CS 10 - Module 1: Introduction to Python and Computer Programming

Getting Started with Python

How does a computer program work?

This course aims to show you what the Python language is and what it is used for. Let's start from the absolute basics.

A program makes a computer usable. Without a program, a computer, even the most powerful one, is nothing more than an object. Similarly, without a player, a piano is nothing more than a wooden box.

Computers can perform very complex tasks, but this ability is not innate. A computer's nature is quite different.

It can execute only extremely simple operations. For example, a computer cannot understand the value of a complicated mathematical function by itself, although this isn't beyond the realms of possibility soon.

Contemporary computers can only evaluate the results of very fundamental operations, like adding or dividing, but they can do it very fast, and can repeat these actions virtually any number of times.

Imagine that you want to know the average speed you've reached during a long journey. You know the distance and the time it took. The computer will be able to compute the speed, but the computer is not aware of such things as distance, speed, or time. Therefore, it is necessary to instruct the computer to:

- accept a number representing the distance.
- accept a number representing the travel time.
- divide the former value by the latter and store the result in the memory.
- display the result (representing the average speed) in a readable format.

These four simple actions form a **program**. Of course, these examples are not formalized, and they are very far from what the computer can understand, but they are good enough to be translated into a language the computer can accept.

Natural languages vs. programming languages

A language is a means (a tool) for expressing and recording thoughts. There are many languages all around us. Some of them require neither speaking nor writing, such as body language.

Another language you use each day is your mother tongue, which you use to manifest your will and to ponder reality. Computers have their own language, too, called **machine language**, which is very rudimentary. Machine languages are developed by humans.

A computer, even the most technically sophisticated, is devoid of even a trace of intelligence. It responds only to a predetermined set of known commands.

The commands it recognizes are very simple. We can imagine that the computer responds to orders like "take that number, divide by another and save the result".

A complete set of known commands is called an **instruction list** (**IL**). Different types of computers may vary depending on the size of their ILs, and the instructions could be completely different in different models.

No computer is currently capable of creating a new language. However, that may change soon. Just as people use several very different languages, machines have many different languages, too. The difference, though, is that human languages developed naturally.

Moreover, they are still evolving, and new words are created every day as old words disappear. These languages are called **natural languages**.

What makes a language?

We can say that each language (machine or natural) consists of the following elements:

- an **alphabet**: a set of symbols used to build words of a certain language
- a **lexis**: (aka a dictionary) a set of words the language offers its users
- a **syntax**: a set of rules used to determine if a certain string of words forms a valid sentence
- **semantics**: a set of rules determining if a certain phrase makes sense

The IL is, in fact, the **alphabet of a machine language**. This is the simplest and most primary set of symbols we can use to give commands to a computer. It's the computer's mother tongue.

The computer's mother tongue is a far cry from a human mother tongue. We need a common language for computers and humans. A bridge between the two different worlds. A language in which humans can write their programs and a language that computers may use to execute the programs. One that is far more complex than machine language and yet far simpler than natural language.

Such languages are often called **high-level programming languages**. They are at least somewhat like natural ones in that they use symbols, words, and conventions readable to humans. These languages enable humans to express commands to computers that are much more complex than those offered by ILs.

A program written in a high-level programming language is called a **source code**. Similarly, the file containing the source code is called the **source file**.

Compilation vs. interpretation

Computer programming is the act of composing the selected programming language's elements in the order that will cause the desired effect. The effect could be different in every specific case - it's up to the programmer's imagination, knowledge, and experience.

Of course, such a composition must be correct in many senses:

- **alphabetically** - a program needs to be written in a recognizable script.
- **lexically** - each programming language has its dictionary, and you need to master it.
- **syntactically** - each language has its rules, and they must be obeyed.
- **semantically** - the program must make sense.

Unfortunately, a programmer can also make mistakes with each of the above items. Each of them can cause the program to become completely useless.

Let's assume that you've successfully written a program. How do we persuade the computer to execute it? You must render your program into machine language. Luckily, the translation can be done by a computer itself.

There are two different ways of transforming a program from a high-level programming language into machine language. There are very few languages that can be both compiled and interpreted.

