Python Essentials 1:
Module 1

Introduction to Python and computer programming

In this module, you will learn about:

- the fundamentals of computer programming, i.e., how the computer works, how the program is executed, how the programming language is defined and constructed;
- the difference between compilation and interpretation
- what Python is, how it is positioned among other programming languages, and what distinguishes the different versions of 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 are able to 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 in the near future.

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, you know the time, you need the speed.

Naturally, the computer will be able to compute this, 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.

Language is the keyword.

Natural languages vs. programming languages

A language is a means (and 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; it's possible to express your deepest feelings very precisely without saying a word.

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.

A computer, even the most technically sophisticated, is devoid of even a trace of intelligence. You could say that it is like a well-trained dog - 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**, sometimes abbreviated to **IL**. Different types of computers may vary depending on the size of their ILs, and the instructions could be completely different in different models.

Note: machine languages are developed by humans.

What makes a language?

We can say that each language (machine or natural, it doesn't matter) consists of the following elements:

- an **alphabet**: a set of symbols used to build words of a certain language (e.g., the Latin alphabet for English, the Cyrillic alphabet for Russian, Kanji for Japanese, and so on)
- a **lexis**: (aka a dictionary) a set of words the language offers its users (e.g., the word "computer" comes from the English language dictionary, while "cmoptrue" doesn't; the word "chat" is present both in English and French dictionaries, but their meanings are different)
- a **syntax**: a set of rules (formal or informal, written or felt intuitively) used to determine if a certain string of words forms a valid sentence (e.g., "I am a python" is a syntactically correct phrase, while "I a python am" isn't)
- **semantics**: a set of rules determining if a certain phrase makes sense (e.g., "I ate a doughnut" makes sense, but "A doughnut ate me" doesn't)

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.

Unfortunately, this tongue is a far cry from a human mother tongue. We all (both computers and humans) need something else, a common language for computers and humans, or a bridge between the two different worlds.

We need 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 similar to 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** (in contrast to the machine code executed by computers). 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 has to be correct in many senses:

- **alphabetically** - a program needs to be written in a recognizable script, such as Roman, Cyrillic, etc.
- **lexically** - each programming language has its dictionary and you need to master it; thankfully, it's much simpler and smaller than the dictionary of any natural language;
- **syntactically** - each language has its rules and they must be obeyed;
- **semantically** - the program has to make sense.

Unfortunately, a programmer can also make mistakes with each of the above four senses. 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 have to render your program into machine language. Luckily, the translation can be done by a computer itself, making the whole process fast and efficient.

There are two different ways of **transforming a program from a high-level programming language into machine language**:

COMPILATION - the source program is translated once (however, this act must be repeated each time you modify the source code) by getting a file (e.g., an .exe file if the code is intended to be run under MS Windows) containing the machine code; now you can distribute the file worldwide; the program that performs this translation is called a compiler or translator;

INTERPRETATION - you (or any user of the code) can translate the source program each time it has to be run; the program performing this kind of transformation is called an interpreter, as it interprets the code every time it is intended to be executed; it also means that you cannot just distribute the source code as-is, because the end-user also needs the interpreter to execute it.

Due to some very fundamental reasons, a particular high-level programming language is designed to fall into one of these two categories.

There are very few languages that can be both compiled and interpreted. Usually, a programming language is projected with this factor in its constructors' minds - will it be compiled or interpreted?

What does the interpreter actually do?

Let's assume once more that you have written a program. Now, it exists as a **computer file**: a computer program is actually a piece of text, so the source code is usually placed in **text files**.

Note: it has to be **pure text**, without any decorations like different fonts, colors, embedded images or other media. Now you have to invoke the interpreter and let it read your source file.

The interpreter reads the source code in a way that is common in Western culture: from top to bottom and from left to right. There are some exceptions - they'll be covered later in the course.

First of all, the interpreter checks if all subsequent lines are correct (using the four aspects covered earlier).

If the compiler finds an error, it finishes its work immediately. The only result in this case is an **error message**.

The interpreter will inform you where the error is located and what caused it. However, these messages may be misleading, as the interpreter isn't able to follow your exact intentions, and may detect errors at some distance from their real causes.

For example, if you try to use an entity of an unknown name, it will cause an error, but the error will be discovered in the place where it tries to use the entity, not where the new entity's name was introduced.

In other words, the actual reason is usually located a little earlier in the code, for example, in the place where you had to inform the interpreter that you were going to use the entity of the name.

Compilation vs. interpretation - advantages and disadvantages

	COMPILATION	INTERPRETATION
ADVANTAGES	• the execution of the translated code is usually faster; • only the user has to have the compiler - the end-user may use the code without it; • the translated code is stored using machine language - as it is very hard to understand it, your own inventions and programming tricks are likely to remain your secret.	• you can run the code as soon as you complete it - there are no additional phases of translation; • the code is stored using programming language, not the machine one - this means that it can be run on computers using different machine languages; you don't compile your code separately for each different architecture.
DISADVANTAGES	• the compilation itself may be a very time-consuming process - you may not be able to run your code immediately after any amendment; • you have to have as many compilers as hardware platforms you want your code to be run on.	• don't expect that interpretation will ramp your code to high speed - your code will share the computer's power with the interpreter, so it can't be really fast; • both you and the end user have to have the interpreter to run your code.

What does this all mean for you?

- Python is an **interpreted language**. This means that it inherits all the described advantages and disadvantages. Of course, it adds some of its unique features to both sets.
- If you want to program in Python, you'll need the **Python interpreter**. You won't be able to run your code without it. Fortunately, **Python is free**. This is one of its most important advantages.

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

What is Python?

Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.

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

At the height of its success, the Monty Python team were performing their sketches to live audiences across the world, including at the Hollywood Bowl.

Since Monty Python is considered one of the two fundamental nutrients to a programmer (the other being pizza), Python's creator named the language in honor of the TV show.

Who created Python?

One of the amazing features of Python is the fact that it is actually one person's work. Usually, new programming languages are developed and published by large companies employing lots of professionals, and due to copyright rules, it is very hard to name any of the people involved in the project. Python is an exception.

There are not many languages whose authors are known by name. Python was created by **Guido van Rossum**, born in 1956 in Haarlem, the Netherlands. Of course, Guido van Rossum did not develop and evolve all the Python components himself

The speed with which Python has spread around the world is a result of the continuous work of thousands (very often anonymous) programmers, testers, users (many of them aren't IT specialists) and enthusiasts, but it must be said that the very first idea (the seed from which Python sprouted) came to one head - Guido's.

A hobby programming project

The circumstances in which Python was created are a bit puzzling. According to Guido van Rossum:

> In December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office (...) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).*Guido van Rossum*

Python goals

In 1999, Guido van Rossum defined his goals for Python:

- an **easy and intuitive** language just as powerful as those of the major competitors;
- **open source**, so anyone can contribute to its development;
- code that is as **understandable** as plain English;
- **suitable for everyday tasks**, allowing for short development times.

About 20 years later, it is clear that all these intentions have been fulfilled. Some sources say that Python is the most popular programming language in the world, while others claim it's the second or the third.

Either way, it still occupies a high rank in the top ten of the PYPL PopularitY of Programming Language and the TIOBE Programming Community Index.

Python isn't a young language anymore. It is **mature and trustworthy**. It's not a one-hit wonder. It's a bright star in the programming firmament, and time spent learning Python is a very good investment.

Python Essentials 1: Module 2

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

In this module, you will learn:

- how to write and run simple Python programs;
- what Python literals, operators, and expressions are;
- what variables are and what are the rules that govern them;
- how to perform basic input and output operations.

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

You have already seen a computer program that contains one function invocation. A function invocation is one of many possible kinds of Python **instructions**.

Of course, any complex program usually contains many more instructions than one. The question is: how do you couple more than one instruction into the Python code?

Python's syntax is quite specific in this area. Unlike most programming languages, Python requires that **there cannot be more than one instruction in a line**.

A line can be empty (i.e., it may contain no instruction at all) but it must not contain two, three or more instructions. This is strictly prohibited.

Note: Python makes one exception to this rule - it allows one instruction to spread across more than one line (which may be helpful when your code contains complex constructions).

Let's expand the code a bit, you can see it in the editor. Run it and note what you see in the console.

Your Python console should now look like this:

```
The itsy bitsy spider climbed up the waterspout.
Down came the rain and washed the spider out.
```

output

This is a good opportunity to make some observations:

- the program **invokes the** `print()` **function twice**, and you can see two separate lines in the console - this means that `print()` begins its output from a new line each time it starts its execution; you can change this behavior, but you can also use it to your advantage;
- each `print()` invocation contains a different string, as its argument and the console content reflects it - this means that **the instructions in the code are executed in the same order** in which they have been placed in the source file; no next instruction is executed until the previous one is completed (there are some exceptions to this rule, but you can ignore them for now)

You can see it in the editor window. Run the code.

What happens?

If everything goes right, you should see something like this:

```
The itsy bitsy spider climbed up the waterspout.

Down came the rain and washed the spider out.
```

output

As you can see, the empty `print()` invocation is not as empty as you may have expected - it does output an empty line, or (this interpretation is also correct) its output is just a newline.

This is not the only way to produce a **newline** in the output console. We're now going to show you another way.

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

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

There are two very subtle changes - we've inserted a strange pair of characters inside the rhyme. They look like this: `\n`.

Interestingly, while **you can see two characters, Python sees one.**

The backslash (`\`) has a very special meaning when used inside strings - this is called **the escape character**.

The word *escape* should be understood specifically - it means that the series of characters in the string escapes for the moment (a very short moment) to introduce a special inclusion.

In other words, the backslash doesn't mean anything in itself, but is only a kind of announcement, that the next character after the backslash has a different meaning too.

The letter `n` placed after the backslash comes from the word *newline*.

Both the backslash and the *n* form a special symbol named **a newline character**, which urges the console to start a **new output line**.

Run the code. Your console should now look like this:

```
The itsy bitsy spider
climbed up the waterspout.

Down came the rain
and washed the spider out.
```

output

As you can see, two newlines appear in the nursery rhyme, in the places where the `\n` have been used.

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

This convention has two important consequences:

1. If you want to put just one backslash inside a string, don't forget its escaping nature - you have to double it, e.g., such an invocation will cause an error:

```
print("\")
```

while this one won't:

```
print("\\")
```

2. Not all escape pairs (the backslash coupled with another character) mean something.

Experiment with your code in the editor, run it, and see what happens.

The `print()` function - using multiple arguments

So far we have tested the `print()` function behavior with no arguments, and with one argument. It's also worth trying to feed the print() function with more than one argument.

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

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

There is one `print()` function invocation, but it contains **three arguments**. All of them are strings.

The arguments are **separated by commas**. We've surrounded them with spaces to make them more visible, but it's not really necessary, and we won't be doing it anymore.

In this case, the commas separating the arguments play a completely different role than the comma inside the string. The former is a part of Python's syntax, the latter is intended to be shown in the console.

If you look at the code again, you'll see that there are no spaces inside the strings.

Run the code and see what happens.

The console should now be showing the following text:

```
The itsy bitsy spider climbed up the waterspout.
```

output

The spaces, removed from the strings, have appeared again. Can you explain why?

Two conclusions emerge from this example:

- a `print()` function invoked with more than one argument **outputs them all on one line**;
- the `print()` function **puts a space between the outputted arguments** on its own initiative.

The `print()` function - the positional way of passing the arguments

Now that you know a bit about `print()` function customs, we're going to show you how to change them.

You should be able to predict the output without running the code in the editor.

The way in which we are passing the arguments into the `print()` function is the most common in Python, and is called **the positional way** (this name comes from the fact that the meaning of the argument is dictated by its position, e.g., the second argument will be outputted after the first, not the other way round).

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

The `print()` function - the keyword arguments

Python offers another mechanism for the passing of arguments, which can be helpful when you want to convince the `print()` function to change its behavior a bit.

We aren't going to explain it in depth right now. We plan to do this when we talk about functions. For now, we simply want to show you how it works. Feel free to use it in your own programs.

The mechanism is called **keyword arguments**. The name stems from the fact that the meaning of these arguments is taken not from its location (position) but from the special word (keyword) used to identify them.

The `print()` function has two keyword arguments that you can use for your purposes. The first of them is named `end`.

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

In order to use it, it is necessary to know some rules:

- a keyword argument consists of three elements: a **keyword** identifying the argument (`end` here); an **equal sign** (`=`); and a **value** assigned to that argument;
- any keyword arguments have to be put **after the last positional argument** (this is very important)

In our example, we have made use of the `end` keyword argument, and set it to a string containing one space.

Run the code to see how it works.

The console should now be showing the following text:

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

output

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

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

The `print()` function - the keyword arguments

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.

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

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

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.

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

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

Code

```
1
2
3
```

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

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

Console

```
my_name_is*Monty*Python.*
```

Key takeaways

1. The `print()` function is a **built-in** function. It prints/outputs a specified message to the screen/consol 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.

3. To call a function (this process is known as **function invocation** or **function call**), you need to use the function name followed by parentheses. You can pass arguments into a function by placing them inside the parentheses. You must separate arguments with a comma, e.g., `print("Hello,", "world!")`. An "empty" `print()` function outputs an empty line to the screen.

4. Python strings are delimited with **quotes**, e.g., `"I am a string"` (double quotes), or `'I am a string, too'` (single quotes).

5. Computer programs are collections of **instructions**. An instruction is a command to perform a specific task when executed, e.g., to print a certain message to the screen.

6. In Python strings the **backslash** (`\`) is a special character which announces that the next character has a different meaning, e.g., `\n` (the **newline character**) starts a new output line.

7. **Positional arguments** are the ones whose meaning is dictated by their position, e.g., the second argument is outputted after the first, the third is outputted after the second, etc.

8. **Keyword arguments** are the ones whose meaning is not dictated by their location, but by a special word (keyword) used to identify them.

9. The `end` and `sep` parameters can be used for formatting the output of the `print()` function. The `sep` parameter specifies the separator between the outputted arguments (e.g., `print("H", "E", "L", "L", "O", sep="-")`, whereas the `end` parameter specifies what to print at the end of the print statement.

Literals - the data in itself

Now that you have a little knowledge of some of the powerful features offered by the `print()` function, it's time to learn about some new issues, and one important new term - the **literal**.

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

As this is a difficult concept to understand, a good example may be helpful.

Take a look at the following set of digits:

```
123
```

Can you guess what value it represents? Of course you can - it's *one hundred twenty three*.

But what about this:

```
c
```

Does it represent any value? Maybe. It can be the symbol of the speed of light, for example. It also can be the constant of integration. Or even the length of a hypotenuse in the sense of a Pythagorean theorem. There are many possibilities.

You cannot choose the right one without some additional knowledge.

