PCAP - Programming Essentials in Python

**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.

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

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

# What makes a language?

We can say that each language (machine or natural, 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.



If the line looks good, the interpreter tries to execute it (note: each line is usually executed separately, so the trio "read-check-execute" can be repeated many times - more times than the actual number of lines in the source file, as some parts of the code may be executed more than once).

It is also possible that a significant part of the code may be executed successfully before the interpreter finds an error. This is normal behavior in this execution model.

You may ask now: which is better? The "compiling" model or the "interpreting" model? There is no obvious answer. If there had been, one of these models would have ceased to exist a long time ago. Both of them have their advantages and their disadvantages.

[Prev](https://edube.org/learn/python-essentials-1/programming-absolute-basics-compilation-vs-interpretation-3) [Next](https://edube.org/learn/python-essentials-1/programming-absolute-basics-compilation-vs-interpretation-5)

**ompilation 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**](https://en.wikipedia.org/wiki/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.

**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](http://pypl.github.io/PYPL.html) and the [TIOBE Programming Community Index](https://www.tiobe.com/tiobe-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.

**What makes Python special?**

How does it happen that programmers, young and old, experienced and novice, want to use it? How did it happen that large companies adopted Python and implemented their flagship products using it?

There are many reasons - we've listed some of them already, but let's enumerate them again in a more practical manner:

* it's **easy to learn** - the time needed to learn Python is shorter than for many other languages; this means that it's possible to start the actual programming faster;
* it's **easy to teach** - the teaching workload is smaller than that needed by other languages; this means that the teacher can put more emphasis on general (language-independent) programming techniques, not wasting energy on exotic tricks, strange exceptions and incomprehensible rules;
* it's **easy to use** for writing new software - it's often possible to write code faster when using Python;
* it's **easy to understand** - it's also often easier to understand someone else's code faster if it is written in Python;
* it's **easy to obtain, install and deploy** - Python is free, open and multiplatform; not all languages can boast that.

Of course, Python has its drawbacks, too:

* it's not a speed demon - Python does not deliver exceptional performance;
* in some cases it may be resistant to some simpler testing techniques - this may mean that debugging Python's code can be more difficult than with other languages; fortunately, making mistakes is always harder in Python.



It should also be stated that Python is not the only solution of its kind available on the IT market.

It has lots of followers, but there are many who prefer other languages and don't even consider Python for their projects.

**Python rivals?**

Python has two direct competitors, with comparable properties and predispositions. These are:

* **Perl** - a scripting language originally authored by Larry Wall;
* **Ruby** - a scripting language originally authored by Yukihiro Matsumoto.

The former is more traditional, more conservative than Python, and resembles some of the good old languages derived from the classic C programming language.

In contrast, the latter is more innovative and more full of fresh ideas than Python. Python itself lies somewhere between these two creations.

The Internet is full of forums with infinite discussions on the superiority of one of these three over the others, should you wish to learn more about each of them.

**Where can we see Python in action?**

We see it every day and almost everywhere. It's used extensively to implement complex **Internet services** like search engines, cloud storage and tools, social media and so on. Whenever you use any of these services, you are actually very close to Python, although you wouldn't know it.

Many **developing tools** are implemented in Python. More and more **everyday use applications** are being written in Python. Lots of **scientists** have abandoned expensive proprietary tools and switched to Python. Lots of IT project **testers** have started using Python to carry out repeatable test procedures. The list is long.

# There is more than one Python

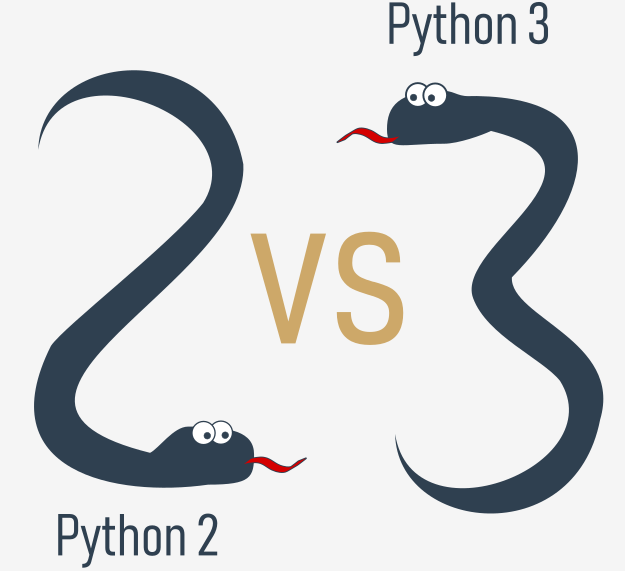
There are two main kinds of Python, called Python 2 and Python 3.

Python 2 is an older version of the original Python. Its development has since been intentionally stalled, although that doesn't mean that there are no updates to it. On the contrary, the updates are issued on a regular basis, but they are not intended to modify the language in any significant way. They rather fix any freshly discovered bugs and security holes. Python 2's development path has reached a dead end already, but Python 2 itself is still very much alive.

**Python 3 is the newer (to be precise, the current) version of the language. It's going through its own evolution path, creating its own standards and habits.**

These two versions of Python aren't compatible with each other. Python 2 scripts won't run in a Python 3 environment and vice versa, so if you want the old Python 2 code to be run by a Python 3 interpreter, the only possible solution is to rewrite it, not from scratch, of course, as large parts of the code may remain untouched, but you do have to revise all the code to find all possible incompatibilities. Unfortunately, this process cannot be fully automatized.

It's too hard, too time-consuming, too expensive, and too risky to migrate an old Python 2 application to a new platform. It's possible that rewriting the code will introduce new bugs to it. It's easier and more sensible to leave these systems alone and to improve the existing interpreter, instead of trying to work inside the already functioning source code.

Python 3 isn't just a better version of Python 2 - it is a completely different language, although it's very similar to its predecessor. When you look at them from a distance, they appear to be the same, but when you look closely, though, you notice a lot of differences.  
  
If you're modifying an old existing Python solution, then it's highly likely that it was coded in Python 2. This is the reason why Python 2 is still in use. There are too many existing Python 2 applications to discard it altogether.

**NOTE**

If you're going to start a new Python project, **you should use Python 3, and this is the version of Python that will be used during this course.**

It is important to remember that there may be smaller or bigger differences between subsequent Python 3 releases (e.g., Python 3.6 introduced ordered dictionary keys by default under the CPython implementation) - the good news, though, is that all the newer versions of Python 3 are **backwards compatible** with the previous versions of Python 3. Whenever meaningful and important, we will always try to highlight those differences in the course.

All the code samples you will find during the course have been tested against Python 3.4, Python 3.6, Python 3.7, and Python 3.8.

[Prev](https://edube.org/learn/python-essentials-1/python-a-tool-not-a-reptile-why-python-why-not-1) [Next](https://edube.org/learn/python-essentials-1/there-is-more-than-one-python-cpython-and-cython-1)

# Python aka CPython

In addition to Python 2 and Python 3, there is more than one version of each.

First of all, there are the Pythons which are maintained by the people gathered around the PSF ([Python Software Foundation](https://www.python.org/psf-landing/)), a community that aims to develop, improve, expand, and popularize Python and its environment. The PSF's president is Guido von Rossum himself, and for this reason, these Pythons are called **canonical**. They are also considered to be **reference Pythons**, as any other implementation of the language should follow all standards established by the PSF.

Guido van Rossum used the "C" programming language to implement the very first version of his language and this decision is still in force. All Pythons coming from the PSF are written in the "C" language. There are many reasons for this approach and it has many consequences. One of them (probably the most important) is that thanks to it, Python may be easily ported and migrated to all platforms with the ability to compile and run "C" language programs (virtually all platforms have this feature, which opens up many expansion opportunities for Python).

This is why the PSF implementation is often referred to as **CPython**. This is the most influential Python among all the Pythons in the world.

# Cython

Another Python family member is **Cython**.

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

How are these two contradictions reconciled? One solution is to write your mathematical ideas using Python, and when you're absolutely sure that your code is correct and produces valid results, you can translate it into "C". Certainly, "C" will run much faster than pure Python.

This is what Cython is intended to do - to automatically translate the Python code (clean and clear, but not too swift) into "C" code (complicated and talkative, but agile).

# Jython

Another version of Python is called **Jython**.

"J" is for "Java". Imagine a Python written in Java instead of C. This is useful, for example, if you develop large and complex systems written entirely in Java and want to add some Python flexibility to them. The traditional CPython may be difficult to integrate into such an environment, as C and Java live in completely different worlds and don't share many common ideas.

Jython can communicate with existing Java infrastructure more effectively. This is why some projects find it usable and needful.

Note: the current Jython implementation follows Python 2 standards. There is no Jython conforming to Python 3, so far.



# PyPy and RPython

Take a look at the logo below. It's a rebus. Can you solve it?

It's a logo of the **PyPy** - a Python within a Python. In other words, it represents a Python environment written in Python-like language named **RPython** (Restricted Python). It is actually a subset of Python.  
  
The source code of PyPy is not run in the interpretation manner, but is instead translated into the C programming language and then executed separately.

This is useful because if you want to test any new feature that may be (but doesn't have to be) introduced into mainstream Python implementation, it's easier to check it with PyPy than with CPython. This is why PyPy is rather a tool for people developing Python than for the rest of the users.

This doesn't make PyPy any less important or less serious than CPython, of course.

In addition, PyPy is compatible with the Python 3 language.

There are many more different Pythons in the world. You'll find them if you look, but **this course will focus on CPython**.

[Prev](https://edube.org/learn/python-essentials-1/there-is-more-than-one-python-cpython-and-cython-1) [Next](https://edube.org/learn/python-essentials-1/begin-your-python-journey-8)

# How to get Python and how to get to use it

There are several ways to get your own copy of Python 3, depending on the operating system you use.

**Linux users most probably have Python already installed** - this is the most likely scenario, as Python's infrastructure is intensively used by many Linux OS components.

For example, some distributors may couple their specific tools together with the system and many of these tools, like package managers, are often written in Python. Some parts of graphical environments available in the Linux world may use Python, too.

If you're a Linux user, open the terminal/console, and type:

python3

at the shell prompt, press *Enter* and wait.

If you see something like this:

Python 3.4.5 (default, Jan 12 2017, 02:28:40)

[GCC 4.2.1 Compatible Clang 3.7.1 (tags/RELEASE\_371/final)] on linux

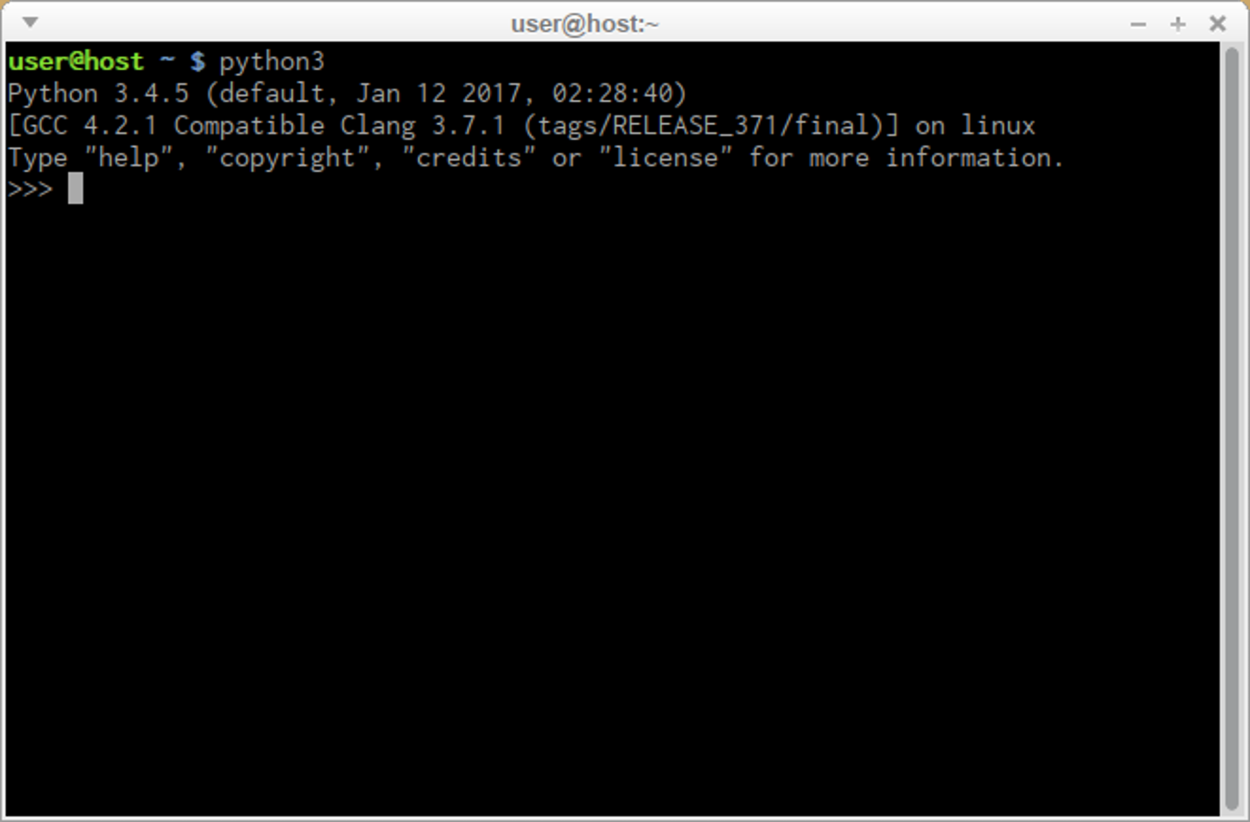
Type "help", "copyright", "credits" or "license" for more information.

>>>

then you don't have to do anything else.

If Python 3 is absent, then refer to your Linux documentation in order to find how to use your package manager to download and install a new package - the one you need is named **python3** or its name begins with that.

All non-Linux users can download a copy at <https://www.python.org/downloads/>.



# Downloading and installing Python

Because the browser tells the site you've entered the OS you use, the only step you have to take is to click the appropriate Python version you want.

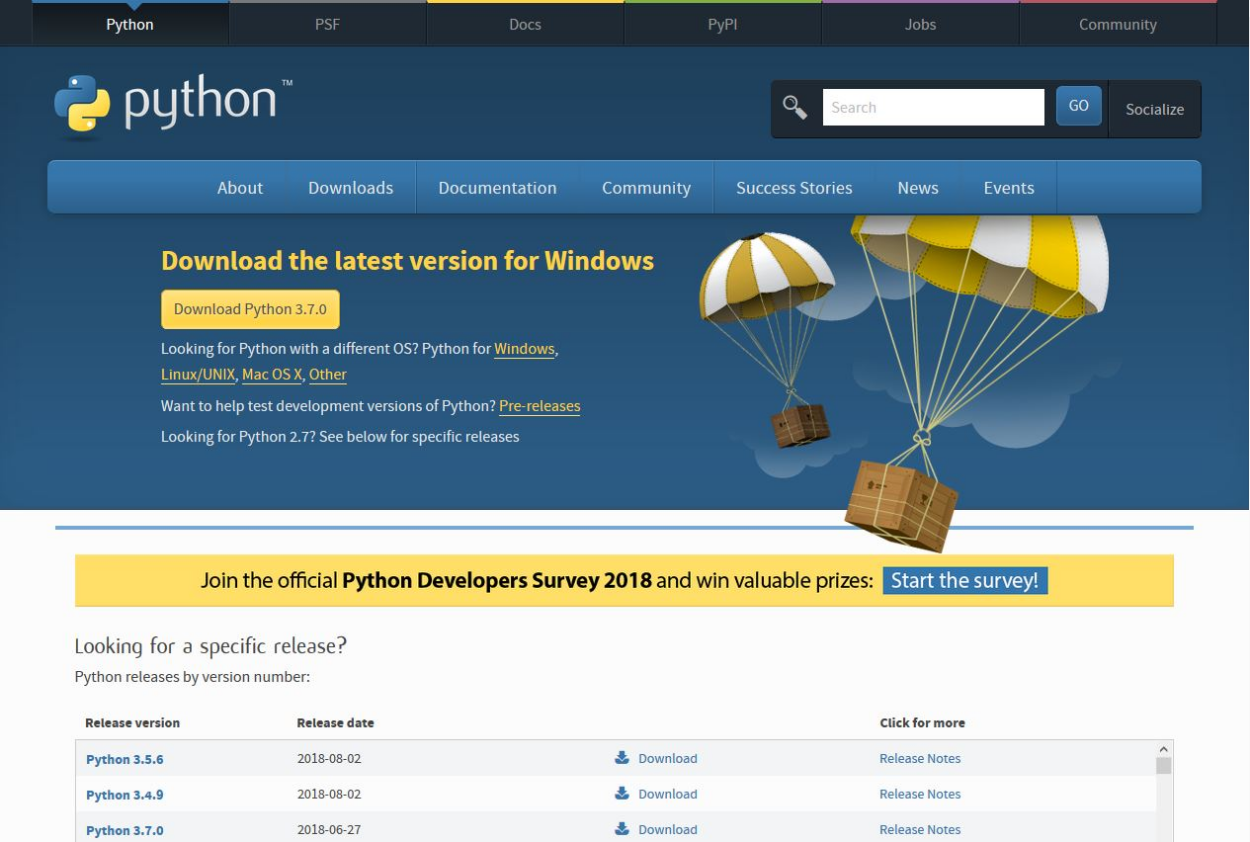
In this case, select Python 3. The site always offers you the latest version of it.

If you're a **Windows user**, start the downloaded .exe file and follow all the steps.

Leave the default settings the installer suggests for now, with one exception - look at the checkbox named **Add Python 3.x to PATH** and check it.

This will make things easier.

If you're a **macOS user**, a version of Python 2 may already have been preinstalled on your computer, but since we will be working with Python 3, you will still need to download and install the relevant .pkg file from the Python site.



**Starting your work with Python**

Now that you have Python 3 installed, it's time to check if it works and make the very first use of it.

This will be a very simple procedure, but it should be enough to convince you that the Python environment is complete and functional.

There are many ways of utilizing Python, especially if you're going to be a Python developer.

To start your work, you need the following tools:

* an **editor** which will support you in writing the code (it should have some special features, not available in simple tools); this dedicated editor will give you more than the standard OS equipment;
* a **console** in which you can launch your newly written code and stop it forcibly when it gets out of control;
* a tool named a **debugger**, able to launch your code step by step and allowing you to inspect it at each moment of execution.

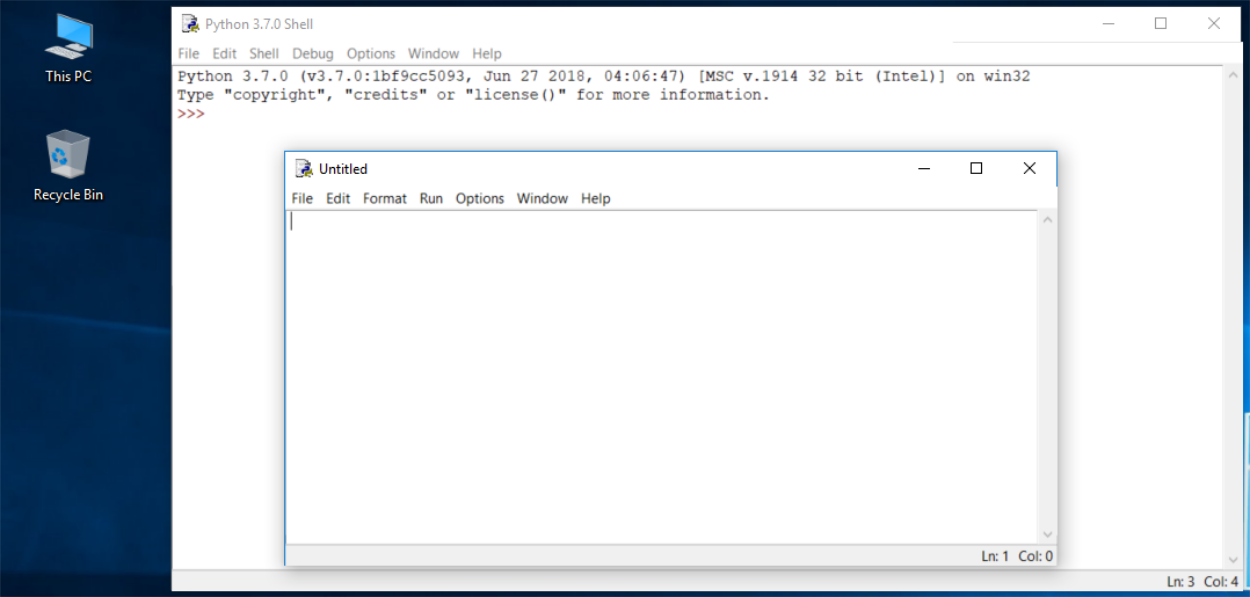
Besides its many useful components, the Python 3 standard installation contains a very simple but extremely useful application named IDLE.

**IDLE** is an acronym: Integrated Development and Learning Environment.

# How to write and run your very first program

It is now time to write and run your first Python 3 program. It will be very simple, for now.

The first step is to create a new source file and fill it with code. Click *File* in the IDLE’s menu and choose *New file*.

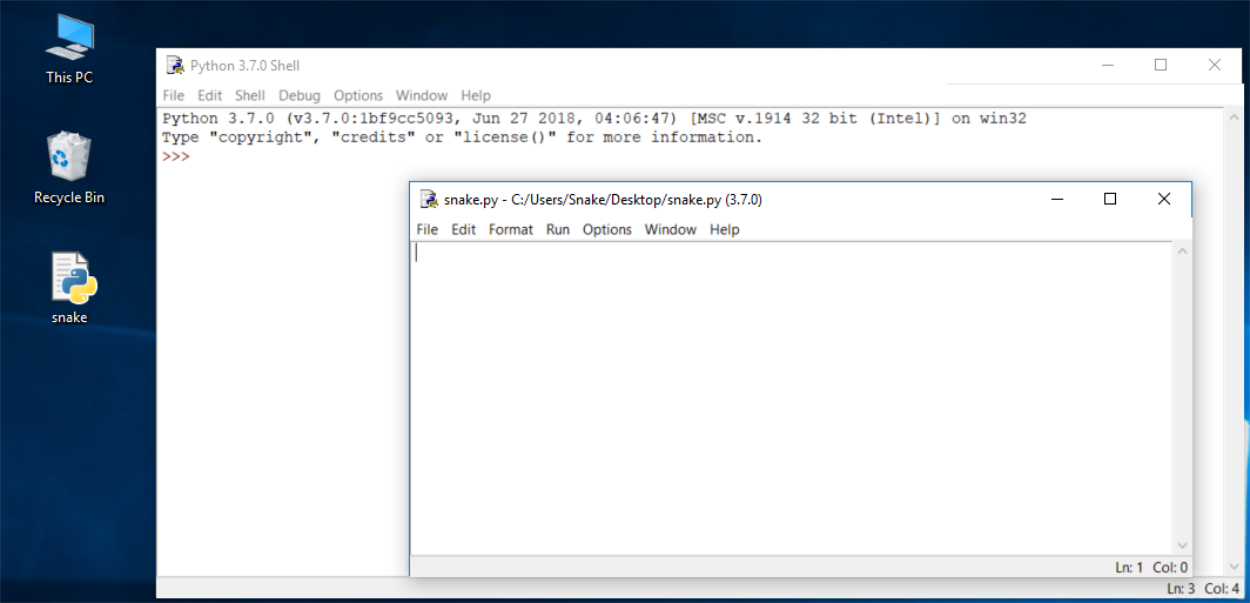


As you can see, IDLE opens a new window for you. You can use it to write and amend your code.

This is the **editor window**. Its only purpose is to be a workplace in which your source code is treated. Do not confuse the editor window with the shell window. They perform different functions.

The editor window is currently untitled, but it's good practice to start work by naming the source file.

Click *File* (in the new window), then click *Save as...*, select a folder for the new file (the desktop is a good place for your first programming attempts) and chose a name for the new file.



Note: don't set any extension for the file name you are going to use. Python needs its files to have the *.py* extension, so you should rely on the dialog window's defaults. Using the standard .py extension enables the OS to properly open these files.

# How to write and run your very first program

Now put just one line into your newly opened and named editor window.

