SYLLABUS:

Course syllabus

Module 1 - Introduction to Python and Computer Programming

- Python - a tool, not a reptile
- There is more than one Python
- Let's start our Python adventure

Module 2 - Data Types, Variables, Basic Input-Output Operations, Basic Operators

- Your first program
- Python literals
- Operators - data manipulation tools
- Variables - data-shaped boxes
- How to talk to computer?

Module 3 - Boolean Values, Conditional Execution, Loops, Lists and List Processing, Logical and Bitwise Operations

- Making decisions in Python
- Python's loops
- Logic and bit operations in Python
- Lists - collections of data
- Sorting simple lists - the bubble sort algorithm
- Lists - some more details
- Lists in advanced applications

Module 5 - Modules, Packages, String and List Methods, and Exceptions

- Using modules
- Some useful modules
- What is package?
- Errors - the programmer's daily bread
- The anatomy of exception
- Some of the most useful exceptions
- Characters and strings vs. computers
- Python's nature of strings
- String methods
- Strings in action
- Four simple programs

Module 6 - The Object-Oriented Approach: Classes, Methods, Objects, and the Standard Objective Features; Exception Handling, and Working with Files

- Basic concepts of object programming
- A short journey from procedural to object approach
- Properties
- Methods
- Inheritance - one of object programming foundations
- Exceptions once again
- Generators and closures
- Processing files
- Working with real files

Welcome to Programming Essentials in Python - Part 1

Module 1
Introduction to Python and computer programming

Module 2
Data types, variables, basic input-output operations, basic operators

Module 3
Boolean values, conditional execution, loops, lists and list processing, logical and bitwise operations

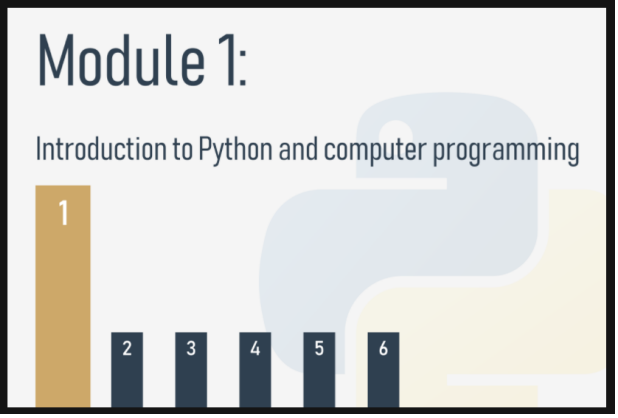
Module 4
Functions, tuples, dictionaries, and data processing

MODULE: 1

Programming Essentials in Python: Module 1

In this module, you will learn about:

- the fundamentals of computer programming;
- setting up your programming environment;
- compilation vs. interpretation;
- introduction to Python.



How does a computer program work?

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

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

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

It can execute only extremely simple operations, e.g., a computer cannot evaluate the value of a complicated mathematical function by itself, although this isn't beyond the realms of possibility in the near future.

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

Imagine that you want to know the average speed you've reached during a long journey. You know the distance, you know the time, you need the speed.

Naturally, the computer will be able to compute this, but the computer is not aware of such things as distance, speed or time. Therefore, it is necessary to instruct the computer to:

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

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

Language is the keyword.

Natural languages vs. programming languages

A language is a means (and a tool) for expressing and recording thoughts. There are many languages all around us. Some of them require neither speaking nor writing, such as body language; it's possible to express your deepest feelings very precisely without saying a word.

Another language you use each day is your mother tongue, which you use to manifest your will and to think about 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. On the other hand, people use a number of very different languages, too, but these languages developed naturally. Moreover, they are still evolving.

New words are created every day and 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.

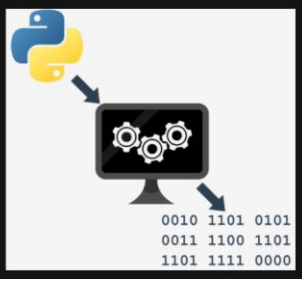
For example, if you try to use an entity of an unknown name, it will cause an error, but the error will be discovered in the place where it tries to use the entity, not where the new entity's name was introduced.

