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 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.

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)

Example:

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

print("Monty Python.")

Result:

My name is Python. Monty Python.

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

The default behavior reflects the situation where the end keyword argument is **implicitly** used in the following way: end="\n".

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.

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

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

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

result:

My\_name\_is\*Monty\*Python.\*

Example:

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

print("Python")

result:

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

Example:

print(" \*")

print(" \* \*")

print(" \* \*")

print(" \* \*")

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

print(" \* \*")

print(" \* \*")

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

result:

\*

\* \*

\* \*

\* \*

\*\*\* \*\*\*

\* \*

\* \*

\*\*\*\*\*

Same result:

print(" \*\n"," \* \*\n"," \* \*\n","\* \*")

print("\*\*\* \*\*\*\n"," \* \*\n"," \* \*\n"," \*\*\*\*\*")

example:

print(" \* " \* 2)

print(" \* \* " \* 2)

print(" \* \* " \* 2)

print(" \* \* " \* 2)

print("\*\*\* \*\*\*" \* 2)

print(" \* \* " \* 2)

print(" \* \* " \* 2)

print(" \*\*\*\*\* " \* 2)

result:

\* \*

\* \* \* \*

\* \* \* \*

\* \* \* \*

\*\*\* \*\*\*\*\*\* \*\*\*

\* \* \* \*

\* \* \* \*

\*\*\*\*\* \*\*\*\*\*

**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.

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.

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)

equals = 83

The second convention allows us to use **hexadecimal** numbers. Such numbers should be preceded by the prefix 0x or 0X (zero-x).

0x123 is a **hexadecimal** number with a (decimal) value equal to 291. The print() function can manage these values too. Try this:

print(0x123)

**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.

**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 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

Result = 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

1

**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.

Reserved words:

['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']

**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

sample problem:

x = -1 # hardcode your test data here

x = float(x)

# write your code here

y = (3 \* x\*\*3) - (2 \* x\*\*2) + (3 \* x) -1

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)

**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).

Example:

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

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

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

result:

+----------+

| |

| |

| |

| |

| |

+----------+

Sandbox Lab Operators and expressions:

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

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

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

# Write your code here.

start\_total\_min = (hour \* 60) + mins

end\_total\_min = start\_total\_min + dura

end\_hour = end\_total\_min // 60

if end\_hour > 23:

end\_hour = end\_hour - 24

end\_min = round(end\_total\_min % 60, 2)

print("The end of the event is at: " + str(end\_hour) + ":" + str(end\_min))

correct code:

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

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

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

# Write your code here.

start\_total\_min = (hour \* 60) + mins

end\_total\_min = start\_total\_min + dura

end\_hour = end\_total\_min // 60

end\_hour = end\_hour % 24

end\_min = end\_total\_min % 60

print("The end of the event is at: " + str(end\_hour) + ":" + str(end\_min))

**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