	Compilation	**Interpretation**
Advantages	Execution of the translated code is usually faster	You can run the code as soon as you complete it
	Only the user must have the compiler. The end-user may use the code without it	the code is stored using programming language, not the machine one. So, it can be run on computers using different machine languages
	The translated code is stored using machine language	
Disadvantages	The compilation itself may be a very time-consuming process	Your code will share the computer's power with the interpreter, so it can't be fast
	You must have as many compilers as hardware platforms you want your code to be run on	Both you and the end user must have the interpreter to run your code

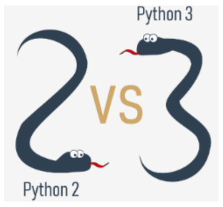
What is Python?

Python is an object-oriented **interpreted language**. This means that it inherits all the advantages and disadvantages. If you want to program in Python, you'll need the **Python interpreter**. You won't be able to run your code without it.

Languages designed to be utilized in the interpretation manner are often called **scripting languages**, while the source programs encoded using them are called **scripts**.

While you may know the python as a large snake, the name of the Python programming language comes from an old comedy sketch series called **Monty Python's Flying Circus**.

Python was created by Guido van Rossum, born in 1956 in Haarlem, the Netherlands.



There are two main kinds of Python, called Python 2 and Python 3. **Python 2** is an older version. **Python 3** is the current version of the language and this is the version of Python that will be used during this course. All the code in the course have been tested against Python 3.4, Python 3.6, Python 3.7, and Python 3.8.

Other versions of Python:

Guido used the "C" programming language to implement the first version of his language. All Pythons coming from the PSF group are written in the "C" language. Therefore, the PSF implementation is often referred to as **CPython**. This is the most influential Python among all the Pythons in the world.

Cython is one solution to the most painful of Python's trait - the lack of efficiency. Large and complex mathematical calculations may be easily coded in Python (much easier than in "C" or any other traditional language), but the resulting code's execution may be extremely time-consuming.

Another version of Python is called **Jython**. "J" is for "Java". Imagine a Python written in Java instead of C. This is useful if you develop large and complex systems written entirely in Java and want to add some Python flexibility to them.

The source code of PyPy is not run in the interpretation manner but is instead translated into the C programming language and then executed separately. This is useful if you want to test any new feature that may be introduced into mainstream Python implementation. Therefore, PyPy is rather a tool for people developing Python than for the rest of the users.

Python Tools

To start your work, you need the following tools:

- an **editor** which will support you in writing the code
- a **console** in which you can launch your newly written code and stop it forcibly
- a tool named a **debugger**, able to launch your code step by step and to inspect it

Python 3 standard installation contains a very simple but extremely useful application named IDLE.

IDLE is an acronym: Integrated Development and Learning Environment.

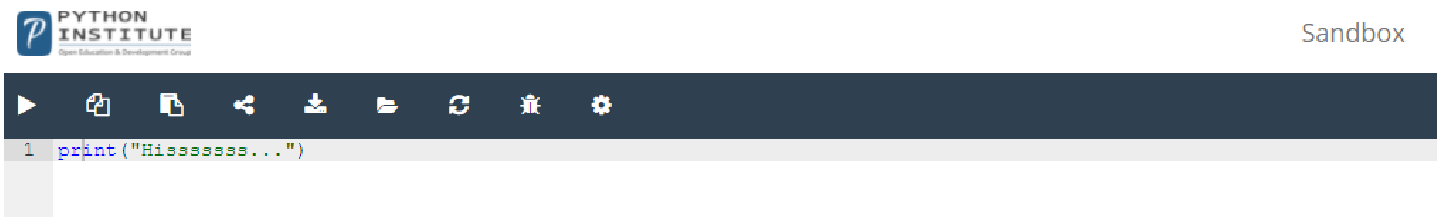
How to write and run your very first program

In this course we will not install and use Python's IDLE. Instead, we will enter and run all the code from an online Sandbox for Python 3.

Open a browser and enter the following URL: https://edube.org/sandbox

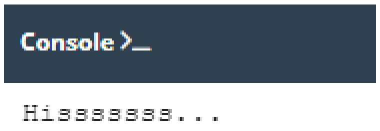
It is now time to write and run your first Python. Now put just one line into your newly opened editor window. The line looks like this:

print("Hisssssss...")



```
print("Hisssssss...")
```

Run the program, press this button on the Sandbox:



```
Hisssssss...
```

If everything goes okay and there are no mistakes in the code, the console window will show you the effects caused by running the program.