In other words, the actual reason is usually located a little earlier in the code, e.g., 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.



Compilation vs. interpretation - advantages and disadvantages

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

What does this all mean for you?

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

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

What is Python?

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

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

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

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

Who created Python?

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

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

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

A hobby programming project

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

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

Python goals

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

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

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

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

Python isn't a young language. 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.

Why not Python?

Despite Python's growing popularity, there are still some niches where Python is absent, or is rarely seen:

- **low-level programming** (sometimes called "close to metal" programming): if you want to implement an extremely effective driver or graphical engine, you wouldn't use Python;
- **applications for mobile devices**: although this territory is still waiting to be conquered by Python, it will most likely happen someday.

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.

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

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, and Python 3.7.

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), 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.**

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.

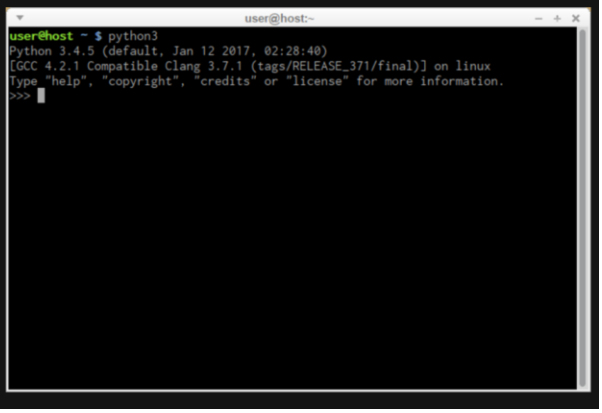
If you see something like this:

```
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/.



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.

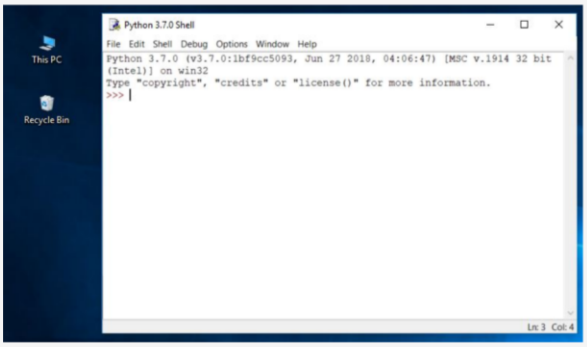
To start your work, you need the following tools:

- 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.

Navigate through your OS menus, find IDLE somewhere under Python 3.x and launch it. This is what you should see:



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.

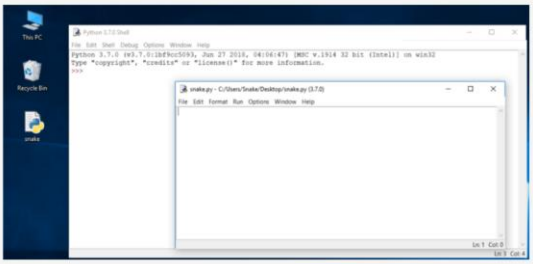
The first step is to create a new source file and fill it with code. Click *File* in the IDLE's menu and choose *New file*.

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.

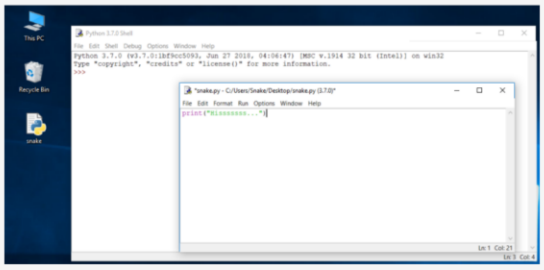
The line looks like this:

```
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.

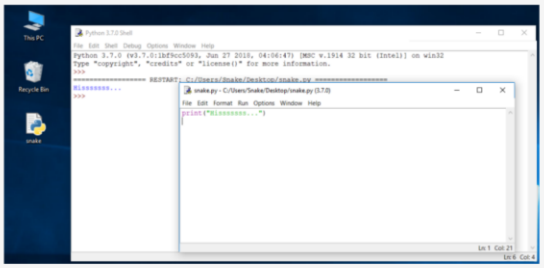


If everything goes okay and there are no mistakes in the code, the console window will show you the effects caused by running the program.