And this is the clue: `123` is a literal, and `c` is not.

You use literals **to encode data and to put them into your code**. We're now going to show you some conventions you have to obey when using Python.

Literals - the data in itself

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

The first line looks familiar. The second seems to be erroneous due to the visible lack of quotes.

Try to run it.

If everything went okay, you should now see two identical lines.

What happened? What does it mean?

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

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

The `print()` function presents them in exactly the same way - this example is obvious, as their human-readable representation is also the same. Internally, in the computer's memory, these two values are stored in completely different ways - the string exists as just a string - a series of letters.

The number is converted into machine representation (a set of bits). The `print()` function is able to show them both in a form readable to humans.

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

Integers

You may already know a little about how computers perform calculations on numbers. Perhaps you've heard of the **binary system**, and know that it's the system computers use for storing numbers, and that they can perform any operation upon them.

We won't explore the intricacies of positional numeral systems here, but we'll say that the numbers handled by modern computers are of two types:

- **integers**, that is, those which are devoid of the fractional part;
- and **floating-point** numbers (or simply **floats**), that contain (or are able to contain) the fractional part.

This definition is not entirely accurate, but quite sufficient for now. The distinction is very important, and the boundary between these two types of numbers is very strict. Both of these kinds of numbers differ significantly in how they're stored in a computer memory and in the range of acceptable values.

The characteristic of the numeric value which determines its kind, range, and application, is called the **type**.

If you encode a literal and place it inside Python code, the form of the literal determines the representation (type) Python will use to **store it in the memory**.

For now, let's leave the floating-point numbers aside (we'll come back to them soon) and consider the question of how Python recognizes integers.

The process is almost like how you would write them with a pencil on paper - it's simply a string of digits that make up the number. But there's a reservation - you must not interject any characters that are not digits inside the number.

Take, for example, the number eleven million one hundred and eleven thousand one hundred and eleven. If you took a pencil in your hand right now, you would write the number like this: `11,111,111`, or like this: `11.111.111`, or even like this: `11 111 111`.

It's clear that this provision makes it easier to read, especially when the number consists of many digits. However, Python doesn't accept things like these. It's **prohibited**. What Python does allow, though, is the use of **underscores** in numeric literals.*

Therefore, you can write this number either like this: `11111111`, or like that: `11_111_111`.

NOTE *Python 3.6 has introduced underscores in numeric literals, allowing for placing single underscores between digits and after base specifiers for improved readability. This feature is not available in older versions of Python.

And how do we code negative numbers in Python? As usual - by adding a **minus**. You can write: `-11111111`, or `-11_111_111`.

Positive numbers do not need to be preceded by the plus sign, but it's permissible, if you wish to do it. The following lines describe the same number: `+11111111` and `11111111`.

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: 3×10^8.

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

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

```
3E8
```

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

Note:

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

Coding floats

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

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

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

```
6.62607E-34
```

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

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

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

```
0.00000000000000000001
```

When you run this literal through Python:

```
print(0.00000000000000000001)
```

this is the result:

```
1e-22
```

output

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

Strings

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

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

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

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

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

```
I like "Monty Python"
```

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

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

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

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

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

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

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

This example will work too:

```
print('I like "Monty Python"')
```

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

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

Exercise 2

What types of literals are the following four examples?

`"1.5", 2.0, 528, False`

Check

Exercise 3

What is the decimal value of the following binary number?

`1011`

Check

Python as a calculator

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

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

```
print(2+2)
```

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

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

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

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

Basic operators

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

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

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

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

`+`, `-`, `*`, `/`, `//`, `%`, `**`

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

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

Arithmetic operators: 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`

Run the code. What do you see?

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

Module 3

Boolean Values, Conditional Execution, Loops, Lists and List Processing, Logical and Bitwise Operations

In this module, you will cover the following topics:

- the Boolean data type;
- relational operators;
- making decisions in Python (if, if-else, if-elif,else)
- how to repeat code execution using loops (while, for)
- how to perform logic and bitwise operations in Python;
- lists in Python (constructing, indexing, and slicing; content manipulation)
- how to sort a list using bubble-sort algorithms;
- multidimensional lists and their applications.

Questions and answers

A programmer writes a program and **the program asks questions**.

A computer 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 will never get a response like *Let me think....*, *I don't know*, or *Probably yes, but I don't know for sure*.

To ask questions, Python uses a set of very special operators. Let's go through them one after another, illustrating their effects on some simple examples.

Comparison: equality operator

Question: **are two values equal**?

To ask this question, you use the `==` (equal equal) operator.

Don't forget this important distinction:

- `=` is an **assignment operator**, e.g., `a = b` assigns `a` with the value of `b`;
- `==` is the question *are these values equal?*; `a == b` **compares** `a` and `b`.

It is a **binary operator with left-sided binding**. It needs two arguments and **checks if they are equal**.

Exercises

Now let's ask a few questions. Try to guess the answers.

Question #1: What is the result of the following comparison?

`2 == 2` Check

`True` - of course, 2 is equal to 2. Python will answer `True` (remember this pair of predefined literals, `True` and `False` - they're Python keywords, too).

Question #2: What is the result of the following comparison?

`2 == 2.` Check

This question is not as easy as the first one. Luckily, Python is able to convert the integer value into its real equivalent, and consequently, the answer is `True`.

Question #3: What is the result of the following comparison?

`1 == 2` Check

This should be easy. The answer will be (or rather, always is) `False`.

Equality: the *equal to* operator (==)

The `==` (equal to) operator compares the values of two operands. If they are equal, the result of the comparison is `True`. If they are not equal, the result of the comparison is `False`.

Look at the equality comparison below - what is the result of this operation?

```
var == 0
```

Note that we cannot find the answer if we do not know what value is currently stored in the variable `var`.

If the variable has been changed many times during the execution of your program, or its initial value is entered from the console, the answer to this question can be given only by Python and only at runtime.

Now imagine a programmer who suffers from insomnia, and has to count black and white sheep separately as long as there are exactly twice as many black sheep as white ones.

The question will be as follows:

```
black_sheep == 2 * white_sheep
```

Due to the low priority of the `==` operator, the question shall be treated as equivalent to this one:

```
black_sheep == (2 * white_sheep)
```

So, let's practice your understanding of the `==` operator now - can you guess the output of the code below?

```
var = 0  # Assigning 0 to var
print(var == 0)

var = 1  # Assigning 1 to var
print(var == 0)
```

Run the code and check if you were right.

Inequality: the *not equal to* operator (`!=`)

The `!=` (not equal to) operator compares the values of two operands, too. Here is the difference: if they are equal, the result of the comparison is `False`. If they are not equal, the result of the comparison is `True`.

Now take a look at the inequality comparison below - can you guess the result of this operation?

```
var = 0  # Assigning 0 to var
print(var != 0)

var = 1  # Assigning 1 to var
print(var != 0)
```

Run the code and check if you were right.

Comparison operators: greater than

You can also ask a comparison question using the `>` (greater than) operator.

If you want to know if there are more black sheep than white ones, you can write it as follows:

```
black_sheep > white_sheep  # Greater than
```

`True` confirms it; `False` denies it.

Comparison operators: greater than or equal to

The *greater than* operator has another special, **non-strict** variant, but it's denoted differently than in classical arithmetic notation: `>=` (greater than or equal to).

There are two subsequent signs, not one.

Both of these operators (strict and non-strict), as well as the two others discussed in the next section, are **binary operators with left-sided binding**, and their **priority is greater than that shown by `==` and `!=`**.

If we want to find out whether or not we have to wear a warm hat, we ask the following question:

```
centigrade_outside ≥ 0.0  # Greater than or equal to
```

Comparison operators: less than or equal to

As you've probably already guessed, the operators used in this case are: the `<` (less than) operator and its non-strict sibling: `<=` (less than or equal to).

Look at this simple example:

```
current_velocity_mph < 85  # Less than
```

```
current_velocity_mph ≤ 85  # Less than or equal to
```

We're going to check if there's a risk of being fined by the highway police (the first question is strict, the second isn't).

Making use of the answers

What can you do with the answer (i.e., the result of a comparison operation) you get from the computer?

There are at least two possibilities: first, you can memorize it (**store it in a variable**) and make use of it later. How do you do that? Well, you would use an arbitrary variable like this:

```
answer = number_of_lions >= number_of_lionesses
```

The content of the variable will tell you the answer to the question asked.

The second possibility is more convenient and far more common: you can use the answer you get to **make a decision about the future of the program**.

You need a special instruction for this purpose, and we'll discuss it very soon.

Now we need to update our **priority table**, and put all the new operators into it. It now looks as follows:

Priority	Operator	
1	`+`, `-`	unary
2	`**`	
3	`*`, `/`, `//`, `%`	
4	`+`, `-`	binary
5	`<`, `<=`, `>`, `>=`	
6	`==`, `!=`	

Conditions and conditional execution

You already know how to ask Python questions, but you still don't know how to make reasonable use of the answers. You have to have a mechanism which will allow you to do something **if a condition is met, and not do it if it isn't**.

It's just like in real life: you do certain things or you don't when a specific condition is met or not, e.g., you go for a walk if the weather is good, or stay home if it's wet and cold.

To make such decisions, Python offers a special instruction. Due to its nature and its application, it's called a **conditional instruction** (or conditional statement).

There are several variants of it. We'll start with the simplest, increasing the difficulty slowly.

The first form of a conditional statement, which you can see below is written very informally but figuratively:

```
if true_or_not:
    do_this_if_true
```

This conditional statement consists of the following, strictly necessary, elements in this and this order only:

- the `if` keyword;
- one or more white spaces;
- an expression (a question or an answer) whose value will be interpreted solely in terms of `True` (when its value is non-zero) and `False` (when it is equal to zero);
- a **colon** followed by a newline;
- an **indented** instruction or set of instructions (at least one instruction is absolutely required); the **indentation** may be achieved in two ways - by inserting a particular number of spaces (the recommendation is to use **four spaces of indentation**), or by using the *tab* character; note: if there is more than one instruction in the indented part, the indentation should be the same in all lines; even though it may look the same if you use tabs mixed with spaces, it's important to make all indentations **exactly the same** - Python 3 **does not allow mixing spaces and tabs** for indentation.

Conditional execution: the `if` statement

If a certain sleepless Python developer falls asleep when he or she counts 120 sheep, and the sleep-inducing procedure may be implemented as a special function named `sleep_and_dream()`, the whole code takes the following shape:

```
if sheep_counter >= 120: # Evaluate a test expression
    sleep_and_dream() # Execute if test expression is True
```

You can read it as: if `sheep_counter` is greater than or equal to `120`, then fall asleep and dream (i.e., execute the `sleep_and_dream` function.)

We've said that **conditionally executed statements have to be indented**. This creates a very legible structure, clearly demonstrating all possible execution paths in the code.

Take a look at the following code:

```
if sheep_counter >= 120:
    make_a_bed()
    take_a_shower()
    sleep_and_dream()
feed_the_sheepdogs()
```

As you can see, making a bed, taking a shower and falling asleep and dreaming are all **executed conditionally** - when `sheep_counter` reaches the desired limit.

Feeding the sheepdogs, however, is **always done** (i.e., the `feed_the_sheepdogs()` function is not indented and does not belong to the `if` block, which means it is always executed.)

Now we're going to discuss another variant of the conditional statement, which also allows you to perform an additional action when the condition is not met.

Conditional execution: the `if-else` statement

We started out with a simple phrase which read: *If the weather is good, we will go for a walk*.

Note - there is not a word about what will happen if the weather is bad. We only know that we won't go outdoors, but what we could do instead is not known. We may want to plan something in case of bad weather, too.

We can say, for example: *If the weather is good, we will go for a walk, otherwise we will go to a theater*.

Now we know what we'll do **if the conditions are met**, and we know what we'll do **if not everything goes our way**. In other words, we have a "Plan B".

Python allows us to express such alternative plans. This is done with a second, slightly more complex form of the conditional statement, the *if-else* statement:

```
if true_or_false_condition:
    perform_if_condition_true
else:
    perform_if_condition_false
```

Thus, there is a new word: `else` - this is a **keyword**.

The part of the code which begins with `else` says what to do if the condition specified for the `if` is not met (note the **colon** after the word).

The *if-else* execution goes as follows:

- if the condition evaluates to **True** (its value is not equal to zero),
  the `perform_if_condition_true` statement is executed, and the conditional statement comes to an end;
- if the condition evaluates to **False** (it is equal to zero),
  the `perform_if_condition_false` statement is executed, and the conditional statement comes to an end.

The `if-else` statement: more conditional execution

By using this form of conditional statement, we can describe our plans as follows:

```
if the_weather_is_good:
    go_for_a_walk()
else:
    go_to_a_theater()
have_lunch()
```

If the weather is good, we'll go for a walk. Otherwise, we'll go to a theatre. No matter if the weather is good or bad, we'll have lunch afterwards (after the walk or after going to the theatre).

Everything we've said about indentation works in the same manner inside **the *else* branch**:

```
if the_weather_is_good:
    go_for_a_walk()
    have_fun()
else:
    go_to_a_theater()
    enjoy_the_movie()
have_lunch()
```

Nested `if-else` statements

Now let's discuss two special cases of the conditional statement.

First, consider the case where the **instruction placed after the `if` is another `if`**.

Read what we have planned for this Sunday. If the weather is fine, we'll go for a walk. If we find a nice restaurant, we'll have lunch there. Otherwise, we'll eat a sandwich. If the weather is poor, we'll go to the theater. If there are no tickets, we'll go shopping in the nearest mall.

Let's write the same in Python. Consider carefully the code here:

```
if the_weather_is_good:
    if nice_restaurant_is_found:
        have_lunch()
    else:
        eat_a_sandwich()
else:
    if tickets_are_available:
        go_to_the_theater()
    else:
        go_shopping()
```

Here are two important points:

- this use of the `if` statement is known as **nesting**; remember that every `else` refers to the `if` which lies **at the same indentation level**; you need to know this to determine how the *if*s and *else*s pair up;
- consider how the **indentation improves readability**, and makes the code easier to understand and trace.

The `elif` statement

The second special case introduces another new Python keyword: **elif**. As you probably suspect, it's a shorter form of **else if**.

`elif` is used to **check more than just one condition**, and to **stop** when the first statement which is true is found.

Our next example resembles nesting, but the similarities are very slight. Again, we'll change our plans and express them as follows: If the weather is fine, we'll go for a walk, otherwise if we get tickets, we'll go to the theater, otherwise if there are free tables at the restaurant, we'll go for lunch; if all else fails, we'll return home and play chess.

Have you noticed how many times we've used the word *otherwise*? This is the stage where the `elif` keyword plays its role.

