", round(kilometers\_to\_miles, 2), "miles")

**LAB**

## Estimated time

10-15 minutes

## Level of difficulty

Easy

## Objectives

* becoming familiar with the concept of numbers, operators, and arithmetic operations in Python;
* performing basic calculations.

## Scenario

Take a look at the code in the editor: it reads a float value, puts it into a variable named x, and prints the value of a variable named y. Your task is to complete the code in order to evaluate the following expression:

3x3 - 2x2 + 3x - 1

The result should be assigned to y.

Remember that classical algebraic notation likes to omit the multiplication operator - you need to use it explicitly. Note how we change data type to make sure that x is of type float.

Keep your code clean and readable, and test it using the data we've provided, each time assigning it to the x variable (by hardcoding it). Don't be discouraged by any initial failures. Be persistent and inquisitive.

## Test Data

Sample input

x = 0

x = 1

x = -1

Expected Output

y = -1.0

y = 3.0

y = -9.0

**output**

x = # hardcode your test data here

x = float(x)

# write your code here

print("y =", y)

**Key takeaways**

1. A **variable** is a named location reserved to store values in the memory. A variable is created or initialized automatically when you assign a value to it for the first time. (2.1.4.1)

2. Each variable must have a unique name - an **identifier**. A legal identifier name must be a non-empty sequence of characters, must begin with the underscore(\_), or a letter, and it cannot be a Python keyword. The first character may be followed by underscores, letters, and digits. Identifiers in Python are case-sensitive. (2.1.4.1)

3. Python is a **dynamically-typed** language, which means you don't need to *declare* variables in it. (2.1.4.3) To assign values to variables, you can use a simple assignment operator in the form of the equal (=) sign, i.e., var = 1.

4. You can also use **compound assignment operators** (shortcut operators) to modify values assigned to variables, e.g., var += 1, or var /= 5 \* 2. (2.1.4.8)

5. You can assign new values to already existing variables using the assignment operator or one of the compound operators, e.g.: (2.1.4.5)

var = 2

print(var)

var = 3

print(var)

var += 1

print(var)

6. You can combine text and variables using the + operator, and use the print() function to output strings and variables, e.g.: (2.1.4.4)

var = "007"

print("Agent " + var)

**Exercise 1**

What is the output of the following snippet?

var = 2

var = 3

print(var)

Check

3

**Exercise 2**

Which of the following variable names are illegal in Python?

my\_var

m

101

averylongvariablename

m101

m 101

Del

del  
Check

my\_var

m

101 # incorrect (starts with a digit)

averylongvariablename

m101

m 101 # incorrect (contains a space)

Del

del # incorrect (is a keyword)

**Exercise 3**

What is the output of the following snippet?

a = '1'

b = "1"

print(a + b)

Check

11

**Exercise 4**

What is the output of the following snippet?

a = 6

b = 3

a /= 2 \* b

print(a)

Check

1.0  
2 \* b = 6  
a = 6 → 6 / 6 = 1.0

**Leaving comments in code: why, how, and when**

You may want to put in a few words addressed not to Python but to humans, usually to explain to other readers of the code how the tricks used in the code work, or the meanings of the variables, and eventually, in order to keep stored information on who the author is and when the program was written.

A remark inserted into the program, which is **omitted at runtime**, is called a **comment**.

How do you leave this kind of comment in the source code? It has to be done in a way that won't force Python to interpret it as part of the code.

Whenever Python encounters a comment in your program, the comment is completely transparent to it - from Python's point of view, this is only one space (regardless of how long the real comment is).

In Python, a comment is a piece of text that begins with a # (hash) sign and extends to the end of the line.

If you want a comment that spans several lines, you have to put a hash in front of them all.

Just like here:

# This program evaluates the hypotenuse c.

# a and b are the lengths of the legs.

a = 3.0

b = 4.0

c = (a \*\* 2 + b \*\* 2) \*\* 0.5 # We use \*\* instead of square root.

print("c =", c)

Good, responsible developers **describe each important piece of code**, e.g., explaining the role of the variables; although it must be stated that the best way of commenting variables is to name them in an unambiguous manner.

For example, if a particular variable is designed to store an area of some unique square, the name square\_area will obviously be better than aunt\_jane.

We say that the first name is **self-commenting**.