Run it once again. Now clear the Console window, press this button on the Sandbox:

Download this code and save it to your Python folder for this course.

Rename the code as **YourName_Hisssss**

How to spoil and fix your code – Example 1

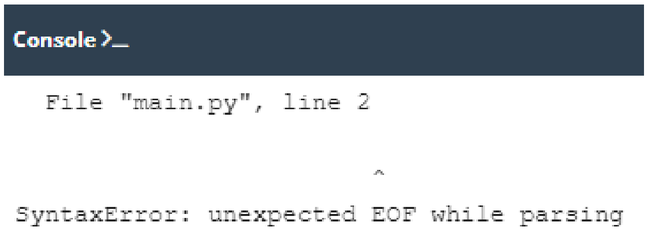
You will now explore the IDLE's helpful feature.

Open and load your **Hisssss** program into the Sandbox:

First, remove the closing parenthesis (bracket at the end). Try to run the program.

The code becomes erroneous. It contains a syntax error now. IDLE should not let you run it.

In the Console window, it says that the interpreter has encountered an **EOF** (end-of-file) although (in its opinion) the code should contain some more text.



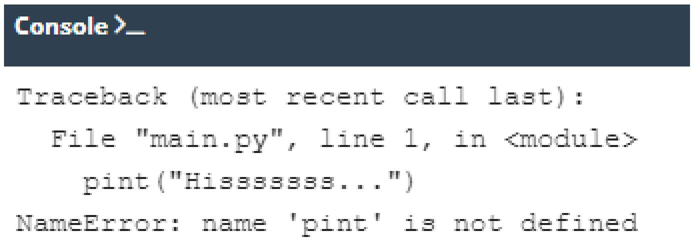
```
  File "main.py", line 2

                     ^
SyntaxError: unexpected EOF while parsing
```

Fix the code now. Run it to see if it "hisses" at you.

How to spoil and fix your code – Example 2

Let's spoil the code one more time. Remove one letter from the word print. Run the code. As you can see, Python is not able to recognize the error.



```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    pint("Hisssssss...")
NameError: name 'pint' is not defined
```

You may have noticed that the error message generated for this error is quite different from the first one. This is because the nature of the error is different, and the error is discovered at a different stage of interpretation.

The console window might provide some useful information regarding the error.

The message is explained below

- **traceback**: Which is the path that the code traverses through different parts of the program.
- **location of the error**: The file name, line number, and module name.
- **content of the erroneous line**: Displays the current cursor location at the bottom-right corner, use it to locate the erroneous line in a long source code.
- the **name of the error** and a short explanation.

Experiment with creating new code. Try to output a different message to the screen, e.g., roar, meow, or even maybe an oink. Try to spoil and fix your code - see what happens.

Congratulations! You have completed Module 1

Hand in:

Python Module 1 – Worksheet and Python code for marking

Python Module 2:

Ask for a copy of Python Module 2 Worksheet

Python Module 2 course content continues below….

CS 10 - Module 2: Output Operations, Data Types, and Operators

The Print Function

Hello, World!

It's time to start writing some real, working Python code. It'll be very simple for the time being, as I show you some fundamental concepts and terms.

Open a browser and the Python Sandbox: https://edube.org/sandbox

Enter the following code: **print (“Hello, World!”)**

Now run it. If everything goes okay, you'll see the output in the console window.

Now we'll spend some time showing and explaining what you're 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 marks**
- a line of text: **Hello, World!**
- another **quotation marks**
- a **closing parenthesis**

Each of the above plays a very important role in the code. We will look at each of these below.

The `print()` function

Look at the line of code: **print ("Hello, World!")**

The word print() 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 that can:

- Cause some effect (ex. send text to a terminal, create a file, draw an image, play a sound, etc)
- Evaluate a value (ex. the square root of a value, the length of a given text) and return the result

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 built-in Python environment
- 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
- You can write them yourself, placing as many functions as you want inside your program

The name of the function should be significant (the name of the print function is self-evident).

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 carefully consider your choice of names.

Three Function components:

- An effect
- A result
- The argument(s)

Mathematical functions usually take one argument: sin(x) takes an x, which is an angle measure.