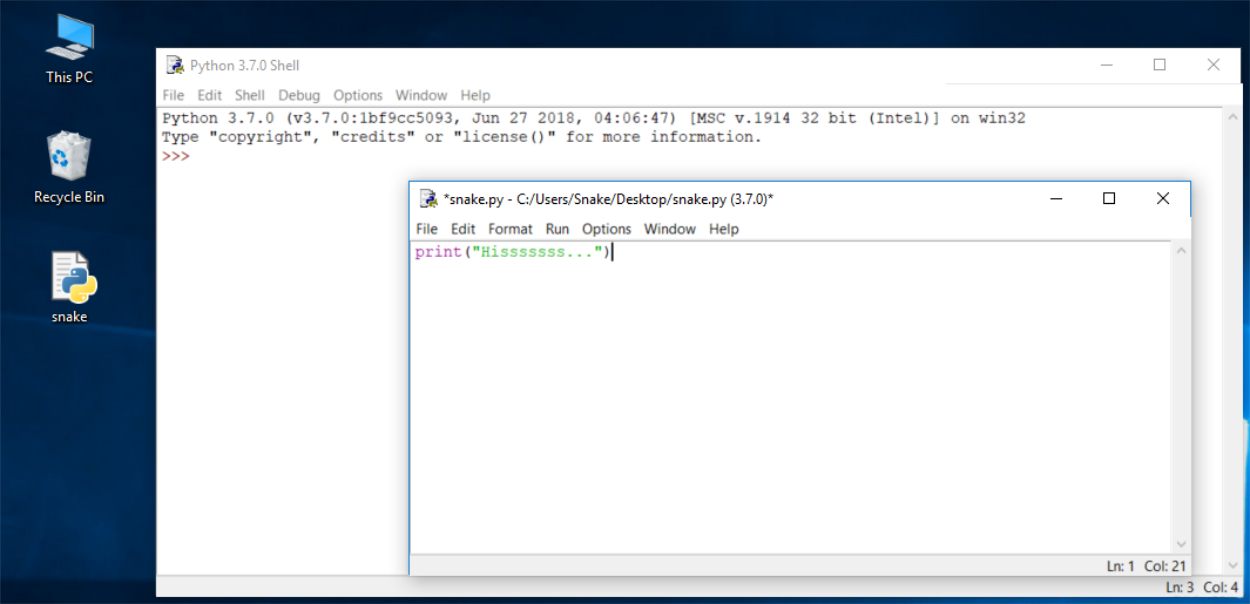
The line looks like this:

print("Hisssssss...")

You can use the clipboard to copy the text into the file.

We're not going to explain the meaning of the program right now. You'll find a detailed discussion in the next chapter.

Take a closer look at the quotation marks. These are the simplest form of quotation marks (neutral, straight, dumb, etc.) commonly used in source files. Do not try to use typographic quotes (curved, curly, smart, etc.), used by advanced text processors, as Python doesn’t accept them.



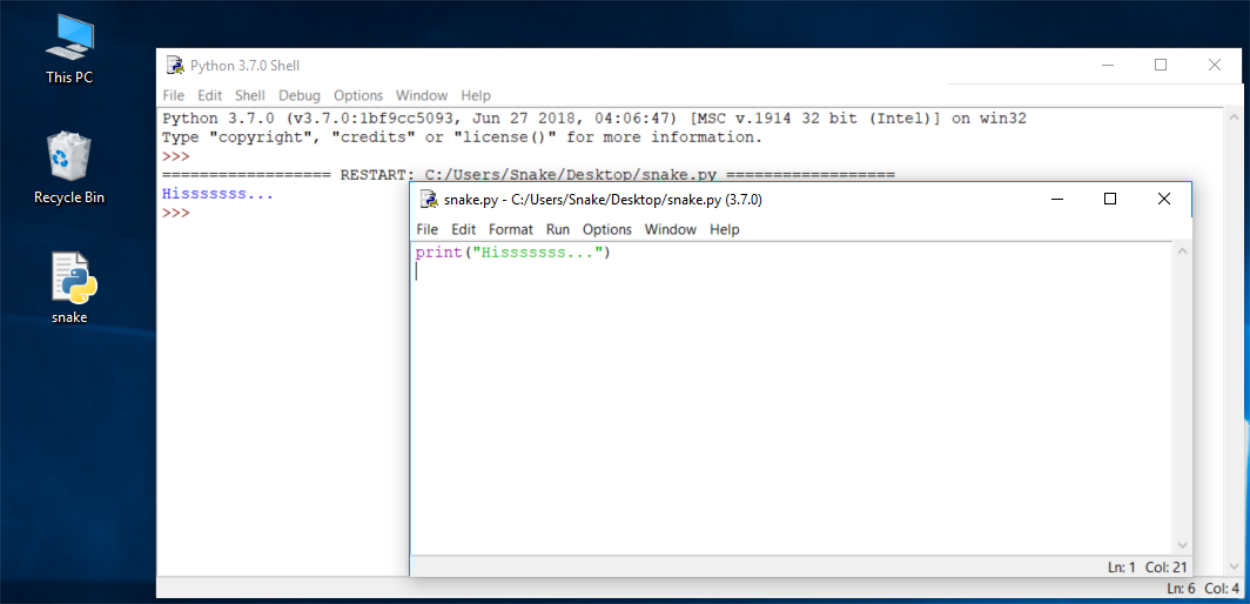
Save the file (*File* -> *Save*) and run the program (*Run* -> *Run Module*).

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

In this case, the program **hisses**.

Try to run it once again. And once more.

Now close both windows now and return to the desktop.



[Prev](https://edube.org/learn/python-essentials-1/begin-your-python-journey-10) [Next](https://edube.org/learn/python-essentials-1/begin-your-python-journey-12)

**How to spoil and fix your code**

Now start IDLE again.

* Click *File*, *Open*, point to the file you saved previously and let IDLE read it in.
* Try to run it again by pressing *F5* when the editor window is active.

As you can see, IDLE is able to save your code and retrieve it when you need it again.

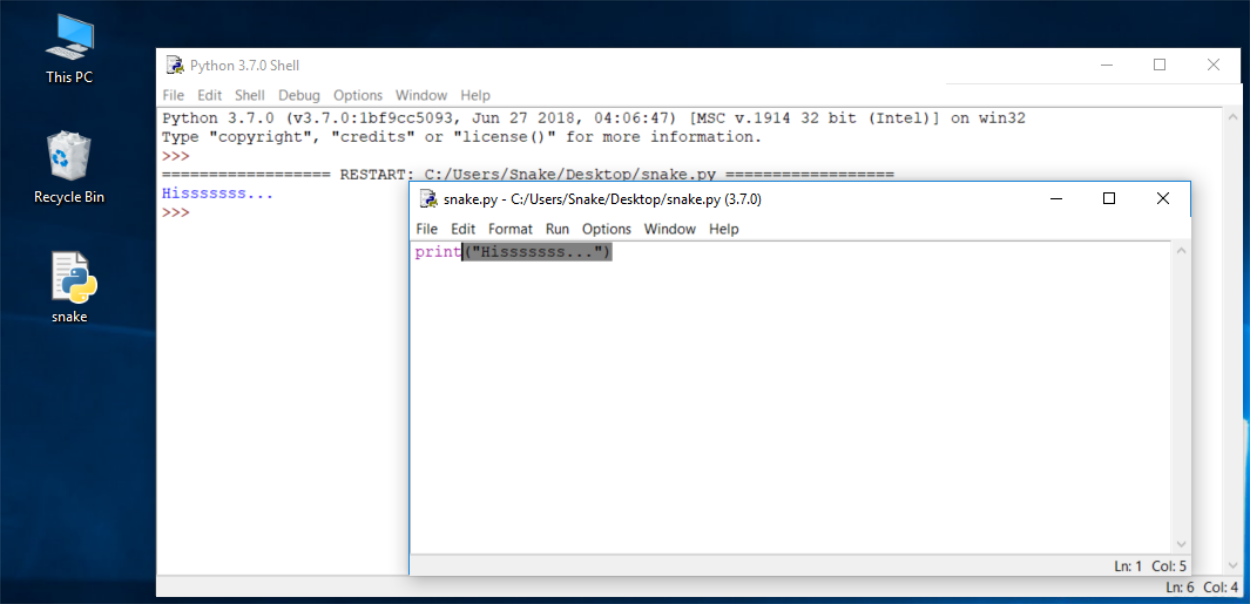
IDLE contains one additional and helpful feature.

* First, remove the closing parenthesis.
* Then enter the parenthesis again.

Your code should look like the one down here:

Hisssssss...

**output**



Every time you put the closing parenthesis in your program, IDLE will show the part of the text limited with a pair of corresponding parentheses. This helps you to remember to **place them in pairs.**

Remove the closing parenthesis again. The code becomes erroneous. It contains a syntax error now. IDLE should not let you run it.

Try to run the program again. IDLE will remind you to save the modified file. Follow the instructions.

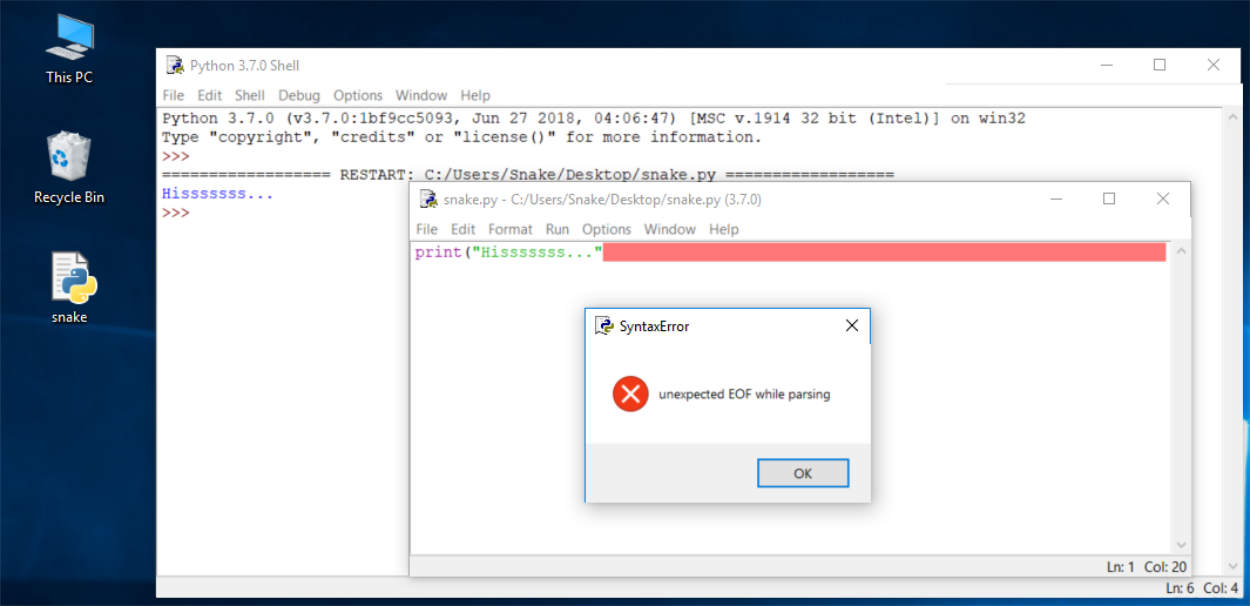
[Prev](https://edube.org/learn/python-essentials-1/begin-your-python-journey-11) [Next](https://edube.org/learn/python-essentials-1/begin-your-python-journey-13)

# How to spoil and fix your code

Watch all the windows carefully.

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

The editor window shows clearly where it happened.

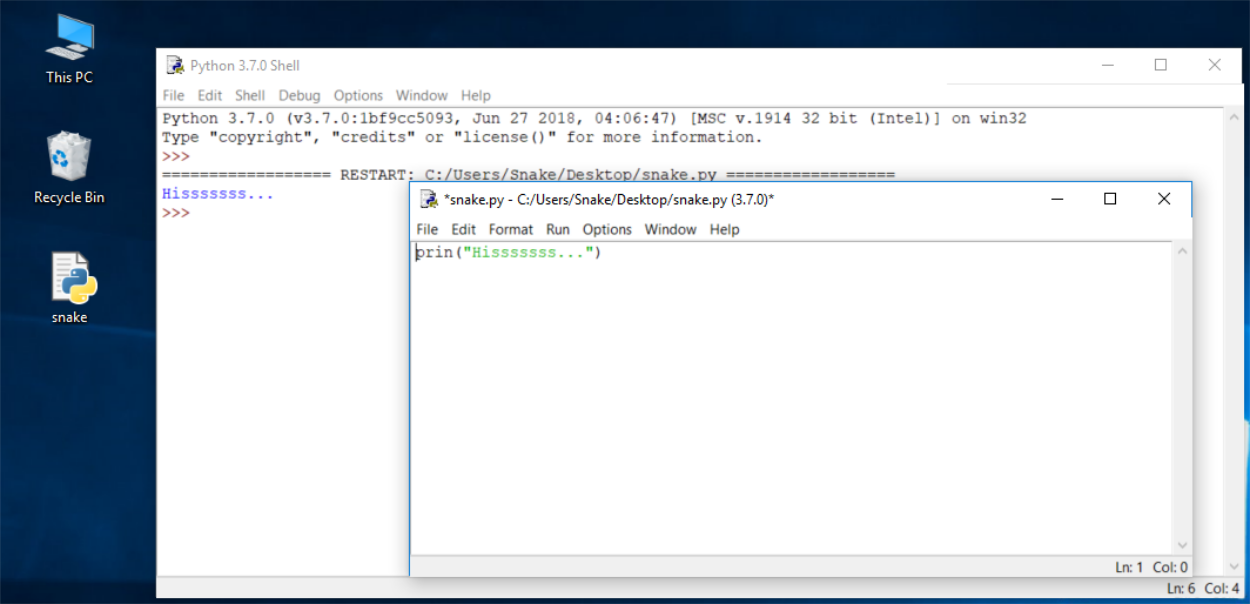


Fix the code now. It should look like this:

print("Hisssssss...")

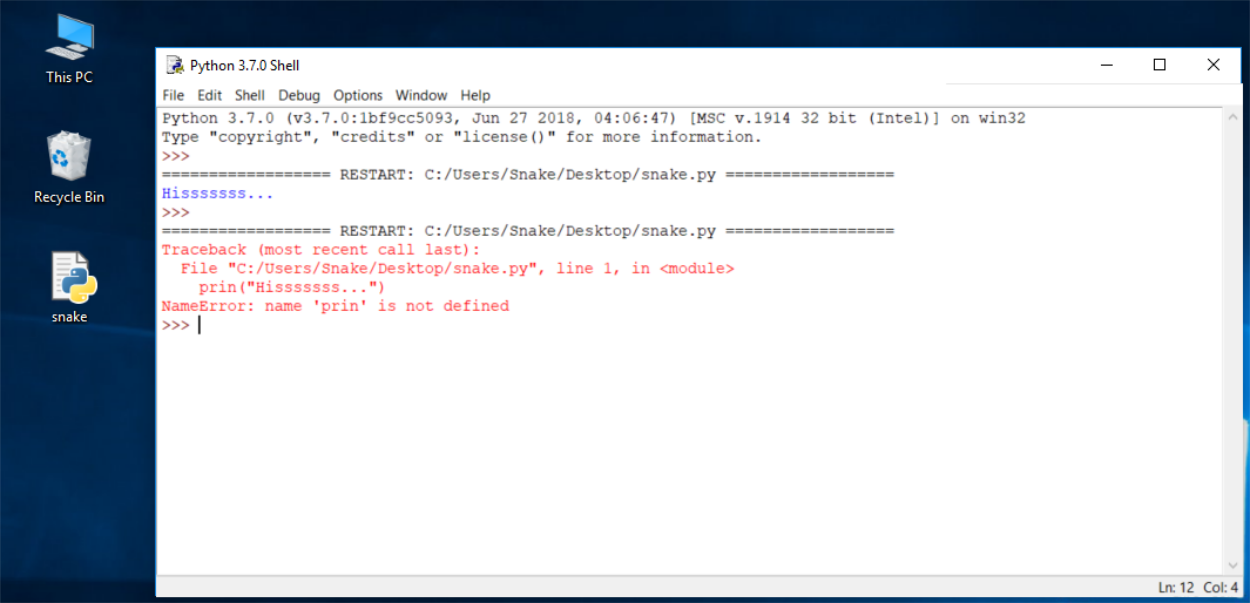
Run it to see if it "hisses" again.

Let's spoil the code one more time. Remove one letter from the word print. Run the code by pressing *F5*. As you can see, Python is not able to recognize the error.



**How to spoil and fix your code**

You may have noticed that the error message generated for the previous error is quite different from the first one.



This is because the nature of the error is **different** and the error is discovered at a **different stage** of interpretation.

The editor window will not provide any useful information regarding the error, but the console windows might.

The message (in red) shows (in the subsequent lines):

* the **traceback** (which is the path that the code traverses through different parts of the program - you can ignore it for now, as it is empty in such a simple code);
* the **location of the error** (the name of the file containing the error, line number and module name); note: the number may be misleading, as Python usually shows the place where it first notices the effects of the error, not necessarily the error itself;
* the **content of the erroneous line**; note: IDLE’s editor window doesn’t show line numbers, but it displays the current cursor location at the bottom-right corner; use it to locate the erroneous line in a long source code;
* the **name of the error** and a short explanation.

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

**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**

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

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.

# he 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)

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

print("The itsy bitsy spider climbed up the waterspout.")  
print("Down came the rain and washed the spider out.")



1

2

3

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

print("Down came the rain and washed the spider out.")

* **Console**

The itsy bitsy spider climbed up the waterspout.

Down came the rain and washed the spider out.

# 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".

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

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



1

2

3

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

print("Monty Python.")

* **Console**

My name is Python. Monty Python.

# 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")

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

## Scenario

Modify the first line of code in the editor, using the sep and end keywords, to match the expected output. Use the two print() functions in the editor.

Don't change anything in the second print() invocation.

## Expected output

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

print("Python")

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

print("Python")

# 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.

**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.

# Integers: octal and hexadecimal numbers

There are two additional conventions in Python that are unknown to the world of mathematics. The first allows us to use numbers in an **octal** representation.

If an integer number is preceded by an 0O or 0o prefix (zero-o), it will be treated as an octal value. This means that the number must contain digits taken from the [0..7] range only.

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

The print() function does the conversion automatically. Try this:

print(0o123)

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

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

print(0x123)

* [**Sandbox**](https://edube.org/sandbox)

## **Code**



1

print(0o123)

* **Console**

Traceback (most recent call last):

File "main.py", line 1, in <module>

print(O01)

NameError: name 'O01' is not defined

Traceback (most recent call last):

File "main.py", line 1, in <module>

print(O001)

NameError: name 'O001' is not defined

Traceback (most recent call last):

File "main.py", line 1, in <module>

print(O001)

NameError: name 'O001' is not defined

File "main.py", line 1

print(001)

^

SyntaxError: invalid token

Traceback (most recent call last):

File "main.py", line 1, in <module>

print(o001)

NameError: name 'o001' is not defined

Traceback (most recent call last):

File "main.py", line 1, in <module>

print(O001)

NameError: name 'O001' is not defined

83

# Floats

Now it's time to talk about another type, which is designed to represent and to store the numbers that (as a mathematician would say) have a **non-empty decimal fraction**.

They are the numbers that have (or may have) a fractional part after the decimal point, and although such a definition is very poor, it's certainly sufficient for what we wish to discuss.

Whenever we use a term like *two and a half* or *minus zero point four*, we think of numbers which the computer considers **floating-point** numbers:

2.5

-0.4

Note: *two and a half* looks normal when you write it in a program, although if your native language prefers to use a comma instead of a point in the number, you should ensure that your **number doesn't contain any commas** at all.

Python will not accept that, or (in very rare but possible cases) may misunderstand your intentions, as the comma itself has its own reserved meaning in Python.

If you want to use just a value of two and a half, you should write it as shown above. Note once again - there is a point between *2* and *5* - not a comma.

As you can probably imagine, the value of **zero point four** could be written in Python as:

0.4

But don't forget this simple rule - you can omit zero when it is the only digit in front of or after the decimal point.

In essence, you can write the value 0.4 as:

.4

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

4.

This will change neither its type nor its value.

**Ints vs. floats**

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

Look at these two numbers:

4

4.0

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

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

The point is what makes a float.

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

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

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

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

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

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

3E8

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

Note:

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

# Coding floats

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

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

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

6.62607E-34

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

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

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

0.0000000000000000000001

When you run this literal through Python:

print(0.0000000000000000000001)

this is the result:

1e-22

**output**

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

# Strings

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

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

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

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

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

I like "Monty Python"

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

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

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

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

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

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

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

This example will work too:

print('I like "Monty Python"')

# Coding strings

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

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

Try to print out a string containing the following message:

I'm Monty Python.

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

Check

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

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

An empty string still remains a string:

print("I\'m Monty Python\"")

**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?

**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**.

256

# List of priorities

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

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

Look at the table below:

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

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

Try to work through the following expression:

print(2\*3%5)

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

Check

# Operators and parentheses

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

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

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

An example of an expression with multiple parentheses is here:

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

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

**What are variables?**

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

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

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

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

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

What does every Python variable have?

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

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

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

You must also name them.

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

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



**orrect and incorrect variable names**

Note that the same restrictions apply to function names.

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

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

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

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

These variable names are also correct:

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

And now for some **incorrect names**:

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

**NOTE**

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

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

**Keywords**

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

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

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

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

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

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

import

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

Import

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

**Creating variables**

What can you put inside a variable?

Anything.

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

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

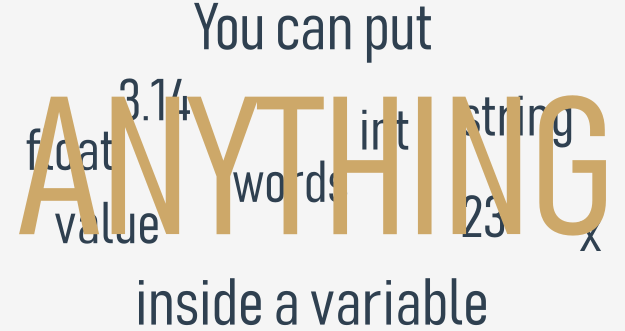
Let's talk now about two important things - **how variables are created**, and **how to put values inside them** (or rather - how to give or **pass values** to them).

**REMEMBER**

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

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

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



Take a look at the snippet:

var =1

print(var)

It consists of two simple instructions:

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

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

Check

# Assigning a new value to an already existing variable

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

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

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

Look at the code below:

var=1

print(var)

var=var+1

print(var)

The code sends two lines to the console:

1

2

**output**

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

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

We can say it shorter: assign 1 to var.

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

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

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

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

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

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

var=100

var=200+300

print(var)

Check

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

[Prev](https://edube.org/learn/python-essentials-1/variables-data-shaped-boxes-9) [Next](https://edube.org/learn/python-essentials-1/variables-data-shaped-boxes-11)

# Shortcut operators

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

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

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

x = x \* 2

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

sheep = sheep + 1

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

x \*= 2

sheep += 1

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

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

variable = variable op expression

It can be simplified and shown as follows:

variable op= expression

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

i = i + 2 \* j ⇒ i += 2 \* j

var = var / 2 ⇒ var /= 2

rem = rem % 10 ⇒ rem %= 10

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

x = x \*\* 2 ⇒ x \*\*= 2

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

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

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

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

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

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

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

Just like here:

# This program evaluates the hypotenuse c.

# a and b are the lengths of the legs.

a=3.0

b=4.0

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

print("c =",c)

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

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

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

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

# This is a test program.

x = 1

y = 2

# y = y + x

print(x + y)

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

**TIP**

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

**The input() function**

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

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

The new function gets data from it.

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

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

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

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

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

Take a look at our example:

print("Tell me anything...")

anything =input()

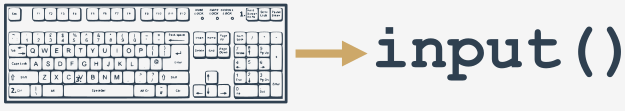
print("Hmm...", anything, "... Really?")

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

Note:

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

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



**The input() function with an argument**

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

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

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

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

Note:

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

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

**The result of the input() function**

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

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

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

anything = input("Enter a number: ")

something = anything \*\* 2.0

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

**Type casting**

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

Their names are self-commenting:

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

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

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

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

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

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

# 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

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

2 == 2.    Check

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

1 == 2    Check

# 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:

**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.

How does that statement work?

* If the true\_or\_not expression **represents the truth** (i.e., its value is not equal to zero), **the indented statement(s) will be executed**;
* if the true\_or\_not expression **does not represent the truth** (i.e., its value is equal to zero), **the indented statement(s) will be omitted** (ignored), and the next executed instruction will be the one after the original indentation level.

In real life, we often express a desire:

*if the weather is good, we'll go for a walk*

*then, we'll have lunch*

As you can see, having lunch is **not a conditional activity** and doesn't depend on the weather.

Knowing what conditions influence our behavior, and assuming that we have the parameterless functions go\_for\_a\_walk() and have\_lunch(), we can write the following snippet:

if the\_weather\_is\_good:

go\_for\_a\_walk()

have\_lunch()

# 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.

[Prev](https://edube.org/learn/python-essentials-1/making-decisions-in-python-11) [Next](https://edube.org/learn/python-essentials-1/making-decisions-in-python-13)

# 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()

eliftickets\_are\_available:

go\_to\_the\_theater()

eliftable\_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.

[Prev](https://edube.org/learn/python-essentials-1/making-decisions-in-python-12) [Next](https://edube.org/learn/python-essentials-1/making-decisions-in-python-14)

# Pseudocode and introduction to loops

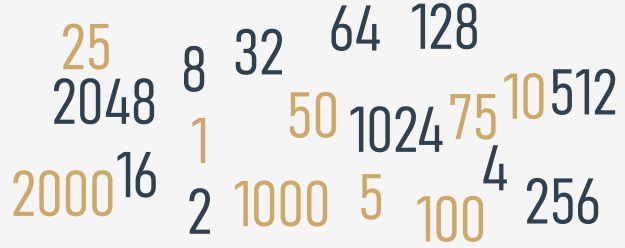
You should now be able to write a program which finds the largest of four, five, six, or even ten numbers.

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())

ifnumber==-1:

print(largest\_number)

exit()

ifnumber>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.