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.



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.

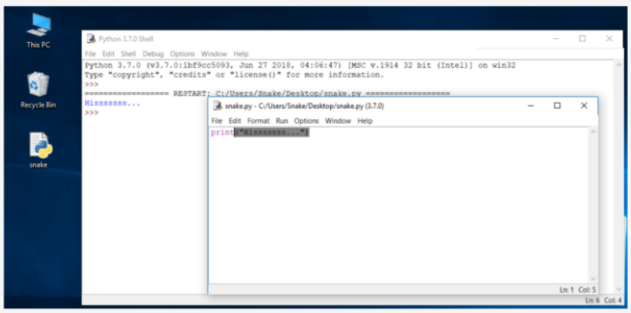
As you can see, IDLE is able to save your code and retrieve it when you need it again.

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:



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.

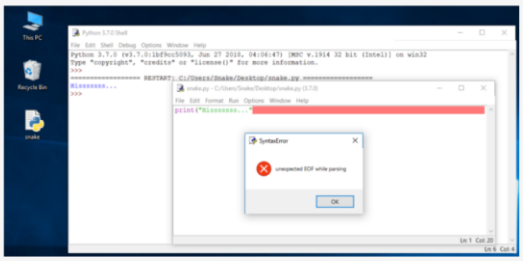
Try to run the program again. IDLE will remind you to save the modified file. Follow the instructions.

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.

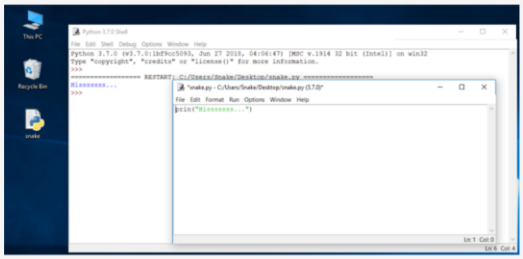


Fix the code now. It should look like this:

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

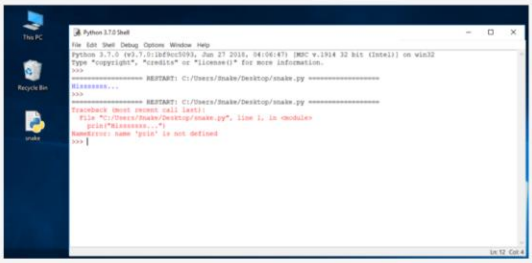
Run it to see if it "hisses" again.

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.

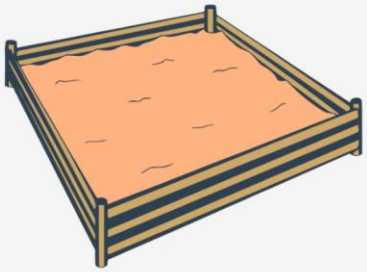
Experiment with creating new files and running your code. Try to output a different message to the screen, e.g., `roar!`, `meow`, or even maybe an `oink!`. Try to spoil and fix your code - see what happens.

Sandbox

This course does not require you to install any software applications to test your code and do the exercises.

To test or experiment with your code, you can use a dedicated, interactive on-line programming environment.

Sandbox allows Python code to be run in an Internet browser.



It is a tool integrated within the course, which can be used as **a browser-based Python sandbox** that allows you to test code discussed throughout the course, as well as **an interpreter that enables you to launch, perform and test the lab exercises** specifically designed for this course.

The Sandbox interface consists of three main parts:

- the **Editor** window, which lets you type in your code,
- the **Console** window, which lets you see the output of your programs,
- a tool named the **Action Buttons** bar, which lets you run your code, refresh the editor window, download your program as a .py file, upload a .py file that will be displayed in the editor, report a bug (in case you spot anything wrong, do let us know about it!)
- and the **Settings** button, which lets you adjust display settings and switch between Python/C/C++ environments.