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. Despite the number of needed/provided arguments, Python functions demand the presence of a pair of parentheses - opening and closing ones.

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 must 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 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, the quotes make the string. They cut out a part of the code and assign a different meaning to it. Think of 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 as data.

So far, you have learned about two important parts of the code: the function and the string. We've talked about them in terms of syntax, but now it's time to discuss them in terms of semantics.

The function name (print in this case) along with the parentheses and argument(s), forms the function invocation. We'll discuss this in more depth soon, but we should just shed a little light on it right now.

What happens when Python encounters an invocation like this one below?

function_name(argument) As an example: print("Hello, World!")

- 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, it takes your argument(s) too and passes it/them to the function
- Fourth, the function executes its code, causes the desired effect (if any), evaluates the desired result(s) (if any) and finishes its task
- Finally, Python returns to your code (to the place just after the invocation) and resumes its execution.

Three important questions about the print() function need be answered:

1. What is the effect the print() function causes?

The function:

- Takes its arguments (zero or more)
- Converts them into human-readable form if needed
- Sends the resulting data to the output device (your screen)

2. What arguments does print() expect?

Any, print() is able to operate with strings, numbers, characters, logical values, objects

3. What value does the print() function return? None. Its effect is enough.

At this point you need to complete the following lab located in Python Lab Manual.

Lab – Print1

The print() function - instructions

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).

The print() function - the escape and newline characters

Carefully look at the modified the code below.

print("The itsy bitsy spider\nclimbed up the waterspout.")

There are two very subtle changes - we've inserted a strange pair of characters inside the string. They look like this: **\n** Interestingly. While you can see two characters, Python sees only one.

The backslash (****) has a very special meaning when used inside strings - this is called the escape character. The word escape 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, but is only a kind of announcement, that the next character after the backslash has a different meaning. 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 forces the console to start a new output line.

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 must 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.

In the code there is one print() function invocation, but it contains three arguments. All are strings.
print("The itsy bitsy spider" , "climbed up" , "the waterspout.")

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

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.

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).

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.

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.

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 must be put after the last positional argument (this is very important)

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).

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

The sep argument's value may be an empty string, too. Try it for yourself. Also, both keyword arguments may be **mixed in one invocation**.

At this point you need to complete the following lab located in Python Lab Manual.

Lab – Print2　　　　**Lab – Print3a, 3b, 3c, 3d**

Literals

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 one important new term - the literal.

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

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 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.

Look at the code below. The second seems to be erroneous due to the visible lack of quotes.

```
print("2")

print(2)
```

However, the output is:

```
2

2
```

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. 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.

Integers

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.

Numbers handled by modern computers are of two types:

- **Integer**, those which are devoid of the fractional part
- **Floating-point** numbers (or simply float), that can contain the fractional part.

Both 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. We will focus on integers now.

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. Like these 11,111,111, or this: 11.111.111, or even this: 11 111 111.

Python does allow the use of underscores in numeric literals. Therefore, you can write this number either like: 11111111, or: 11_111_111.

Code negative numbers in Python? You can write: -11111111, or -11_111_111.

Code positive numbers in Python? You can write: +11111111 or +11_111_111 or 11111111

Integers: octal and hexadecimal numbers

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 to 7] range only.

0o123 is an **octal** number with a (decimal) value equal to 83.

Hexadecimal numbers are preceded by the prefix 0x or 0X (zero-x).

0x123 is a **hexadecimal** number with a (decimal) value equal to 291.

Floats

They are the numbers that have (or may have) a fractional part after the decimal point 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
Don't forget this rule - you can omit zero when it is the only digit in front of or after the decimal point.

You can write 0.4 as: **.4**   The value of 4.0 as: **4.**   This will change neither its type nor its value.

Integers vs. floats

Look at these two numbers: 4 and 4.0 You may think that they are 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: 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. The exponent (the value after the E) must be an integer but the base (the value in front of the E) may be an integer.

Coding floats

A physical constant called Planck's constant (and denoted as h), has the value: 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. 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.

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 must close it with a quote. If you start a string with an apostrophe, you must end it with an apostrophe.

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. The backslash is a very powerful tool - it can escape not only quotes, but also apostrophes.