[Prev](https://edube.org/learn/python-essentials-1/making-decisions-in-python-14) [Next](https://edube.org/learn/python-essentials-1/lab-comparison-operators-and-conditional-execution-1)

**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: 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

[Prev](https://edube.org/learn/python-essentials-1/section-summary-1-2) [Next](https://edube.org/learn/python-essentials-1/loops-in-python-while-3)

# 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:

whileTrue:

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.

whilenumber!=-1:

# Is number larger than largest\_number?

ifnumber>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 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

whilei<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.

Take a look at the snippet below. Can you predict its output?

foriinrange(10):

print("The value of i is currently", i)

Run the code to check if you were right.

Note:

* the loop has been executed ten times (it's the range() function's argument)
* the last control variable's value is 9 (not 10, as **it starts from**0, not from 1)

The range() function invocation may be equipped with two arguments, not just one:

for i in range(2, 8):

print("The value of i is currently", i)

In this case, the first argument determines the initial (first) value of the control variable.

The last argument shows the first value the control variable will not be assigned.

Note: the range() function **accepts only integers as its arguments**, and generates sequences of integers.

Can you guess the output of the program? Run it to check if you were right now, too.

The first value shown is 2 (taken from the range()'s first argument.)

The last is 7 (although the range()'s second argument is 8).

**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.

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 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

whileTrue:

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

ifnumber==-1:

break

counter+=1

ifnumber>largest\_number:

largest\_number=number

ifcounter!=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: "))

whilenumber!=-1:

ifnumber==-1:

continue

counter+=1

ifnumber>largest\_number:

largest\_number=number

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

ifcounter:

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.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**



1

* **Console**

# 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

whilei<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.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

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



1

2

3

4

5

6

7

i=1

whilei<5:

print(i)

i+=1

else:

print("else:",i)

* **Console**

1

2

3

4

else: 5

# 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

foriinrange(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**.

# 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

# Computer 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 |

[Prev](https://edube.org/learn/python-essentials-1/section-summary-2-2-2) [Next](https://edube.org/learn/python-essentials-1/logic-and-bit-operations-in-python-and-or-not-3)

# 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(pandq)==(notp)or(notq)

not(porq)==(notp)and(notq)

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=notnoti

# 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 1s 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).

[Prev](https://edube.org/learn/python-essentials-1/logic-and-bit-operations-in-python-and-or-not-2) [Next](https://edube.org/learn/python-essentials-1/logic-and-bit-operations-in-python-1)

# 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=iandj

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=noti

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 = 0000000000000000000000000000x000

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
* 0000000000000000000000000000000 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 23 = 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 ihere it is:

ifflag\_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

[Prev](https://edube.org/learn/python-essentials-1/logic-and-bit-operations-in-python-1) [Next](https://edube.org/learn/python-essentials-1/logic-and-bit-operations-in-python-bit-shifting-1)

**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 | =, +=, -=, \*=, /=, %=, &=, ^=, |=, >>=, <<= |  |

**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**.

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.

# 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!

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

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



1

2

3

4

numbers=[111,7,2,1]

print(numbers[-1])

print(numbers[-2])

* **Console**

1

2

# Functions vs. methods

A **method is a specific kind of function** - it behaves like a function and looks like a function, but differs in the way in which it acts, and in its invocation style.

A **function doesn't belong to any data** - it gets data, it may create new data and it (generally) produces a result.

A method does all these things, but is also able to **change the state of a selected entity**.

**A method is owned by the data it works for, while a function is owned by the whole code**.

This also means that invoking a method requires some specification of the data from which the method is invoked.

It may sound puzzling here, but we'll deal with it in depth when we delve into object-oriented programming.

In general, a typical function invocation may look like this:

result = function(arg)

The function takes an argument, does something, and returns a result.

A typical method invocation usually looks like this:

result = data.method(arg)

Note: the name of the method is preceded by the name of the data which owns the method. Next, you add a **dot**, followed by the **method name**, and a pair of **parenthesis enclosing the arguments**.

The method will behave like a function, but can do something more - it can **change the internal state of the data** from which it has been invoked.

You may ask: why are we talking about methods, not about lists?

This is an essential issue right now, as we're going to show you how to add new elements to an existing list. This can be done with methods owned by all the lists, not by functions.

# Key takeaways

1. The **list is a type of data** in Python used to **store multiple objects**. It is an **ordered and mutable collection** of comma-separated items between square brackets, e.g.:

my\_list=[1,None,True,"I am a string",256,0]

2. Lists can be **indexed and updated**, e.g.:

my\_list=[1,None,True,'I am a string',256,0]

print(my\_list[3])# outputs: I am a string

print(my\_list[-1])# outputs: 0

my\_list[1]='?'

print(my\_list)# outputs: [1, '?', True, 'I am a string', 256, 0]

my\_list.insert(0,"first")

my\_list.append("last")

print(my\_list)# outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']

3. Lists can be **nested**, e.g.:

my\_list=[1,'a',["list",64,[0,1],False]]

You will learn more about nesting in module 3.1.7 - for the time being, we just want you to be aware that something like this is possible, too.

4. List elements and lists can be **deleted**, e.g.:

my\_list=[1,2,3,4]

delmy\_list[2]

print(my\_list)# outputs: [1, 2, 4]

delmy\_list# deletes the whole list

Again, you will learn more about this in module 3.1.6 - don't worry. For the time being just try to experiment with the above code and check how changing it affects the output.

5. Lists can be **iterated** through using the for loop, e.g.:

my\_list=["white","purple","blue","yellow","green"]

forcolorinmy\_list:

print(color)

6. The len() function may be used to **check the list's length**, e.g.:

my\_list=["white","purple","blue","yellow","green"]

print(len(my\_list))# outputs 5

delmy\_list[2]

print(len(my\_list))# outputs 4

7. A typical **function** invocation looks as follows: result = function(arg), while a typical **method** invocation looks like this:result = data.method(arg).

**Exercise 1**

What is the output of the following snippet?

lst = [1, 2, 3, 4, 5]

lst.insert(1, 6)

del lst[0]

lst.append(1)

print(lst)

Check

**Exercise 2**

What is the output of the following snippet?

lst = [1, 2, 3, 4, 5]

lst\_2 = []

add = 0

for number in lst:

add += number

lst\_2.append(add)

print(lst\_2)

Check

**Exercise 3**

What happens when you run the following snippet?

lst = []

del lst

print(lst)

Check

**Exercise 4**

What is the output of the following snippet?

lst = [1, [2, 3], 4]

print(lst[1])

print(len(lst))

Check

**The bubble sort**

Now that you can effectively juggle the elements of lists, it's time to learn how to **sort** them. Many sorting algorithms have been invented so far, which differ a lot in speed, as well as in complexity. We are going to show you a very simple algorithm, easy to understand, but unfortunately not too efficient, either. It's used very rarely, and certainly not for large and extensive lists.

Let's say that a list can be sorted in two ways:

* increasing (or more precisely - non-decreasing) - if in every pair of adjacent elements, the former element is not greater than the latter;
* decreasing (or more precisely - non-increasing) - if in every pair of adjacent elements, the former element is not less than the latter.

In the following sections, we'll sort the list in increasing order, so that the numbers will be ordered from the smallest to the largest.

Here's the list:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 10 | 6 | 2 | 4 |

We'll try to use the following approach: we'll take the first and the second elements and compare them; if we determine that they're in the wrong order (i.e., the first is greater than the second), we'll swap them round; if their order is valid, we'll do nothing. A glance at our list confirms the latter - the elements 01 and 02 are in the proper order, as in 8 < 10.

Now look at the second and the third elements. They're in the wrong positions. We have to swap them:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | **6** | **10** | 2 | 4 |

We go further, and look at the third and the fourth elements. Again, this is not what it's supposed to be like. We have to swap them:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 6 | **2** | **10** | 4 |

Now we check the fourth and the fifth elements. Yes, they too are in the wrong positions. Another swap occurs:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 8 | 6 | 2 | **4** | **10** |

The first pass through the list is already finished. We're still far from finishing our job, but something curious has happened in the meantime. The largest element, 10, has already gone to the end of the list. Note that this is the **desired place** for it. All the remaining elements form a picturesque mess, but this one is already in place.

Now, for a moment, try to imagine the list in a slightly different way - namely, like this:

|  |
| --- |
| 10 |
| 4 |
| 2 |
| 6 |
| 8 |

Look - 10 is at the top. We could say that it floated up from the bottom to the surface, just like the **bubble in a glass of champagne**. The sorting method derives its name from the same observation - it's called a **bubble sort**.

Now we start with the second pass through the list. We look at the first and second elements - a swap is necessary:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **6** | **8** | 2 | 4 | 10 |

Time for the second and third elements: we have to swap them too:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | **2** | **8** | 4 | 10 |

Now the third and fourth elements, and the second pass is finished, as 8 is already in place:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 2 | **4** | **8** | 10 |

We start the next pass immediately. Watch the first and the second elements carefully - another swap is needed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **6** | 4 | 8 | 10 |

Now 6 needs to go into place. We swap the second and the third elements:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | **4** | **6** | 8 | 10 |

The list is already sorted. We have nothing more to do. This is exactly what we want.

As you can see, the essence of this algorithm is simple: **we compare the adjacent elements, and by swapping some of them, we achieve our goal**.

Let's code in Python all the actions performed during a single pass through the list, and then we'll consider how many passes we actually need to perform it. We haven't explained this so far, and we'll do that a little later.

# Sorting a list

How many passes do we need to sort the entire list?

We solve this issue in the following way: **we introduce another variable**; its task is to observe if any swap has been done during the pass or not; if there is no swap, then the list is already sorted, and nothing more has to be done. We create a variable named swapped, and we assign a value of False to it, to indicate that there are no swaps. Otherwise, it will be assigned True.

my\_list=[8,10,6,2,4]# list to sort

foriinrange(len(my\_list)-1):# we need (5 - 1) comparisons

ifmy\_list[i]>my\_list[i+1]:# compare adjacent elements

my\_list[i],my\_list[i+1]=my\_list[i+1],my\_list[i]# If we end up here, we have to swap the elements.

You should be able to read and understand this program without any problems:

my\_list=[8,10,6,2,4]# list to sort

swapped=True# It's a little fake, we need it to enter the while loop.

whileswapped:

swapped=False# no swaps so far

foriinrange(len(my\_list)-1):

ifmy\_list[i]>my\_list[i+1]:

swapped=True# a swap occurred!

my\_list[i],my\_list[i+1]=my\_list[i+1],my\_list[i]

print(my\_list)

Run the program and test it.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**



1

* **Console**

my\_list = [8, 10, 6, 2, 4] # list to sort

swapped = True # It's a little fake, we need it to enter the while loop.

while swapped:

swapped = False # no swaps so far

for i in range(len(my\_list) - 1):

if my\_list[i] >my\_list[i + 1]:

swapped = True # a swap occurred!

my\_list[i], my\_list[i + 1] = my\_list[i + 1], my\_list[i]

print(my\_list)

my\_list = []

swapped = True

num = int(input("How many elements do you want to sort: "))

for i in range(num):

val = float(input("Enter a list element: "))

my\_list.append(val)

while swapped:

swapped = False

for i in range(len(my\_list) - 1):

if my\_list[i] >my\_list[i + 1]:

swapped = True

my\_list[i], my\_list[i + 1] = my\_list[i + 1], my\_list[i]

print("\nSorted:")

print(my\_list)

# Key takeaways

1. You can use the sort() method to sort elements of a list, e.g.:

lst=[5,3,1,2,4]

print(lst)

lst.sort()

print(lst)# outputs: [1, 2, 3, 4, 5]

2. There is also a list method called reverse(), which you can use to reverse the list, e.g.:

lst=[5,3,1,2,4]

print(lst)

lst.reverse()

print(lst)# outputs: [4, 2, 1, 3, 5]

**Exercise 1**

What is the output of the following snippet?

lst = ["D", "F", "A", "Z"]

lst.sort()

print(lst)

Check

**Exercise 2**

What is the output of the following snippet?

a = 3

b = 1

c = 2

lst = [a, c, b]

lst.sort()

print(lst)

Check

**Exercise 3**

What is the output of the following snippet?

a = "A"

b = "B"

c = "C"

d = " "

lst = [a, b, c, d]

lst.reverse()

print(lst)

**The inner life of lists**

Now we want to show you one important, and very surprising, feature of lists, which strongly distinguishes them from ordinary variables.

We want you to memorize it - it may affect your future programs, and cause severe problems if forgotten or overlooked.

Take a look at the snippet in the editor.

The program:

* creates a one-element list named list\_1;
* assigns it to a new list named list\_2;
* changes the only element of list\_1;
* prints out list\_2.

The surprising part is the fact that the program will output: [2], not [1], which seems to be the obvious solution.

Lists (and many other complex Python entities) are stored in different ways than ordinary (scalar) variables.

You could say that:

* the name of an ordinary variable is the **name of its content**;
* the name of a list is the name of a **memory location where the list is stored**.

Read these two lines once more - the difference is essential for understanding what we are going to talk about next.

The assignment: list\_2 = list\_1 copies the name of the array, not its contents. In effect, the two names (list\_1 and list\_2) identify the same location in the computer memory. Modifying one of them affects the other, and vice versa.

How do you cope with that?

# Powerful slices

Fortunately, the solution is at your fingertips - its name is the **slice**.

A slice is an element of Python syntax that allows you to **make a brand new copy of a list, or parts of a list**.

It actually copies the list's contents, not the list's name.

This is exactly what you need. Take a look at the snippet below:

list\_1 = [1]

list\_2 = list\_1[:]

list\_1[0] = 2

print(list\_2)

Its output is [1].

This inconspicuous part of the code described as [:] is able to produce a brand new list.

One of the most general forms of the slice looks as follows:

my\_list[start:end]

As you can see, it resembles indexing, but the colon inside makes a big difference.

A slice of this form **makes a new (target) list, taking elements from the source list - the elements of the indices from start to**end - 1.

Note: not to end but to end - 1. An element with an index equal to end is the first element which **does not take part in the slicing**.

Using negative values for both start and end is possible (just like in indexing).

Take a look at the snippet:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[1:3]

print(new\_list)

The new\_list list will have end - start (3 - 1 = 2) elements - the ones with indices equal to 1 and 2 (but not 3).

The snippet's output is: [8, 6]

# Copying the entire list.

list\_1 = [1]

list\_2 = list\_1[:]

list\_1[0] = 2

print(list\_2)

# Copying some part of the list.

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[1:3]

print(new\_list)

# Slices: continued

If you omit the start in your slice, it is assumed that you want to get a slice beginning at the element with index 0.

In other words, the slice of this form:

my\_list[:end]

is a more compact equivalent of:

my\_list[0:end]

Look at the snippet below:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[:3]

print(new\_list)

This is why its output is: [10, 8, 6].

Similarly, if you omit the end in your slice, it is assumed that you want the slice to end at the element with the index len(my\_list).

In other words, the slice of this form:

my\_list[start:]

is a more compact equivalent of:

my\_list[start:len(my\_list)]

Look at the following snippet:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[3:]

print(new\_list)

Its output is therefore: [4, 2].

# Slices: continued

As we've said before, omitting both start and end makes a **copy of the whole list**:

my\_list = [10, 8, 6, 4, 2]

new\_list = my\_list[:]

print(new\_list)

The snippet's output is: [10, 8, 6, 4, 2].

The previously described del instruction is able to **delete more than just a list's element at once - it can delete slices too**:

my\_list = [10, 8, 6, 4, 2]

del my\_list[1:3]

print(my\_list)

Note: in this case, the slice **doesn't produce any new list**!

The snippet's output is: [10, 4, 2].

Deleting **all the elements** at once is possible too:

my\_list = [10, 8, 6, 4, 2]

del my\_list[:]

print(my\_list)

The list becomes empty, and the output is: [].

Removing the slice from the code changes its meaning dramatically.

Take a look:

my\_list = [10, 8, 6, 4, 2]

del my\_list

print(my\_list)

The del instruction will **delete the list itself, not its content**.

The print() function invocation from the last line of the code will then cause a runtime error.

my\_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]

largest = my\_list[0]

for i in range(1, len(my\_list)):

if my\_list[i] > largest:

largest = my\_list[i]

print(largest)

my\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

to\_find = 5

found = False

for i in range(len(my\_list)):

found = my\_list[i] == to\_find

if found:

break

if found:

print("Element found at index", i)

else:

print("absent")

# lists in lists

Lists can consist of scalars (namely numbers) and elements of a much more complex structure (you've already seen such examples as strings, booleans, or even other lists in the previous Section Summary lessons). Let's have a closer look at the case where a **list's elements are just lists**.

We often find such **arrays** in our lives. Probably the best example of this is a **chessboard**.

A chessboard is composed of rows and columns. There are eight rows and eight columns. Each column is marked with the letters A through H. Each line is marked with a number from one to eight.

The location of each field is identified by letter-digit pairs. Thus, we know that the bottom left corner of the board (the one with the white rook) is A1, while the opposite corner is H8.

Let's assume that we're able to use the selected numbers to represent any chess piece. We can also assume that **every row on the chessboard is a list**.

Look at the code below:

row=[]

foriinrange(8):

row.append(WHITE\_PAWN)

It builds a list containing eight elements representing the second row of the chessboard - the one filled with pawns (assume that WHITE\_PAWN is a **predefined symbol** representing a white pawn).

The same effect may be achieved by means of a **list comprehension**, the special syntax used by Python in order to fill massive lists.

A list comprehension is actually a list, but **created on-the-fly during program execution, and is not described statically**.

Take a look at the snippet:

row=[WHITE\_PAWNforiinrange(8)]

The part of the code placed inside the brackets specifies:

* the data to be used to fill the list (WHITE\_PAWN)
* the clause specifying how many times the data occurs inside the list (for i in range(8))

Let us show you some other **list comprehension examples**:

Example #1:

squares=[x\*\*2forxinrange(10)]

The snippet produces a ten-element list filled with squares of ten integer numbers starting from zero (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

Example #2:

twos=[2\*\*iforiinrange(8)]

The snippet creates an eight-element array containing the first eight powers of two (1, 2, 4, 8, 16, 32, 64, 128)

Example #3:

odds=[xforxinsquaresifx%2!=0]

The snippet makes a list with only the odd elements of the squares list.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**



1

* **Console**

**Lists in lists: two-dimensional arrays**

Let's also assume that a **predefined symbol** named EMPTY designates an empty field on the chessboard.

So, if we want to create a list of lists representing the whole chessboard, it may be done in the following way:

board = []

foriinrange(8):

row = [EMPTY foriinrange(8)]

board.append(row)

Note:

* the inner part of the loop creates a row consisting of eight elements (each of them equal to EMPTY) and appends it to the board list;
* the outer part repeats it eight times;
* in total, the board list consists of 64 elements (all equal to EMPTY)

This model perfectly mimics the real chessboard, which is in fact an eight-element list of elements, all being single rows. Let's summarize our observations:

* the elements of the rows are fields, eight of them per row;
* the elements of the chessboard are rows, eight of them per chessboard.

The board variable is now a **two-dimensional array**. It's also called, by analogy to algebraic terms, a **matrix**.

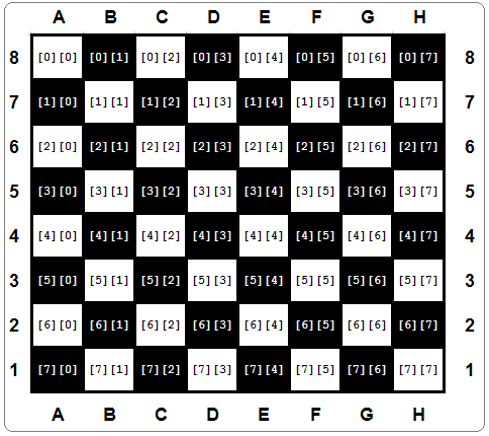
As list comprehensions can be **nested**, we can shorten the board creation in the following way:

board = [[EMPTY foriinrange(8)] for j inrange(8)]

# Lists in lists: two-dimensional arrays - continued

Access to the selected field of the board requires two indices - the first selects the row; the second - the field number inside the row, which is de facto a column number.

Take a look at the chessboard. Every field contains a pair of indices which should be given to access the field's content:



Glancing at the figure shown above, let's set some chess pieces on the board. First, let's add all the rooks:

board[0][0]=ROOK

board[0][7]=ROOK

board[7][0]=ROOK

board[7][7]=ROOK

If you want to add a knight to C4, you do it as follows:

board[4][2]=KNIGHT

And now a pawn to E5:

board[3][4]=PAWN

EMPTY = "-"

ROOK = "ROOK"

board = []

for i in range(8):

row = [EMPTY for i in range(8)]

board.append(row)

board[0][0] = ROOK

board[0][7] = ROOK

board[7][0] = ROOK

board[7][7] = ROOK

print(board)

**Multidimensional nature of lists: advanced applications**

Let's go deeper into the multidimensional nature of lists. To find any element of a two-dimensional list, you have to use two *coordinates*:

* a vertical one (row number)
* and a horizontal one (column number).

Imagine that you develop a piece of software for an automatic weather station. The device records the air temperature on an hourly basis and does it throughout the month. This gives you a total of 24 × 31 = 744 values. Let's try to design a list capable of storing all these results.

First, you have to decide which data type would be adequate for this application. In this case, a float would be best, since this thermometer is able to measure the temperature with an accuracy of 0.1 ℃.

Then you take an arbitrary decision that the rows will record the readings every hour on the hour (so the row will have 24 elements) and each of the rows will be assigned to one day of the month (let's assume that each month has 31 days, so you need 31 rows). Here's the appropriate pair of comprehensions (h is for hour, d for day):

temps = [[0.0for h inrange(24)] for dinrange(31)]

The whole matrix is filled with zeros now. You can assume that it's updated automatically using special hardware agents. The thing you have to do is to wait for the matrix to be filled with measurements.

Now it's time to determine the monthly average noon temperature. Add up all 31 readings recorded at noon and divide the sum by 31. You can assume that the midnight temperature is stored first. Here's the relevant code:

temps = [[0.0for h inrange(24)] for dinrange(31)]

#

# The matrix is magically updated here.

#

total =0.0

for day in temps:

total +=day[11]

average = total /31

print("Average temperature at noon:", average)

Note: the day variable used by the for loop is not a scalar - each pass through the temps matrix assigns it with the subsequent rows of the matrix; hence, it's a list. It has to be indexed with 11 to access the temperature value measured at noon.

Now find the highest temperature during the whole month - see the code:

temps = [[0.0for h inrange(24)] for dinrange(31)]

#

# The matrix is magically updated here.

#

highest =-100.0

for day in temps:

for temp in day:

if temp > highest:

highest = temp

print("The highest temperature was:", highest)

Note:

* the day variable iterates through all the rows in the temps matrix;
* the temp variable iterates through all the measurements taken in one day.

Now count the days when the temperature at noon was at least 20 ℃:

temps = [[0.0for h inrange(24)] for dinrange(31)]

#

# The matrix is magically updated here.

#

hot\_days=0

for day in temps:

ifday[11] >20.0:

hot\_days+=1

print(hot\_days, "days were hot.")

# Three-dimensional arrays

Python does not limit the depth of list-in-list inclusion. Here you can see an example of a three-dimensional array:

Imagine a hotel. It's a huge hotel consisting of three buildings, 15 floors each. There are 20 rooms on each floor. For this, you need an array which can collect and process information on the occupied/free rooms.

First step - the type of the array's elements. In this case, a Boolean value (True/False) would fit.

Step two - calm analysis of the situation. Summarize the available information: three buildings, 15 floors, 20 rooms.

Now you can create the array:

rooms=[[[Falseforrinrange(20)]forfinrange(15)]fortinrange(3)]

The first index (0 through 2) selects one of the buildings; the second (0 through 14) selects the floor, the third (0 through 19) selects the room number. All rooms are initially free.

Now you can book a room for two newlyweds: in the second building, on the tenth floor, room 14:

rooms[1][9][13]=True

and release the second room on the fifth floor located in the first building:

rooms[0][4][1]=False

Check if there are any vacancies on the 15th floor of the third building:

vacancy=0

forroom\_numberinrange(20):

ifnotrooms[2][14][room\_number]:

vacancy+=1

The vacancy variable contains 0 if all the rooms are occupied, or the number of available rooms otherwise.

# 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.

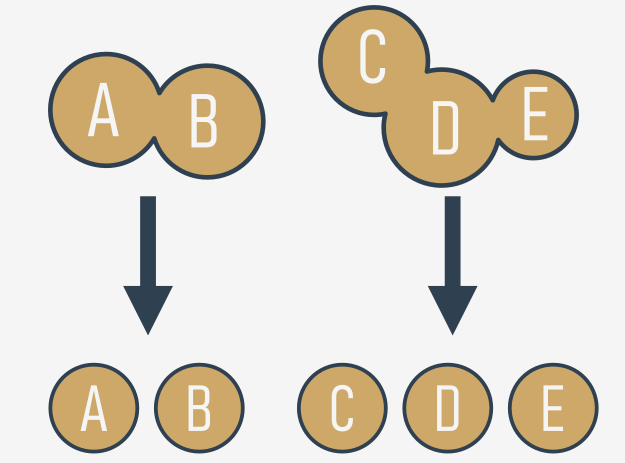
We can now define the first condition which can help you decide when to start writing your own functions: **if a particular fragment of the code begins to appear in more than one place, consider the possibility of isolating it in the form of a function** invoked from the points where the original code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

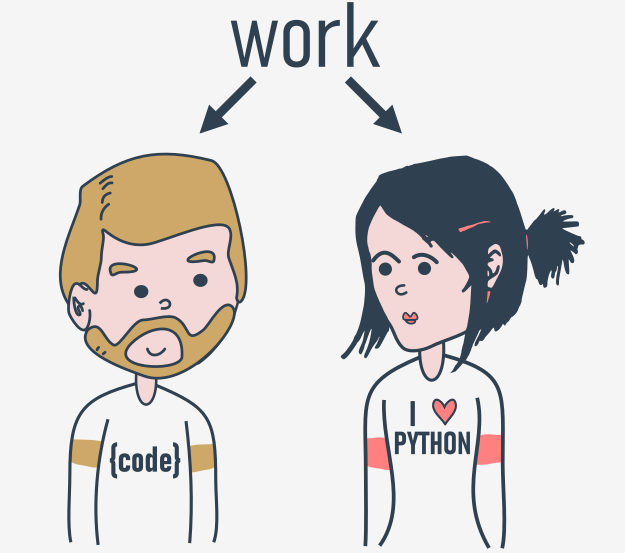
This considerably simplifies the work of the program, because each piece of code can be encoded separately, and tested separately. The process described here is often called **decomposition**.