Let's write the same scenario using Python:

```
if the_weather_is_good:
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_theater()
elif table_is_available:
    go_for_lunch()
else:
    play_chess_at_home()
```

The way to assemble subsequent *if-elif-else* statements is sometimes called a **cascade**.

Notice again how the indentation improves the readability of the code.

Some additional attention has to be paid in this case:

- you **mustn't use** `else` **without a preceding** `if`;
- `else` is always the **last branch of the cascade**, regardless of whether you've used `elif` or not;
- `else` is an **optional** part of the cascade, and may be omitted;
- if there is an else branch in the cascade, only one of all the branches is executed;
- if there is no else branch, it's possible that none of the available branches is executed.

This may sound a little puzzling, but hopefully some simple examples will help shed more light.

Analyzing code samples

Now we're going to show you some simple yet complete programs. We won't explain them in detail, because we consider the comments (and the variable names) inside the code to be sufficient guides.

All the programs solve the same problem - they **find the largest of several numbers and print it out**.

Example 1:

We'll start with the simplest case - **how to identify the larger of two numbers**:

```
# Read two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))

# Choose the larger number
if number1 > number2:
    larger_number = number1
else:
    larger_number = number2

# Print the result
print("The larger number is:", larger_number)
```

The above snippet should be clear - it reads two integer values, compares them, and finds which is the larger.

Example 2:

Now we're going to show you one intriguing fact. Python has an interesting feature, look at the code below:

```
# Read two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))

# Choose the larger number
```

```
if number1 > number2: larger_number = number1
else: larger_number = number2

# Print the result
print("The larger number is:", larger_number)
```

Note: if any of the *if-elif-else* branches contains just one instruction, you may code it in a more comprehensive form (you don't need to make an indented line after the keyword, but just continue the line after the colon).

This style, however, may be misleading, and we're not going to use it in our future programs, but it's definitely worth knowing if you want to read and understand someone else's programs.

There are no other differences in the code.

Example 3:

It's time to complicate the code - let's find the largest of three numbers. Will it enlarge the code? A bit.

We assume that the first value is the largest. Then we verify this hypothesis with the two remaining values.

Look at the code below:

```
# Read three numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))

# We temporarily assume that the first number
# is the largest one.
# We will verify this soon.
largest_number = number1
```

```
# We check if the second number is larger than current largest_number
# and update largest_number if needed.
if number2 > largest_number:
    largest_number = number2

# We check if the third number is larger than current largest_number
# and update largest_number if needed.
if number3 > largest_number:
    largest_number = number3

# Print the result
print("The largest number is:", largest_number)
```

This method is significantly simpler than trying to find the largest number all at once, by comparing all possible pairs of numbers (i.e., first with second, second with third, third with first). Try to rebuild the code for yourself.

Pseudocode and introduction to loops

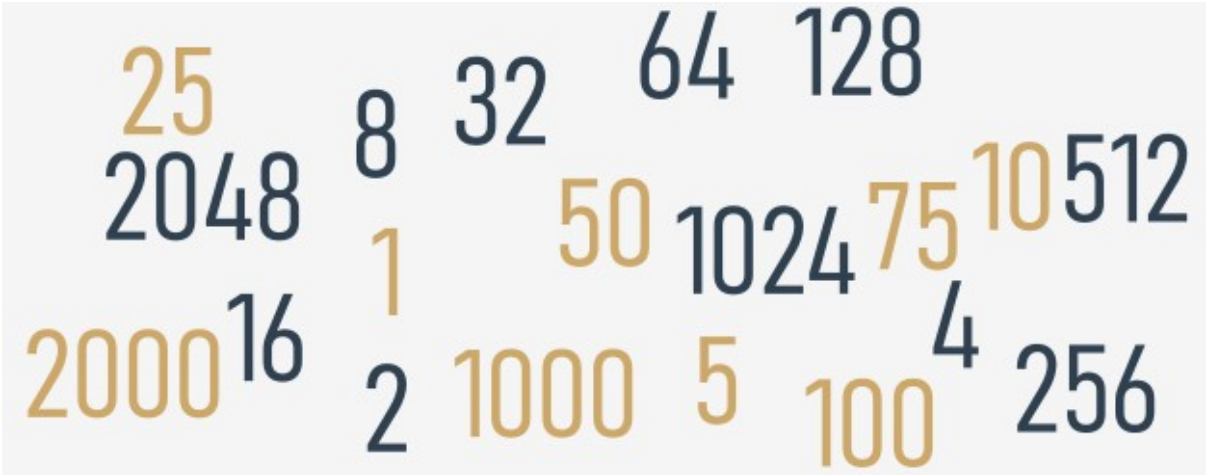
You should now be able to write a program which finds the largest of four, five, six, or even ten numbers.

You already know the scheme, so extending the size of the problem will not be particularly complex.

But what happens if we ask you to write a program that finds the largest of two hundred numbers? Can you imagine the code?

You'll need two hundred variables. If two hundred variables isn't bad enough, try to imagine searching for the largest of a million numbers.

Imagine a code that contains 199 conditional statements and two hundred invocations of the `input()` function. Luckily, you don't need to deal with that. There's a simpler approach.



We'll ignore the requirements of Python syntax for now, and try to analyze the problem without thinking about the real programming. In other words, we'll try to write the **algorithm**, and when we're happy with it, we'll implement it.

In this case, we'll use a kind of notation which is not an actual programming language (it can be neither compiled nor executed), but it is formalized, concise and readable. It's called **pseudocode**.

Let's look at our pseudocode below:

```
largest_number = -999999999
number = int(input())
if number == -1:
    print(largest_number)
    exit()
if number > largest_number:
    largest_number = number
# Go to line 02
```

What's happening in it?

Firstly, we can simplify the program if, at the very beginning of the code, we assign the variable `largest_number` with a value which will be smaller than any of the entered numbers. We'll use `-999999999` for that purpose.

Secondly, we assume that our algorithm will not know in advance how many numbers will be delivered to the program. We expect that the user will enter as many numbers as she/he wants - the algorithm will work well with one hundred and with one thousand numbers. How do we do that?

We make a deal with the user: when the value `-1` is entered, it will be a sign that there are no more data and the program should end its work.

Otherwise, if the entered value is not equal to `-1`, the program will read another number, and so on.

The trick is based on the assumption that any part of the code can be performed more than once - precisely, as many times as needed.

Performing a certain part of the code more than once is called a **loop**. The meaning of this term is probably obvious to you.

Lines `02` through `08` make a loop. We'll **pass through them as many times as needed** to review all the entered values.

Can you use a similar structure in a program written in Python? Yes, you can.

Extra Info

Python often comes with a lot of built-in functions that will do the work for you. For example, to find the largest number of all, you can use a Python built-in function called `max()`. You can use it with multiple arguments. Analyze the code below:

```
# Read three numbers.
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))

# Check which one of the numbers is the greatest
# and pass it to the largest_number variable.

largest_number = max(number1, number2, number3)

# Print the result.
```

```
print("The largest number is:", largest_number)
```

By the same fashion, you can use the `min()` function to return the lowest number. You can rebuild the above code and experiment with it in the Sandbox.

We're going to talk about these (and many other) functions soon. For the time being, our focus will be put on conditional execution and loops to let you gain more confidence in programming and teach you the skills that will let you fully understand and apply the two concepts in your code. So, for now, we're not taking any shortcuts.

Key takeaways

1. The **comparison** (or the so-called *relational*) operators are used to compare values. The table below illustrates how the comparison operators work, assuming that `x = 0`, `y = 1`, and `z = 0`:

Operator	Description	Example
`==`	returns if operands' values are equal, and `False` otherwise	`x == y  # False` `x == z  # True`
`!=`	returns `True` if operands' values are not equal, and `False` otherwise	`x != y  # True` `x != z  # False`
`>`	`True` if the left operand's value is greater than the right operand's value, and `False` otherwise	`x > y  # False` `y > z  # True`
`<`	`True` if the left operand's value is less than the right operand's value, and `False` otherwise	`x < y  # True` `y < z  # False`

`≥`	`True` if the left operand's value is greater than or equal to the right operand's value, and `False` otherwise	`x >= y  # False`
		`x >= z  # True`
		`y >= z  # True`

`≤`	`True` if the left operand's value is less than or equal to the right operand's value, and `False` otherwise	`x <= y  # True`
		`x <= z  # True`
		`y <= z  #`

```
False
```

2. When you want to execute some code only if a certain condition is met, you can use a **conditional statement**:

- a single `if` statement, e.g.:

```
x = 10

if x == 10: # condition
    print("x is equal to 10")  # Executed if the condition is True.
```

- a series of `if` statements, e.g.:

```
x = 10

if x > 5: # condition one
    print("x is greater than 5")  # Executed if condition one is True.

if x < 10: # condition two
    print("x is less than 10")  # Executed if condition two is True.

if x == 10: # condition three
    print("x is equal to 10")  # Executed if condition three is True.
```

Each `if` statement is tested separately.

- an `if-else` statement, e.g.:

```
x = 10

if x < 10:  # Condition
    print("x is less than 10")  # Executed if the condition is True.
```

```
else:
    print("x is greater than or equal to 10")  # Executed if the
condition is False.
```

- a series of `if` statements followed by an `else`, e.g.:

```
x = 10

if x > 5:  # True
    print("x > 5")

if x > 8:  # True
    print("x > 8")

if x > 10:  # False
    print("x > 10")

else:
    print("else will be executed")
```

Each `if` is tested separately. The body of `else` is executed if the last `if` is `False`.

- The `if-elif-else` statement, e.g.:

```
x = 10

if x == 10:  # True
    print("x == 10")

if x > 15:  # False
    print("x > 15")

elif x > 10:  # False
    print("x > 10")

elif x > 5:  # True
    print("x > 5")

else:
    print("else will not be executed")
```

If the condition for `if` is `False`, the program checks the conditions of the subsequent `elif` blocks - the first `elif` block that is `True` is executed. If all the conditions are `False`, the `else` block will be executed.

- Nested conditional statements, e.g.:

```
x = 10

if x > 5:  # True
    if x == 6:  # False
        print("nested: x == 6")
    elif x == 10:  # True
        print("nested: x == 10")
    else:
        print("nested: else")
else:
    print("else")
```

Key takeaways

1. The **comparison** (or the so-called *relational*) operators are used to compare values. The table below illustrates how the comparison operators work, assuming that `x = 0`, `y = 1`, and `z = 0`:

Operator	Description	Example
`==`	returns if operands' values are equal, and `False` otherwise	`x == y  # False` `x == z  # True`
`!=`	returns `True` if operands' values are not equal, and `False` otherwise	`x != y  # True` `x != z  # False`
`>`	`True` if the left operand's value is greater than the right operand's value, and `False` otherwise	`x > y  # False` `y > z  # True`
`<`	`True` if the left operand's value is less than the right operand's value, and `False` otherwise	`x < y  # True` `y < z  # False`
`≥`	`True` if the left operand's value is greater than or equal to the right operand's value,	`x >= y  # False` `x >= z  # True`

	and `False` otherwise	`y >= z  # True`

≤	`True` if the left operand's value is less than or equal to the right operand's value, and `False` otherwise	`x <= y  # True`
		`x <= z  # True`
		`y <= z  # False`

2. When you want to execute some code only if a certain condition is met, you can use a **conditional statement**:

- a single `if` statement, e.g.:

```
x = 10

if x == 10: # condition
    print("x is equal to 10")  # Executed if the condition is True.
```

- a series of `if` statements, e.g.:

```
x = 10

if x > 5: # condition one
    print("x is greater than 5")  # Executed if condition one is True.

if x < 10: # condition two
    print("x is less than 10")  # Executed if condition two is True.

if x == 10: # condition three
    print("x is equal to 10")  # Executed if condition three is True.
```

Each `if` statement is tested separately.

- an `if-else` statement, e.g.:

```
x = 10

if x < 10:  # Condition
    print("x is less than 10")  # Executed if the condition is True.

else:
    print("x is greater than or equal to 10")  # Executed if the condition is False.
```

- a series of `if` statements followed by an `else`, e.g.:

```
x = 10

if x > 5:  # True
    print("x > 5")

if x > 8:  # True
    print("x > 8")

if x > 10:  # False
    print("x > 10")

else:
    print("else will be executed")
```

Each `if` is tested separately. The body of `else` is executed if the last `if` is `False`.

- The `if-elif-else` statement, e.g.:

```
x = 10

if x == 10:  # True
    print("x == 10")

if x > 15:  # False
    print("x > 15")

elif x > 10:  # False
    print("x > 10")

elif x > 5:  # True
    print("x > 5")
```

```
else:
    print("else will not be executed")
```

If the condition for `if` is `False`, the program checks the conditions of the subsequent `elif` blocks - the first `elif` block that is `True` is executed. If all the conditions are `False`, the `else` block will be executed.

- Nested conditional statements, e.g.:

```
x = 10

if x > 5:  # True
    if x == 6:  # False
        print("nested: x == 6")
    elif x == 10:  # True
        print("nested: x == 10")
    else:
        print("nested: else")
else:
    print("else")
```

Key takeaways: continued

Exercise 1

What is the output of the following snippet?

```
x = 5

y = 10

z = 8


print(x > y)

print(y > z)
```

Check

Exercise 2

What is the output of the following snippet?

```
x, y, z = 5, 10, 8


print(x > z)
print((y - 5) == x)
```

Check

Exercise 3

What is the output of the following snippet?

```
x, y, z = 5, 10, 8
x, y, z = z, y, x


print(x > z)
print((y - 5) == x)
```

Check

Exercise 4

What is the output of the following snippet?

```
x = 10


if x == 10:
    print(x == 10)
if x > 5:
    print(x > 5)
if x < 10:
    print(x < 10)
```

```
else:
    print("else")
```

Check

Exercise 5

What is the output of the following snippet?

```
x = "1"

if x == 1:
    print("one")
elif x == "1":
    if int(x) > 1:
        print("two")
    elif int(x) < 1:
        print("three")
    else:
        print("four")
if int(x) == 1:
    print("five")
else:
    print("six")
```

Check

Exercise 6

What is the output of the following snippet?

```
x = 1
y = 1.0
z = "1"


if x == y:
    print("one")
if y == int(z):
    print("two")
elif x == y:
    print("three")
else:
    print("four")
```

Check

Looping your code with `while`

Do you agree with the statement presented below?

```
while there is something to do
    do it
```

Note that this record also declares that if there is nothing to do, nothing at all will happen.

In general, in Python, a loop can be represented as follows:

```
while conditional_expression:
    instruction
```

If you notice some similarities to the *if* instruction, that's quite all right. Indeed, the syntactic difference is only one: you use the word `while` instead of the word `if`.

The semantic difference is more important: when the condition is met, *if* performs its statements **only once**; *while* **repeats the execution as long as the condition evaluates to** `True`.