Now copy the following code:

```
print("Hello!")
print("Welcome to Python Essentials!")
print("THIS IS SANDBOX MODE.")
```

...then click the **Sandbox** button to enter Sandbox Mode, paste the code in the editor window, and click the Run button to see what happens.

To get back to our course, click **Back to course** in the top right-hand corner of the Sandbox interface.

Practice Interface

This course contains four different types of interfaces.

So far you've seen the **Study Interface** (one or two windows with text and images/animation), and **Sandbox Interface**, which you can use to try out your own code (click *Sandbox* to switch to the Sandbox interface).

The one you see now is the **Practice Interface**, which enables you to study new things and do coding tasks at the same time. You will use this kind of interface for most of the time during the course.

The Practice Interface consists of a text area on the left and the Editor/Console windows on the right.

One more type of interface that you will see in the future is the Quiz/Test Interface, which will let you check your knowledge and skills to see how well you have mastered the study material.

Sandbox

```
1 # Click the Run button
2 print("WELCOME TO PYTHON ESSENTIALS!")
3 print("\"welcome to python essentials\"")
4 print("one thousand program is completed in python")
5
```

Console >_

```
WELCOME TO PYTHON ESSENTIALS!
WELCOME TO PYTHON ESSENTIALS!
"welcome to python essentials"
WELCOME TO PYTHON ESSENTIALS!
"welcome to python essentials"
one thousand program is completed in python
```

Prev Next

Congratulations! You have completed Module 1.

Well done! You've reached the end of Module 1 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 1:

- the fundamentals of computer programming, i.e., how the computer works, how the program is executed, how the programming language is defined and constructed,
- the difference between compilation and interpretation;
- the basic information about Python and how it is positioned among other programming languages, and what distinguishes its different versions
- the study resources and different types of interfaces you will be using in the course.

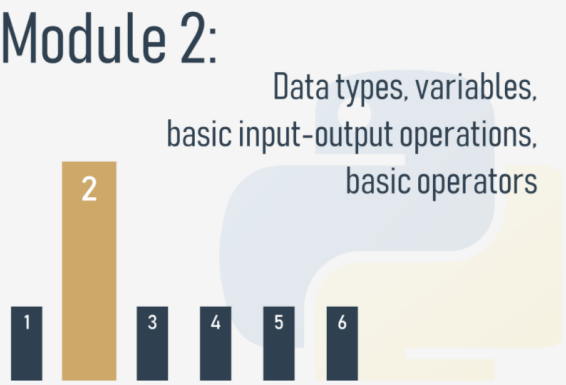
You are now ready to take the module quiz, which will help you gauge what you've learned so far.

MODULE: 2

Programming Essentials in Python: Module 2

In this module, you will learn about:

- data types and the basic methods of formatting, converting, inputting and outputting data;
- operators;
- variables.



Hello, World!

It's time to start writing some **real, working Python code**. It'll be very simple for the time being.

As we're going to show you some fundamental concepts and terms, these snippets of code won't be serious or complex.

Run the code in the editor window on the right. If everything goes okay here, you'll see the **line of text** in the console window.

Alternatively, launch IDLE, create a new Python source file, fill it with this code, name the file and save it. Now run it. If everything goes okay, you'll see the rhyme's line in the IDLE console window. The code you have run should look familiar. You saw something very similar when we led you through the setting up of the IDLE environment.

Now we'll spend some time showing and explaining to you what you're actually seeing, and why it looks like this.

As you can see, the first program consists of the following parts:

- the word `print`;
- an opening parenthesis;
- a quotation mark;
- a line of text: `Hello, World!`;
- another quotation mark;
- a closing parenthesis.

Each of the above plays a very important role in the code.

```
1 print("Hello, World!")
```

```
Console >_
Hello, World!
```

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's 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 or some values** (e.g., the square root of a value or the length of a given text); this is what makes Python's functions the relatives of mathematical concepts.

Moreover, many of Python's functions can do the above two things together.

Where do the functions come from?