Comments may be useful in another respect - you can use them to **mark a piece of code that currently isn't needed** for whatever reason. Look at the example below, if you **uncomment** the highlighted line, this will affect the output of the code:

# This is a test program.

x = 1

y = 2

# y = y + x

print(x + y)

This is often done during the testing of a program, in order to isolate the place where an error might be hidden.

**TIP**

If you'd like to quickly comment or uncomment multiple lines of code, select the line(s) you wish to modify and use the following keyboard shortcut: **CTRL** + **/** (Windows) or **CMD** + **/** (Mac OS). It's a very useful trick, isn't it? Try [this code](https://edube.org/sandbox/ffa32cfe-a181-11e9-ade3-0242e71d5f55) in Sandbox.

**LAB**

## Estimated time

5 minutes

## Level of difficulty

Very Easy

## Objectives

* becoming familiar with the concept of comments in Python;
* using and not using comments;
* replacing comments with code;
* experimenting with Python code.

## Scenario

The code in the editor contains comments. Try to improve it: add or remove comments where you find it appropriate (yes, sometimes removing a comment can make the code more readable), and change variable names where you think this will improve code comprehension.

**NOTE**

Comments are very important. They are used not only to make your programs **easier to understand**, but also to **disable those pieces of code that are currently not needed** (e.g., when you need to test some parts of your code only, and ignore other). Good programmers **describe** each important piece of code, and give **self-commenting names** to variables, as sometimes it is simply much better to leave information in the code.

It's good to use **readable** variable names, and sometimes it's better to **divide your code** into named pieces (e.g., functions). In some situations, it's a good idea to write the steps of computations in a clearer way.

One more thing: it may happen that a comment contains a wrong or incorrect piece of information - you should never do that on purpose!

#this program computes the number of seconds in a given number of hours

# this program has been written two days ago

a = 2 # number of hours

seconds = 3600 # number of seconds in 1 hour

print("Hours: ", a) #printing the number of hours

# print("Seconds in Hours: ", a \* seconds) # printing the number of seconds in a given number of hours

#here we should also print "Goodbye", but a programmer didn't have time to write any code

#this is the end of the program that computes the number of seconds in 3 hour

**Key takeaways**

1. Comments can be used to leave additional information in code. They are omitted at runtime. The information left in source code is addressed to human readers. In Python, a comment is a piece of text that begins with #. The comment extends to the end of line.

2. If you want to place a comment that spans several lines, you need to place # in front of them all. Moreover, you can use a comment to mark a piece of code that is not needed at the moment (see the last line of the snippet below), e.g.:

# This program prints

# an introduction to the screen.

print("Hello!") # Invoking the print() function

# print("I'm Python.")

3. Whenever possible and justified, you should give **self-commenting names** to variables, e.g., if you're using two variables to store a length and width of something, the variable names length and width may be a better choice than myvar1 and myvar2.

4. It's important to use comments to make programs easier to understand, and to use readable and meaningful variable names in code. However, it's equally important **not to use** variable names that are confusing, or leave comments that contain wrong or incorrect information!

5. Comments can be important when *you* are reading your own code after some time (trust us, developers do forget what their own code does), and when *others* are reading your code (can help them understand what your programs do and how they do it more quickly).

**Exercise 1**

What is the output of the following snippet?

# print("String #1")

print("String #2")

Check

String #2

**Exercise 2**

What will happen when you run the following code?

# This is

a multiline

comment. #

print("Hello!")

Check

SyntaxError: invalid syntax

# The input() function

We're now going to introduce you to a completely new function, which seems to be a mirror reflection of the good old print() function.

Why? Well, print() sends data to the console.

The new function gets data from it.

print() has no usable result. The meaning of the new function is to **return a very usable result**.

The function is named input(). The name of the function says everything.

The input() function is able to read data entered by the user and to return the same data to the running program.

The program can manipulate the data, making the code truly interactive.

Virtually all programs **read and process data**. A program which doesn't get a user's input is a **deaf program**.

Take a look at our example:

print("Tell me anything...")

anything = input()

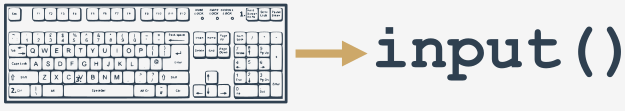
print("Hmm...", anything, "... Really?")

It shows a very simple case of using the input() function.