Note: all the rules regarding **indentation** are applicable here, too. We'll show you this soon.

Look at the algorithm below:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    :
    instruction_n
```

It is now important to remember that:

- if you want to execute **more than one statement inside one** `while`, you must (as with `if`) **indent** all the instructions in the same way;
- an instruction or set of instructions executed inside the `while` loop is called the **loop's body**;
- if the condition is `False` (equal to zero) as early as when it is tested for the first time, the body is not executed even once (note the analogy of not having to do anything if there is nothing to do);
- the body should be able to change the condition's value, because if the condition is `True` at the beginning, the body might run continuously to infinity - notice that doing a thing usually decreases the number of things to do).

An infinite loop

An infinite loop, also called an **endless loop**, is a sequence of instructions in a program which repeat indefinitely (loop endlessly.)

Here's an example of a loop that is not able to finish its execution:

```
while True:
    print("I'm stuck inside a loop.")
```

This loop will infinitely print `"I'm stuck inside a loop."` on the screen.

NOTE

If you want to get the best learning experience from seeing how an infinite loop behaves, launch IDLE, create a New File, copy-paste the above code, save your file, and run the program. What you will see is the never-ending sequence of `"I'm stuck inside a loop."` strings printed to the Python console window. To terminate your program, just press *Ctrl-C* (or *Ctrl-Break* on some computers). This will cause the so-called `KeyboardInterrupt` exception and let your program get out of the loop. We'll talk about it later in the course.

Let's go back to the sketch of the algorithm we showed you recently. We're going to show you how to use this newly learned loop to find the largest number from a large set of entered data.

Analyze the program carefully. See where the loop starts (line 8). Locate the loop's body and find out **how the body is exited**:

```
# Store the current largest number here.
largest_number = -999999999
```

```
# Input the first value.
number = int(input("Enter a number or type -1 to stop: "))

# If the number is not equal to -1, continue.
while number != -1:
    # Is number larger than largest_number?
    if number > largest_number:
        # Yes, update largest_number.
        largest_number = number
    # Input the next number.
    number = int(input("Enter a number or type -1 to stop: "))

# Print the largest number.
print("The largest number is:", largest_number)
```

Check how this code implements the algorithm we showed you earlier.

Looping your code with `while`

Do you agree with the statement presented below?

```
while there is something to do
    do it
```

Note that this record also declares that if there is nothing to do, nothing at all will happen.

In general, in Python, a loop can be represented as follows:

```
while conditional_expression:
    instruction
```

If you notice some similarities to the *if* instruction, that's quite all right. Indeed, the syntactic difference is only one: you use the word `while` instead of the word `if`.

The semantic difference is more important: when the condition is met, *if* performs its statements **only once**; *while* **repeats the execution as long as the condition evaluates to** `True`.

Note: all the rules regarding **indentation** are applicable here, too. We'll show you this soon.

Look at the algorithm below:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    :
    instruction_n
```

It is now important to remember that:

- if you want to execute **more than one statement inside one** `while`, you must (as with `if`) **indent** all the instructions in the same way;
- an instruction or set of instructions executed inside the `while` loop is called the **loop's body**;
- if the condition is `False` (equal to zero) as early as when it is tested for the first time, the body is not executed even once (note the analogy of not having to do anything if there is nothing to do);
- the body should be able to change the condition's value, because if the condition is `True` at the beginning, the body might run continuously to infinity - notice that doing a thing usually decreases the number of things to do).

An infinite loop

An infinite loop, also called an **endless loop**, is a sequence of instructions in a program which repeat indefinitely (loop endlessly.)

Here's an example of a loop that is not able to finish its execution:

```
while True:
    print("I'm stuck inside a loop.")
```

This loop will infinitely print `"I'm stuck inside a loop."` on the screen.

NOTE

If you want to get the best learning experience from seeing how an infinite loop behaves, launch IDLE, create a New File, copy-paste the above code, save your file, and run the program. What you will see is the never-ending sequence of `"I'm stuck inside a loop."` strings printed to the Python console window. To terminate your program, just press *Ctrl-C* (or *Ctrl-Break* on some computers). This will cause the so-called `KeyboardInterrupt` exception and let your program get out of the loop. We'll talk about it later in the course.

Let's go back to the sketch of the algorithm we showed you recently. We're going to show you how to use this newly learned loop to find the largest number from a large set of entered data.

Analyze the program carefully. See where the loop starts (line 8). Locate the loop's body and find out **how the body is exited**:

```
# Store the current largest number here.
largest_number = -999999999

# Input the first value.
number = int(input("Enter a number or type -1 to stop: "))

# If the number is not equal to -1, continue.
while number != -1:
    # Is number larger than largest_number?
    if number > largest_number:
        # Yes, update largest_number.
        largest_number = number
    # Input the next number.
    number = int(input("Enter a number or type -1 to stop: "))
```

```
# Print the largest number.
print("The largest number is:", largest_number)
```

Check how this code implements the algorithm we showed you earlier.

The `while` loop: more examples

Let's look at another example employing the `while` loop. Follow the comments to find out the idea and the solution.

```
# A program that reads a sequence of numbers
# and counts how many numbers are even and how many are odd.
# The program terminates when zero is entered.

odd_numbers = 0
even_numbers = 0

# Read the first number.
number = int(input("Enter a number or type 0 to stop: "))

# 0 terminates execution.
while number != 0:
    # Check if the number is odd.
    if number % 2 == 1:
        # Increase the odd_numbers counter.
        odd_numbers += 1
    else:
        # Increase the even_numbers counter.
        even_numbers += 1
```

```
    # Read the next number.
    number = int(input("Enter a number or type 0 to stop: "))

# Print results.
print("Odd numbers count:", odd_numbers)
print("Even numbers count:", even_numbers)
```

Certain expressions can be simplified without changing the program's behavior.

Try to recall how Python interprets the truth of a condition, and note that these two forms are equivalent:

`while number != 0:` and `while number:`.

The condition that checks if a number is odd can be coded in these equivalent forms, too:

`if number % 2 == 1:` and `if number % 2:`.

Using a `counter` variable to exit a loop

Look at the snippet below:

```
counter = 5
while counter != 0:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

This code is intended to print the string `"Inside the loop."` and the value stored in the `counter` variable during a given loop exactly five times. Once the condition has not

been met (the `counter` variable has reached `0`), the loop is exited, and the message `"Outside the loop."` as well as the value stored in `counter` is printed.

But there's one thing that can be written more compactly - the condition of the `while` loop.

Can you see the difference?

```
counter = 5
while counter:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

Is it more compact than previously? A bit. Is it more legible? That's disputable.

REMEMBER

Don't feel obliged to code your programs in a way that is always the shortest and the most compact. Readability may be a more important factor. Keep your code ready for a new programmer.

Objectives

Familiarize the student with:

- using the `while` loop;
- reflecting real-life situations in computer code.

Scenario

A junior magician has picked a secret number. He has hidden it in a variable named `secret_number`. He wants everyone who run his program to play the *Guess the secret number* game, and guess what number he has picked for them. Those who don't guess the number will be stuck in an endless loop forever! Unfortunately, he does not know how to complete the code.

Your task is to help the magician complete the code in the editor in such a way so that the code:

- will ask the user to enter an integer number;
- will use a `while` loop;

- will check whether the number entered by the user is the same as the number picked by the magician. If the number chosen by the user is different than the magician's secret number, the user should see the message `"Ha ha! You're stuck in my loop!"` and be prompted to enter a number again. If the number entered by the user matches the number picked by the magician, the number should be printed to the screen, and the magician should say the following words: `"Well done, muggle! You are free now."`

The magician is counting on you! Don't disappoint him.

EXTRA INFO

By the way, look at the `print()` function. The way we've used it here is called *multi-line printing*. You can use **triple quotes** to print strings on multiple lines in order to make text easier to read, or create a special text-based design. Experiment with it.

Looping your code with `for`

Another kind of loop available in Python comes from the observation that sometimes it's more important to **count the "turns" of the loop** than to check the conditions.

Imagine that a loop's body needs to be executed exactly one hundred times. If you would like to use the `while` loop to do it, it may look like this:

```
i = 0
while i < 100:
    # do_something()
    i += 1
```

It would be nice if somebody could do this boring counting for you. Is that possible?

Of course it is - there's a special loop for these kinds of tasks, and it is named `for`.

Actually, the `for` loop is designed to do more complicated tasks - **it can "browse" large collections of data item by item**. We'll show you how to do that soon, but right now we're going to present a simpler variant of its application.

Take a look at the snippet:

```
for i in range(100):
    # do_something()
```

```
    pass
```

There are some new elements. Let us tell you about them:

- the *for* keyword opens the `for` loop; note - there's no condition after it; you don't have to think about conditions, as they're checked internally, without any intervention;
- any variable after the *for* keyword is the **control variable** of the loop; it counts the loop's turns, and does it automatically;
- the *in* keyword introduces a syntax element describing the range of possible values being assigned to the control variable;
- the `range()` function (this is a very special function) is responsible for generating all the desired values of the control variable; in our example, the function will create (we can even say that it will **feed** the loop with) subsequent values from the following set: 0, 1, 2 .. 97, 98, 99; note: in this case, the `range()` function starts its job from 0 and finishes it one step (one integer number) before the value of its argument;
- note the *pass* keyword inside the loop body - it does nothing at all; it's an **empty instruction** - we put it here because the `for` loop's syntax demands at least one instruction inside the body (by the way - `if`, `elif`, `else` and `while` express the same thing)

Our next examples will be a bit more modest in the number of loop repetitions.

More about the `for` loop and the `range()` function with three arguments

The `range()` function may also accept **three arguments** - take a look at the code in the editor.

The third argument is an **increment** - it's a value added to control the variable at every loop turn (as you may suspect, the **default value of the increment is 1**).

Can you tell us how many lines will appear in the console and what values they will contain?

Run the program to find out if you were right.

You should be able to see the following lines in the console window:

```
The value of i is currently 2
```

```
The value of i is currently 5
```

output

Do you know why? The first argument passed to the `range()` function tells us what the **starting** number of the sequence is (hence `2` in the output). The second argument tells the function where to **stop** the sequence (the function generates numbers up to the number indicated by the second argument, but does not include it). Finally, the third argument indicates the **step**, which actually means the difference between each number in the sequence of numbers generated by the function.

`2` (starting number) → `5` (`2` increment by 3 equals `5` - the number is within the range from 2 to 8) → `8` (`5` increment by 3 equals `8` - the number is not within the range from 2 to 8, because the stop parameter is not included in the sequence of numbers generated by the function.)

Note: if the set generated by the `range()` function is empty, the loop won't execute its body at all.

Just like here - there will be no output:

```
for i in range(1, 1):
    print("The value of i is currently", i)
```

Note: the set generated by the `range()` has to be sorted in **ascending order**. There's no way to force the `range()` to create a set in a different form when the `range()` function accepts exactly two arguments. This means that the `range()`'s second argument must be greater than the first.

Thus, there will be no output here, either:

```
for i in range(2, 1):
    print("The value of i is currently", i)
```

Let's have a look at a short program whose task is to write some of the first powers of two:

```
power = 1

for expo in range(16):

    print("2 to the power of", expo, "is", power)

    power *= 2
```

The `expo` variable is used as a control variable for the loop, and indicates the current value of the *exponent*. The exponentiation itself is replaced by multiplying by two. Since 2^0 is equal to 1, then 2×1 is equal to 2^1, 2×2^1 is equal to 2^2, and so on. What is the greatest exponent for which our program still prints the result?

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

Objectives

Familiarize the student with:

- using the `for` loop;
- reflecting real-life situations in computer code.

Scenario

Do you know what Mississippi is? Well, it's the name of one of the states and rivers in the United States. The Mississippi River is about 2,340 miles long, which makes it the second longest river in the United States (the longest being the Missouri River). It's so long that a single drop of water needs 90 days to travel its entire length!

The word *Mississippi* is also used for a slightly different purpose: to *count mississippily*.

If you're not familiar with the phrase, we're here to explain to you what it means: it's used to count seconds.

The idea behind it is that adding the word *Mississippi* to a number when counting seconds aloud makes them sound closer to clock-time, and therefore "one Mississippi, two Mississippi, three Mississippi" will take approximately an actual three seconds of time! It's often used by children playing hide-and-seek to make sure the seeker does an honest count.

Your task is very simple here: write a program that uses a `for` loop to "count mississippily" to five. Having counted to five, the program should print to the screen the final message `"Ready or not, here I come!"`

Use the skeleton we've provided in the editor.

EXTRA INFO

Note that the code in the editor contains two elements which may not be fully clear to you at this moment: the `import time` statement, and the `sleep()` method. We're going to talk about them soon.

For the time being, we'd just like you to know that we've imported the `time` module and used the `sleep()` method to suspend the execution of each subsequent `print()` function inside the `for` loop for one second, so that the message outputted to the console resembles an actual counting. Don't worry - you'll soon learn more about modules and methods.

Expected output

```
1 Mississippi
```

```
2 Mississippi
```

```
3 Mississippi
```

```
4 Mississippi
```

```
5 Mississippi
```

The `break` and `continue` statements

So far, we've treated the body of the loop as an indivisible and inseparable sequence of instructions that are performed completely at every turn of the loop. However, as developer, you could be faced with the following choices:

- it appears that it's unnecessary to continue the loop as a whole; you should refrain from further execution of the loop's body and go further;
- it appears that you need to start the next turn of the loop without completing the execution of the current turn.

Python provides two special instructions for the implementation of both these tasks. Let's say for the sake of accuracy that their existence in the language is not necessary - an experienced programmer is able to code any algorithm without these instructions. Such additions, which don't improve the language's expressive power, but only simplify the developer's work, are sometimes called **syntactic candy**, or syntactic sugar.

These two instructions are:

- `break` - exits the loop immediately, and unconditionally ends the loop's operation; the program begins to execute the nearest instruction after the loop's body;
- `continue` - behaves as if the program has suddenly reached the end of the body; the next turn is started and the condition expression is tested immediately.

Both these words are **keywords**.

Now we'll show you two simple examples to illustrate how the two instructions work. Look at the code in the editor. Run the program and analyze the output. Modify the code and experiment.

```
# break - example


print("The break instruction:")

for i in range(1, 6):

    if i == 3:

        break

    print("Inside the loop.", i)

print("Outside the loop.")




# continue - example


print("\nThe continue instruction:")

for i in range(1, 6):

    if i == 3:

        continue

    print("Inside the loop.", i)

print("Outside the loop.")
```

```
The break instruction:
Inside the loop. 1
Inside the loop. 2
Outside the loop.

The continue instruction:
Inside the loop. 1
Inside the loop. 2
Inside the loop. 4
Inside the loop. 5
Outside the loop.
```

The `break` and `continue` statements: more examples

Let's return to our program that recognizes the largest among the entered numbers. We'll convert it twice, using the `break` and `continue` instructions.