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**.

# 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.

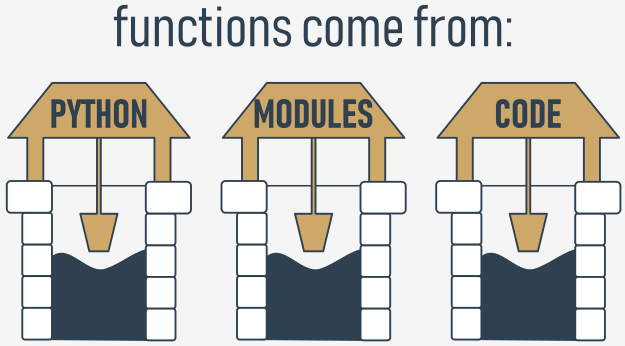
Each of them writes a clearly defined and described set of functions, which when **combined into the module** (we'll tell you about this a bit later) will give the final product.

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?

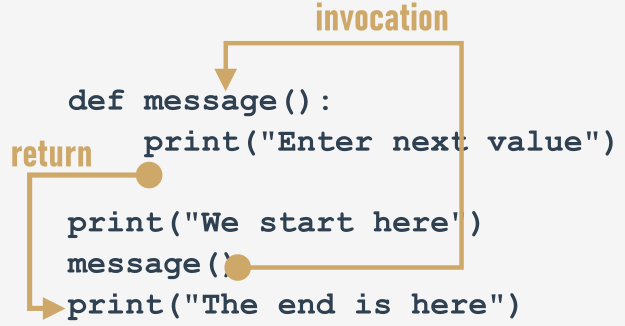
In general, functions come from at least three places:

* 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.



**ow 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.")

defmessage():

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:

defmessage():

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.

[Prev](https://edube.org/learn/python-essentials-1/writing-functions-4) [Next](https://edube.org/learn/python-essentials-1/section-summary-14)

**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.

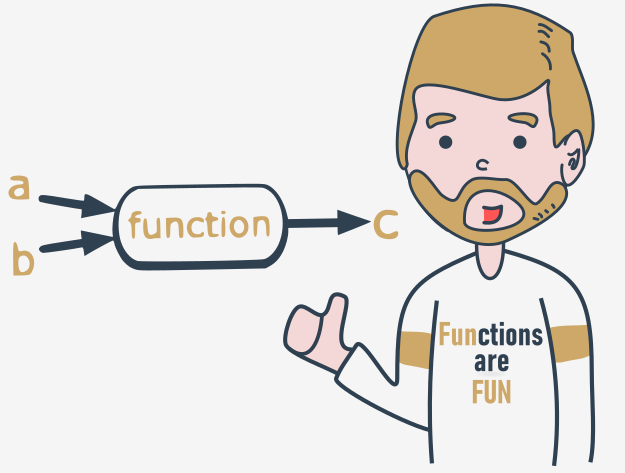
A value for the parameter will arrive from the function's environment.

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:

defmessage(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 numbernumberPositional parameter passing

A technique which assigns the ith (first, second, and so on) argument to the ith (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.

defmy\_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:

defintroduction(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:

defintroduction(first\_name,last\_name):

print("Hello, my name is",first\_name,last\_name)

introduction("Skywalker","Luke")

introduction("Quick","Jesse")

introduction("Kent","Clark")

# Keyword 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:

defintroduction(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")

**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:

defadding(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?

# 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:

defhappy\_new\_year(wishes=True):

print("Three...")

print("Two...")

print("One...")

ifnotwishes:

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:

defboring\_function():

return123

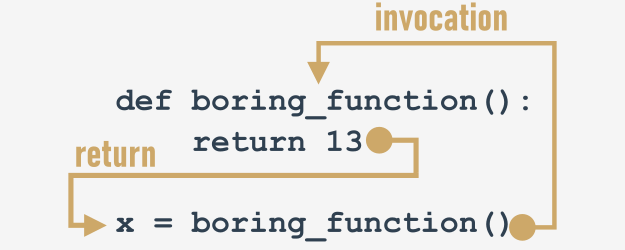
x=boring\_function()

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):

defboring\_function():

print("'Boredom Mode' ON.")

return123

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.

[Prev](https://edube.org/learn/python-essentials-1/section-summary-15) [Next](https://edube.org/learn/python-essentials-1/returning-a-result-from-a-function-6)

# Effects and results: lists and functions

There are two additional questions that should be answered here.

The first is: **may a list be sent to a function as an argument?**

Of course it may! Any entity recognizable by Python can play the role of a function argument, although it has to be assured that the function is able to cope with it.

So, if you pass a list to a function, the function has to handle it like a list.

A function like this one here:

deflist\_sum(lst):

s=0

foreleminlst:

s+=elem

returns

and invoked like this:

print(list\_sum([5,4,3]))

will return 12 as a result, but you should expect problems if you invoke it in this risky way:

print(list\_sum(5))

# Functions and scopes: the global keyword

Hopefully, you should now have arrived at the following question: does this mean that a function is not able to modify a variable defined outside it? This would create a lot of discomfort.

Fortunately, the answer is *no*.

There's a special Python method which can **extend a variable's scope in a way which includes the functions' bodies** (even if you want not only to read the values, but also to modify them).

Such an effect is caused by a keyword named global:

globalname

globalname1,name2, ...

Using this keyword inside a function with the name (or names separated with commas) of a variable(s), forces Python to refrain from creating a new variable inside the function - the one accessible from outside will be used instead.

In other words, this name becomes global (it has a **global scope**, and it doesn't matter whether it's the subject of read or assign).

Look at the code in the editor.

We've added global to the function.

# Some simple functions: recursion

There's one more thing we want to show you to make everything complete - it's **recursion**.

This term may describe many different concepts, but one of them is especially interesting - the one referring to computer programming.

In this field, recursion is a **technique where a function invokes itself**.

These two cases seem to be the best to illustrate the phenomenon - factorials and Fibonacci numbers. Especially the latter.

**The Fibonacci numbers definition is a clear example of recursion**. We already told you that:

**Fibi = Fibi-1 + Fibi-2**

The definition of the ith number refers to the i-1 number, and so on, till you reach the first two.

Can it be used in the code? Yes, it can. It can also make the code shorter and clearer.

The second version of our fib() function makes direct use of this definition:

deffib(n):

ifn<1:

returnNone

ifn<3:

return1

returnfib(n-1)+fib(n-2)

The code is much clearer now.

But is it really safe? Does it entail any risk?

Yes, there is a little risk indeed. **If you forget to consider the conditions which can stop the chain of recursive invocations, the program may enter an infinite loop**. You have to be careful.

The factorial has a second, **recursive** side too. Look:

n! = 1 × 2 × 3 × ... × n-1 × n

It's obvious that:

1 × 2 × 3 × ... × n-1 = (n-1)!

So, finally, the result is:

n! = (n-1)! × n

This is in fact a ready recipe for our new solution.

Here it is:

deffactorial\_function(n):

ifn<0:

returnNone

ifn<2:

return1

returnn\*factorial\_function(n-1)

# Sequence types and mutability

Before we start talking about **tuples** and **dictionaries**, we have to introduce two important concepts: **sequence types** and **mutability**.

A **sequence type is a type of data in Python which is able to store more than one value (or less than one, as a sequence may be empty), and these values can be sequentially (hence the name) browsed**, element by element.

As the for loop is a tool especially designed to iterate through sequences, we can express the definition as: **a sequence is data which can be scanned by the**for**loop**.

You've encountered one Python sequence so far - the list. The list is a classic example of a Python sequence, although there are some other sequences worth mentioning, and we're going to present them to you now.

The second notion - **mutability** - is a property of any of Python's data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: **mutable** and **immutable**.

**Mutable data can be freely updated at any time** - we call such an operation in situ.

*In situ* is a Latin phrase that translates as literally *in position*. For example, the following instruction modifies the data in situ:

list.append(1)

**Immutable data cannot be modified in this way**.

Imagine that a list can only be assigned and read over. You would be able neither to append an element to it, nor remove any element from it. This means that appending an element to the end of the list would require the recreation of the list from scratch.

You would have to build a completely new list, consisting of the all elements of the already existing list, plus the new element.

The data type we want to tell you about now is a **tuple**. **A tuple is an immutable sequence type**. It can behave like a list, but it mustn't be modified in situ.

## What is a tuple?

The first and the clearest distinction between lists and tuples is the syntax used to create them - **tuples prefer to use parenthesis**, whereas lists like to see brackets, although it's also **possible to create a tuple just from a set of values separated by commas**.

Look at the example:

tuple\_1=(1,2,4,8)

tuple\_2=1.,.5,.25,.125

There are two tuples, both containing **four elements**.

Let's print them:

tuple\_1=(1,2,4,8)

tuple\_2=1.,.5,.25,.125

print(tuple\_1)

print(tuple\_2)

This is what you should see in the console:

(1, 2, 4, 8)

(1.0, 0.5, 0.25, 0.125)

**output**

Note: **each tuple element may be of a different type** (floating-point, integer, or any other not-as-yet-introduced kind of data).

## How to create a tuple?

It is possible to create an empty tuple - parentheses are required then:

empty\_tuple=()

If you want to create a **one-element tuple**, you have to take into consideration the fact that, due to syntax reasons (a tuple has to be distinguishable from an ordinary, single value), you must end the value with a comma:

one\_element\_tuple\_1 = (1, )

one\_element\_tuple\_2 = 1.,

Removing the commas won't spoil the program in any syntactical sense, but you will instead get two single variables, not tuples.

[Prev](https://edube.org/learn/python-essentials-1/section-summary-18) [Next](https://edube.org/learn/python-essentials-1/tuples-and-dictionaries-8)

# What is a dictionary?

The **dictionary** is another Python data structure. It's **not a sequence** type (but can be easily adapted to sequence processing) and it is **mutable**.

To explain what the Python dictionary actually is, it is important to understand that it is literally a dictionary.

The Python dictionary works in the same way as **a bilingual dictionary**. For example, you have an English word (e.g., cat) and need its French equivalent. You browse the dictionary in order to find the word (you may use different techniques to do that - it doesn't matter) and eventually you get it. Next, you check the French counterpart and it is (most probably) the word "chat".



In Python's world, the word you look for is named a key. The word you get from the dictionary is called a value.

This means that a dictionary is a set of **key-value** pairs. Note:

* each key must be **unique** - it's not possible to have more than one key of the same value;
* a key may be **any immutable type of object**: it can be a number (integer or float), or even a string, but not a list;
* a dictionary is not a list - a list contains a set of numbered values, while a **dictionary holds pairs of values**;
* the len() function works for dictionaries, too - it returns the numbers of key-value elements in the dictionary;
* a dictionary is a **one-way tool** - if you have an English-French dictionary, you can look for French equivalents of English terms, but not vice versa.

Now we can show you some working examples.

## How to make a dictionary?

If you want to assign some initial pairs to a dictionary, you should use the following syntax:

dictionary={"cat":"chat","dog":"chien","horse":"cheval"}

phone\_numbers={'boss':5551234567,'Suzy':22657854310}

empty\_dictionary={}

print(dictionary)

print(phone\_numbers)

print(empty\_dictionary)

In the first example, the dictionary uses keys and values which are both strings. In the second one, the keys are strings, but the values are integers. The reverse layout (keys → numbers, values → strings) is also possible, as well as number-number combination.

The list of pairs is **surrounded by curly braces**, while the pairs themselves are **separated by commas**, and the **keys and values by colons**.

The first of our dictionaries is a very simple English-French dictionary. The second - a very tiny telephone directory.

The empty dictionaries are constructed by an **empty pair of curly braces** - nothing unusual.

The dictionary as a whole can be printed with a single print() invocation. The snippet **may** produce the following output:

{'dog': 'chien', 'horse': 'cheval', 'cat': 'chat'}

{'Suzy': 5557654321, 'boss': 5551234567}

{}

**output**

Have you noticed anything surprising? The order of the printed pairs is different than in the initial assignment. What does that mean?

First of all, it's a confirmation that **dictionaries are not lists** - they don't preserve the order of their data, as the order is completely meaningless (unlike in real, paper dictionaries). The order in which a dictionary **stores its data is completely out of your control**, and your expectations. That's normal. (\*)

NOTE

(\*) In Python 3.6x dictionaries have become **ordered** collections by default. Your results may vary depending on what Python version you're using.

# How to use a dictionary: the keys()

Can dictionaries be **browsed** using the for loop, like lists or tuples?

No and yes.

No, because a dictionary is **not a sequence type** - the for loop is useless with it.

Yes, because there are simple and very effective tools that can **adapt any dictionary to the**for**loop requirements** (in other words, building an intermediate link between the dictionary and a temporary sequence entity).

The first of them is a method named keys(), possessed by each dictionary. The method **returns an iterable object consisting of all the keys gathered within the dictionary**. Having a group of keys enables you to access the whole dictionary in an easy and handy way.

Just like here:

dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for key in dictionary.keys():

print(key, "->", dictionary[key]

The code's output looks as follows:

horse -> cheval

dog ->chien

cat -> chat

## The sorted() function

Do you want it **sorted**? Just enrich the for loop to get such a form:

for key in sorted(dictionary.keys()):

The sorted() function will do its best - the output will look like this:

cat -> chat

dog ->chien

horse -> cheval

# Errors – the developer's daily bread

It seems indisputable that all programmers (including you) want to write error-free code and do their best to achieve this goal. Unfortunately, nothing is perfect in this world and software is no exception. Pay attention to the word **exception** as we’ll see it again very soon in a meaning that has nothing in common with the absolute.



To err is human. It's impossible to make no mistakes, and it's impossible to write error-free code. Don't get us wrong – we don't want to convince you that writing messy and faulty programs is a virtue. We rather want to explain that even the most careful programmer is not able to avoid minor or major defects. It's only those who do nothing that make no mistakes.

Paradoxically, accepting this difficult truth can make you a better programmer and may improve your code quality.

"How could this be possible?", you may ask.

We'll try to show you.

# Errors in data vs. errors in code

Dealing with programming errors has (at least) two sides. The one appears when you get into trouble because your – apparently correct – code is fed with bad data. For example, you expect the code will input an integer value, but your careless user enters some random letters instead.

It may happen that your code will be terminated then, and the user will be left alone with a terse and ambiguous error message on the screen. The user will be unsatisfied, and you should be unsatisfied, too.

We're going to show you how to protect your code from this kind of failure and how not to provoke the user's anger.

The other side of dealing with programming errors reveals itself when undesirable code behavior is caused by mistakes you made when you were writing your program. This kind of error is commonly called a “bug”, which is a manifestation of a well-established belief that if a program works badly, it must be caused by malicious bugs which live inside the computer hardware and cause short circuits or other interference.

This idea is not as mad as it may look – such incidents were common in times when computers occupied large halls, consumed kilowatts of electricity, and produced enormous amounts of heat. Fortunately or not, these times are gone forever and the only bugs which can spoil your code are those you sowed in the code yourself. Therefore, we will try to show you how to find and eliminate your bugs, in other words, how to debug your code.

Let's start the journey through the land of errors and bugs.

**The *try-except* branch**

In the Python world, there is a rule that says: *"It’s better to beg for forgiveness than to ask for permission"*.

Let's stop here for a moment. Don't get us wrong – we don't want you to apply the rule in your everyday life. Don't take anyone's car without permission in the hope that you can be so convincing that you will avoid conviction. The rule is about something else.

Actually, the rule reads: *"it's better to handle an error when it happens than to try to avoid it"*.

*"Okay,"* you may say now, *'but how should I beg for forgiveness when the program is terminated and there is nothing left that can be done?"* This is where the **exception** comes on the scene.

Look at the code in the editor.

You can see two branches here:

* first, starting with the try keyword – this is the place where you put the code you suspect is risky and may be terminated in case of error; note: this kind of error is called an **exception**, while the exception occurrence is called **raising** – we can say that an exception is (or was) raised;

* second, the part of the code starting with the except keyword is designed to handle the exception; it's up to you what you want to do here: you can clean up the mess or you can just sweep the problem under the carpet (although we would prefer the first solution).

**The exception proves the rule**

Let's rewrite the code to adopt the Python approach to life:

Let us summarize what we talked about:

* any part of the code placed between try and except is executed in a very special way – any error which occurs here **won't terminate program execution**. Instead, the control will immediately jump to the first line situated after the except keyword, and no other part of the try branch is executed;

* the code in the except branch is activated only when an exception has been encountered inside the try block. There is no way to get there by any other means;

* when either the try block or the except block is executed successfully, the control returns to the normal path of execution, and any code located beyond in the source file is executed as if nothing happened.

Now we want to ask you an innocent question: is ValueError the only way the control could fall into the except branch?

## How to take PCEP exam?

Taking the exam requires that you have an **exam voucher**. You can purchase an exam voucher at OpenEDG Voucher Store and assign it to your Test Candidate Account.

**Purchase exam voucher**

To purchase an exam voucher, you should go to [OpenEDG Voucher Store](https://ums.edube.org/store" \t "_blank), add a PCEP voucher to the cart, and place an order.

During the process you be requested to provide billing information (invoice details and the email address where the invoice should be sent), shipping information (email address where the voucher should be sent) and read and accept the Voucher Policies.

In the final stage of the order placement process, you will be requested to make payment (PayPal/Credit card, or Wire Transfer based on a Pro Forma Invoice issued).

You can pay for the exam by all major credit cards (i.e., Visa, MasterCard, American Express), Debit Cards or via PayPal account.

**Assign voucher to your account**

Test candidates who already have an exam voucher can assign it to their User Account before taking the exam. The exam vouchers assigned are displayed on the Test Candidate Account and may be used to launch exams via OpenEDG Testing Service.

You can also assign the voucher to your account during order placement. The voucher will be linked with your account as soon as we receive the payment.

**Take the exam**

After logging in at <https://edube.org/login>, you can launch your exam session by:

* clicking Take Exam (if you have an exam voucher and wish to begin your exam session immediately) or by
* clicking Launch Exam next to the voucher assigned to the OpenEDG User Account (if the voucher has already been assigned to your account.)

For more information about the technical requirements, code of honor, and exam policies, go to <https://pythoninstitute.org/pcep-testing-policies/>.

For example, you can run the following code to print the names of all entities within the math module:

import math

for name indir(math):

print(name, end="\t")

The example code should produce the following output:

\_\_doc\_\_ \_\_loader\_\_ \_\_name\_\_ \_\_package\_\_ \_\_spec\_\_ acosacoshasinasinhatan atan2 atanh ceil copysign cos cosh degrees e erf erfc exp expm1 fabs factorial floor fmodfrexpfsum gamma hypotisfiniteisinfisnanldexplgamma log log10 log1p log2 modf pi pow radians sin sinh sqrt tan tanh trunc

**Selected functions from the math module: continued**

Another group of the math's functions is formed by functions which are connected with **exponentiation**:

* e → a constant with a value that is an approximation of Euler's number (e)
* exp(x) → finding the value of ex;
* log(x) → the natural logarithm of x
* log(x, b) → the logarithm of x to base b
* log10(x) → the decimal logarithm of x (more precise than log(x, 10))
* log2(x) → the binary logarithm of x (more precise than log(x, 2))

Note: the pow() function:

* pow(x, y) → finding the value of xy (mind the domains)

This is a built-in function, and doesn't have to be imported.

The last group consists of some general-purpose functions like:

* ceil(x) → the ceiling of x (the smallest integer greater than or equal to x)
* floor(x) → the floor of x (the largest integer less than or equal to x)
* trunc(x) → the value of x truncated to an integer (be careful - it's not an equivalent either of ceil or floor)
* factorial(x) → returns x! (x has to be an integral and not a negative)
* hypot(x, y) → returns the length of the hypotenuse of a right-angle triangle with the leg lengths equal to x and y (the same as sqrt(pow(x, 2) + pow(y, 2)) but more precise)

Look at the code in the editor. Analyze the program carefully.

It demonstrates the fundamental differences between ceil(), floor() and trunc().

**Selected functions from the random module: continued**

**The randrange and randint functions**

If you want integer random values, one of the following functions would fit better:

* randrange(end)
* randrange(beg, end)
* randrange(beg, end, step)
* randint(left, right)

The first three invocations will generate an integer taken (pseudorandomly) from the range (respectively):

* range(end)
* range(beg, end)
* range(beg, end, step)

Note the implicit **right-sided exclusion**!

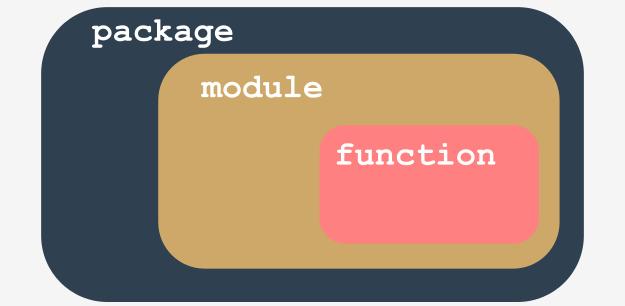
The last function is an equivalent of randrange(left, right+1) - it generates the integer value i, which falls in the range [left, right] (no exclusion on the right side).

Look at the code in the editor. This sample program will consequently output a line consisting of three zeros and either a zero or one at the fourth place.

**What is a package?**

Writing your own modules doesn't differ much from writing ordinary scripts.

There are some specific aspects you must be aware of, but it definitely isn't rocket science. You'll see this soon enough.



Let's summarize some important issues:

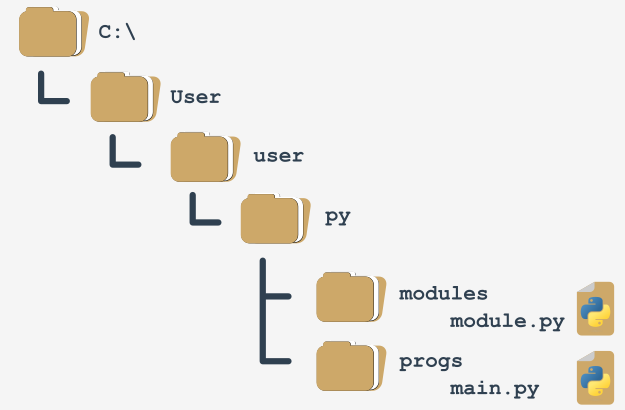
* a **module is a kind of container filled with functions** - you can pack as many functions as you want into one module and distribute it across the world;
* of course, it's generally a good idea not to mix functions with different application areas within one module (just like in a library - nobody expects scientific works to be put among comic books), so group your functions carefully and name the module containing them in a clear and intuitive way (e.g., don't give the name arcade\_games to a module containing functions intended to partition and format hard disks)
* making many modules may cause a little mess - sooner or later you'll want to **group your modules** exactly in the same way as you've previously grouped functions - is there a more general container than a module?
* yes, there is - it's a **package**; in the world of modules, a package plays a similar role to a folder/directory in the world of files.

**Your first module: step 11**

It's time to make our example more complicated - so far we've assumed that the main Python file is located in the same folder/directory as the module to be imported.

Let's give up this assumption and conduct the following thought experiment:

* we are using Windows ® OS (this assumption is important, as the file name's shape depends on it)
* the main Python script lies in C:\Users\user\py\progs and is named main.py
* the module to import is located in C:\Users\user\py\modules



How to deal with it?

To answer this question, we have to talk about **how Python searches for modules**. There's a special variable (actually a list) storing all locations (folders/directories) that are searched in order to find a module which has been requested by the import instruction.

Python browses these folders in the order in which they are listed in the list - if the module cannot be found in any of these directories, the import fails.

Otherwise, the first folder containing a module with the desired name will be taken into consideration (if any of the remaining folders contains a module of that name, it will be ignored).

The variable is named path, and it's accessible through the module named sys. This is how you can check its regular value:

import sys

for p insys.path:

print(p)

We've launched the code inside the C:\User\user folder, and this is what we've got:

C:\Users\user

C:\Users\user\AppData\Local\Programs\Python\Python36-32\python36.zip

C:\Users\user\AppData\Local\Programs\Python\Python36-32\DLLs

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib

C:\Users\user\AppData\Local\Programs\Python\Python36-32

C:\Users\user\AppData\Local\Programs\Python\Python36-32\lib\site-packages

**sample output**

**How to use pip: a simple test program**

Now that *pygame* is finally accessible, we can try to use it in a very simple test program. Let’s comment on it briefly.