A string can contain no characters at all. An empty string remains a string: ' ' or " "

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 type, a Boolean value. 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 and 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.

These two Boolean values have strict denotations in Python: True and False

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

At this point you need to complete the following lab located in Python Lab Manual.

Lab – Literal1

Operators

Basic operators

An operator is a symbol of the programming language, which can operate on the values. Just as in arithmetic, the + (plus) sign is the operator, which can add two numbers, giving the result of the addition.

Not all Python operators are as obvious as the plus sign. 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 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: 2^3. Pure text editors don't accept that, so Python uses ** instead, e.g., 2 ** 3.

You can surrounded the double asterisks with spaces as in the example. It's not compulsory, but it improves the readability of the code.

Rules when using asterisks:

- When both ** arguments are integers, the result is an integer
- When at least one ** argument is a float, the result is a float

This is an important distinction to remember.

Arithmetic operators: multiplication

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

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.

The result produced by the division operator is always a float, regardless of whether 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:

- The 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.

Imagine that we used / instead of //, it would be 1.5 in both cases. 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.

If some of the values are negative (like -6//4 or -6.//-4), this will obviously affect the result. 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 and -2.0. Integer division can also be called floor division.

Operators: remainder (modulo)

The next operator is quite a peculiar one, because it has no equivalent among traditional arithmetic operators. Its graphical representation is the % (percent) sign, which may look a bit confusing. 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.

Look at the code. print(14 % 4)

The result is two.

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.

As you know, division by zero doesn't work. You will get incorrect answers if you do the following:

- 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.

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 one of the binary operators, just like the addition, multiplication, and division operators.

But the minus and plus operator may be used in a different (unary) way.

The unary + 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.

Operators and their priorities

Consider the following expression: 2 + 3 * 5
You probably remember from math courses that multiplications precede additions. First multiply 3 by 5 and, 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.

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: 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

The correct answer is 1. This operator has left-sided binding. **But there's one interesting exception**.

Operators and their bindings: exponentiation

Exponentiation operator uses right-sided binding.

This simple example will show you how it works: print(2 ** 2 ** 3)

The two possible results are:

2 ** 2 → 4; 4 ** 3 → 64

2 ** 3 → 8; 2 ** 8 → 256

The correct answer is 256. This operator has right-sided binding.

List of priorities

This is not the complete list of operator priorities but a truncated form

Priority	Operator	
1	+, -	unary
2	**	
3	*, /, //, %	
4	+, -	binary

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.

Congratulations! You have completed Module 2

Hand in:

Python Module 2 – Worksheet and Python code (7)

Python Module 3:

Ask for a copy of Python Module 3 Worksheet

Python Module 3 course content continues below….

CS 10 - Module 3: Variables, Commenting, and Input Operations

Variables

What are variables?

It's quite a normal question to ask how to store the results of operations, 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 offers "boxes" (containers) that are called variables.

Every variable has.

- A name
- A value (the content of the container)

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.

Naming variables follow some strict rules:

- The name must be composed of upper-case or lower-case letters, digits, and the character _ (underscore)
- The name must begin with a letter
- The underscore character is a letter
- Upper- and lower-case letters are treated as different - Cat and CAT are the same words, but are two different variable names
- the name of the variable must not be any of Python's reserved words (keywords)

Correct and incorrect variable 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, days_to_christmas, TheNameIsSoLongThatYouWillMakeMistakesWithIt, _.

Python lets characters specific to languages that use other alphabets.

Use the following naming convention for variables and functions. Variable names should be lowercase, with words separated by underscores to improve readability (e.g., var, my_variable)

Keywords

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 reserved keywords. They are reserved because you mustn't use them as names. The meaning of the reserved word is predefined and mustn't be changed in any way. Since 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.

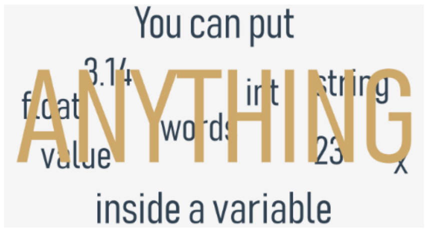
You can't name your variable:	You can do this instead
import	Import

Creating variables

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. **This is not normal for most programming languages.**

Let's talk 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).

IMPORTANT: In Python, variable comes into existence because 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 (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.