Analyze the code, and judge whether and how you would use either of them.

The `break` variant goes here:

```
largest_number = -99999999
counter = 0

while True:
    number = int(input("Enter a number or type -1 to end program: "))
    if number == -1:
        break
    counter += 1
    if number > largest_number:
        largest_number = number

if counter != 0:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

Run it, test it, and experiment with it.

And now the `continue` variant:

```
largest_number = -99999999
counter = 0
```

```
number = int(input("Enter a number or type -1 to end program: "))

while number != -1:
    if number == -1:
        continue
    counter += 1

    if number > largest_number:
        largest_number = number
    number = int(input("Enter a number or type -1 to end program: "))

if counter:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

Look carefully, the user enters the first number **before** the program enters the `while` loop. The subsequent number is entered when the program is **already in the loop**.

Again - run the program, test it, and experiment with it.

Objectives

Familiarize the student with:

- using the `break` statement in loops;
- reflecting real-life situations in computer code.

Scenario

The `break` statement is used to exit/terminate a loop.

Design a program that uses a `while` loop and continuously asks the user to enter a word unless the user enters `"chupacabra"` as the secret exit word, in which case the

message `"You've successfully left the loop."` should be printed to the screen, and the loop should terminate.

Don't print any of the words entered by the user. Use the concept of conditional execution and the `break` statement.

Objectives

Familiarize the student with:

- using the `continue` statement in loops;
- reflecting real-life situations in computer code.

Scenario

The `continue` statement is used to skip the current block and move ahead to the next iteration, without executing the statements inside the loop.

It can be used with both the `while` and `for` loops.

Your task here is very special: you must design a vowel eater! Write a program that uses:

- a `for` loop;
- the concept of conditional execution (*if-elif-else*)
- the `continue` statement.

Your program must:

- ask the user to enter a word;
- use `user_word = user_word.upper()` to convert the word entered by the user to upper case; we'll talk about the so-called **string methods** and the `upper()` method very soon - don't worry;
- use conditional execution and the `continue` statement to "eat" the following vowels *A*, *E*, *I*, *O*, *U* from the inputted word;
- print the uneaten letters to the screen, each one of them on a separate line.

Test your program with the data we've provided for you.

Test data

Sample input: `Gregory`

Expected output:

`G`

`R`

`G`

`R`

`Y`

Sample input: `abstemious`

Expected output:

`B`

`S`

`T`

`M`

`S`

Sample input: `IOUEA`

Expected output:

Objectives

Familiarize the student with:

- using the `continue` statement in loops;
- modifying and upgrading the existing code;
- reflecting real-life situations in computer code.

Scenario

Your task here is even more special than before: you must redesign the (ugly) vowel eater from the previous lab (3.1.2.10) and create a better, upgraded (pretty) vowel eater! Write a program that uses:

- a `for` loop;
- the concept of conditional execution (*if-elif-else*)
- the `continue` statement.

Your program must:

- ask the user to enter a word;
- use `user_word = user_word.upper()` to convert the word entered by the user to upper case; we'll talk about the so-called **string methods** and the `upper()` method very soon - don't worry;
- use conditional execution and the `continue` statement to "eat" the following vowels *A*, *E*, *I*, *O*, *U* from the inputted word;
- assign the uneaten letters to the `word_without_vowels` variable and print the variable to the screen.

Look at the code in the editor. We've created `word_without_vowels` and assigned an empty string to it. Use concatenation operation to ask Python to combine selected letters into a longer string during subsequent loop turns, and assign it to the `word_without_vowels` variable.

Test your program with the data we've provided for you.

Test data

Sample input: `Gregory`

Expected output:

`GRGRY`

Sample input: `abstemious`

Expected output:

`BSTMS`

Sample input: `IOUEA`

Expected output:

The `while` loop and the `else` branch

Both loops, `while` and `for`, have one interesting (and rarely used) feature.

We'll show you how it works - try to judge for yourself if it's usable and whether you can live without it or not.

In other words, try to convince yourself if the feature is valuable and useful, or is just syntactic sugar.

Take a look at the snippet in the editor. There's something strange at the end - the `else` keyword.

As you may have suspected, **loops may have the `else` branch too, like `if`s**.

The loop's `else` branch is **always executed once, regardless of whether the loop has entered its body or not**.

Can you guess the output? Run the program to check if you were right.

Modify the snippet a bit so that the loop has no chance to execute its body even once:

```
i = 5
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)
```

The `while`'s condition is `False` at the beginning - can you see it?

Run and test the program, and check whether the `else` branch has been executed or not.

The `for` loop and the `else` branch

`for` loops behave a bit differently - take a look at the snippet in the editor and run it.

The output may be a bit surprising.

The `i` variable retains its last value.

Modify the code a bit to carry out one more experiment.

```
i = 111
```

```
for i in range(2, 1):
    print(i)
else:
    print("else:", i)
```

Can you guess the output?

The loop's body won't be executed here at all. Note: we've assigned the `i` variable before the loop.

Run the program and check its output.

When the loop's body isn't executed, the control variable retains the value it had before the loop.

Note: **if the control variable doesn't exist before the loop starts, it won't exist when the execution reaches the `else` branch**.

How do you feel about this variant of `else`?

Now we're going to tell you about some other kinds of variables. Our current variables can only **store one value at a time**, but there are variables that can do much more - they can **store as many values as you want**.

Objectives

Familiarize the student with:

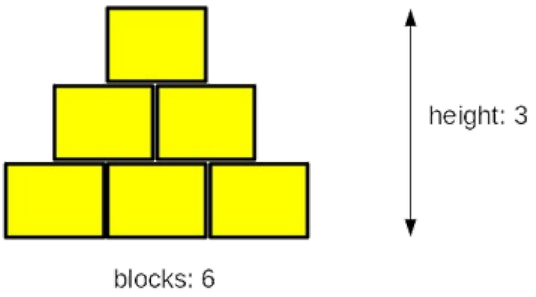
- using the `while` loop;
- finding the proper implementation of verbally defined rules;
- reflecting real-life situations in computer code.

Scenario

Listen to this story: a boy and his father, a computer programmer, are playing with wooden blocks. They are building a pyramid.

Their pyramid is a bit weird, as it is actually a pyramid-shaped wall - it's flat. The pyramid is stacked according to one simple principle: each lower layer contains one block more than the layer above.

The figure illustrates the rule used by the builders:



Your task is to write a program which reads the number of blocks the builders have, and outputs the height of the pyramid that can be built using these blocks.

Note: the height is measured by the number of **fully completed layers** - if the builders don't have a sufficient number of blocks and cannot complete the next layer, they finish their work immediately.

Test your code using the data we've provided.

Test Data

Sample input: `6`

Expected output: `The height of the pyramid: 3`

Sample input: `20`

Expected output: `The height of the pyramid: 5`

Sample input: `1000`

Expected output: `The height of the pyramid: 44`

Sample input: `2`

Expected output: `The height of the pyramid: 1`

Objectives

Familiarize the student with:

- using the `while` loop;
- converting verbally defined loops into actual Python code.

Scenario

In 1937, a German mathematician named Lothar Collatz formulated an intriguing hypothesis (it still remains unproven) which can be described in the following way:

1. take any non-negative and non-zero integer number and name it `c0`;
2. if it's even, evaluate a new `c0` as `c0 ÷ 2`;
3. otherwise, if it's odd, evaluate a new `c0` as `3 × c0 + 1`;
4. if `c0 ≠ 1`, skip to point 2.

The hypothesis says that regardless of the initial value of `c0`, it will always go to 1.

Of course, it's an extremely complex task to use a computer in order to prove the hypothesis for any natural number (it may even require artificial intelligence), but you can use Python to check some individual numbers. Maybe you'll even find the one which would disprove the hypothesis.

Write a program which reads one natural number and executes the above steps as long as `c0` remains different from 1. We also want you to count the steps needed to achieve the goal. Your code should output all the intermediate values of `c0`, too.

Hint: the most important part of the problem is how to transform Collatz's idea into a `while` loop - this is the key to success.

Test your code using the data we've provided.

Test Data

Sample input: `15`

Expected output:

```
46
23
70
35
106
53
160
80
40
20
10
5
16
8
4
2
1
steps = 17
```

Sample input: `16`

Key takeaways

1. There are two types of loops in Python: `while` and `for`:

- the `while` loop executes a statement or a set of statements as long as a specified boolean condition is true, e.g.:

```
# Example 1
while True:
    print("Stuck in an infinite loop.")

# Example 2
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

- the `for` loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the `for` loop to iterate over a sequence of numbers using the built-in `range` function. Look at the examples below:

```
# Example 1
word = "Python"
for letter in word:
    print(letter, end="*")

# Example 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

2. You can use the `break` and `continue` statements to change the flow of a loop:

- You use `break` to exit a loop, e.g.:

```
text = "OpenEDG Python Institute"
for letter in text:
    if letter == "P":
        break
    print(letter, end="")
```

- You use `continue` to skip the current iteration, and continue with the next iteration, e.g.:

```
text = "pyxpyxpyx"
for letter in text:
    if letter == "x":
        continue
    print(letter, end="")
```

Key takeaways: continued

Exercise 1

Create a `for` loop that counts from 0 to 10, and prints odd numbers to the screen. Use the skeleton below:

```
for i in range(1, 11):
    # Line of code.
        # Line of code.
```

Check

Exercise 2

Create a `while` loop that counts from 0 to 10, and prints odd numbers to the screen. Use the skeleton below:

```
x = 1
while x < 11:
    # Line of code.
        # Line of code.
    # Line of code.
```

Check

Exercise 3

Create a program with a `for` loop and a `break` statement. The program should iterate over characters in an email address, exit the loop when it reaches the `@` symbol, and print the part before `@` on one line. Use the skeleton below:

```
for ch in "john.smith@pythoninstitute.org":
    if ch == "@":
        # Line of code.
    # Line of code.
```

Check

Exercise 4

Create a program with a `for` loop and a `continue` statement. The program should iterate over a string of digits, replace each `0` with `x`, and print the modified string to the screen. Use the skeleton below:

```
for digit in "0165031806510":
    if digit == "0":
        # Line of code.
        # Line of code.
    # Line of code.
```

Check

omputer logic

Have you noticed that the conditions we've used so far have been very simple, not to say, quite primitive? The conditions we use in real life are much more complex. Let's look at this sentence:

If we have some free time, and the weather is good, we will go for a walk.

We've used the conjunction `and`, which means that going for a walk depends on the simultaneous fulfilment of these two conditions. In the language of logic, such a connection of conditions is called a **conjunction**. And now another example:

If you are in the mall or I am in the mall, one of us will buy a gift for Mom.

The appearance of the word `or` means that the purchase depends on at least one of these conditions. In logic, such a compound is called a **disjunction**.

It's clear that Python must have operators to build conjunctions and disjunctions. Without them, the expressive power of the language would be substantially weakened. They're called **logical operators**.

and

One logical conjunction operator in Python is the word *and*. It's a **binary operator with a priority that is lower than the one expressed by the comparison operators**. It allows us to code complex conditions without the use of parentheses like this one:

```
counter > 0 and value == 100
```

The result provided by the `and` operator can be determined on the basis of the **truth table**.

If we consider the conjunction of `A and B`, the set of possible values of arguments and corresponding values of the conjunction looks as follows:

Argument `A`	**Argument** `B`	`A and B`
`False`	`False`	`False`
`False`	`True`	`False`
`True`	`False`	`False`
`True`	`True`	`True`

or

A disjunction operator is the word `or`. It's a **binary operator with a lower priority than** `and` (just like `+` compared to `*`). Its truth table is as follows:

Argument `A`	Argument `B`	`A or B`
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

`not`

In addition, there's another operator that can be applied for constructing conditions. It's a **unary operator performing a logical negation**. Its operation is simple: it turns truth into falsehood and falsehood into truth.

This operator is written as the word `not`, and its **priority is very high: the same as the unary `+` and `-`**. Its truth table is simple:

Argument	`not` Argument
`False`	`True`
`True`	`False`

Logical expressions

Let's create a variable named `var` and assign `1` to it. The following conditions are **pairwise equivalent**:

```
# Example 1:
print(var > 0)
print(not (var <= 0))



# Example 2:
print(var != 0)
print(not (var == 0))
```

You may be familiar with De Morgan's laws. They say that:

The negation of a conjunction is the disjunction of the negations.

The negation of a disjunction is the conjunction of the negations.

Let's write the same thing using Python:

```
not (p and q) == (not p) or (not q)
```

```
not (p or q) == (not p) and (not q)
```

Note how the parentheses have been used to code the expressions - we put them there to improve readability.

We should add that none of these two-argument operators can be used in the abbreviated form known as `op=`. This exception is worth remembering.

Logical values vs. single bits

Logical operators take their arguments as a whole regardless of how many bits they contain. The operators are aware only of the value: zero (when all the bits are reset) means `False`; not zero (when at least one bit is set) means `True`.

The result of their operations is one of these values: `False` or `True`. This means that this snippet will assign the value `True` to the `j` variable if `i` is not zero; otherwise, it will be `False`.

```
i = 1
```

```
j = not not i
```

Bitwise operators

However, there are four operators that allow you to **manipulate single bits of data**. They are called **bitwise operators**.

They cover all the operations we mentioned before in the logical context, and one additional operator. This is the `xor` (as in **exclusive or**) operator, and is denoted as `^` (caret).

Here are all of them:

- `&` (ampersand) - bitwise conjunction;
- `|` (bar) - bitwise disjunction;
- `~` (tilde) - bitwise negation;
- `^` (caret) - bitwise exclusive or (xor).

Bitwise operations (`&`, `|`, and `^`)

Argument `A`	Argument `B`	`A & B`	`A | B`	`A ^ B`
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise operations (~)

Argument	`~` Argument
0	1
1	0

Let's make it easier:

- `&` requires exactly two `1`s to provide `1` as the result;
- `|` requires at least one `1` to provide `1` as the result;
- `^` requires exactly one `1` to provide `1` as the result.

Let us add an important remark: the arguments of these operators **must be integers**; we must not use floats here.

The difference in the operation of the logical and bit operators is important: **the logical operators do not penetrate into the bit level of its argument**. They're only interested in the final integer value.

Bitwise operators are stricter: they deal with **every bit separately**. If we assume that the integer variable occupies 64 bits (which is common in modern computer systems), you can imagine the bitwise operation as a 64-fold evaluation of the logical operator for each pair of bits of the arguments. This analogy is obviously imperfect, as in the real world all these 64 operations are performed at the same time (simultaneously).