* line 1: import *pygame* and let it serve us;
* line 3: the program will run as long as the run variable is True;
* lines 4 and 5: determine the window's size;
* line 6: initialize the *pygame* environment;
* line 7: prepare the application window and set its size;
* line 8: make an object representing the default font of size 48 points;
* line 9: make an object representing a given text – the text will be anti-aliased (True) and white (255,255,255)
* line 10: insert the text into the (currently invisible) screen buffer;
* line 11: flip the screen buffers to make the text visible;
* line 12: the *pygame* main loop starts here;
* line 13: get a list of all pending *pygame* events;
* lines 14 through 16: check whether the user has closed the window or clicked somewhere inside it or pressed any key;
* line 15: if yes, stop executing the code.

import pygame

run = True

width = 400

height = 100

pygame.init()

screen = pygame.display.set\_mode((width, height))

font = pygame.font.SysFont(None, 48)

text = font.render("Welcome to pygame", True, (255, 255, 255))

screen.blit(text, ((width - text.get\_width()) // 2, (height - text.get\_height()) // 2))

pygame.display.flip()

while run:

for event in pygame.event.get():

if event.type == pygame.QUIT\

or event.type == pygame.MOUSEBUTTONUP\

or event.type == pygame.KEYUP:

run = False

# The basic concepts of the object-oriented approach

Let's take a step outside of computer programming and computers in general, and discuss object programming issues.

Nearly all of the programs and techniques you have used till now fall under the procedural style of programming. Admittedly, you have made use of some built-in objects, but when referring to them, we just mentioned the absolute minimum.

The procedural style of programming was the dominant approach to software development for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).

The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers.

This kind of understanding of a project's structure makes many important tasks easier, e.g., dividing the project into small, independent parts, and independent development of different project elements.

**Python is a universal tool for both object and procedural programming**. It may be successfully utilized in both spheres.

Furthermore, you can create lots of useful applications, even if you know nothing about classes and objects, but you have to keep in mind that some of the problems (e.g., graphical user interface handling) may require a strict object approach.

Fortunately, object programming is relatively simple.



[Prev](https://edube.org/learn/python-essentials-2/python-essentials-2-module-3) [Next](https://edube.org/learn/python-essentials-2/the-foundations-of-oop-7)

**Procedural vs. the object-oriented approach**

In the **procedural approach**, it's possible to distinguish two different and completely separate worlds: **the world of data, and the world of code**. The world of data is populated with variables of different kinds, while the world of code is inhabited by code grouped into modules and functions.

Functions are able to use data, but not vice versa. Furthermore, functions are able to abuse data, i.e., to use the value in an unauthorized manner (e.g., when the sine function gets a bank account balance as a parameter).

We said in the past that data cannot use functions. But is this entirely true? Are there some special kinds of data that can use functions?

Yes, there are - the ones named methods. These are functions which are invoked from within the data, not beside them. If you can see this distinction, you've taken the first step into object programming.

The **object approach** suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.

Every **class is like a recipe which can be used when you want to create a useful object** (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.

Every object has a set of traits (they are called properties or attributes - we'll use both words synonymously) and is able to perform a set of activities (which are called methods).

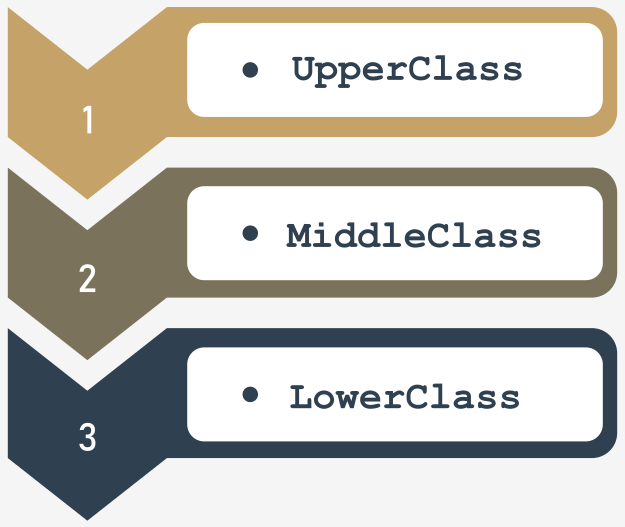
The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.

**Objects are incarnations** of ideas expressed in classes, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in an old cookbook.

The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) are able to protect the sensible data and hide it from unauthorized modifications.

There is no clear border between data and code: they live as one in objects.

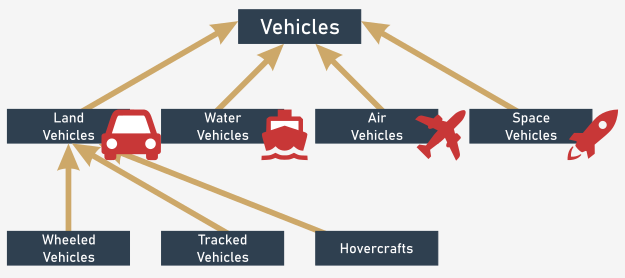
All these concepts are not as abstract as you may at first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programming: they don't create artificial life - **they reflect real facts, relationships, and circumstances**.



**Class hierarchies**

The word *class* has many meanings, but not all of them are compatible with the ideas we want to discuss here. The *class* that we are concerned with is like a *category*, as a result of precisely defined similarities.

We'll try to point out a few classes which are good examples of this concept.



Let's look for a moment at vehicles. All existing vehicles (and those that don't exist yet) are **related by a single, important feature**: the ability to move. You may argue that a dog moves, too; is a dog a vehicle? No, it isn't. We have to improve the definition, i.e., enrich it with other criteria, distinguishing vehicles from other beings, and creating a stronger connection. Let's take the following circumstances into consideration: vehicles are artificially created entities used for transportation, moved by forces of nature, and directed (driven) by humans.

Based on this definition, a dog is not a vehicle.

The *vehicles* class is very broad. Too broad. We have to define some more **specialized classes**, then. The specialized classes are the **subclasses**. The *vehicles* class will be a **superclass** for them all.

Note: **the hierarchy grows from top to bottom, like tree roots, not branches**. The most general, and the widest, class is always at the top (the superclass) while its descendants are located below (the subclasses).

By now, you can probably point out some potential subclasses for the *Vehicles* superclass. There are many possible classifications. We've chosen subclasses based on the environment, and say that there are (at least) four subclasses:

* land vehicles;
* water vehicles;
* air vehicles;
* space vehicles.

In this example, we'll discuss the first subclass only - land vehicles. If you wish, you can continue with the remaining classes.

Land vehicles may be further divided, depending on the method with which they impact the ground. So, we can enumerate:

* wheeled vehicles;
* tracked vehicles;
* hovercrafts.

The hierarchy we've created is illustrated by the figure.

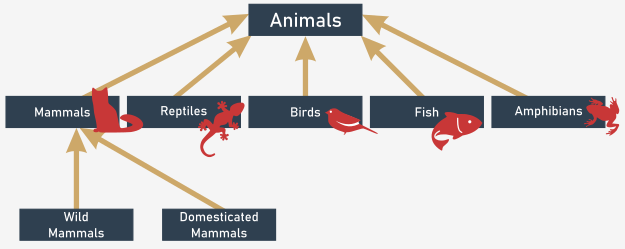
Note the direction of the arrows - they always point to the superclass. The top-level class is an exception - it doesn't have its own superclass.

* fish;
* amphibians.

We'll take the first one for further analysis.

We have identified the following subclasses:

* wild mammals;
* domesticated mammals.



Try to extend the hierarchy any way you want, and find the right place for humans.

# What is an object?

A class (among other definitions) is a **set of objects**. An object is **a being belonging to a class**.

An object is **an incarnation of the requirements, traits, and qualities assigned to a specific class**. This may sound simple, but note the following important circumstances. Classes form a hierarchy.

This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.

For example: any personal car is an object belonging to the *wheeled vehicles* class. It also means that the same car belongs to all superclasses of its home class; therefore, it is a member of the *vehicles* class, too.

Your dog (or your cat) is an object included in the *domesticated mammals* class, which explicitly means that it is included in the *animals* class as well.

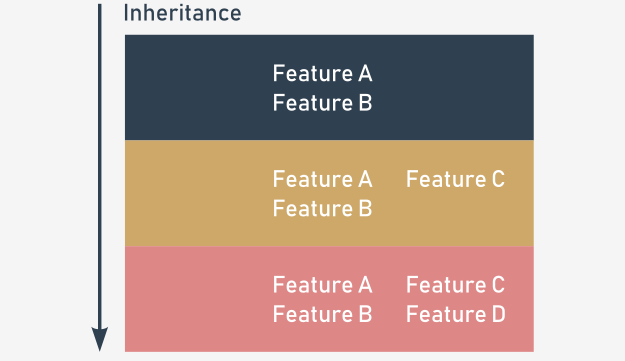
Each **subclass is more specialized** (or more specific) than its superclass. Conversely, each **superclass is more general** (more abstract) than any of its subclasses.

Note that we've presumed that a class may only have one superclass - this is not always true, but we'll discuss this issue more a bit later.

## Inheritance

Let's define one of the fundamental concepts of object programming, named **inheritance**. Any object bound to a specific level of a class hierarchy **inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses**.

The object's home class may define new traits (as well as requirements and qualities) which will be inherited by any of its subclasses.



You shouldn't have any problems matching this rule to specific examples, whether it applies to animals, or to vehicles.

# What does an object have?

The object programming convention assumes that **every existing object may be equipped with three groups of attributes**:

* an object has a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
* an object has a **set of individual properties** which make it original, unique, or outstanding (although it's possible that some objects may have no properties at all)
* an object has a **set of abilities to perform specific activities**, able to change the object itself, or some of the other objects.

There is a hint (although this doesn't always work) which can help you identify any of the three spheres above. Whenever you describe an object and you use:

* a noun – you probably define the object's name;
* an adjective – you probably define the object's property;
* a verb – you probably define the object's activity.

Two sample phrases should serve as a good example:

* A pink Cadillac went quickly.  
    
  Object name = Cadillac  
  Home class = Wheeled vehicles  
  Property = Color (pink)  
  Activity = Go (quickly)

* Rudolph is a large cat who sleeps all day.  
    
  Object name = Rudolph  
  Home class = Cat  
  Property = Size (large)  
  Activity = Sleep (all day)



[Prev](https://edube.org/learn/python-essentials-2/the-foundations-of-oop-10) [Next](https://edube.org/learn/python-essentials-2/the-foundations-of-oop-12)

# Your first class

Object programming is **the art of defining and expanding classes**. A class is a model of a very specific part of reality, reflecting properties and activities found in the real world.

The classes defined at the beginning are too general and imprecise to cover the largest possible number of real cases.

There's no obstacle to defining new, more precise subclasses. They'll inherit everything from their superclass, so the work that went into its creation isn't wasted.

The new class may add new properties and new activities, and therefore may be more useful in specific applications. Obviously, it may be used as a superclass for any number of newly created subclasses.

The process doesn't need to have an end. You can create as many classes as you need.

The class you define has nothing to do with the object: **the existence of a class does not mean that any of the compatible objects will automatically be created**. The class itself isn't able to create an object - you have to create it yourself, and Python allows you to do this.

It's time to define the simplest class and to create an object. Take a look at the example below:

class TheSimplestClass:

pass

We've defined a class there. The class is rather poor: it has neither properties nor activities. It's **empty**, actually, but that doesn't matter for now. The simpler the class, the better for our purposes.

**The definition begins with the keyword**class. The keyword is followed by an **identifier which will name the class** (note: don't confuse it with the object's name - these are two different things).

Next, you add a **colon** (:), as classes, like functions, form their own nested block. The content inside the block define all the class's properties and activities.

The pass keyword fills the class with nothing. It doesn't contain any methods or properties.

## Your first object

The newly defined class becomes a tool that is able to create new objects. The tool has to be used explicitly, on demand.

Imagine that you want to create one (exactly one) object of the TheSimplestClass class.

To do this, you need to assign a variable to store the newly created object of that class, and create an object at the same time.

You do it in the following way:

my\_first\_object = TheSimplestClass()

Note:

* the class name tries to pretend that it's a function - can you see this? We'll discuss it soon;
* the newly created object is equipped with everything the class brings; as this class is completely empty, the object is empty, too.

The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an **instance of the class**).

Let's leave classes alone for a short moment, as we're now going to tell you a few words about *stacks*. We know the concept of classes and objects may not be fully clear yet. Don't worry, we'll explain everything very soon.

[Prev](https://edube.org/learn/python-essentials-2/the-foundations-of-oop-11) [Next](https://edube.org/learn/python-essentials-2/section-summary-22)

# Key takeaways

1. A **class** is an idea (more or less abstract) which can be used to create a number of incarnations – such an incarnation is called an **object**.

2. When a class is derived from another class, their relation is named **inheritance**. The class which derives from the other class is named a **subclass**. The second side of this relation is named **superclass**. A way to present such a relation is an **inheritance diagram**, where:

* superclasses are always presented **above** their subclasses;
* relations between classes are shown as arrows directed **from the subclass toward its superclass**.

3. Objects are equipped with:

* a **name** which identifies them and allows us to distinguish between them;
* a set of **properties** (the set can be empty)
* a set of **methods** (can be empty, too)

4. To define a Python class, you need to use the class keyword. For example:

class This\_Is\_A\_Class:

pass

5. To create an object of the previously defined class, you need to use the class as if it were a function. For example:

this\_is\_an\_object = This\_Is\_A\_Class()

**Exercise 1**

If we assume that pythons, vipers, and cobras are subclasses of the same superclass, how would you call it?

Check

Snake, reptile, vertebrate, animal – all these answers are acceptable.

**Exercise 2**

Try to name a few python class subclasses.

Check

Indian python, African rock python, ball python, Burmese python – the list is long.

**Exercise 3**

Can you name one of your classes just "class"?

Check

No, you can't – class is a keyword!

[Prev](https://edube.org/learn/python-essentials-2/the-foundations-of-oop-12) [Next](https://edube.org/learn/python-essentials-2/a-short-journey-from-procedural-to-object-approach-24)

**What is a stack?**

**A stack is a structure developed to store data in a very specific way**. Imagine a stack of coins. You aren't able to put a coin anywhere else but on the top of the stack.

Similarly, you can't get a coin off the stack from any place other than the top of the stack. If you want to get the coin that lies on the bottom, you have to remove all the coins from the higher levels.

The alternative name for a stack (but only in IT terminology) is **LIFO**.

It's an abbreviation for a very clear description of the stack's behavior: **Last In - First Out**. The coin that came last onto the stack will leave first.

**A stack is an object** with two elementary operations, conventionally named **push** (when a new element is put on the top) and **pop** (when an existing element is taken away from the top).

Stacks are used very often in many classical algorithms, and it's hard to imagine the implementation of many widely used tools without the use of stacks.



Let's implement a stack in Python. This will be a very simple stack, and we'll show you how to do it in two independent approaches: procedural and objective.

Let's start with the first one.

**The stack - the procedural approach**

First, you have to decide how to store the values which will arrive onto the stack. We suggest using the simplest of methods, and **employing a list** for this job. Let's assume that the size of the stack is not limited in any way. Let's also assume that the last element of the list stores the top element.

The stack itself is already created:

stack = []

We're ready to **define a function that puts a value onto the stack**. Here are the presuppositions for it:

* the name for the function is push;
* the function gets one parameter (this is the value to be put onto the stack)
* the function returns nothing;
* the function appends the parameter's value to the end of the stack;

This is how we've done it - take a look:

def push(val):

stack.append(val)

Now it's time for a **function to take a value off the stack**. This is how you can do it:

* the name of the function is pop;
* the function doesn't get any parameters;
* the function returns the value taken from the stack
* the function reads the value from the top of the stack and removes it.

The function is here:

def pop():

val = stack[-1]

del stack[-1]

return val

Note: the function doesn't check if there is any element in the stack.

Let's assemble all the pieces together to set the stack in motion. The **complete program** pushes three numbers onto the stack, pulls them off, and prints their values on the screen. You can see it in the editor window.

The program outputs the following text to the screen:

1

2

3

**output**

Test it.

stack = []

def push(val):

stack.append(val)

def pop():

val = stack[-1]

del stack[-1]

return val

push(3)

push(2)

push(1)

print(pop())

print(pop())

print(pop())

# The stack - the procedural approach vs. the object-oriented approach

The procedural stack is ready. Of course, there are some weaknesses, and the implementation could be improved in many ways (harnessing exceptions to work is a good idea), but in general the stack is fully implemented, and you can use it if you need to.

But the more often you use it, the more disadvantages you'll encounter. Here are some of them:

* the essential variable (the stack list) is highly **vulnerable**; anyone can modify it in an uncontrollable way, destroying the stack, in effect; this doesn't mean that it's been done maliciously - on the contrary, it may happen as a result of carelessness, e.g., when somebody confuses variable names; imagine that you have accidentally written something like this:
* stack[0] = 0

The functioning of the stack will be completely disorganized;

* it may also happen that one day you need more than one stack; you'll have to create another list for the stack's storage, and probably other push and pop functions too;

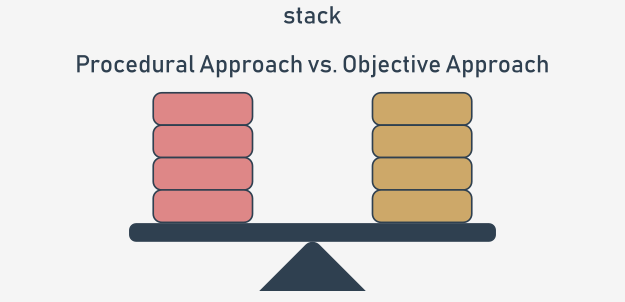
* it may also happen that you need not only push and pop functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacks.

The objective approach delivers solutions for each of the above problems. Let's name them first:

* the ability to hide (protect) selected values against unauthorized access is called **encapsulation; the encapsulated values can be neither accessed nor modified if you want to use them exclusively**;

* when you have a class implementing all the needed stack behaviors, you can produce as many stacks as you want; you needn't copy or replicate any part of the code;

* the ability to enrich the stack with new functions comes from inheritance; you can create a new class (a subclass) which inherits all the existing traits from the superclass, and adds some new ones.



Let's now write a brand new stack implementation from scratch. This time, we'll use the objective approach, guiding you step by step into the world of object programming.

[Prev](https://edube.org/learn/python-essentials-2/a-short-journey-from-procedural-to-object-approach-25) [Next](https://edube.org/learn/python-essentials-2/a-short-journey-from-procedural-to-object-approach-27)

**The stack - the object approach**

Of course, the main idea remains the same. We'll use a list as the stack's storage. We only have to know how to put the list into the class.

Let's start from the absolute beginning - this is how the objective stack begins:

class Stack:

Now, we expect two things from it:

* we want the class to have **one property as the stack's storage** - we have to **"install" a list inside each object of the class** (note: each object has to have its own list - the list mustn't be shared among different stacks)
* then, we want **the list to be hidden** from the class users' sight.

How is this done?

In contrast to other programming languages, Python has no means of allowing you to declare such a property just like that.

Instead, you need to add a specific statement or instruction. The properties have to be added to the class manually.

How do you guarantee that such an activity takes place every time the new stack is created?

There is a simple way to do it - you have to **equip the class with a specific function** - its specificity is dual:

* it has to be named in a strict way;
* it is invoked implicitly, when the new object is created.

Such a function is called a **constructor**, as its general purpose is to **construct a new object**. The constructor should know everything about the object's structure, and must perform all the needed initializations.

Let's add a very simple constructor to the new class. Take a look at the snippet:

class Stack:

def \_\_init\_\_(self):

print("Hi!")

stack\_object = Stack()

And now:

* the constructor's name is always \_\_init\_\_;
* it has to have **at least one parameter** (we'll discuss this later); the parameter is used to represent the newly created object - you can use the parameter to manipulate the object, and to enrich it with the needed properties; you'll make use of this soon;
* note: the obligatory parameter is usually named self - it's only **a convention, but you should follow it** - it simplifies the process of reading and understanding your code.

The code is in the editor. Run it now.

Here is its output:

Hi!

**output**

Note - there is no trace of invoking the constructor inside the code. It has been invoked implicitly and automatically. Let's make use of that now.

class Stack: # Defining the Stack class.

def \_\_init\_\_(self): # Defining the constructor function.

print("Hi!")

stack\_object = Stack() # Instantiating the object.

# The stack - the object approach: continued

Any change you make inside the constructor that modifies the state of the self parameter will be reflected in the newly created object.

This means you can add any property to the object and the property will remain there until the object finishes its life or the property is explicitly removed.

Now let's **add just one property to the new object** - a list for a stack. We'll name it stack\_list.

Just like here:

class Stack:

def \_\_init\_\_(self):

self.stack\_list = []

stack\_object = Stack()

print(len(stack\_object.stack\_list))

Note:

* we've used the **dotted notation**, just like when invoking methods; this is the general convention for accessing an object's properties - you need to name the object, put a dot (.) after it, and specify the desired property's name; don't use parentheses! You don't want to invoke a method - you want to **access a property**;
* if you set a property's value for the very first time (like in the constructor), you are creating it; from that moment on, the object has got the property and is ready to use its value;
* we've done something more in the code - we've tried to access the stack\_list property from outside the class immediately after the object has been created; we want to check the current length of the stack - have we succeeded?

Yes, we have - the code produces the following output:

0

**output**

This is not we want from the stack. We prefer stack\_list to be **hidden from the outside world**. Is that possible?

Yes, and it's simple, but not very intuitive.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Stack:  
def \_\_init\_\_(self):  
self.stack\_list = []  
  
  
stack\_object = Stack()  
print(len(stack\_object.stack\_list))



1

2

3

4

5

6

7

8

class Stack:

def \_\_init\_\_(self):

self.stack\_list = []

stack\_object = Stack()

print(len(stack\_object.stack\_list))

* **Console**

class Stack:

def \_\_init\_\_(self):

self.stack\_list = []

stack\_object = Stack()

print(len(stack\_object.stack\_list))

# The stack - the object approach: continued

Take a look - we've added two underscores before the stack\_list name - nothing more:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

stack\_object = Stack()

print(len(stack\_object.\_\_stack\_list))

The change invalidates the program.

Why?

When any class component has a **name starting with two underscores (**\_\_**), it becomes private** - this means that it can be accessed only from within the class.

You cannot see it from the outside world. This is how Python implements the **encapsulation** concept.

Run the program to test our assumptions - an AttributeError exception should be raised.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Stack:  
def \_\_init\_\_(self):  
self.\_\_stack\_list = []  
  
  
stack\_object = Stack()  
print(len(stack\_object.\_\_stack\_list))



1

2

3

4

5

6

7

8

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

stack\_object = Stack()

print(len(stack\_object.\_\_stack\_list))

* **Console**

# The stack - the object approach: continued

Take a look - we've added two underscores before the stack\_list name - nothing more:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

stack\_object = Stack()

print(len(stack\_object.\_\_stack\_list))

The change invalidates the program.

Why?

When any class component has a **name starting with two underscores (**\_\_**), it becomes private** - this means that it can be accessed only from within the class.

You cannot see it from the outside world. This is how Python implements the **encapsulation** concept.

Run the program to test our assumptions - an AttributeError exception should be raised.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Stack:  
def \_\_init\_\_(self):  
self.\_\_stack\_list = []  
  
  
stack\_object = Stack()  
print(len(stack\_object.\_\_stack\_list))



1

2

3

4

5

6

7

8

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

stack\_object = Stack()

print(len(stack\_object.\_\_stack\_list))

* **Console**

# The object approach: a stack from scratch

Now it's time for the two functions (methods) implementing the *push* and *pop* operations. Python assumes that a function of this kind (a class activity) should be **immersed inside the class body** - just like a constructor.

We want to invoke these functions to push and pop values. This means that they should both be accessible to every class's user (in contrast to the previously constructed list, which is hidden from the ordinary class's users).

Such a component is called **public**, so you **can't begin its name with two (or more) underscores**. There is one more requirement - **the name must have no more than one trailing underscore**. As no trailing underscores at all fully meets the requirement, you can assume that the name is acceptable.

The functions themselves are simple. Take a look:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object = Stack()

stack\_object.push(3)

stack\_object.push(2)

stack\_object.push(1)

print(stack\_object.pop())

print(stack\_object.pop())

print(stack\_object.pop())

However, there's something really strange in the code. The functions look familiar, but they have more parameters than their procedural counterparts.

Here, both functions have a parameter named self at the first position of the parameters list.

Is it needed? Yes, it is.

All methods have to have this parameter. It plays the same role as the first constructor parameter.

**It allows the method to access entities (properties and activities/methods) carried out by the actual object**. You cannot omit it. Every time Python invokes a method, it implicitly sends the current object as the first argument.

This means that a **method is obligated to have at least one parameter, which is used by Python itself** - you don't have any influence on it.

If your method needs no parameters at all, this one must be specified anyway. If it's designed to process just one parameter, you have to specify two, and the first one's role is still the same.

There is one more thing that requires explanation - the way in which methods are invoked from within the \_\_stack\_list variable.

Fortunately, it's much simpler than it looks:

* the first stage delivers the object as a whole → self;
* next, you need to get to the \_\_stack\_list list → self.\_\_stack\_list;
* with \_\_stack\_list ready to be used, you can perform the third and last step → self.\_\_stack\_list.append(val).