Note:

* The program **prompts the user to input some data** from the console (most likely using a keyboard, although it is also possible to input data using voice or image);
* the input() function is invoked without arguments (this is the simplest way of using the function); the function will **switch the console to input mode**; you'll see a blinking cursor, and you'll be able to input some keystrokes, finishing off by hitting the *Enter* key; all the inputted data will be **sent to your program** through the function's result;
* note: you need to assign the result to a variable; this is crucial - missing out this step will cause the entered data to be lost;
* then we use the print() function to output the data we get, with some additional remarks.

Try to run the code and let the function show you what it can do for you.



# The input() function with an argument

The input() function can do something else: it can prompt the user without any help from print().

We've modified our example a bit, look at the code:

anything = input("Tell me anything...")

print("Hmm...", anything, "...Really?")

Note:

* the input() function is invoked with one argument - it's a string containing a message;
* the message will be displayed on the console before the user is given an opportunity to enter anything;
* input() will then do its job.

This variant of the input() invocation simplifies the code and makes it clearer.

# The result of the input() function

We've said it already, but it must be unambiguously stated once again: the **result of the**input()**function is a string**.

A string containing all the characters the user enters from the keyboard. It is not an integer or a float.

This means that **you mustn't use it as an argument of any arithmetic operation**, e.g., you can't use this data to square it, divide it by anything, or divide anything by it.

anything = input("Enter a number: ")

something = anything \*\* 2.0

print(anything, "to the power of 2 is", something)

**The input() function - prohibited operations**

Look at the code in the editor. Run it, enter any number, and press *Enter*.

What happens?

Python should have given you the following output:

Traceback (most recent call last):

File ".main.py", line 4, in <module>

something = anything \*\* 2.0

TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'float'

**output**

The last line of the sentence explains everything - you tried to apply the \*\* operator to 'str' (string) accompanied with 'float'.

This is prohibited.

This should be obvious - can you predict the value of "to be or not to be" raised to the power of 2?

We can't. Python can't either.

Have we fallen into a deadlock? Is there a solution to this issue? Of course there is.

# Testing TypeError message.

anything = input("Enter a number: ")

something = anything \*\* 2.0

print(anything, "to the power of 2 is", something)

# Type casting

Python offers two simple functions to specify a type of data and solve this problem - here they are: int() and float().

Their names are self-commenting:

* the int() function **takes one argument** (e.g., a string: int(string)) and tries to convert it into an integer; if it fails, the whole program will fail too (there is a workaround for this situation, but we'll show you this a little later);
* the float() function takes one argument (e.g., a string: float(string)) and tries to convert it into a float (the rest is the same).

This is very simple and very effective. Moreover, you can invoke any of the functions by passing the input() results directly to them. There's no need to use any variable as an intermediate storage.

We've implemented the idea in the editor - take a look at the code.

Can you imagine how the string entered by the user flows from input() into print()?

Try to run the modified code. Don't forget to enter a **valid number**.

Check some different values, small and big, negative and positive. Zero is a good input, too.

anything = float(input("Enter a number: "))

something = anything \*\* 2.0

print(anything, "to the power of 2 is", something)

**More about input() and type casting**

Having a team consisting of the trio input()-int()-float() opens up lots of new possibilities.

You'll eventually be able to write complete programs, accepting data in the form of numbers, processing them and displaying the results.

Of course, these programs will be very primitive and not very usable, as they cannot make decisions, and consequently are not able to react differently to different situations.

This is not really a problem, though; we'll show you how to overcome it soon.

Our next example refers to the earlier program to find the length of a hypotenuse. Let's rewrite it and make it able to read the lengths of the legs from the console.

Check out the editor window - this is how it looks now.

The program asks the user twice for both legs' lengths, evaluates the hypotenuse and prints the result.

Run it and try to input some negative values.

The program - unfortunately - doesn't react to this obvious error.

Let's ignore this weakness for now. We'll come back to it soon.

Note that in the program that you can see in the editor, the hypo variable is used for only one purpose - to save the calculated value between the execution of the adjoining line of code.

As the print() function accepts an expression as its argument, you can **remove the variable** from the code.

Just like this:

leg\_a = float(input("Input first leg length: "))

leg\_b = float(input("Input second leg length: "))

print("Hypotenuse length is", (leg\_a\*\*2 + leg\_b\*\*2) \*\* .5)

leg\_a = float(input("Input first leg length: "))

leg\_b = float(input("Input second leg length: "))

hypo = (leg\_a\*\*2 + leg\_b\*\*2) \*\* .5

print("Hypotenuse length is", hypo)

**String operators - introduction**

It's time to return to these two arithmetic operators: + and \*.