Logical vs. bit operations: continued

We'll now show you an example of the difference in operation between the logical and bit operations. Let's assume that the following assignments have been performed:

```
i = 15
```

```
j = 22
```

If we assume that the integers are stored with 32 bits, the bitwise image of the two variables will be as follows:

```
i: 00000000000000000000000000001111
```

```
j: 00000000000000000000000000010110
```

The assignment is given:

```
log = i and j
```

We are dealing with a logical conjunction here. Let's trace the course of the calculations. Both variables `i` and `j` are not zeros, so will be deemed to represent `True`. Consulting the truth table for the `and` operator, we can see that the result will be `True`. No other operations are performed.

```
log: True
```

Now the bitwise operation - here it is:

```
bit = i & j
```

The `&` operator will operate with each pair of corresponding bits separately, producing the values of the relevant bits of the result. Therefore, the result will be as follows:

`i`	`00000000000000000000000000001111`
`j`	`00000000000000000000000000010110`

`bit = i & j`	`00000000000000000000000000000110`

These bits correspond to the integer value of six.

Let's look at the negation operators now. First the logical one:

```
logneg = not i
```

The `logneg` variable will be set to `False` - nothing more needs to be done.

The bitwise negation goes like this:

```
bitneg = ~i
```

It may be a bit surprising: the `bitneg` variable value is `-16`. This may seem strange, but isn't at all. If you wish to learn more, you should check out the binary numeral system and the rules governing two's complement numbers.

`i`	`00000000000000000000000000001111`
`bitneg = ~i`	`11111111111111111111111111110000`

Each of these two-argument operators can be used in **abbreviated form**. These are the examples of their equivalent notations:

`x = x & y`	`x &= y`
`x = x | y`	`x |= y`
`x = x ^ y`	`x ^= y`

How do we deal with single bits?

We'll now show you what you can use bitwise operators for. Imagine that you're a developer obliged to write an important piece of an operating system. You've been told that you're allowed to use a variable assigned in the following way:

```
flag_register = 0x1234
```

The variable stores the information about various aspects of system operation. **Each bit of the variable stores one yes/no value**. You've also been told that only one of these bits is yours - the third (remember that bits are numbered from zero, and bit number zero is the lowest one, while the highest is number 31). The remaining bits are not allowed to change, because they're intended to store other data. Here's your bit marked with the letter `x`:

```
flag_register = 00000000000000000000000000000x000
```

You may be faced with the following tasks:

1. **Check the state of your bit** - you want to find out the value of your bit; comparing the whole variable to zero will not do anything, because the remaining bits can have completely unpredictable values, but you can use the following conjunction property:

```
x & 1 = x
x & 0 = 0
```

If you apply the `&` operation to the `flag_register` variable along with the following bit image:

```
00000000000000000000000000001000
```

(note the `1` at your bit's position) as the result, you obtain one of the following bit strings:

- `00000000000000000000000000001000` if your bit was set to `1`
- `00000000000000000000000000000000` if your bit was reset to `0`

Such a sequence of zeros and ones, whose task is to grab the value or to change the selected bits, is called a **bit mask**.

Let's build a bit mask to detect the state of your bit. It should point to **the third bit**. That bit has the weight of $2^3 = 8$. A suitable mask could be created by the following declaration:

```
the_mask = 8
```

You can also make a sequence of instructions depending on the state of your bit i here it is:

```
if flag_register & the_mask:
    # My bit is set.
else:
    # My bit is reset.
```

2. **Reset your bit** - you assign a zero to the bit while all the other bits must remain unchanged; let's use the same property of the conjunction as before, but let's use a slightly different mask - exactly as below:

```
11111111111111111111111111110111
```

Note that the mask was created as a result of the negation of all the bits of `the_mask` variable. Resetting the bit is simple, and looks like this (choose the one you like more):

```
flag_register = flag_register & ~the_mask
flag_register &= ~the_mask
```

3. **Set your bit** - you assign a `1` to your bit, while all the remaining bits must remain unchanged; use the following disjunction property:

```
x | 1 = 1
x | 0 = x
```

You're now ready to set your bit with one of the following instructions:

```
flag_register = flag_register | the_mask
flag_register |= the_mask
```

4. **Negate your bit** - you replace a `1` with a `0` and a `0` with a `1`. You can use an interesting property of the `xor` operator:

```
x ^ 1 = ~x
```

```
x ^ 0 = x
```

and negate your bit with the following instructions:

```
flag_register = flag_register ^ the_mask
flag_register ^= the_mask
```

Binary left shift and binary right shift

Python offers yet another operation relating to single bits: **shifting**. This is applied only to **integer** values, and you mustn't use floats as arguments for it.

You already apply this operation very often and quite unconsciously. How do you multiply any number by ten? Take a look:

12345 × 10 = 123450

As you can see, **multiplying by ten is in fact a shift** of all the digits to the left and filling the resulting gap with zero.

Division by ten? Take a look:

12340 ÷ 10 = 1234

Dividing by ten is nothing but shifting the digits to the right.

The same kind of operation is performed by the computer, but with one difference: as two is the base for binary numbers (not 10), **shifting a value one bit to the left thus corresponds to multiplying it by two**; respectively, **shifting one bit to the right is like dividing by two** (notice that the rightmost bit is lost).

The **shift operators** in Python are a pair of **digraphs**: `<<` and `>>`, clearly suggesting in which direction the shift will act.

```
value << bits
value >> bits
```

The left argument of these operators is an integer value whose bits are shifted. The right argument determines the size of the shift.

It shows that this operation is certainly not commutative.

The priority of these operators is very high. You'll see them in the updated table of priorities, which we'll show you at the end of this section.

Take a look at the shifts in the editor window.

The final `print()` invocation produces the following output:

```
17 68 8
```

output

Note:

- `17 >> 1` → `17 // 2` (**17** floor-divided by **2 to the power of 1**) → `8` (shifting to the right by one bit is the same as integer division by two)
- `17 << 2` → `17 * 4` (**17** multiplied by **2 to the power of 2**) → `68` (shifting to the left by two bits is the same as integer multiplication by four)

And here is the **updated priority table**, containing all the operators introduced so far:

Priority	Operator	
1	`~`, `+`, `-`	unary
2	`**`	
3	`*`, `/`, `//`, `%`	
4	`+`, `-`	binary
5	`<<`, `>>`	
6	`<`, `<=`, `>`, `>=`	
7	`==`, `!=`	
8	`&`	
9	`|`	
10	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `>>=`, `<<=`	

Key takeaways

1. Python supports the following logical operators:

- `and` → if both operands are true, the condition is true, e.g., `(True and True)` is `True`,
- `or` → if any of the operands are true, the condition is true, e.g., `(True or False)` is `True`,
- `not` → returns false if the result is true, and returns true if the result is false, e.g., `not True` is `False`.

2. You can use bitwise operators to manipulate single bits of data. The following sample data:

- `x = 15`, which is `0000 1111` in binary,
- `y = 16`, which is `0001 0000` in binary.

will be used to illustrate the meaning of bitwise operators in Python. Analyze the examples below:

- `&` does a *bitwise and*, e.g., `x & y = 0`, which is `0000 0000` in binary,
- `|` does a *bitwise or*, e.g., `x | y = 31`, which is `0001 1111` in binary,
- `~` does a *bitwise not*, e.g., `~x = 240`*, which is `1111 0000` in binary,
- `^` does a *bitwise xor*, e.g., `x ^ y = 31`, which is `0001 1111` in binary,
- `>>` does a *bitwise right shift*, e.g., `y >> 1 = 8`, which is `0000 1000` in binary,
- `<<` does a *bitwise left shift*, e.g., `y << 3 =` , which is `1000 0000` in binary,

* `-16` (decimal from signed 2's complement) -- read more about the Two's complement operation.

Exercise 1

What is the output of the following snippet?

```
x = 1

y = 0


z = ((x == y) and (x == y)) or not(x == y)

print(not(z))
```

Check

Exercise 2

What is the output of the following snippet?

```
x = 4

y = 1


a = x & y

b = x | y

c = ~x  # tricky!

d = x ^ 5
```

```
e = x >> 2

f = x << 2



print(a, b, c, d, e, f)
```

Check

Why do we need lists?

It may happen that you have to read, store, process, and finally, print dozens, maybe hundreds, perhaps even thousands of numbers. What then? Do you need to create a separate variable for each value? Will you have to spend long hours writing statements like the one below?

```
var1 = int(input())
var2 = int(input())
var3 = int(input())
var4 = int(input())
var5 = int(input())
var6 = int(input())
:
:
```

If you don't think that this is a complicated task, then take a piece of paper and write a program that:

- reads five numbers,
- prints them in order from the smallest to the largest (NB, this kind of processing is called **sorting**).

You should find that you don't even have enough paper to complete the task.

So far, you've learned how to declare variables that are able to store exactly one given value at a time. Such variables are sometimes called **scalars** by analogy with mathematics. All the variables you've used so far are actually scalars.

Think of how convenient it would be to declare a variable that could **store more than one value**. For example, a hundred, or a thousand or even ten thousand. It would still be one and the same variable, but very wide and capacious. Sounds appealing? Perhaps, but how would it handle such a container full of different values? How would it choose just the one you need?

What if you could just number them? And then say: *give me the value number 2; assign the value number 15; increase the value number 10000*.

We'll show you how to declare such **multi-value variables**. We'll do this with the example we just suggested. We'll write a **program that sorts a sequence of numbers**. We won't be particularly ambitious - we'll assume that there are exactly five numbers.

Let's create a variable called `numbers`; it's assigned with not just one number, but is filled with a list consisting of five values (note: the **list starts with an open square bracket and ends with a closed square bracket**; the space between the brackets is filled with five numbers separated by commas).

```
numbers = [10, 5, 7, 2, 1]
```

Let's say the same thing using adequate terminology: `numbers` **is a list consisting of five values, all of them numbers**. We can also say that this statement creates a list of length equal to five (as in there are five elements inside it).

The elements inside a list **may have different types**. Some of them may be integers, others floats, and yet others may be lists.

Python has adopted a convention stating that the elements in a list are **always numbered starting from zero**. This means that the item stored at the beginning of the list will have the number zero. Since there are five elements in our list, the last of them is assigned the number four. Don't forget this.

You'll soon get used to it, and it'll become second nature.

Before we go any further in our discussion, we have to state the following: our **list is a collection of elements, but each element is a scalar**.

Indexing lists

How do you change the value of a chosen element in the list?

Let's **assign a new value of** `111` **to the first element** in the list. We do it this way:

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers)  # Printing original list content.


numbers[0] = 111

print("New list content: ", numbers)  # Current list content.
```

And now we want **the value of the fifth element to be copied to the second element** - can you guess how to do it?

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers)  # Printing original list content.


numbers[0] = 111

print("\nPrevious list content:", numbers)  # Printing previous list content.


numbers[1] = numbers[4]  # Copying value of the fifth element to the second.

print("New list content:", numbers)  # Printing current list content.
```

The value inside the brackets which selects one element of the list is called an **index**, while the operation of selecting an element from the list is known as **indexing**.

We're going to use the `print()` function to print the list content each time we make the changes. This will help us follow each step more carefully and see what's going on after a particular list modification.

Note: all the indices used so far are literals. Their values are fixed at runtime, but **any expression can be the index**, too. This opens up lots of possibilities.

Accessing list content

Each of the list's elements may be accessed separately. For example, it can be printed:

```
print(numbers[0]) # Accessing the list's first element.
```

Assuming that all of the previous operations have been completed successfully, the snippet will send `111` to the console.

As you can see in the editor, the list may also be printed as a whole - just like here:

```
print(numbers)  # Printing the whole list.
```

As you've probably noticed before, Python decorates the output in a way that suggests that all the presented values form a list. The output from the example snippet above looks like this:

```
[111, 1, 7, 2, 1]
```

output

The `len()` function

The **length of a list** may vary during execution. New elements may be added to the list, while others may be removed from it. This means that the list is a very dynamic entity.

If you want to check the list's current length, you can use a function named `len()` (its name comes from *length*).

The function takes the **list's name as an argument, and returns the number of elements currently stored** inside the list (in other words - the list's length).

Look at the last line of code in the editor, run the program and check what value it will print to the console. Can you guess?

Removing elements from a list

Any of the list's elements may be **removed** at any time - this is done with an instruction named `del` (delete). Note: it's an **instruction**, not a function.

You have to point to the element to be removed - it'll vanish from the list, and the list's length will be reduced by one.

Look at the snippet below. Can you guess what output it will produce? Run the program in the editor and check.

```
del numbers[1]
print(len(numbers))
print(numbers)
```

You can't access an element which doesn't exist - you can neither get its value nor assign it a value. Both of these instructions will cause runtime errors now:

```
print(numbers[4])
```

```
numbers[4] = 1
```

Add the snippet above after the last line of code in the editor, run the program and check what happens.

Note: we've removed one of the list's elements - there are only four elements in the list now. This means that element number four doesn't exist.

Code

```
1
2
3
4
5
6
7
8
9
10
11
```

```
numbers = [10, 5, 7, 2, 1]

print("Original list content:", numbers)  # Printing original list content.

numbers[0] = 111

print("\nPrevious list content:", numbers)  # Printing previous list content.

numbers[1] = numbers[4]  # Copying value of the fifth element to the second.

print("Previous list content:", numbers)  # Printing previous list content.

print("\nList's length:", len(numbers))  # Printing previous list length.

###

del numbers[1]  # Removing the second element from the list.

print("New list's length:", len(numbers))  # Printing new list length.

print("\nNew list content:", numbers)  # Printing current list content.

###
```

- **Console**

Negative indices are legal

It may look strange, but negative indices are legal, and can be very useful.

An element with an index equal to `-1` is **the last one in the list**.

```
print(numbers[-1])
```

The example snippet will output `1`. Run the program and check.

Similarly, the element with an index equal to `-2` is **the one before last in the list**.

```
print(numbers[-2])
```

The example snippet will output `2`.

The last accessible element in our list is `numbers[-4]` (the first one) - don't try to go any further!

Code

```
1
2
3
4
```

```
numbers = [111, 7, 2, 1]
print(numbers[-1])
print(numbers[-2])
```

Module 4

Functions, Tuples, Dictionaries, Exceptions, and Data Processing

In this module, you will cover the following topics:

- code structuring and the concept of function;
- function invocation and returning a result from a function;
- name scopes and variable shadowing;
- tuples and their purpose, constructing and using tuples;
- dictionaries and their purpose, constructing and using dictionaries;
- exceptions – the *try* statement and the *except* clause, Python built-in exceptions, code testing and debugging.

Why do we need functions?

You've come across **functions** many times so far, but the view on their merits that we have given you has been rather one-sided. You've only invoked the functions by using them as tools to make life easier, and to simplify time-consuming and tedious tasks.