The class declaration is complete, and all its components have been listed. The class is ready for use.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Stack:  
def \_\_init\_\_(self):  
self.\_\_stack\_list = []  
  
  
def push(self, val):  
self.\_\_stack\_list.append(val)  
  
  
def pop(self):  
val = self.\_\_stack\_list[-1]  
del self.\_\_stack\_list[-1]  
return val  
  
  
stack\_object = Stack()  
  
stack\_object.push(3)  
stack\_object.push(2)  
stack\_object.push(1)  
  
print(stack\_object.pop())  
print(stack\_object.pop())  
print(stack\_object.pop())



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object = Stack()

stack\_object.push(3)

stack\_object.push(2)

* **Console**

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object = Stack()

stack\_object.push(3)

stack\_object.push(2)

stack\_object.push(1)

print(stack\_object.pop())

print(stack\_object.pop())

print(stack\_object.pop())

**The object approach: a stack from scratch**

Having such a class opens up some new possibilities. For example, you can now have more than one stack behaving in the same way. Each stack will have its own copy of private data, but will utilize the same set of methods.

This is exactly what we want for this example.

Analyze the code:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object\_1 = Stack()

stack\_object\_2 = Stack()

stack\_object\_1.push(3)

stack\_object\_2.push(stack\_object\_1.pop())

print(stack\_object\_2.pop())

There are **two stacks created from the same base class**. They work **independently**. You can make more of them if you want to.

Run the code in the editor and see what happens. Carry out your own experiments.

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object\_1 = Stack()

stack\_object\_2 = Stack()

stack\_object\_1.push(3)

stack\_object\_2.push(stack\_object\_1.pop())

print(stack\_object\_2.pop())

# The object approach: a stack from scratch (continued)

Analyze the snippet below - we've created three objects of the class Stack. Next, we've juggled them up. Try to predict the value outputted to the screen.

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

little\_stack = Stack()

another\_stack = Stack()

funny\_stack = Stack()

little\_stack.push(1)

another\_stack.push(little\_stack.pop() + 1)

funny\_stack.push(another\_stack.pop() - 2)

print(funny\_stack.pop())

So, what's the result? Run the program and check if you were right.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Stack:  
def \_\_init\_\_(self):  
self.\_\_stack\_list = []  
  
def push(self, val):  
self.\_\_stack\_list.append(val)  
  
def pop(self):  
val = self.\_\_stack\_list[-1]  
del self.\_\_stack\_list[-1]  
return val  
  
  
# Enter code here.



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

# Enter code here.

* **Console**

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

# Enter code here.

# The object approach: a stack from scratch (continued)

Now let's go a little further. Let's **add a new class for handling stacks**.

The new class should be able to **evaluate the sum of all the elements currently stored on the stack**.

We don't want to modify the previously defined stack. It's already good enough in its applications, and we don't want it changed in any way. We want a new stack with new capabilities. In other words, we want to construct a subclass of the already existing Stack class.

The first step is easy: just **define a new subclass pointing to the class which will be used as the superclass**.

This is what it looks like:

class AddingStack(Stack):

pass

The class doesn't define any new component yet, but that doesn't mean that it's empty. **It gets all the components defined by its superclass** - the name of the superclass is written before the colon directly after the new class name.

This is what we want from the new stack:

* we want the push method not only to push the value onto the stack but also to add the value to the sum variable;
* we want the pop function not only to pop the value off the stack but also to subtract the value from the sum variable.

Firstly, let's add a new variable to the class. It'll be a **private variable**, like the stack list. We don't want anybody to manipulate the sum value.

As you already know, adding a new property to the class is done by the constructor. You already know how to do that, but there is something really intriguing inside the constructor. Take a look:

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

The second line of the constructor's body creates a property named \_\_sum - it will store the total of all the stack's values.

But the line before it looks different. What does it do? Is it really necessary? Yes, it is.

Contrary to many other languages, Python forces you to **explicitly invoke a superclass's constructor**. Omitting this point will have harmful effects - the object will be deprived of the \_\_stack\_list list. Such a stack will not function properly.

This is the only time you can invoke any of the available constructors explicitly - it can be done inside the subclass's constructor.

Note the syntax:

* you specify the superclass's name (this is the class whose constructor you want to run)
* you put a dot (.)after it;
* you specify the name of the constructor;
* you have to point to the object (the class's instance) which has to be initialized by the constructor - this is why you have to specify the argument and use the self variable here; note: **invoking any method (including constructors) from outside the class never requires you to put the**self**argument at the argument's list** - invoking a method from within the class demands explicit usage of the self argument, and it has to be put first on the list.

Note: it's generally a recommended practice to invoke the superclass's constructor before any other initializations you want to perform inside the subclass. This is the rule we have followed in the snippet.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Stack:  
def \_\_init\_\_(self):  
self.\_\_stack\_list = []  
  
def push(self, val):  
self.\_\_stack\_list.append(val)  
  
def pop(self):  
val = self.\_\_stack\_list[-1]  
del self.\_\_stack\_list[-1]  
return val  
  
  
class AddingStack(Stack):  
def \_\_init\_\_(self):  
Stack.\_\_init\_\_(self)  
self.\_\_sum = 0



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

* **Console**

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

# self.\_\_sum = 0he object approach: a stack from scratch (continued)

Secondly, let's add two methods. But let us ask you: is it really adding? We have these methods in the superclass already. Can we do something like that?

Yes, we can. It means that we're going to **change the functionality of the methods, not their names**. We can say more precisely that the interface (the way in which the objects are handled) of the class remains the same when changing the implementation at the same time.

Let's start with the implementation of the push function. This is what we expect from it:

* to add the value to the \_\_sum variable;
* to push the value onto the stack.

Note: the second activity has already been implemented inside the superclass - so we can use that. Furthermore, we have to use it, as there's no other way to access the \_\_stackList variable.

This is how the push method looks in the subclass:

def push(self, val):

self.\_\_sum += val

Stack.push(self, val)

Note the way we've invoked the previous implementation of the push method (the one available in the superclass):

* we have to specify the superclass's name; this is necessary in order to clearly indicate the class containing the method, to avoid confusing it with any other function of the same name;
* we have to specify the target object and to pass it as the first argument (it's not implicitly added to the invocation in this context.)

We say that the push method has been overridden - the same name as in the superclass now represents a different functionality.

class Stack:

def \_\_init\_\_(self):

self.\_\_stackList = []

def push(self, val):

self.\_\_stackList.append(val)

def pop(self):

val = self.\_\_stackList[-1]

del self.\_\_stackList[-1]

return val

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

# Enter code here.

**The object approach: a stack from scratch (continued)**

This is the new pop function:

def pop(self):

val = Stack.pop(self)

self.\_\_sum -= val

return val

So far, we've defined the \_\_sum variable, but we haven't provided a method to get its value. It seems to be hidden. How can we reveal it and do it in a way that still protects it from modifications?

We have to define a new method. We'll name it get\_sum. Its only task will be to **return the**\_\_sum**value**.

Here it is:

def get\_sum(self):

return self.\_\_sum

So, let's look at the program in the editor. The complete code of the class is there. We can check its functioning now, and we do it with the help of a very few additional lines of code.

As you can see, we add five subsequent values onto the stack, print their sum, and take them all off the stack.

Okay, this has been a very brief introduction to Python's object programming. Soon we're going to tell you about it all in more detail.

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

def get\_sum(self):

return self.\_\_sum

def push(self, val):

self.\_\_sum += val

Stack.push(self, val)

def pop(self):

val = Stack.pop(self)

self.\_\_sum -= val

return val

stack\_object = AddingStack()

for i in range(5):

stack\_object.push(i)

print(stack\_object.get\_sum())

for i in range(5):

print(stack\_object.pop())

# Key takeaways

1. A **stack** is an object designed to store data using the **LIFO** model. The stack usually accomplishes at least two operations, named **push()** and **pop()**.

2. Implementing the stack in a procedural model raises several problems which can be solved by the techniques offered by **OOP** (**O**bject **O**riented **P**rogramming):

3. A class **method** is actually a function declared inside the class and able to access all the class's components.

4. The part of the Python class responsible for creating new objects is called the **constructor**, and it's implemented as a method of the name \_\_init\_\_.

5. Each class method declaration must contain at least one parameter (always the first one) usually referred to as self, and is used by the objects to identify themselves.

6. If we want to hide any of a class's components from the outside world, we should start its name with \_\_. Such components are called **private**.

**Exercise 1**

Assuming that there is a class named Snakes, write the very first line of the Python class declaration, expressing the fact that the new class is actually a subclass of Snake.

Check

class Python(Snakes):

**Exercise 2**

Something is missing from the following declaration – what?

class Snakes

def \_\_init\_\_():

self.sound = 'Sssssss'

Check

The \_\_init\_\_() constructor lacks the obligatory parameter (we should name it self to stay compliant with the standards).

**Exercise 3**

Modify the code to guarantee that the venomous property is private.

class Snakes

def \_\_init\_\_(self):

self.venomous = True

Check

The code should look as follows:  
  
class Snakes

def \_\_init\_\_(self):

self.\_\_venomous = True

[Prev](https://edube.org/learn/python-essentials-2/a-short-journey-from-procedural-to-object-approach-35) [Next](https://edube.org/learn/python-essentials-2/counting-stack)

# Instance variables

In general, a class can be equipped with two different kinds of data to form a class's properties. You already saw one of them when we were looking at stacks.

This kind of class property exists when and only when it is explicitly created and added to an object. As you already know, this can be done during the object's initialization, performed by the constructor.

Moreover, it can be done in any moment of the object's life. Furthermore, any existing property can be removed at any time.

Such an approach has some important consequences:

* different objects of the same class **may possess different sets of properties**;
* there must be a way to **safely check if a specific object owns the property** you want to utilize (unless you want to provoke an exception - it's always worth considering)
* each object **carries its own set of properties** - they don't interfere with one another in any way.

Such variables (properties) are called **instance variables**.

The word *instance* suggests that they are closely connected to the objects (which are class instances), not to the classes themselves. Let's take a closer look at them.

Here is an example:

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.first = val

def set\_second(self, val):

self.second = val

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_2.set\_second(3)

example\_object\_3 = ExampleClass(4)

example\_object\_3.third = 5

print(example\_object\_1.\_\_dict\_\_)

print(example\_object\_2.\_\_dict\_\_)

print(example\_object\_3.\_\_dict\_\_)

It needs one additional explanation before we go into any more detail. Take a look at the last three lines of the code.

Python objects, when created, are gifted with a **small set of predefined properties and methods**. Each object has got them, whether you want them or not. One of them is a variable named \_\_dict\_\_ (it's a dictionary).

The variable contains the names and values of all the properties (variables) the object is currently carrying. Let's make use of it to safely present an object's contents.

Let's dive into the code now:

* the class named ExampleClass has a constructor, which **unconditionally creates an instance variable** named first, and sets it with the value passed through the first argument (from the class user's perspective) or the second argument (from the constructor's perspective); note the default value of the parameter - any trick you can do with a regular function parameter can be applied to methods, too;

* the class also has a **method which creates another instance variable**, named second;

* we've created three objects of the class ExampleClass, but all these instances differ:

* + example\_object\_1 only has the property named first;

* + example\_object\_2 has two properties: first and second;

* + example\_object\_3 has been enriched with a property named third just on the fly, outside the class's code - this is possible and fully permissible.

The program's output clearly shows that our assumptions are correct - here it is:

{'first': 1}

{'second': 3, 'first': 2}

{'third': 5, 'first': 4}

**output**

There is one additional conclusion that should be stated here: **modifying an instance variable of any object has no impact on all the remaining objects**. Instance variables are perfectly isolated from each other.

[Prev](https://edube.org/learn/python-essentials-2/queue-aka-fifo-part-2) [Next](https://edube.org/learn/python-essentials-2/oop-properties-9)

**Instance variables: continued**

Take a look at the modified example in the editor.

It's nearly the same as the previous one. The only difference is in the property names. We've **added two underscores (**\_\_**)** in front of them.

As you know, such an addition makes the instance variable **private** - it becomes inaccessible from the outer world.

The actual behavior of these names is a bit more complicated, so let's run the program. This is the output:

{'\_ExampleClass\_\_first': 1}

{'\_ExampleClass\_\_first': 2, '\_ExampleClass\_\_second': 3}

{'\_ExampleClass\_\_first': 4, '\_\_third': 5}

**output**

Can you see these strange names full of underscores? Where did they come from?

When Python sees that you want to add an instance variable to an object and you're going to do it inside any of the object's methods, it **mangles the operation** in the following way:

* it puts a class name before your name;
* it puts an additional underscore at the beginning.

This is why the \_\_first becomes \_ExampleClass\_\_first.

**The name is now fully accessible from outside the class**. You can run a code like this:

print(example\_object\_1.\_ExampleClass\_\_first)

and you'll get a valid result with no errors or exceptions.

As you can see, making a property private is limited.

**The mangling won't work if you add a private instance variable outside the class code**. In this case, it'll behave like any other ordinary property.

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

def set\_second(self, val = 2):

self.\_\_second = val

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_2.set\_second(3)

example\_object\_3 = ExampleClass(4)

example\_object\_3.\_\_third = 5

print(example\_object\_1.\_\_dict\_\_)

print(example\_object\_2.\_\_dict\_\_)

print(example\_object\_3.\_\_dict\_\_)

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

def set\_second(self, val = 2):

self.\_\_second = val

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_2.set\_second(3)

example\_object\_3 = ExampleClass(4)

example\_object\_3.\_\_third = 5

print(example\_object\_1.\_\_dict\_\_)

print(example\_object\_2.\_\_dict\_\_)

print(example\_object\_3.\_\_dict\_\_)

**Class variables**

A class variable is **a property which exists in just one copy and is stored outside any object**.

Note: no instance variable exists if there is no object in the class; a class variable exists in one copy even if there are no objects in the class.

Class variables are created differently to their instance siblings. The example will tell you more:

class ExampleClass:

counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.counter += 1

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_3 = ExampleClass(4)

print(example\_object\_1.\_\_dict\_\_, example\_object\_1.counter)

print(example\_object\_2.\_\_dict\_\_, example\_object\_2.counter)

print(example\_object\_3.\_\_dict\_\_, example\_object\_3.counter)

Look:

* there is an assignment in the first list of the class definition - it sets the variable named counter to 0; initializing the variable inside the class but outside any of its methods makes the variable a class variable;
* accessing such a variable looks the same as accessing any instance attribute - you can see it in the constructor body; as you can see, the constructor increments the variable by one; in effect, the variable counts all the created objects.

Running the code will cause the following output:

{'\_ExampleClass\_\_first': 1} 3

{'\_ExampleClass\_\_first': 2} 3

{'\_ExampleClass\_\_first': 4} 3

**output**

Two important conclusions come from the example:

* class variables **aren't shown in an object's**\_\_dict\_\_ (this is natural as class variables aren't parts of an object) but you can always try to look into the variable of the same name, but at the class level - we'll show you this very soon;
* a class variable **always presents the same value** in all class instances (objects)

**Class variables: continued**

Mangling a class variable's name has the same effects as those you're already familiar with.

Look at the example in the editor. Can you guess its output?

Run the program and check if your predictions were correct. Everything works as expected, doesn't it?

class ExampleClass:

\_\_counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.\_\_counter += 1

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_3 = ExampleClass(4)

print(example\_object\_1.\_\_dict\_\_, example\_object\_1.\_ExampleClass\_\_counter)

print(example\_object\_2.\_\_dict\_\_, example\_object\_2.\_ExampleClass\_\_counter)

print(example\_object\_3.\_\_dict\_\_, example\_object\_3.\_ExampleClass\_\_counter)

# Class variables: continued

We told you before that class variables exist even when no class instance (object) had been created.

Now we're going to take the opportunity to show you **the difference between these two**\_\_dict\_\_**variables**, the one from the class and the one from the object.

Look at the code in the editor. The proof is there.

Let's take a closer look at it:

1. We define one class named ExampleClass;

1. The class defines one class variable named varia;

1. The class constructor sets the variable with the parameter's value;

1. Naming the variable is the most important aspect of the example because:
   * Changing the assignment to self.varia = val would create an instance variable of the same name as the class's one;
   * Changing the assignment to varia = val would operate on a method's local variable; (we strongly encourage you to test both of the above cases - this will make it easier for you to remember the difference)
2. The first line of the off-class code prints the value of the ExampleClass.varia attribute; note - we use the value before the very first object of the class is instantiated.

Run the code in the editor and check its output.

As you can see, the class' \_\_dict\_\_ contains much more data than its object's counterpart. Most of them are useless now - the one we want you to check carefully shows the current varia value.

Note that the object's \_\_dict\_\_ is empty - the object has no instance variables.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class ExampleClass:  
varia = 1  
def \_\_init\_\_(self, val):  
ExampleClass.varia = val  
  
  
print(ExampleClass.\_\_dict\_\_)  
example\_object = ExampleClass(2)  
  
print(ExampleClass.\_\_dict\_\_)  
print(example\_object.\_\_dict\_\_)



1

2

3

4

5

6

7

8

9

10

11

12

class ExampleClass:

varia = 1

def \_\_init\_\_(self, val):

ExampleClass.varia = val

print(ExampleClass.\_\_dict\_\_)

example\_object = ExampleClass(2)

print(ExampleClass.\_\_dict\_\_)

print(example\_object.\_\_dict\_\_)

* **Console**

class ExampleClass:

varia = 1

def \_\_init\_\_(self, val):

ExampleClass.varia = val

print(ExampleClass.\_\_dict\_\_)

example\_object = ExampleClass(2)

print(ExampleClass.\_\_dict\_\_)

print(example\_object.\_\_dict\_\_)

**Checking an attribute's existence**

Python's attitude to object instantiation raises one important issue - in contrast to other programming languages, **you may not expect that all objects of the same class have the same sets of properties**.

Just like in the example in the editor. Look at it carefully.

The object created by the constructor can have only one of two possible attributes: a or b.

Executing the code will produce the following output:

1

Traceback (most recent call last):

File ".main.py", line 11, in

print(example\_object.b)

AttributeError: 'ExampleClass' object has no attribute 'b'

**output**

As you can see, accessing a non-existing object (class) attribute causes an AttributeError exception.

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

print(example\_object.b)

# Checking an attribute's existence: continued

The try-except instruction gives you the chance to avoid issues with non-existent properties.

It's easy - look at the code in the editor.

As you can see, this action isn't very sophisticated. Essentially, we've just swept the issue under the carpet.

Fortunately, there is one more way to cope with the issue.

Python provides a **function which is able to safely check if any object/class contains a specified property**. The function is named hasattr, and expects two arguments to be passed to it:

* the class or the object being checked;
* the name of the property whose existence has to be reported (note: it has to be a string containing the attribute name, not the name alone)

The function returns True or False.

This is how you can utilize it:

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

if hasattr(example\_object, 'b'):

print(example\_object.b)

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class ExampleClass:  
def \_\_init\_\_(self, val):  
if val % 2 != 0:  
self.a = 1  
else:  
self.b = 1  
  
  
example\_object = ExampleClass(1)  
print(example\_object.a)  
  
try:  
print(example\_object.b)  
except AttributeError:  
pass



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

try:

print(example\_object.b)

except AttributeError:

pass

* **Console**

**Checking an attribute's existence: continued**

Don't forget that the hasattr() function can operate on classes, too. You can use it **to find out if a class variable is available**, just like here in the example in the editor.

The function returns True if the specified class contains a given attribute, and False otherwise.

Can you guess the code's output? Run it to check your guesses.

And one more example - look at the code below and try to predict its output:

class ExampleClass:

a = 1

def \_\_init\_\_(self):

self.b = 2

example\_object = ExampleClass()

print(hasattr(example\_object, 'b'))

print(hasattr(example\_object, 'a'))

print(hasattr(ExampleClass, 'b'))

print(hasattr(ExampleClass, 'a'))

Were you successful? Run the code to check your predictions.

Okay, we've made it to the end of this section. In the next section we're going to talk about methods, as methods drive the objects and make them active.

**Key takeaways**

1. An **instance variable** is a property whose existence depends on the creation of an object. Every object can have a different set of instance variables.

Moreover, they can be freely added to and removed from objects during their lifetime. All object instance variables are stored inside a dedicated dictionary named \_\_dict\_\_, contained in every object separately.

2. An instance variable can be private when its name starts with \_\_, but don't forget that such a property is still accessible from outside the class using a **mangled name** constructed as \_ClassName\_\_PrivatePropertyName.

3. A **class variable** is a property which exists in exactly one copy, and doesn't need any created object to be accessible. Such variables are not shown as \_\_dict\_\_ content.

All a class's class variables are stored inside a dedicated dictionary named \_\_dict\_\_, contained in every class separately.

4. A function named hasattr() can be used to determine if any object/class contains a specified property.

For example:

class Sample:

gamma = 0 # Class variable.

def \_\_init\_\_(self):

self.alpha = 1 # Instance variable.

self.\_\_delta = 3 # Private instance variable.

obj = Sample()

obj.beta = 2 # Another instance variable (existing only inside the "obj" instance.)

print(obj.\_\_dict\_\_)

The code outputs:

{'alpha': 1, '\_Sample\_\_delta': 3, 'beta': 2}

**output**

**Exercise 1**

Which of the Python class properties are instance variables and which are class variables? Which of them are private?  
  
class Python:

population = 1

victims = 0

def \_\_init\_\_(self):

self.length\_ft = 3

self.\_\_venomous = False

Check

**Exercise 2**

You're going to negate the \_\_venomous property of the version\_2 object, ignoring the fact that the property is private. How will you do this?

version\_2 = Python()

Check

**Exercise 3**

Write an expression which checks if the version\_2 object contains an instance property named constrictor (yes, constr**i**ctor!).

Check

**Methods in detail**

Let's summarize all the facts regarding the use of methods in Python classes.

As you already know, a **method is a function embedded inside a class**.

There is one fundamental requirement - a **method is obliged to have at least one parameter** (there are no such thing as parameterless methods - a method may be invoked without an argument, but not declared without parameters).

The first (or only) parameter is usually named self. We suggest that you follow the convention - it's commonly used, and you'll cause a few surprises by using other names for it.

The name self suggests the parameter's purpose - **it identifies the object for which the method is invoked**.

If you're going to invoke a method, you mustn't pass the argument for the self parameter - Python will set it for you.

The example in the editor shows the difference.

The code outputs:

method

**output**

Note the way we've created the object - we've **treated the class name like a function**, returning a newly instantiated object of the class.

If you want the method to accept parameters other than self, you should:

* place them after self in the method's definition;
* deliver them during invocation without specifying self (as previously)

Just like here:

class Classy:

def method(self, par):

print("method:", par)

obj = Classy()

obj.method(1)

obj.method(2)

obj.method(3)

The code outputs:

method: 1

method: 2

method: 3

**output**

class Classy:

def method(self):

print("method")

obj = Classy()

obj.method()

**Methods in detail: continued**

The self parameter is used **to obtain access to the object's instance and class variables**.

The example shows both ways of utilizing self:

class Classy:

varia = 2

def method(self):

print(self.varia, self.var)

obj = Classy()

obj.var = 3

obj.method()

The code outputs:

2 3

**output**

The self parameter is also used **to invoke other object/class methods from inside the class**.

Just like here:

class Classy:

def other(self):

print("other")

def method(self):

print("method")

self.other()

obj = Classy()

obj.method()

The code outputs:

method

other

**output**

# Methods in detail: continued

If you name a method like this: \_\_init\_\_, it won't be a regular method - it will be a **constructor**.

If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.

The constructor:

* is **obliged to have the**self**parameter** (it's set automatically, as usual);
* **may (but doesn't need to) have more parameters** than just self; if this happens, the way in which the class name is used to create the object must reflect the \_\_init\_\_ definition;
* **can be used to set up the object**, i.e., properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, etc.

Look at the code in the editor. The example shows a very simple constructor at work.

Run it. The code outputs:

object

**output**

Note that the constructor:

* **cannot return a value**, as it is designed to return a newly created object and nothing else;
* **cannot be invoked directly either from the object or from inside the class** (you can invoke a constructor from any of the object's subclasses, but we'll discuss this issue later.)

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Classy:  
def \_\_init\_\_(self, value):  
self.var = value  
  
  
obj\_1 = Classy("object")  
  
print(obj\_1.var)



1

2

3

4

5

6

7

8

9

class Classy:

def \_\_init\_\_(self, value):

self.var = value

obj\_1 = Classy("object")

print(obj\_1.var)

* **Console**

class Classy:

def \_\_init\_\_(self, value):

self.var = value

obj\_1 = Classy("object")

print(obj\_1.var)

rgument value. Test it.

The code outputs:

object

None

**output**

Everything we've said about **property name mangling** applies to method names, too - a method whose name starts with \_\_ is (partially) hidden.

The example shows this effect:

class Classy:

def visible(self):

print("visible")

def \_\_hidden(self):

print("hidden")

obj = Classy()

obj.visible()

try:

obj.\_\_hidden()

except:

print("failed")

obj.\_Classy\_\_hidden()

The code outputs:

visible

failed

hidden

**output**

Run the program, and test it.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Classy:  
def \_\_init\_\_(self, value = None):  
self.var = value  
  
  
obj\_1 = Classy("object")  
obj\_2 = Classy()  
  
print(obj\_1.var)  
print(obj\_2.var)



1

2

3

4

5

6

7

8

9

10

11

class Classy:

def \_\_init\_\_(self, value = None):

self.var = value

obj\_1 = Classy("object")

obj\_2 = Classy()

print(obj\_1.var)

print(obj\_2.var)

* **Console**

class Classy:

def \_\_init\_\_(self, value = None):

self.var = value

obj\_1 = Classy("object")

obj\_2 = Classy()

print(obj\_1.var)

print(obj\_2.var)

**The inner life of classes and objects**

Each Python class and each Python object is pre-equipped with a set of useful attributes which can be used to examine its capabilities.