We want to show you that they have a second function. They are able to do something more than just **add** and **multiply**.

We've seen them in action where their arguments are numbers (floats or integers, it doesn't matter).

Now we're going to show you that they can handle strings, too, albeit in a very specific way.

**Concatenation**

The + (plus) sign, when applied to two strings, becomes a **concatenation operator**:

string + string

It simply **concatenates** (glues) two strings into one. Of course, like its arithmetic sibling, it can be used more than once in one expression, and in such a context it behaves according to left-sided binding.

In contrast to its arithmetic sibling, the concatenation operator is **not commutative**, i.e., "ab" + "ba" is not the same as "ba" + "ab".

Don't forget - if you want the + sign to be a **concatenator**, not an adder, you must ensure that **both its arguments are strings**.

You cannot mix types here.

This simple program shows the + sign in its second use:

fnam = input("May I have your first name, please? ")

lnam = input("May I have your last name, please? ")

print("Thank you.")

print("\nYour name is " + fnam + " " + lnam + ".")

Note: using + to concatenate strings lets you construct the output in a more precise way than with a pure print() function, even if enriched with the end= and sep= keyword arguments.

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

# Replication

The \* (asterisk) sign, when applied to a string and number (or a number and string, as it remains commutative in this position) becomes a **replication operator**:

string \* number

number \* string

It replicates the string the same number of times specified by the number.

For example:

* "James" \* 3 gives "JamesJamesJames"
* 3 \* "an" gives "ananan"
* 5 \* "2" (or "2" \* 5) gives "22222" (not 10!)

REMEMBER

A number less than or equal to zero produces an **empty string**.

This simple program "draws" a rectangle, making use of an old operator (+) in a new role:

print("+" + 10 \* "-" + "+")

print(("|" + " " \* 10 + "|\n") \* 5, end="")

print("+" + 10 \* "-" + "+")

Note the way in which we've used the parentheses in the second line of the code.

Try practicing to create other shapes or your own artwork!

**Type conversion: str()**

You already know how to use the int() and float() functions to convert a string into a number.

This type of conversion is not a one-way street. You can also **convert a number into a string**, which is way easier and safer ‒ this kind of operation is always possible.

A function capable of doing that is called str():

str(number)

To be honest, it can do much more than just transform numbers into strings, but that can wait for later.

**The "right-angle triangle" again**

Here is our "right-angle triangle" program again:

leg\_a = float(input("Input first leg length: "))

leg\_b = float(input("Input second leg length: "))

print("Hypotenuse length is " + str((leg\_a\*\*2 + leg\_b\*\*2) \*\* .5))

We've modified it a bit to show you how the str() function works. Thanks to this, we can **pass the whole result to the**print()**function as one string**, forgetting about the commas.

You've made some serious strides on your way to Python programming.

You already know the basic data types, and a set of fundamental operators. You know how to organize the output and how to get data from the user. These are very strong foundations for Module 3. But before we move on to the next module, let's do a few labs, and recap all that you've learned in this section.

**LAB**

## Estimated time

5-10 minutes

## Level of difficulty

Easy

## Objectives

* becoming familiar with the inputting and outputting of data in Python;
* evaluating simple expressions.

## Scenario

Your task is to complete the code in order to evaluate the results of four basic arithmetic operations.

The results have to be printed to the console.

You may not be able to protect the code from a user who wants to divide by zero. That's okay, don't worry about it for now.

Test your code - does it produce the results you expect?

We won't show you any test data - that would be too simple.

# input a float value for variable a here

# input a float value for variable b here

# output the result of addition here

# output the result of subtraction here

# output the result of multiplication here

# output the result of division here

print("\nThat's all, folks!")

**LAB**

## Estimated time

20 minutes

## Level of difficulty

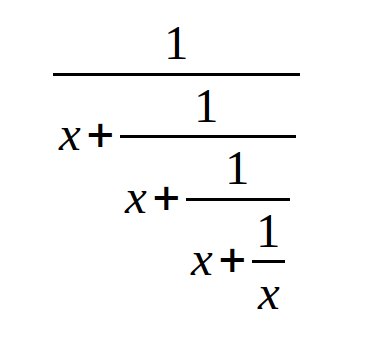
Intermediate

## Objectives

* becoming familiar with the concept of numbers, operators and arithmetic operations in Python;
* understanding the precedence and associativity of Python operators, as well as the proper use of parentheses.