When you want some data to be printed on the console, you use `print()`. When you want to read the value of a variable, you use `input()`, coupled with either `int()` or `float()`.

You've also made use of some **methods**, which are in fact functions, but declared in a very specific way.

Now you'll learn how to write your own functions, and how to use them. We'll write several functions together, from the very simple to the rather complex, which will require your focus and attention.

It often happens that a particular piece of code is **repeated many times in your program**. It's repeated either literally, or with only a few minor modifications, consisting of the use of other variables in the same algorithm. It also happens that a programmer cannot resist simplifying the work, and begins to clone such pieces of code using the clipboard and copy-paste operations.

It could end up as greatly frustrating when suddenly it turns out that there was an error in the cloned code. The programmer will have a lot of drudgery to find all the places that need corrections. There's also a high risk of the corrections causing errors.

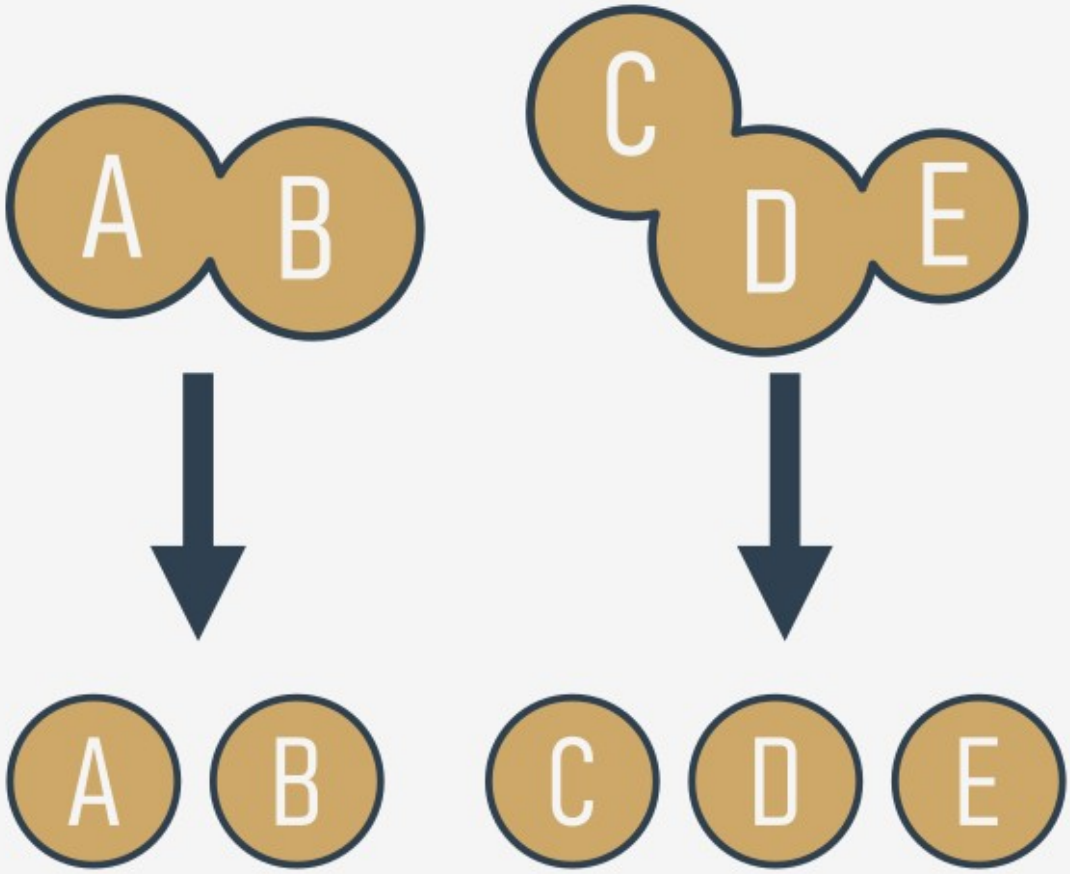
We can now define the first condition which can help you decide when to start writing your own functions: **if a particular fragment of the code begins to appear in more than one place, consider the possibility of isolating it in the form of a function** invoked from the points where the original code was placed before.

It may happen that the algorithm you're going to implement is so complex that your code begins to grow in an uncontrolled manner, and suddenly you notice that you're not able to navigate through it so easily anymore.

You can try to cope with the issue by commenting the code extensively, but soon you find that this dramatically worsens your situation - **too many comments make the code larger and harder to read**. Some say that a **well-written function should be viewed entirely in one glance**.

A good and attentive developer **divides the code** (or more accurately: the problem) into well-isolated pieces, and **encodes each of them in the form of a function**.

This considerably simplifies the work of the program, because each piece of code can be encoded separately, and tested separately. The process described here is often called **decomposition**.



We can now state the second condition: **if a piece of code becomes so large that reading and understating it may cause a problem, consider dividing it into separate, smaller problems, and implement each of them in the form of a separate function**.

This decomposition continues until you get a set of short functions, easy to understand and test.

Decomposition

It often happens that the problem is so large and complex that it cannot be assigned to a single developer, and a **team of developers** have to work on it. The problem must be split between several developers in a way that ensures their efficient and seamless cooperation.

It seems inconceivable that more than one programmer should write the same piece of code at the same time, so the job has to be dispersed among all the team members.

This kind of decomposition has a different purpose to the one described previously - it's not only about **sharing the work**, but also about **sharing the responsibility** among many developers.

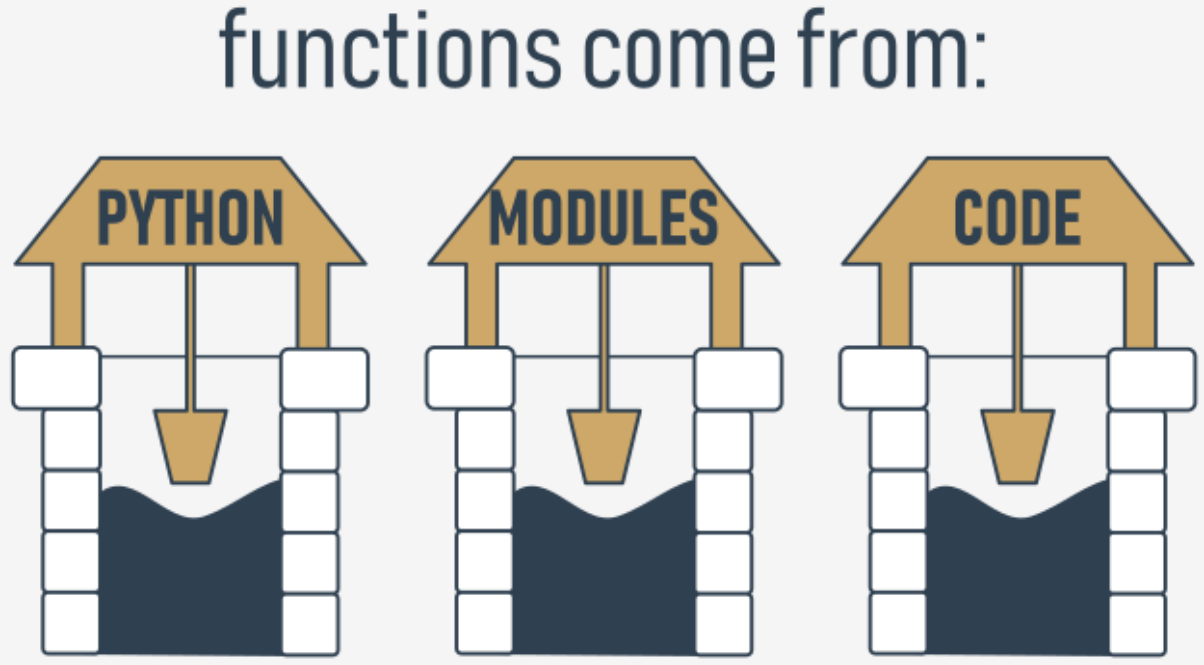
Each of them writes a clearly defined and described set of functions, which when **combined into the module** (we'll tell you about this a bit later) will give the final product.

This leads us directly to the third condition: if you're going to divide the work among multiple programmers, **decompose the problem to allow the product to be implemented as a set of separately written functions packed together in different modules**.

Where do the functions come from?

In general, functions come from at least three places:

- from Python itself - numerous functions (like `print()`) are an **integral part of Python**, and are always available without any additional effort on behalf of the programmer; we call these functions **built-in functions**;
- from Python's **preinstalled modules** - a lot of functions, very useful ones, but used significantly less often than built-in ones, are available in a number of modules installed together with Python; the use of these functions requires some additional steps from the programmer in order to make them fully accessible (we'll tell you about this in a while);
- **directly from your code** - you can write your own functions, place them inside your code, and use them freely;
- there is one other possibility, but it's connected with classes, so we'll omit it for now.



Your first function

How do you make such a function?

You need to **define** it. The word *define* is significant here.

This is what the simplest function definition looks like:

```
def function_name():
    function_body
```

- It always starts with the **keyword** `def` (for *define*)
- next after `def` goes the **name of the function** (the rules for naming functions are exactly the same as for naming variables)
- after the function name, there's a place for a pair of **parentheses** (they contain nothing here, but that will change soon)
- the line has to be ended with a **colon**;
- the line directly after `def` begins the **function body** - a couple (at least one) of necessarily **nested instructions**, which will be executed every time the function is invoked; note: the **function ends where the nesting ends**, so you have to be careful.

We're ready to define our **prompting** function. We'll name it `message` - here it is:

```
def message():
    print("Enter a value: ")
```

The function is extremely simple, but fully **usable**. We've named it `message`, but you can label it according to your taste. Let's use it.

Our code contains the function definition now:

```
def message():
    print("Enter a value: ")

print("We start here.")
print("We end here.")
```

Note: we don't use the function at all - there's no **invocation** of it inside the code.

When you run it, you see the following output:

```
We start here.
We end here.
```

output

This means that Python reads the function's definitions and remembers them, but won't launch any of them without your permission.

We've modified the code now - we've inserted the **function's invocation** between the start and end messages:

```
def message():
    print("Enter a value: ")

print("We start here.")
message()
print("We end here.")
```

The output looks different now:

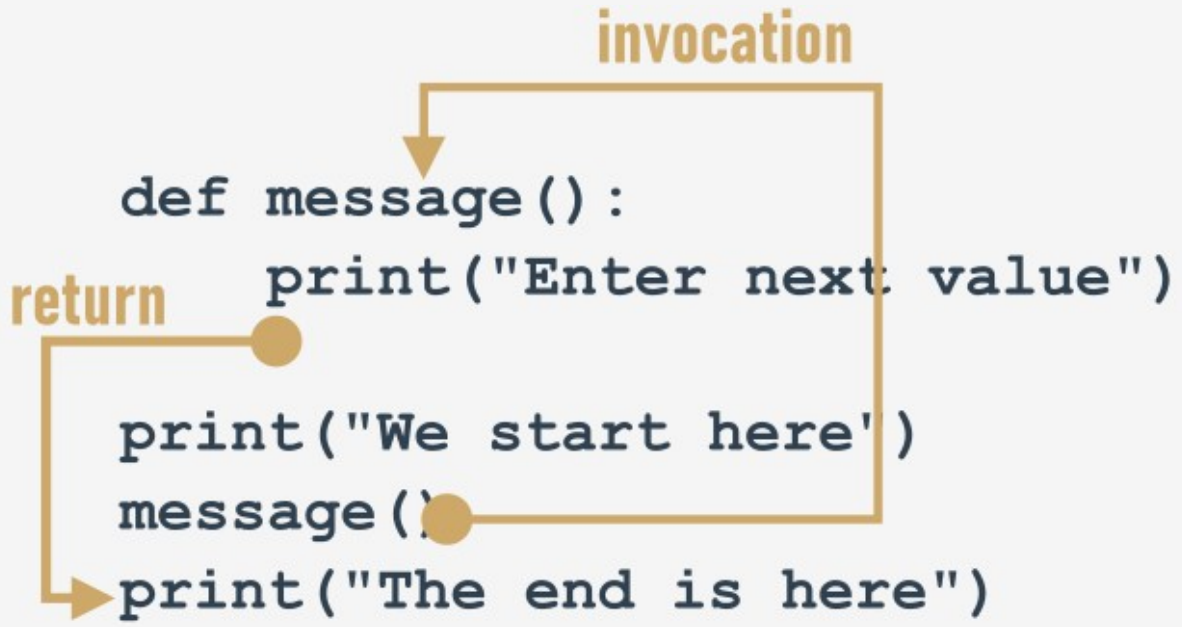
```
We start here.
Enter a value:
We end here.
```

output

Test the code, modify it, experiment with it.

How functions work

Look at the picture below:



It tries to show you the whole process:

- when you **invoke** a function, Python remembers the place where it happened and *jumps* into the invoked function;
- the body of the function is then **executed**;
- reaching the end of the function forces Python to **return** to the place directly after the point of invocation.

There are two, very important, catches. Here's the first of them:

You mustn't invoke a function which is not known at the moment of invocation.

Remember - Python reads your code from top to bottom. It's not going to look ahead in order to find a function you forgot to put in the right place ("right" means "before invocation".)

We've inserted an error into this code - can you see the difference?

```
print("We start here.")
message()
print("We end here.")


def message():
    print("Enter a value: ")
```

We've moved the function to the end of the code. Is Python able to find it when the execution reaches the invocation?

No, it isn't. The error message will read:

```
NameError: name 'message' is not defined
```

output

Don't try to force Python to look for functions you didn't deliver at the right time.

The second catch sounds a little simpler:

You mustn't have a function and a variable of the same name.

The following snippet is erroneous:

```
def message():
    print("Enter a value: ")

message = 1
```

Assigning a value to the name message causes Python to forget its previous role. The function named `message` becomes unavailable.

Fortunately, you're free to **mix your code with functions** - you're not obliged to put all your functions at the top of your source file.

Look at the snippet:

```
print("We start here.")


def message():
    print("Enter a value: ")

message()

print("We end here.")
```

It may look strange, but it's completely correct, and works as intended.

Let's return to our primary example, and employ the function for the right job, like here:

```
def message():
    print("Enter a value: ")

message()
a = int(input())
message()
b = int(input())
message()
c = int(input())
```

Modifying the prompting message is now easy and clear - you can do it by **changing the code in just one place** - inside the function's body.

Open the sandbox, and try to do it yourself.

Parameterized functions

The function's full power reveals itself when it can be equipped with an interface that is able to accept data provided by the invoker. Such data can modify the function's behavior, making it more flexible and adaptable to changing conditions.