You already know one of these - it's the \_\_dict\_\_ property.

Let's observe how it deals with methods - look at the code in the editor.

Run it to see what it outputs. Check the output carefully.

Find all the defined methods and attributes. Locate the context in which they exist: inside the object or inside the class.

class Classy:

varia = 1

def \_\_init\_\_(self):

self.var = 2

def method(self):

pass

def \_\_hidden(self):

pass

obj = Classy()

print(obj.\_\_dict\_\_)

print(Classy.\_\_dict\_\_)

# The inner life of classes and objects: continued

\_\_dict\_\_ is a dictionary. Another built-in property worth mentioning is \_\_name\_\_, which is a string.

The property contains **the name of the class**. It's nothing exciting, just a string.

Note: the \_\_name\_\_ attribute is absent from the object - **it exists only inside classes**.

If you want to **find the class of a particular object**, you can use a function named type(), which is able (among other things) to find a class which has been used to instantiate any object.

Look at the code in the editor, run it, and see for yourself.

The code outputs:

Classy

Classy

**output**

Note that a statement like this one:

print(obj.\_\_name\_\_)

will cause an error.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Classy:  
pass  
  
  
print(Classy.\_\_name\_\_)  
obj = Classy()  
print(type(obj).\_\_name\_\_)



1

2

3

4

5

6

7

8

class Classy:

pass

print(Classy.\_\_name\_\_)

obj = Classy()

print(type(obj).\_\_name\_\_)

* **Console**

class Classy:

pass

print(Classy.\_\_name\_\_)

obj = Classy()

print(type(obj).\_\_name\_\_)

# The inner life of classes and objects: continued

\_\_module\_\_ is a string, too - it **stores the name of the module which contains the definition of the class**.

Let's check it - run the code in the editor.

The code outputs:

\_\_main\_\_

\_\_main\_\_

**output**

As you know, any module named \_\_main\_\_ is actually not a module, but the **file currently being run**.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Classy:  
pass  
  
  
print(Classy.\_\_module\_\_)  
obj = Classy()  
print(obj.\_\_module\_\_)



1

2

3

4

5

6

7

8

class Classy:

pass

print(Classy.\_\_module\_\_)

obj = Classy()

print(obj.\_\_module\_\_)

* **Console**

class Classy:

pass

print(Classy.\_\_module\_\_)

obj = Classy()

print(obj.\_\_module\_\_)

# ontinued

\_\_bases\_\_ is a tuple. The **tuple contains classes** (not class names) which are direct superclasses for the class.

The order is the same as that used inside the class definition.

We'll show you only a very basic example, as we want to highlight **how inheritance works**.

Moreover, we're going to show you how to use this attribute when we discuss the objective aspects of exceptions.

Note: **only classes have this attribute** - objects don't.

We've defined a function named printbases(), designed to present the tuple's contents clearly.

Look at the code in the editor. Analyze it and run it. It will output:

( object )

( object )

( SuperOne SuperTwo )

**output**

Note: **a class without explicit superclasses points to object** (a predefined Python class) as its direct ancestor.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class SuperOne:  
pass  
  
  
class SuperTwo:  
pass  
  
  
class Sub(SuperOne, SuperTwo):  
pass  
  
  
def printBases(cls):  
print('( ', end='')  
  
for x in cls.\_\_bases\_\_:  
print(x.\_\_name\_\_, end=' ')  
print(')')  
  
  
printBases(SuperOne)  
printBases(SuperTwo)  
printBases(Sub)



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

class SuperOne:

pass

class SuperTwo:

pass

class Sub(SuperOne, SuperTwo):

pass

def printBases(cls):

print('( ', end='')

for x in cls.\_\_bases\_\_:

print(x.\_\_name\_\_, end=' ')

print(')')

* **Console**

class SuperOne:

pass

class SuperTwo:

pass

class Sub(SuperOne, SuperTwo):

pass

def printBases(cls):

print('( ', end='')

for x in cls.\_\_bases\_\_:

print(x.\_\_name\_\_, end=' ')

print(')')

printBases(SuperOne)

printBases(SuperTwo)

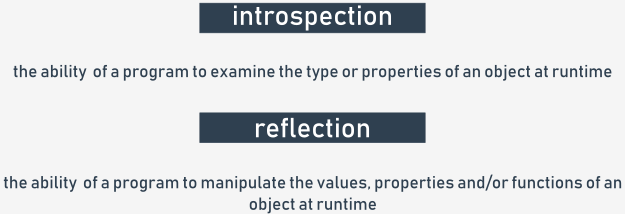
printBases(Sub)

**Reflection and introspection**

All these means allow the Python programmer to perform two important activities specific to many objective languages. They are:

* **introspection**, which is the ability of a program to examine the type or properties of an object at runtime;
* **reflection**, which goes a step further, and is the ability of a program to manipulate the values, properties and/or functions of an object at runtime.

In other words, you don't have to know a complete class/object definition to manipulate the object, as the object and/or its class contain the metadata allowing you to recognize its features during program execution.



y class, scans its contents in order to find all integer attributes with names starting with i, and increments them by one.

Impossible? Not at all!

This is how it works:

* line 1: define a very simple class...
* lines 3 through 10: ... and fill it with some attributes;
* line 12: this is our function!
* line 13: scan the \_\_dict\_\_ attribute, looking for all attribute names;
* line 14: if a name starts with i...
* line 15: ... use the getattr() function to get its current value; note: getattr() takes two arguments: an object, and its property name (as a string), and returns the current attribute's value;
* line 16: check if the value is of type integer, and use the function isinstance() for this purpose (we'll discuss this later);
* line 17: if the check goes well, increment the property's value by making use of the setattr() function; the function takes three arguments: an object, the property name (as a string), and the property's new value.

The code outputs:

{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}

{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}

**output**

That's all!

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class MyClass:  
pass  
  
  
obj = MyClass()  
obj.a = 1  
obj.b = 2  
obj.i = 3  
obj.ireal = 3.5  
obj.integer = 4  
obj.z = 5  
  
  
def incIntsI(obj):  
for name in obj.\_\_dict\_\_.keys():  
if name.startswith('i'):  
val = getattr(obj, name)  
if isinstance(val, int):  
setattr(obj, name, val + 1)  
  
  
print(obj.\_\_dict\_\_)  
incIntsI(obj)  
print(obj.\_\_dict\_\_)



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

class MyClass:

pass

obj = MyClass()

obj.a = 1

obj.b = 2

obj.i = 3

obj.ireal = 3.5

obj.integer = 4

obj.z = 5

def incIntsI(obj):

for name in obj.\_\_dict\_\_.keys():

if name.startswith('i'):

val = getattr(obj, name)

if isinstance(val, int):

setattr(obj, name, val + 1)

* **Console**

import math

class Point:

def \_\_init\_\_(self, x=0.0, y=0.0):

#

# Write code here

#

def getx(self):

#

# Write code here

#

def gety(self):

#

# Write code here

#

def distance\_from\_xy(self, x, y):

#

# Write code here

#

def distance\_from\_point(self, point):

#

# Write code here

#

point1 = Point(0, 0)

point2 = Point(1, 1)

print(point1.distance\_from\_point(point2))

print(point2.distance\_from\_xy(2, 0))

import math

class Point:

#

# The code copied from the previous lab.

#

class Triangle:

def \_\_init\_\_(self, vertice1, vertice2, vertice3):

#

# Write code here

#

def perimeter(self):

#

# Write code here

#

triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))

print(triangle.perimeter())

# Inheritance - why and how?

Before we start talking about inheritance, we want to present a new, handy mechanism utilized by Python's classes and objects - it's **the way in which the object is able to introduce itself**.

Let's start with an example. Look at the code in the editor.

The program prints out just one line of text, which in our case is this:

<\_\_main\_\_.Star object at 0x7f1074cc7c50>

**output**

If you run the same code on your computer, you'll see something very similar, although the hexadecimal number (the substring starting with 0x) will be different, as it's just an internal object identifier used by Python, and it's unlikely that it would appear the same when the same code is run in a different environment.

As you can see, the printout here isn't really useful, and something more specific, or just prettier, may be more preferable.

Fortunately, Python offers just such a function.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Star:  
def \_\_init\_\_(self, name, galaxy):  
self.name = name  
self.galaxy = galaxy  
  
  
sun = Star("Sun", "Milky Way")  
print(sun)



1

2

3

4

5

6

7

8

9

class Star:

def \_\_init\_\_(self, name, galaxy):

self.name = name

self.galaxy = galaxy

sun = Star("Sun", "Milky Way")

print(sun)

* **Console**

# Inheritance - why and how?

When Python needs any class/object to be presented as a string (putting an object as an argument in the print() function invocation fits this condition) it tries to invoke a method named \_\_str\_\_() from the object and to use the string it returns.

The default \_\_str\_\_() method returns the previous string - ugly and not very informative. You can change it just by **defining your own method of the name**.

We've just done it - look at the code in the editor.

This new \_\_str\_\_() method makes a string consisting of the star's and galaxy's names - nothing special, but the print results look better now, doesn't it?

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

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Star:  
def \_\_init\_\_(self, name, galaxy):  
self.name = name  
self.galaxy = galaxy  
  
def \_\_str\_\_(self):  
return self.name + ' in ' + self.galaxy  
  
  
sun = Star("Sun", "Milky Way")  
print(sun)



1

2

3

4

5

6

7

8

9

10

11

12

class Star:

def \_\_init\_\_(self, name, galaxy):

self.name = name

self.galaxy = galaxy

def \_\_str\_\_(self):

return self.name + ' in ' + self.galaxy

sun = Star("Sun", "Milky Way")

print(sun)

* **Console**

[Prev](https://edube.org/learn/python-essentials-2/oop-fundamentals-inheritance-39)

**Inheritance - why and how?**

The term inheritance is older than computer programming, and it describes the common practice of passing different goods from one person to another upon that person's death. The term, when related to computer programming, has an entirely different meaning.



Let's define the term for our purposes:

Inheritance is a common practice (in object programming) of **passing attributes and methods from the superclass (defined and existing) to a newly created class, called the subclass**.

In other words, inheritance is **a way of building a new class, not from scratch, but by using an already defined repertoire of traits**. The new class inherits (and this is the key) all the already existing equipment, but is able to add some new ones if needed.

Thanks to that, it's possible to **build more specialized (more concrete) classes** using some sets of predefined general rules and behaviors.

The most important factor of the process is the relation between the superclass and all of its subclasses (note: if *B* is a subclass of *A* and *C* is a subclass of *B*, this also means than *C* is a subclass of *A*, as the relationship is fully transitive).

A very simple example of **two-level inheritance** is presented here:

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

All the presented classes are empty for now, as we're going to show you how the mutual relations between the super- and subclasses work. We'll fill them with contents soon.

We can say that:

* The Vehicle class is the superclass for both the LandVehicle and TrackedVehicle classes;
* The LandVehicle class is a subclass of Vehicle and a superclass of TrackedVehicle at the same time;
* The TrackedVehicle class is a subclass of both the Vehicle and LandVehicle classes.

The above knowledge comes from reading the code (in other words, we know it because we can see it).

Does Python know the same? Is it possible to ask Python about it? Yes, it is.

**Inheritance: issubclass()**

Python offers a function which is able to **identify a relationship between two classes**, and although its diagnosis isn't complex, it can **check if a particular class is a subclass of any other class**.

This is how it looks:

issubclass(ClassOne, ClassTwo)

The function returns True if ClassOne is a subclass of ClassTwo, and False otherwise.

Let's see it in action - it may surprise you. Look at the code in the editor. Read it carefully.

There are two nested loops. Their purpose is to **check all possible ordered pairs of classes, and to print the results of the check to determine whether the pair matches the subclass-superclass relationship**.

Run the code. The program produces the following output:

True False False

True True False

True True True

**output**

Let's make the result more readable:

| **↓ is a subclass of →** | **Vehicle** | **LandVehicle** | **TrackedVehicle** |
| --- | --- | --- | --- |
| **Vehicle** | True | False | False |
| **LandVehicle** | True | True | False |
| **TrackedVehicle** | True | True | True |

There is one important observation to make: **each class is considered to be a subclass of itself**.

lready know, **an object is an incarnation of a class**. This means that the object is like a cake baked using a recipe which is included inside the class.

This can generate some important issues.

Let's assume that you've got a cake (e.g., as an argument passed to your function). You want to know what recipe has been used to make it. Why? Because you want to know what to expect from it, e.g., whether it contains nuts or not, which is crucial information to some people.

Similarly, it can be crucial if the object does have (or doesn't have) certain characteristics. In other words, **whether it is an object of a certain class or not**.

Such a fact could be detected by the function named isinstance():

isinstance(objectName, ClassName)

The functions returns True if the object is an instance of the class, or False otherwise.

**Being an instance of a class means that the object (the cake) has been prepared using a recipe contained in either the class or one of its superclasses**.

Don't forget: if a subclass contains at least the same equipment as any of its superclasses, it means that objects of the subclass can do the same as objects derived from the superclass, ergo, it's an instance of its home class and any of its superclasses.

Let's test it. Analyze the code in the editor.

We've created three objects, one for each of the classes. Next, using two nested loops, we check all possible object-class pairs **to find out if the objects are instances of the classes**.

Run the code.

This is what we get:

True False False

True True False

True True True

**output**

Let's make the result more readable once again:

| **↓ is an instance of →** | **Vehicle** | **LandVehicle** | **TrackedVehicle** |
| --- | --- | --- | --- |
| **my\_vehicle** | True | False | False |
| **my\_land\_vehicle** | True | True | False |
| **my\_tracked\_vehicle** | True | True | True |

Does the table confirm our expectations?

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

my\_vehicle = Vehicle()

my\_land\_vehicle = LandVehicle()

my\_tracked\_vehicle = TrackedVehicle()

for obj in [my\_vehicle, my\_land\_vehicle, my\_tracked\_vehicle]:

for cls in [Vehicle, LandVehicle, TrackedVehicle]:

print(isinstance(obj, cls), end="\t")

print()

**Inheritance: the is operator**

There is also a Python operator worth mentioning, as it refers directly to objects - here it is:

object\_one is object\_two

**The**is**operator checks whether two variables (**object\_one**and**object\_two**here) refer to the same object**.

Don't forget that **variables don't store the objects themselves, but only the handles pointing to the internal Python memory**.

Assigning a value of an object variable to another variable doesn't copy the object, but only its handle. This is why an operator like is may be very useful in particular circumstances.

Take a look at the code in the editor. Let's analyze it:

* there is a very simple class equipped with a simple constructor, creating just one property. The class is used to instantiate two objects. The former is then assigned to another variable, and its val property is incremented by one.
* afterward, the is operator is applied three times to check all possible pairs of objects, and all val property values are also printed.
* the last part of the code carries out another experiment. After three assignments, both strings contain the same texts, but **these texts are stored in different objects**.

The code prints:

False

False

True

1 2 1

True False

**output**

The results prove that object\_1 and object\_3 are actually the same objects, while string\_1 and string\_2 aren't, despite their contents being the same.

class SampleClass:

def \_\_init\_\_(self, val):

self.val = val

object\_1 = SampleClass(0)

object\_2 = SampleClass(2)

object\_3 = object\_1

object\_3.val += 1

print(object\_1 is object\_2)

print(object\_2 is object\_3)

print(object\_3 is object\_1)

print(object\_1.val, object\_2.val, object\_3.val)

string\_1 = "Mary had a little "

string\_2 = "Mary had a little lamb"

string\_1 += "lamb"

print(string\_1 == string\_2, string\_1 is string\_2)

# ed

Look at the code in the editor. We've modified it to show you another method of accessing any entity defined inside the superclass.

In the last example, we explicitly named the superclass. In this example, we make use of the super() function, which **accesses the superclass without needing to know its name**:

super().\_\_init\_\_(name)

The super() function creates a context in which you don't have to (moreover, you mustn't) pass the self argument to the method being invoked - this is why it's possible to activate the superclass constructor using only one argument.

Note: you can use this mechanism not only to **invoke the superclass constructor, but also to get access to any of the resources available inside the superclass**.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Super:  
def \_\_init\_\_(self, name):  
self.name = name  
  
def \_\_str\_\_(self):  
return "My name is " + self.name + "."  
  
  
class Sub(Super):  
def \_\_init\_\_(self, name):  
super().\_\_init\_\_(name)  
  
  
obj = Sub("Andy")  
  
print(obj)



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

class Super:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name + "."

class Sub(Super):

def \_\_init\_\_(self, name):

super().\_\_init\_\_(name)

obj = Sub("Andy")

print(obj)

* **Console**

**How Python finds properties and methods: continued**

Let's try to do something similar, but with properties (more precisely: with **class variables**).

Take a look at the example in the editor.

As you can see, the Super class defines one class variable named supVar, and the Sub class defines a variable named subVar.

Both these variables are visible inside the object of class Sub - this is why the code outputs:

# Testing properties: class variables.

class Super:

supVar = 1

class Sub(Super):

subVar = 2

obj = Sub()

print(obj.subVar)

print(obj.supVar)

# Testing properties: instance variables.

class Super:

def \_\_init\_\_(self):

self.supVar = 11

class Sub(Super):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.subVar = 12

obj = Sub()

print(obj.subVar)

print(obj.supVar)

# How Python finds properties and methods: continued

It's now possible to formulate a general statement describing Python's behavior.

When you try to access any object's entity, Python will try to (in this order):

* find it **inside the object** itself;
* find it **in all classes** involved in the object's inheritance line from bottom to top;

If both of the above fail, an **exception (**AttributeError**) is raised**.

The first condition may need some additional attention. As you know, all objects deriving from a particular class may have different sets of attributes, and some of the attributes may be added to the object a long time after the object's creation.

The example in the editor summarizes this in a **three-level inheritance line**. Analyze it carefully.

All the comments we've made so far are related to **single inheritance**, when a subclass has exactly one superclass. This is the most common situation (and the recommended one, too).

Python, however, offers much more here. In the next lessons we're going to show you some examples of **multiple inheritance**.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Level1:  
variable\_1 = 100  
def \_\_init\_\_(self):  
self.var\_1 = 101  
  
def fun\_1(self):  
return 102  
  
  
class Level2(Level1):  
variable\_2 = 200  
def \_\_init\_\_(self):  
super().\_\_init\_\_()  
self.var\_2 = 201  
  
def fun\_2(self):  
return 202  
  
  
class Level3(Level2):  
variable\_3 = 300  
def \_\_init\_\_(self):  
super().\_\_init\_\_()  
self.var\_3 = 301  
  
def fun\_3(self):  
return 302  
  
  
obj = Level3()  
  
print(obj.variable\_1, obj.var\_1, obj.fun\_1())  
print(obj.variable\_2, obj.var\_2, obj.fun\_2())  
print(obj.variable\_3, obj.var\_3, obj.fun\_3())



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

class Level1:

variable\_1 = 100

def \_\_init\_\_(self):

self.var\_1 = 101

def fun\_1(self):

return 102

class Level2(Level1):

variable\_2 = 200

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.var\_2 = 201

def fun\_2(self):

return 202

* **Console**

class Level1:

variable\_1 = 100

def \_\_init\_\_(self):

self.var\_1 = 101

def fun\_1(self):

return 102

class Level2(Level1):

variable\_2 = 200

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.var\_2 = 201

def fun\_2(self):

return 202

class Level3(Level2):

variable\_3 = 300

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.var\_3 = 301

def fun\_3(self):

return 302

obj = Level3()

print(obj.variable\_1, obj.var\_1, obj.fun\_1())

print(obj.variable\_2, obj.var\_2, obj.fun\_2())

print(obj.variable\_3, obj.var\_3, obj.fun\_3())

ere:

class SuperA:

var\_a = 10

def fun\_a(self):

return 11

class SuperB:

var\_b = 20

def fun\_b(self):

return 21

class Sub(SuperA, SuperB):

pass

obj = Sub()

print(obj.var\_a, obj.fun\_a())

print(obj.var\_b, obj.fun\_b())

The Sub class has two superclasses: SuperA and SuperB. This means that the Sub class **inherits all the goods offered by both**SuperA**and**SuperB.

The code prints:

10 11

20 21

**output**

Now it's time to introduce a brand new term - **overriding**.

What do you think will happen if more than one of the superclasses defines an entity of a particular name?

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class SuperA:  
var\_a = 10  
def fun\_a(self):  
return 11  
  
  
class SuperB:  
var\_b = 20  
def fun\_b(self):  
return 21  
  
  
class Sub(SuperA, SuperB):  
pass  
  
  
obj = Sub()  
  
print(obj.var\_a, obj.fun\_a())  
print(obj.var\_b, obj.fun\_b())



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

class SuperA:

var\_a = 10

def fun\_a(self):

return 11

class SuperB:

var\_b = 20

def fun\_b(self):

return 21

class Sub(SuperA, SuperB):

pass

obj = Sub()

print(obj.var\_a, obj.fun\_a())

* **Console**

**How Python finds properties and methods: continued**

Let's analyze the example in the editor.

Both, Level1 and Level2 classes define a method named fun() and a property named var. Does this mean that the Level3 class object will be able to access two copies of each entity? Not at all.

**The entity defined later (in the inheritance sense) overrides the same entity defined earlier**. This is why the code produces the following output:

200 201

**output**

As you can see, the var class variable and fun() method from the Level2 class override the entities of the same names derived from the Level1 class.

This feature can be intentionally used to modify default (or previously defined) class behaviors when any of its classes needs to act in a different way to its ancestor.

We can also say that **Python looks for an entity from bottom to top**, and is fully satisfied with the first entity of the desired name.

How does it work when a class has two ancestors offering the same entity, and they lie on the same level? In other words, what should you expect when a class emerges using multiple inheritance? Let's look at this.

class Level1:

var = 100

def fun(self):

return 101

class Level2(Level1):

var = 200

def fun(self):

return 201

class Level3(Level2):

pass

obj = Level3()

print(obj.var, obj.fun())

# How Python finds properties and methods: continued

Let's take a look at the example in the editor.

The Sub class inherits goods from two superclasses, Left and Right (these names are intended to be meaningful).

There is no doubt that the class variable var\_right comes from the Right class, and var\_left comes from Left respectively.

This is clear. But where does var come from? Is it possible to guess it? The same problem is encountered with the fun() method - will it be invoked from Left or from Right? Let's run the program - its output is:

L LL RR Left

**output**

This proves that both unclear cases have a solution inside the Left class. Is this a sufficient premise to formulate a general rule? Yes, it is.

We can say that **Python looks for object components** in the following order:

* **inside the object** itself;
* **in its superclasses**, from bottom to top;
* if there is more than one class on a particular inheritance path, Python scans them from left to right.

Do you need anything more? Just make a small amendment in the code - replace: class Sub(Left, Right): with: class Sub(Right, Left):, then run the program again, and see what happens.

What do you see now? We see:

R LL RR Right

**output**

Do you see the same, or something different?

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

class Left:  
var = "L"  
var\_left = "LL"  
def fun(self):  
return "Left"  
  
  
class Right:  
var = "R"  
var\_right = "RR"  
def fun(self):  
return "Right"  
  
  
class Sub(Left, Right):  
pass  
  
  
obj = Sub()  
  
print(obj.var, obj.var\_left, obj.var\_right, obj.fun())



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

class Left:

var = "L"

var\_left = "LL"

def fun(self):

return "Left"

class Right:

var = "R"

var\_right = "RR"

def fun(self):

return "Right"

class Sub(Left, Right):

pass

obj = Sub()

* **Console**

[Prev](https://edube.org/learn/python-essentials-2/oop-fundamentals-inheritance-31)

**How to build a hierarchy of classes**

Building a hierarchy of classes isn't just art for art's sake.

If you divide a problem among classes and decide which of them should be located at the top and which should be placed at the bottom of the hierarchy, you have to carefully analyze the issue, but before we show you how to do it (and how not to do it), we want to highlight an interesting effect. It's nothing extraordinary (it's just a consequence of the general rules presented earlier), but remembering it may be key to understanding how some codes work, and how the effect may be used to build a flexible set of classes.

Take a look at the code in the editor. Let's analyze it:

* there are two classes, named One and Two, while Two is derived from One. Nothing special. However, one thing looks remarkable - the do\_it() method.
* the do\_it()method is **defined twice**: originally inside One and subsequently inside Two. The essence of the example lies in the fact that it is **invoked just once** - inside One.

The question is - which of the two methods will be invoked by the last two lines of the code?

The first invocation seems to be simple, and it is simple, actually - invoking doanything() from the object named one will obviously activate the first of the methods.

The second invocation needs some attention. It's simple, too if you keep in mind how Python finds class components. The second invocation will launch do\_it() in the form existing inside the Two class, regardless of the fact that the invocation takes place within the One class.

In effect, the code produces the following output:

do\_it from One

do\_it from Two

**output**

Note: the situation in which **the subclass is able to modify its superclass behavior (just like in the example) is called polymorphism**. The word comes from Greek (polys: "many, much" and morphe, "form, shape"), which means that one and the same class can take various forms depending on the redefinitions done by any of its subclasses.