## Scenario

Your task is to complete the code in order to evaluate the following expression:



The result should be assigned to y. Be careful - watch the operators and keep their priorities in mind. Don't hesitate to use as many parentheses as you need.

You can use additional variables to shorten the expression (but it's not necessary). Test your code carefully.

## Test Data

Sample input: 1

Expected output:

y = 0.6000000000000001

Sample input: 10

Expected output:

y = 0.09901951266867294

Sample input: 100

Expected output:

y = 0.009999000199950014

Sample input: -5

Expected output:

y = -0.19258202567760344

x = float(input("Enter value for x: "))

# Write your code here.

print("y =", y)

**LAB**

## Estimated time

15-20 minutes

## Level of difficulty

Easy

## Objectives

* improving the ability to use numbers, operators, and arithmetic operations in Python;
* using the print() function's formatting capabilities;
* learning to express everyday-life phenomena in terms of programming language.

## Scenario

Your task is to prepare a simple code able to evaluate the **end time** of a period of time, given as a number of minutes (it could be arbitrarily large). The start time is given as a pair of hours (0..23) and minutes (0..59). The result has to be printed to the console.

For example, if an event starts at **12:17** and lasts **59 minutes**, it will end at **13:16**.

Don't worry about any imperfections in your code - it's okay if it accepts an invalid time - the most important thing is that the code produce valid results for valid input data.

Test your code carefully. Hint: using the % operator may be the key to success.

## Test Data

Sample input:12

17

59

Expected output: 13:16

Sample input:23

58

642

Expected output: 10:40

Sample input:0

1

2939

Expected output: 1:0

hour = int(input("Starting time (hours): "))

mins = int(input("Starting time (minutes): "))

dura = int(input("Event duration (minutes): "))

# Write your code here.

**Key takeaways**

1. The print() function **sends data to the console**, while the input() function **gets data from the console**.

2. The input() function comes with an optional parameter: **the prompt string**. It allows you to write a message before the user input, e.g.:

name = input("Enter your name: ")

print("Hello, " + name + ". Nice to meet you!")

3. When the input() function is called, the program's flow is stopped, the prompt symbol keeps blinking (it prompts the user to take action when the console is switched to input mode) until the user has entered an input and/or pressed the *Enter* key.

NOTE

You can test the functionality of the input() function in its full scope locally on your machine. For resource optimization reasons, we have limited the maximum program execution time in Edube to a few seconds. Go to the Sandbox, copy-paste the above snippet, run the program, and do nothing ‒ just wait a few seconds to see what happens. Your program should be stopped automatically after a short moment. Now open IDLE, and run the same program there ‒ can you see the difference?

Tip: the above-mentioned feature of the input() function can be used to prompt the user to end a program. Look at the code below:

name = input("Enter your name: ")

print("Hello, " + name + ". Nice to meet you!")

print("\nPress Enter to end the program.")

input()

print("THE END.")

4. The result of the input() function is a string. You can add strings to each other using the concatenation (+) operator. Check out this code:

num\_1 = input("Enter the first number: ") # Enter 12

num\_2 = input("Enter the second number: ") # Enter 21

print(num\_1 + num\_2) # the program returns 1221

5. You can also multiply (\* ‒ replication) strings, e.g.:

my\_input = input("Enter something: ") # Example input: hello

print(my\_input \* 3) # Expected output: hellohellohello

**Exercise 1**

What is the output of the following snippet?

x = int(input("Enter a number: ")) # The user enters 2

print(x \* "5")

Check

55

**Exercise 2**

What is the expected output of the following snippet?

x = input("Enter a number: ") # The user enters 2

print(type(x))

Check

<class 'str'>

# Congratulations! You have completed Module 2.

Well done! You've reached the end of Module 2 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 2:

* the basic methods of formatting and outputting data offered by Python, together with the primary kinds of data and numerical operators, their mutual relations and bindings;
* the concept of variables and variable naming conventions;
* the assignment operator, the rules governing the building of expressions;
* the inputting and converting of data;

You are now ready to take the module quiz and attempt the final challenge: Module 2 Test, which will help you gauge what you've learned so far.