A parameter is actually a variable, but there are two important factors that make parameters different and special:

- **parameters exist only inside functions in which they have been defined**, and the only place where the parameter can be defined is a space between a pair of parentheses in the `def` statement;
- **assigning a value to the parameter is done at the time of the function's invocation**, by specifying the corresponding argument.

```
def function(parameter):
    ###
```

Don't forget:

- **parameters live inside functions** (this is their natural environment)
- **arguments exist outside functions**, and are carriers of values passed to corresponding parameters.

There is a clear and unambiguous frontier between these two worlds.

Let's enrich the function above with just one parameter - we're going to use it to show the user the number of a value the function asks for.

We have to rebuild the `def` statement - this is how it looks now:

```
def message(number):
    ###
```

The definition specifies that our function operates on just one parameter named `number`. You can use it as an ordinary variable, but **only inside the function** - it isn't visible anywhere else.

Let's now improve the function's body:

```
def message(number):
    print("Enter a number:", number)
```

We've made use of the parameter. Note: we haven't assigned the parameter with any value. Is it correct?

Yes, it is.

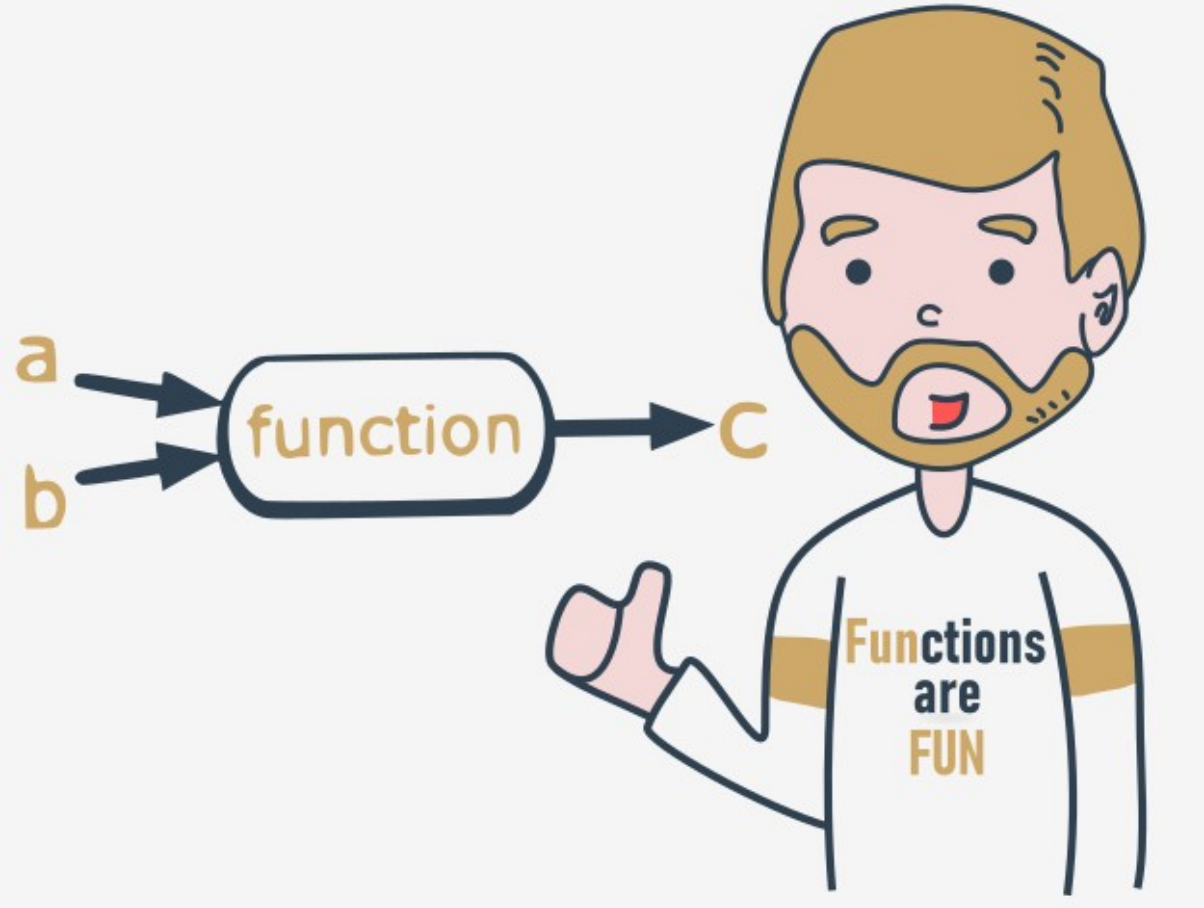
A value for the parameter will arrive from the function's environment.

Remember: **specifying one or more parameters in a function's definition** is also a requirement, and you have to fulfil it during invocation. You must **provide as many arguments as there are defined parameters**.

Failure to do so will cause an error.

Parametrized functions: continued

A function can have **as many parameters as you want**, but the more parameters you have, the harder it is to memorize their roles and purposes.



Let's modify the function - it has **two parameters** now:

```
def message(what, number):
```

```
    print("Enter", what, "number", number)
```

This also means that invoking the function will require **two arguments**.

The first new parameter is intended to carry the name of the desired value.

Here it is:

```
def message(what, number):
    print("Enter", what, "number", number)


message("telephone", 11)
message("price", 5)
message("number", "number")
```

This is the output you're about to see:

```
Enter telephone number 11
Enter price number 5
Enter number number number
```

output

Run the code, modify it, add more parameters, and see how this affects the output.

Positional parameter passing

A technique which assigns the i^{th} (first, second, and so on) argument to the i^{th} (first, second, and so on) function parameter is called **positional parameter passing**, while arguments passed in this way are named **positional arguments**.

You've used it already, but Python can offer a lot more. We're going to tell you about it now.

```
def my_function(a, b, c):
    print(a, b, c)


my_function(1, 2, 3)
```

Note: positional parameter passing is intuitively used by people in many social occasions. For example, it may be generally accepted that when we introduce ourselves we mention our first name(s) before our last name, e.g., "My name's John Doe."

Incidentally, Hungarians do it in reverse order.

Let's implement that social custom in Python. The following function will be responsible for introducing somebody:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)

introduction("Luke", "Skywalker")
introduction("Jesse", "Quick")
introduction("Clark", "Kent")
```

Can you guess the output? Run the code and find out if you were right.

Now imagine that the same function is being used in Hungary. In this case, the code would look like this:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)

introduction("Skywalker", "Luke")
introduction("Quick", "Jesse")
introduction("Kent", "Clark")
```

The output will look different. Can you guess it?

Run the code to see if you were right here, too. Are you surprised?

Can you make the function more culture-independent?

eyword argument passing

Python offers another convention for passing arguments, where **the meaning of the argument is dictated by its name**, not by its position - it's called **keyword argument passing**.

Take a look at the snippet:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)


introduction(first_name = "James", last_name = "Bond")
introduction(last_name = "Skywalker", first_name = "Luke")
```

The concept is clear - the values passed to the parameters are preceded by the target parameters' names, followed by the `=` sign.

The position doesn't matter here - each argument's value knows its destination on the basis of the name used.

You should be able to predict the output. Run the code to check if you were right.

Of course, you **mustn't use a non-existent parameter name**.

The following snippet will cause a runtime error:

```
def introduction(first_name, last_name):
    print("Hello, my name is", first_name, last_name)


introduction(surname="Skywalker", first_name="Luke")
```

This is what Python will tell you:

```
TypeError: introduction() got an unexpected keyword argument 'surname'
```

output

Try it out yourself.

Mixing positional and keyword arguments

You can mix both fashions if you want - there is only one unbreakable rule: you have to put **positional arguments before keyword arguments**.

If you think for a moment, you'll certainly guess why.

To show you how it works, we'll use the following simple three-parameter function:

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)
```

Its purpose is to evaluate and present the sum of all its arguments.

The function, when invoked in the following way:

```
adding(1, 2, 3)
```

will output:

```
1 + 2 + 3 = 6
```

output

It was - as you may suspect - a pure example of **positional argument passing**.

Of course, you can replace such an invocation with a purely keyword variant, like this:

```
adding(c = 1, a = 2, b = 3)
```

Our program will output a line like this:

```
2 + 3 + 1 = 6
```

output

Note the order of the values.

Let's try to mix both styles now.

Look at the function invocation below:

```
adding(3, c = 1, b = 2)
```

Let's analyze it:

- the argument (`3`) for the `a` parameter is passed using the positional way;
- the arguments for `c` and `b` are specified as keyword ones.

This is what you'll see in the console:

```
3 + 2 + 1 = 6
```

output

Be careful, and beware of mistakes. If you try to pass more than one value to one argument, all you'll get is a runtime error.

Look at the invocation below - it seems that we've tried to set `a` twice:

```
adding(3, a = 1, b = 2)
```

Python's response:

```
TypeError: adding() got multiple values for argument 'a'
```

output

Look at the snipet below. A code like this is fully correct, but it doesn't make much sense:

```
adding(4, 3, c = 2)
```

Everything is right, but leaving in just one keyword argument looks a bit weird - what do you think?

Parametrized functions - more details

It happens at times that a particular parameter's values are in use more often than others. Such arguments may have their **default (predefined) values** taken into consideration when their corresponding arguments have been omitted.

They say that the most popular English last name is *Smith*. Let's try to take this into account.

The default parameter's value is set using clear and pictorial syntax:

```
def introduction(first_name, last_name="Smith"):
    print("Hello, my name is", first_name, last_name)
```

You only have to extend the parameter's name with the `=` sign, followed by the default value.

Let's invoke the function as usual:

```
introduction("James", "Doe")
```

Can you guess the output of the program? Run it and check if you were right.

And? Everything looks the same, but when you invoke the function in a way that looks a bit suspicious at first sight, like this:

```
introduction("Henry")
```

or this:

```
introduction(first_name="William")
```

there will be no error, and both invocations will succeed, while the console will show the following output:

```
Hello, my name is Henry Smith
Hello, my name is William Smith
```

output

Test it.

You can go further if it's useful. Both parameters have their default values now, look at the code below:

```
def introduction(first_name="John", last_name="Smith"):
    print("Hello, my name is", first_name, last_name)
```

This makes the following invocation absolutely valid:

```
introduction()
```

And this is the expected output:

```
Hello, my name is John Smith
```

output

If you use one keyword argument, the remaining one will take the default value:

```
introduction(last_name="Hopkins")
```

The output is:

```
Hello, my name is John Hopkins
```

output

Test it.

Congratulations - you have just learned the basic ways of communicating with functions.

Effects and results: the `return` instruction

All the previously presented functions have some kind of effect - they produce some text and send it to the console.

Of course, functions - like their mathematical siblings - may have results.

To get **functions to return a value** (but not only for this purpose) you use the `return` instruction.

This word gives you a full picture of its capabilities. Note: it's a Python **keyword**.

The `return` instruction has **two different variants** - let's consider them separately.

`return` without an expression

The first consists of the keyword itself, without anything following it.

When used inside a function, it causes the **immediate termination of the function's execution, and an instant return (hence the name) to the point of invocation**.

Note: if a function is not intended to produce a result, **using the `return` instruction is not obligatory** - it will be executed implicitly at the end of the function.

Anyway, you can use it to **terminate a function's activities on demand**, before the control reaches the function's last line.

Let's consider the following function:

```
def happy_new_year(wishes = True):
    print("Three...")
    print("Two...")
    print("One...")
    if not wishes:
        return

    print("Happy New Year!")
```

When invoked without any arguments:

```
happy_new_year()
```

The function causes a little noise - the output will look like this:

```
Three...
Two...
One...
Happy New Year!
```

output

Providing `False` as an argument:

```
happy_new_year(False)
```

will modify the function's behavior - the `return` instruction will cause its termination just before the wishes - this is the updated output:

```
Three...
Two...
One...
```

output

`return` with an expression

The second `return` variant is **extended with an expression**:

```
def function():
    return expression
```

There are two consequences of using it:

- it causes the **immediate termination of the function's execution** (nothing new compared to the first variant)
- moreover, the function will **evaluate the expression's value and will return (hence the name once again) it as the function's result**.

Yes, we already know - this example isn't really sophisticated:

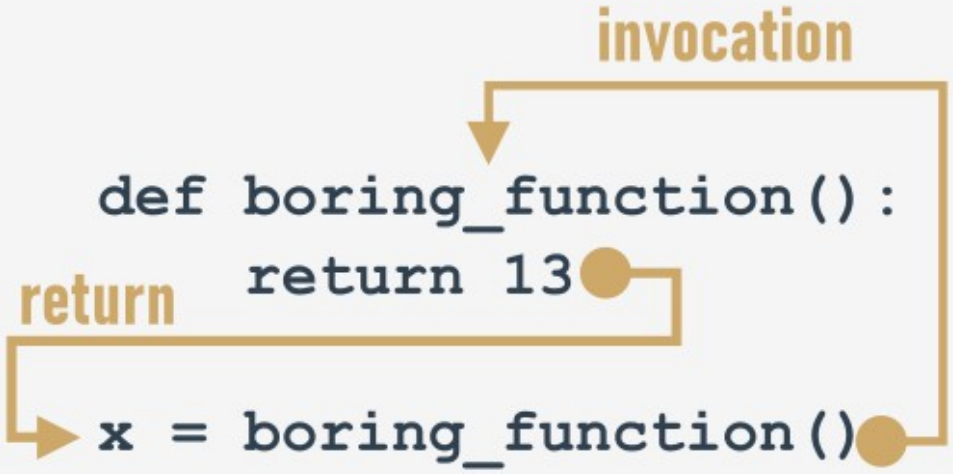
```
def boring_function():
    return 123


x = boring_function()

print("The boring_function has returned its result. It's:", x)
```

The snippet writes the following text to the console:

```
The boring_function has returned its result. It's: 123
```

Let's investigate it for a while.

Analyze the figure
below:



The `return` instruction, enriched with the expression (the expression is very simple here), "transports" the expression's value to the place where the function has been invoked.

The result may be freely used here, e.g., to be assigned to a variable.

It may also be completely ignored and lost without a trace.

Note, we're not being too polite here - the function returns a value, and we ignore it (we don't use it in any way):

```
def boring_function():
    print("'Boredom Mode' ON.")
    return 123


print("This lesson is interesting!")
boring_function()
print("This lesson is boring...")
```

The program produces the following output:

```
This lesson is interesting!
'Boredom Mode' ON.
This lesson is boring...
```

output

Is it punishable? Not at all.

The only disadvantage is that the result has been irretrievably lost.

Don't forget:

- you are always **allowed to ignore the function's result**, and be satisfied with the function's effect (if the function has any)
- if a function is intended to return a useful result, it must contain the second variant of the `return` instruction.

Wait a minute - does this mean that there are useless results, too? Yes - in some sense.