The method, redefined in any of the superclasses, thus changing the behavior of the superclass, is called **virtual**.

In other words, no class is given once and for all. Each class's behavior may be modified at any time by any of its subclasses.

We're going to show you **how to use polymorphism to extend class flexibility**.

class One:

def do\_it(self):

print("do\_it from One")

def doanything(self):

self.do\_it()

class Two(One):

def do\_it(self):

print("do\_it from Two")

one = One()

two = Two()

one.doanything()

two.doanything()

# How to build a hierarchy of classes: continued

Look at the example in the editor.

Does it resemble anything? Yes, of course it does. It refers to the example shown at the beginning of the module when we talked about the general concepts of objective programming.

It may look weird, but we didn't use inheritance in any way - just to show you that it doesn't limit us, and we managed to get ours.

We defined two separate classes able to produce two different kinds of land vehicles. The main difference between them is in how they turn. A wheeled vehicle just turns the front wheels (generally). A tracked vehicle has to stop one of the tracks.

Can you follow the code?

* a tracked vehicle performs a turn by stopping and moving on one of its tracks (this is done by the control\_track() method, which will be implemented later)
* a wheeled vehicle turns when its front wheels turn (this is done by the turn\_front\_wheels() method)
* the turn() method uses the method suitable for each particular vehicle.

Can you see **what's wrong with the code**?

The turn() methods look too similar to leave them in this form.

Let's rebuild the code - we're going to introduce a superclass to gather all the similar aspects of the driving vehicles, moving all the specifics to the subclasses.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

import time  
  
class TrackedVehicle:  
def control\_track(left, stop):  
pass  
  
def turn(left):  
control\_track(left, True)  
time.sleep(0.25)  
control\_track(left, False)  
  
  
class WheeledVehicle:  
def turn\_front\_wheels(left, on):  
pass  
  
def turn(left):  
turn\_front\_wheels(left, True)  
time.sleep(0.25)  
turn\_front\_wheels(left, False)



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

import time

class TrackedVehicle:

def control\_track(left, stop):

pass

def turn(left):

control\_track(left, True)

time.sleep(0.25)

control\_track(left, False)

class WheeledVehicle:

def turn\_front\_wheels(left, on):

pass

def turn(left):

turn\_front\_wheels(left, True)

time.sleep(0.25)

* **Console**

import time

class TrackedVehicle:

def control\_track(left, stop):

pass

def turn(left):

control\_track(left, True)

time.sleep(0.25)

control\_track(left, False)

class WheeledVehicle:

def turn\_front\_wheels(left, on):

pass

def turn(left):

turn\_front\_wheels(left, True)

time.sleep(0.25)

turn\_front\_wheels(left, False)

# How to build a hierarchy of classes: continued

Look at the code in the editor again. This is what we've done:

* we defined a superclass named Vehicle, which uses the turn() method to implement a general scheme of turning, while the turning itself is done by a method named change\_direction(); note: the former method is empty, as we are going to put all the details into the subclass (such a method is often called an **abstract method**, as it only demonstrates some possibility which will be instantiated later)
* we defined a subclass named TrackedVehicle (note: it's derived from the Vehicle class) which instantiated the change\_direction() method by using the specific (concrete) method named control\_track()
* respectively, the subclass named WheeledVehicle does the same trick, but uses the turn\_front\_wheels() method to force the vehicle to turn.

The most important advantage (omitting readability issues) is that this form of code enables you to implement a brand new turning algorithm just by modifying the turn() method, which can be done in just one place, as all the vehicles will obey it.

This is how **polymorphism helps the developer to keep the code clean and consistent**.

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

import time  
  
class Vehicle:  
def change\_direction(left, on):  
pass  
  
def turn(left):  
change\_direction(left, True)  
time.sleep(0.25)  
change\_direction(left, False)  
  
  
class TrackedVehicle(Vehicle):  
def control\_track(left, stop):  
pass  
  
def change\_direction(left, on):  
control\_track(left, on)  
  
  
class WheeledVehicle(Vehicle):  
def turn\_front\_wheels(left, on):  
pass  
  
def change\_direction(left, on):  
turn\_front\_wheels(left, on)



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

import time

class Vehicle:

def change\_direction(left, on):

pass

def turn(left):

change\_direction(left, True)

time.sleep(0.25)

change\_direction(left, False)

class TrackedVehicle(Vehicle):

def control\_track(left, stop):

pass

def change\_direction(left, on):

control\_track(left, on)

* **Console**

import time

class Vehicle:

def change\_direction(left, on):

pass

def turn(left):

change\_direction(left, True)

time.sleep(0.25)

change\_direction(left, False)

class TrackedVehicle(Vehicle):

def control\_track(left, stop):

pass

def change\_direction(left, on):

control\_track(left, on)

class WheeledVehicle(Vehicle):

def turn\_front\_wheels(left, on):

pass

def change\_direction(left, on):

turn\_front\_wheels(left, on)

**ntinued**

Inheritance is not the only way of constructing adaptable classes. You can achieve the same goals (not always, but very often) by using a technique named composition.

**Composition is the process of composing an object using other different objects**. The objects used in the composition deliver a set of desired traits (properties and/or methods) so we can say that they act like blocks used to build a more complicated structure.

It can be said that:

* **inheritance extends a class's capabilities** by adding new components and modifying existing ones; in other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them;
* **composition projects a class as a container** able to store and use other objects (derived from other classes) where each of the objects implements a part of a desired class's behavior.

Let us illustrate the difference by using the previously defined vehicles. The previous approach led us to a hierarchy of classes in which the top-most class was aware of the general rules used in turning the vehicle, but didn't know how to control the appropriate components (wheels or tracks).

The subclasses implemented this ability by introducing specialized mechanisms. Let's do (almost) the same thing, but using composition. The class - like in the previous example - is aware of how to turn the vehicle, but the actual turn is done by a specialized object stored in a property named controller. The controller is able to control the vehicle by manipulating the relevant vehicle's parts.

Take a look into the editor - this is how it could look.

There are two classes named Tracks and Wheels - they know how to control the vehicle's direction. There is also a class named Vehicle which can use any of the available controllers (the two already defined, or any other defined in the future) - the controller itself is passed to the class during initialization.

In this way, the vehicle's ability to turn is composed using an external object, not implemented inside the Vehicle class.

In other words, we have a universal vehicle and can install either tracks or wheels onto it.

The code produces the following output:

wheels: True True

wheels: True False

tracks: False True

tracks: False False

**output**

import time

class Tracks:

def change\_direction(self, left, on):

print("tracks: ", left, on)

class Wheels:

def change\_direction(self, left, on):

print("wheels: ", left, on)

class Vehicle:

def \_\_init\_\_(self, controller):

self.controller = controller

def turn(self, left):

self.controller.change\_direction(left, True)

time.sleep(0.25)

self.controller.change\_direction(left, False)

wheeled = Vehicle(Wheels())

tracked = Vehicle(Tracks())

wheeled.turn(True)

tracked.turn(False)

# Single inheritance vs. multiple inheritance

As you already know, there are no obstacles to using multiple inheritance in Python. You can derive any new class from more than one previously defined classes.

There is only one "but". The fact that you can do it does not mean you have to.

Don't forget that:

* a single inheritance class is always simpler, safer, and easier to understand and maintain;

* multiple inheritance is always risky, as you have many more opportunities to make a mistake in identifying these parts of the superclasses which will effectively influence the new class;

* multiple inheritance may make overriding extremely tricky; moreover, using the super() function becomes ambiguous;

* multiple inheritance violates the **single responsibility principle** (more details here: <https://en.wikipedia.org/wiki/Single_responsibility_principle>) as it makes a new class of two (or more) classes that know nothing about each other;

* we strongly suggest multiple inheritance as the last of all possible solutions - if you really need the many different functionalities offered by different classes, composition may be a better alternative.

**What is Method Resolution Order (MRO) and why is it that not all inheritances make sense?**

MRO, in general, is a way (you can call it a **strategy**) in which a particular programming language scans through the upper part of a class’s hierarchy in order to find the method it currently needs. It's worth emphasizing that different languages use slightly (or even completely) different MROs. Python is a unique creature in this respect, however, and its customs are a bit specific.

We're going to show you how Python's MRO works in two peculiar cases that are clear-cut examples of problems which may occur when you try to use multiple inheritance too recklessly. Let's start with a snippet that initially may look simple. Look at what we've prepared for you in the editor.

We're sure that if you analyze the snippet yourself, you won't see any anomalies in it. Yes, you're perfectly right - it looks clear and simple, and raises no concerns. If you run the code, it will produce the following, predictable output:

bottom

middle

top

**output**

No surprises so far. Let's make a tiny change to this code. Have a look:

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Middle, Top):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Can you see the difference? It's hidden in this line:

class Bottom(Middle, Top):

In this exotic way, we've turned a very simple code with a clear single-inheritance path into a mysterious multiple-inheritance riddle. “Is it valid?” you may ask. Yes, it is. “How is that possible?” you should ask now, and we hope that you really feel the need to ask this question.

As you can see, the order in which the two superclasses have been listed between parenthesis is compliant with the code's structure: the Middle class precedes the Top class, just like in the real inheritance path.

Despite its oddity, the sample is correct and works as expected, but it has to be stated that this notation doesn’t bring any new functionality or additional meaning.

Let's modify the code once again - now we'll swap both superclass names in the Bottom class definition. This is what the snippet looks like now:

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Top, Middle):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

To anticipate your question, we’ll say that this amendment has spoiled the code, and it won't run anymore. What a pity. The order we tried to force (Top, Middle) is incompatible with the inheritance path which is derived from the code's structure. Python won't like it. This is what we'll see:

TypeError: Cannot create a consistent method resolution order (MRO) for bases Top, Middle

**output**

We think that the message speaks for itself. Python's MRO cannot be bent or violated, not just because that's the way Python works, but also because it’s a rule you have to obey.

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Middle):

def m\_bottom(self):

print("bottom")

object = Bottom()

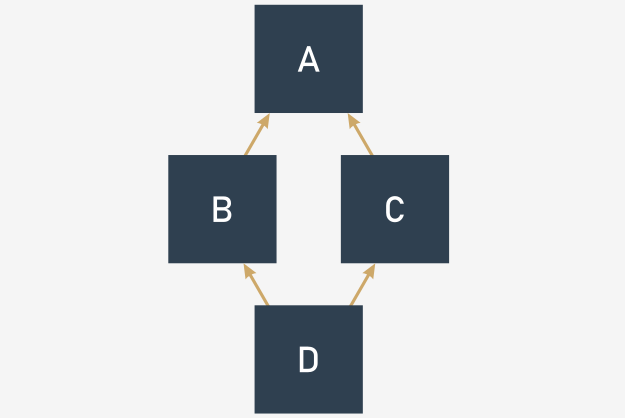
object.m\_bottom()

object.m\_middle()

object.m\_top()

**The diamond problem**

The second example of the spectrum of issues that can possibly arise from multiple inheritance is illustrated by a classic problem named the **diamond problem**. The name reflects the shape of the inheritance diagram - take a look at the picture:



* There is the top-most superclass named A;
* there are two subclasses derived from A: B and C;
* and there is also the bottom-most subclass named D, derived from B and C (or C and B, as these two variants mean different things in Python)

Can you see the diamond there?

Have a look at the code in the editor. The same structure, but expressed in Python.

Some programming languages forbid multiple inheritance at all, and as a consequence, they won't let you build a diamond - this is the route that Java and C# have chosen to follow since their origins.

Python, however, has chosen a different route - it allows multiple inheritance, and it doesn't mind if you write and run code like the one in the editor. But don't forget about MRO - it's always in charge.

Let's rebuild our example from the previous page to make it more diamond-like, just like below:

class Top:

def m\_top(self):

print("top")

class Middle\_Left(Top):

def m\_middle(self):

print("middle\_left")

class Middle\_Right(Top):

def m\_middle(self):

print("middle\_right")

class Bottom(Middle\_Left, Middle\_Right):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Note: both Middle classes define **a method of the same name**: m\_middle().

It introduces a small uncertainty to our sample, although we're absolutely sure that you can answer the following key question: which of the two m\_middle() methods will actually be invoked when the following line is executed?

Object.m\_middle()

In other words, what will you see on the screen: middle\_left or middle\_right?

You don't need to hurry – think twice and keep Python's MRO in mind!

Are you ready?

Yes, you're right. The invocation will activate the m\_middle() method, which comes from the Middle\_Left class. The explanation is simple: the class is listed before Middle\_Right on the Bottom class's inheritance list. If you want to make sure that there’s no doubt about it, try to swap these two classes on the list and check the results.

If you want to experience some more profound impressions about multiple inheritance and precious gemstones, try to modify our snippet and equip the Upper class with another specimen of the m\_middle() method, and investigate its behavior carefully.

As you can see, diamonds may bring some problems into your life – both the real ones and those offered by Python.

class A:

pass

class B(A):

pass

class C(A):

pass

class D(B, C):

pass

d = D()

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

return None

else:

print("Everything went fine")

return n

print(reciprocal(2))

print(reciprocal(0))

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

n = None

else:

print("Everything went fine")

finally:

print("It's time to say goodbye")

return n

print(reciprocal(2))

print(reciprocal(0))

**Exceptions are classes**

All the previous examples were content with detecting a specific kind of exception and responding to it in an appropriate way. Now we're going to delve deeper, and look inside the exception itself.

You probably won't be surprised to learn that **exceptions are classes**. Furthermore, when an exception is raised, an object of the class is instantiated, and goes through all levels of program execution, looking for the except branch that is prepared to deal with it.

Such an object carries some useful information which can help you to precisely identify all aspects of the pending situation. To achieve that goal, Python offers a special variant of the exception clause - you can find it in the editor.

As you can see, the except statement is extended, and contains an additional phrase starting with the as keyword, followed by an identifier. The identifier is designed to catch the exception object so you can analyze its nature and draw proper conclusions.

Note: the identifier's scope covers its except branch, and doesn't go any further.

The example presents a very simple way of utilizing the received object - just print it out (as you can see, the output is produced by the object's \_\_str\_\_() method) and it contains a brief message describing the reason.

The same message will be printed if there is no fitting except block in the code, and Python is forced to handle it alone.

# Exceptions are classes

All the built-in Python exceptions form a hierarchy of classes. There is no obstacle to extending it if you find it reasonable.

Look at the code in the editor.

This program dumps all predefined exception classes in the form of a tree-like printout.

As **a tree is a perfect example of a recursive data structure**, a recursion seems to be the best tool to traverse through it. The print\_exception\_tree() function takes two arguments:

* a point inside the tree from which we start traversing the tree;
* a nesting level (we'll use it to build a simplified drawing of the tree's branches)

Let's start from the tree's root - the root of Python's exception classes is the BaseException class (it's a superclass of all other exceptions).

For each of the encountered classes, perform the same set of operations:

* print its name, taken from the \_\_name\_\_ property;
* iterate through the list of subclasses delivered by the \_\_subclasses\_\_() method, and recursively invoke the print\_exception\_tree() function, incrementing the nesting level respectively.

Note how we've drawn the branches and forks. The printout isn't sorted in any way - you can try to sort it yourself, if you want a challenge. Moreover, there are some subtle inaccuracies in the way in which some branches are presented. That can be fixed, too, if you wish.

This is how it looks:

BaseException

+---Exception

| +---TypeError

| +---StopAsyncIteration

| +---StopIteration

| +---ImportError

| | +---ModuleNotFoundError

| | +---ZipImportError

| +---OSError

| | +---ConnectionError

| | | +---BrokenPipeError

| | | +---ConnectionAbortedError

| | | +---ConnectionRefusedError

| | | +---ConnectionResetError

| | +---BlockingIOError

| | +---ChildProcessError

| | +---FileExistsError

| | +---FileNotFoundError

| | +---IsADirectoryError

| | +---NotADirectoryError

| | +---InterruptedError

| | +---PermissionError

| | +---ProcessLookupError

| | +---TimeoutError

| | +---UnsupportedOperation

| | +---herror

| | +---gaierror

| | +---timeout

| | +---Error

| | | +---SameFileError

| | +---SpecialFileError

| | +---ExecError

| | +---ReadError

| +---EOFError

| +---RuntimeError

| | +---RecursionError

| | +---NotImplementedError

| | +---\_DeadlockError

| | +---BrokenBarrierError

| +---NameError

| | +---UnboundLocalError

| +---AttributeError

| +---SyntaxError

| | +---IndentationError

| | | +---TabError

| +---LookupError

| | +---IndexError

| | +---KeyError

| | +---CodecRegistryError

| +---ValueError

| | +---UnicodeError

| | | +---UnicodeEncodeError

| | | +---UnicodeDecodeError

| | | +---UnicodeTranslateError

| | +---UnsupportedOperation

| +---AssertionError

| +---ArithmeticError

| | +---FloatingPointError

| | +---OverflowError

| | +---ZeroDivisionError

| +---SystemError

| | +---CodecRegistryError

| +---ReferenceError

| +---BufferError

| +---MemoryError

| +---Warning

| | +---UserWarning

| | +---DeprecationWarning

| | +---PendingDeprecationWarning

| | +---SyntaxWarning

| | +---RuntimeWarning

| | +---FutureWarning

| | +---ImportWarning

| | +---UnicodeWarning

| | +---BytesWarning

| | +---ResourceWarning

| +---error

| +---Verbose

| +---Error

| +---TokenError

| +---StopTokenizing

| +---Empty

| +---Full

| +---\_OptionError

| +---TclError

| +---SubprocessError

| | +---CalledProcessError

| | +---TimeoutExpired

| +---Error

| | +---NoSectionError

| | +---DuplicateSectionError

| | +---DuplicateOptionError

| | +---NoOptionError

| | +---InterpolationError

| | | +---InterpolationMissingOptionError

| | | +---InterpolationSyntaxError

| | | +---InterpolationDepthError

| | +---ParsingError

| | | +---MissingSectionHeaderError

| +---InvalidConfigType

| +---InvalidConfigSet

| +---InvalidFgBg

| +---InvalidTheme

| +---EndOfBlock

| +---BdbQuit

| +---error

| +---\_Stop

| +---PickleError

| | +---PicklingError

| | +---UnpicklingError

| +---\_GiveupOnSendfile

| +---error

| +---LZMAError

| +---RegistryError

| +---ErrorDuringImport

+---GeneratorExit

+---SystemExit

+---KeyboardInterrupt

**output**

* [**Sandbox**](https://edube.org/sandbox)

## **Code**

def print\_exception\_tree(thisclass, nest = 0):  
if nest > 1:  
print(" |" \* (nest - 1), end="")  
if nest > 0:  
print(" +---", end="")  
  
print(thisclass.\_\_name\_\_)  
  
for subclass in thisclass.\_\_subclasses\_\_():  
print\_exception\_tree(subclass, nest + 1)  
  
  
print\_exception\_tree(BaseException)



1

2

3

4

5

6

7

8

9

10

11

12

13

14

def print\_exception\_tree(thisclass, nest = 0):

if nest > 1:

print(" |" \* (nest - 1), end="")

if nest > 0:

print(" +---", end="")

print(thisclass.\_\_name\_\_)

for subclass in thisclass.\_\_subclasses\_\_():

print\_exception\_tree(subclass, nest + 1)

print\_exception\_tree(BaseException)

* **Console**

[Prev](https://edube.org/learn/python-essentials-2/exceptions-once-again-10) [Next](https://edube.org/learn/python-essentials-2/exceptions-once-again-12)

**Detailed anatomy of exceptions**

Let's take a closer look at the exception's object, as there are some really interesting elements here (we'll return to the issue soon when we consider Python's input/output base techniques, as their exception subsystem extends these objects a bit).

The BaseException class introduces a property named args. It's a **tuple designed to gather all arguments passed to the class constructor**. It is empty if the construct has been invoked without any arguments, or contains just one element when the constructor gets one argument (we don't count the self argument here), and so on.

We've prepared a simple function to print the args property in an elegant way. You can see the function in the editor.

We've used the function to print the contents of the args property in three different cases, where the exception of the Exception class is raised in three different ways. To make it more spectacular, we've also printed the object itself, along with the result of the \_\_str\_\_() invocation.

The first case looks routine - there is just the name Exception after the raise keyword. This means that the object of this class has been created in a most routine way.

The second and third cases may look a bit weird at first glance, but there's nothing odd here - these are just the constructor invocations. In the second raise statement, the constructor is invoked with one argument, and in the third, with two.

As you can see, the program output reflects this, showing the appropriate contents of the args property:

: :

my exception : my exception : my exception

('my', 'exception') : ('my', 'exception') : ('my', 'exception')

**output**

**How to create your own exception**

The exceptions hierarchy is neither closed nor finished, and you can always extend it if you want or need to create your own world populated with your own exceptions.

It may be useful when you create a complex module which detects errors and raises exceptions, and you want the exceptions to be easily distinguishable from any others brought by Python.

This is done by **defining your own, new exceptions as subclasses derived from predefined ones**.

Note: if you want to create an exception which will be utilized as a specialized case of any built-in exception, derive it from just this one. If you want to build your own hierarchy, and don't want it to be closely connected to Python's exception tree, derive it from any of the top exception classes, like Exception.

Imagine that you've created a brand new arithmetic, ruled by your own laws and theorems. It's clear that division has been redefined, too, and has to behave in a different way than routine dividing. It's also clear that this new division should raise its own exception, different from the built-in ZeroDivisionError, but it's reasonable to assume that in some circumstances, you (or your arithmetic's user) may want to treat all zero divisions in the same way.

Demands like these may be fulfilled in the way presented in the editor. Look at the code, and let's analyze it:

* We've defined our own exception, named MyZeroDivisionError, derived from the built-in ZeroDivisionError. As you can see, we've decided not to add any new components to the class.  
    
  In effect, an exception of this class can be - depending on the desired point of view - treated like a plain ZeroDivisionError, or considered separately.

* The do\_the\_division() function raises either a MyZeroDivisionError or ZeroDivisionError exception, depending on the argument's value.  
    
  The function is invoked four times in total, while the first two invocations are handled using only one except branch (the more general one) and the last two ones with two different branches, able to distinguish the exceptions (don't forget: the order of the branches makes a fundamental difference!)

class MyZeroDivisionError(ZeroDivisionError):

pass

def do\_the\_division(mine):

if mine:

raise MyZeroDivisionError("some worse news")

else:

raise ZeroDivisionError("some bad news")

for mode in [False, True]:

try:

do\_the\_division(mode)

except ZeroDivisionError:

print('Division by zero')

for mode in [False, True]:

try:

do\_the\_division(mode)

except MyZeroDivisionError:

print('My division by zero')

except ZeroDivisionError:

print('Original division by zero')

**How to create your own exception: continued**

Look at the code in the editor. We've coupled together the two previously defined exceptions and harnessed them to work in a small example snippet.

One of these is raised inside the make\_pizza() function when any of these two erroneous situations is discovered: a wrong pizza request, or a request for too much cheese.

Note:

* removing the branch starting with except TooMuchCheeseError will cause all appearing exceptions to be classified as PizzaError;
* removing the branch starting with except PizzaErrorwill cause the TooMuchCheeseError exceptions to remain unhandled, and will cause the program to terminate.

The previous solution, although elegant and efficient, has one important weakness. Due to the somewhat easygoing way of declaring the constructors, the new exceptions cannot be used as-is, without a full list of required arguments.

We'll remove this weakness by **setting the default values for all constructor parameters**. Take a look:

class PizzaError(Exception):

def \_\_init\_\_(self, pizza='uknown', message=''):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza='uknown', cheese='>100', message=''):

PizzaError.\_\_init\_\_(self, pizza, message)

self.cheese = cheese

def make\_pizza(pizza, cheese):

if pizza not in ['margherita', 'capricciosa', 'calzone']:

raise PizzaError

if cheese > 100:

raise TooMuchCheeseError

print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:

try:

make\_pizza(pz, ch)

except TooMuchCheeseError as tmce:

print(tmce, ':', tmce.cheese)

except PizzaError as pe:

print(pe, ':', pe.pizza)

Now, if the circumstances permit, it is possible to use the class names alone.

**Key takeaways**

1. The else: branch of the try statement is executed when there has been no exception during the execution of the try: block.

2. The finally: branch of the try statement is **always** executed.

3. The syntax except *Exception\_Name* as an *exception\_object*: lets you intercept an object carrying information about a pending exception. The object's property named args (a tuple) stores all arguments passed to the object's constructor.

4. The exception classes can be extended to enrich them with new capabilities, or to adopt their traits to newly defined exceptions.

For example:

try:

assert \_\_name\_\_ == "\_\_main\_\_"

except:

print("fail", end=' ')

else:

print("success", end=' ')

finally:

print("done")

The code outputs: success done.

**Exercise 1**

What is the expected output of the following code?

import math

try:

print(math.sqrt(9))

except ValueError:

print("inf")

else:

print("fine")

Check

3.0

fine

**Exercise 2**

What is the expected output of the following code?

import math

try:

print(math.sqrt(-9))

except ValueError:

print("inf")

else:

print("fine")

finally:

print("the end")

Check

inf

the end

**Exercise 3**

What is the expected output of the following code?

import math

class NewValueError(ValueError):

def \_\_init\_\_(self, name, color, state):

self.data = (name, color, state)

try:

raise NewValueError("Enemy warning", "Red alert", "High readiness")

except NewValueError as nve:

for arg in nve.args:

print(arg, end='! ')

Check

Enemy warning! Red alert! High readiness!