**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.

**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**

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 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

**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**

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?

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.

## 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

**output**

# 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

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

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.

Through this example, you encounter two different types of literals:

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

print("2")

print(2)

**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.
* **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"')
* **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
* print('I\'m Monty Python.')  
  or  
    
  print("I'm Monty Python.")
* 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?

**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

They're both strings/string literals.

**Exercise 2**

What types of literals are the following four examples?

"1.5", 2.0, 528, False  
Check

The first is a string, the second is a numerical literal (a float), the third is a numerical literal (an integer), and the fourth is a boolean literal.

**Exercise 3**

What is the decimal value of the following binary number?

1011  
Check

It's 11, because (2\*\*0) + (2\*\*1) + (2\*\*3) = 11

**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.

**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

3.0 - not 3 but 3.0 (the rule still works: 12 // 4.5 gives 2.0; 2.0 \* 4.5 gives 9.0; 12 - 9.0 gives 3.0)

**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

correct is 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)

answer 10.0

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

16 8.0 8

**Exercise 2**

What is the output of the following snippet?

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

Check

-0.5 0.5 0 -1

**Exercise 3**

What is the output of the following snippet?

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

Check

-2 2 512