Look at the code:

```
sum = 1

print(var)
```

It consists of two simple instructions. The first of them creates a variable named sum and assigns a literal with an integer value equal to 1.

The second prints the value of the newly created variable to the console. print() can handle variables too.

Using variables

You're allowed to use as many variable declarations as you need to achieve your goal.

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

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

Assigning a new value to an already existing variable

To assign a new value to an already created variable 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
```

The first line of code creates a new variable named var and assigns 1 to it. The statement reads: assign a value of 1 to a variable named var.

The third line takes the current value of the variable var, add 1 to it and store the result in the variable var. Python treats the sign = not as equal to, but as assign a value. In effect, the value of variable var has been incremented by one, which has nothing to do with comparing the variable with any value.

At this point you need to complete the following lab located in Module 2 Lab Manual.

Lab – Variable1

Shortcut operators

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

For example:

```
x = x * 2
sheep = sheep + 1
```

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

```
x *= 2
sheep += 1
```

At this point you need to complete the following lab located in Python Lab Manual.

Lab – Variable2a

Lab – Variable2b

Lab – Variable3

Commenting Code

Leaving comments in code: why, how, and when

You may want to put in a few words usually to explain to other readers of the code, or the meanings of the variables etc. The remark inserted into, which is omitted at runtime, is called a comment.

Whenever Python encounters a comment, the comment is completely transparent to it. 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 must put a hash in front of them all.

Example:

```
# 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. However, the best way of commenting variables is to name them in an unambiguous manner. If a particular variable is designed to store an area of a square, the name square_area will obviously be better than aunt_jane. We say that the name is self-commenting. You can use comments to mark a piece of code that currently isn't needed for whatever reason. This is often done during the testing of a program.

```
# This is a test program.
x = 1
y = 2
# y = y + x
print(x + y)
```

TIP

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).

At this point you need to complete the following lab located in Python Lab Manual.

Lab – Comment1

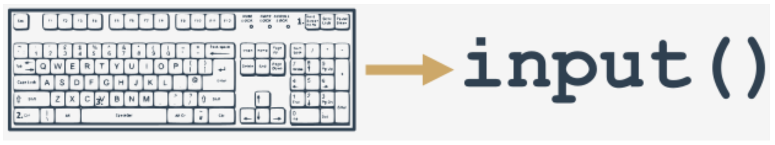
Input Operations

The input() function

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 (most likely using a keyboard, although it is also possible to input data using voice, files, image). The program can manipulate the data, making the code truly interactive.

Virtually all programs read and process data.

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



print("Tell me anything...")

anything = input()

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

Note:

- The program prompts the user to input some data from the console
- The input() function is invoked without arguments. The function will switch the console to input mode, you'll see a blinking cursor, and you'll be able to input some keystrokes. Then press the Enter key. All the inputted data is sent to the program through the function's result
- You need to assign the result to a variable; this is crucial - missing out this step will cause the entered data to be lost
- We use the print() function to output the data we get, with some additional remarks

The input() function with an argument

The input() function can also prompt the user We've modified our example a bit.

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 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 or divide it by anything, etc.

Type casting

Python offers two simple functions to specify a type of data: int() and float().

Their names are self-commenting:

- The int() function takes one argument (int(string)) and tries to convert it into an integer
- The float() function takes one argument (float(string)) and tries to convert it into a float

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.

Having code consisting of 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.

String operators - introduction

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. Don't forget you must ensure that both its arguments are strings. You cannot mix types.

The * (asterisk) sign, when applied to a string and number (or a number and string) becomes a replication operator. It replicates the string the same number of times specified by the number. A number less than or equal to zero produces an empty string.

string * number

number * string

Examples:

"James" * 3 gives "JamesJamesJames"

3 * "an" gives "ananan"

5 * "2" (or "2" * 5) gives "22222"

You can also convert a number into a string. A function capable of doing that is: str(number)

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.

At this point you need to complete the following lab located in Module 2 Lab Manual.

Lab – Input1

Lab – Input2

Lab – Input3

Hand in:

Python Module 3 –Worksheet and Python code

Congratulations! You have completed Module 3

Well done! You've reached the end of Module 3 and completed a major milestone in your Python programming education.

You are now ready to take the **Python Quiz**