- They may come **from Python itself**; the print function is one of this kind; such a function is an added value received together with Python and its environment (it is **built-in**); you don't have to do anything special (e.g., ask anyone for anything) if you want to make use of it;
- they may come from one or more of Python's add-ons named **modules**; some of the modules come with Python, others may require separate installation - whatever the case, they all need to be explicitly connected with your code (we'll show you how to do that soon);
- you can **write them yourself**, placing as many functions as you want and need inside your program to make it simpler, clearer and more elegant.

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

Of course, if you're going to make use of any already existing function, you have no influence on its name, but when you start writing your own functions, you should consider carefully your choice of names.

The `print()` function

As we said before, a function may have:

- an **effect**;
- a **result**.

There's also a third, very important, function component - the **argument**(s).

Mathematical functions usually take one argument, e.g., sin(x) takes an x, which is the measure of an angle.

Python functions, on the other hand, are more versatile. Depending on the individual needs, they may accept any number of arguments - as many as necessary to perform their tasks. Note: any number includes zero - some Python functions don't need any argument.

```
print("Hello, World!")
```

In spite of the number of needed/provided arguments, Python functions strongly demand the presence of **a pair of parentheses** - opening and closing ones, respectively.

If you want to deliver one or more arguments to a function, you place them **inside the parentheses**. If you're going to use a function which doesn't take any argument, you still have to have the parentheses.

Note: to distinguish ordinary words from function names, place **a pair of empty parentheses** after their names, even if the corresponding function wants one or more arguments. This is a standard convention.

The function we're talking about here is `print()`.

Does the `print()` function in our example have any arguments?

Of course it does, but what are they?

The `print()` function

The only argument delivered to the `print()` function in this example is a **string**:

```
print("Hello, World!")
```

As you can see, the **string is delimited with quotes** - in fact, the quotes make the string - they cut out a part of the code and assign a different meaning to it.

You can imagine that the quotes say something like: the text between us is not code. It isn't intended to be executed, and you should take it as is.

Almost anything you put inside the quotes will be taken literally, not as code, but as **data**. Try to play with this particular string - modify it, enter some new content, delete some of the existing content.

There's more than one way to specify a string inside Python's code, but for now, though, this one is enough.

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

The `print()` function

The function name (***print*** in this case) along with the parentheses and argument(s), forms the **function invocation**.

We'll discuss this in more depth soon, but we should just shed a little light on it right now.

```
print("Hello, World!")
```

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

```
function_name(argument)
```

Let's see:

- First, Python checks if the name specified is **legal** (it browses its internal data in order to find an existing function of the name; if this search fails, Python aborts the code);
- second, Python checks if the function's requirements for the number of arguments **allows you to invoke** the function in this way (e.g., if a specific function demands exactly two arguments, any invocation delivering only one argument will be considered erroneous, and will abort the code's execution);
- third, Python **leaves your code for a moment** and jumps into the function you want to invoke; of course, it takes your argument(s) too and passes it/them to the function;
- fourth, the function **executes its code**, causes the desired effect (if any), evaluates the desired result(s) (if any) and finishes its task;
- finally, Python **returns to your code** (to the place just after the invocation) and resumes its execution.

The `print()` function

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.

LAB

Estimated time

5 minutes

Level of difficulty

Very easy

Objectives

- becoming familiar with the `print()` function and its formatting capabilities;
- experimenting with Python code.

Scenario

The `print()` command, which is one of the easiest directives in Python, simply prints out a line to the screen.

In your first lab:

- use the `print()` function to print the line `Hello, Python!` to the screen. Use double quotes around the string;
- having done that, use the `print()` function again, but this time print your first name;
- remove the double quotes and run your code. Watch Python's reaction. What kind of error is thrown?
- then, remove the parentheses, put back the double quotes, and run your code again. What kind of error is thrown this time?
- experiment as much as you can. Change double quotes to single quotes, use multiple `print()` functions on the same line, and then on different lines. See what happens.

```
1 print("\"Hello, Python!\"")
2 print('golu')
3 print("golu")
4 print('''golu''')
```

Console >...

```
"Hello, Python!"
"Hello, Python!"
"golu"
  File "main.py", line 2
    print(\"golu\")
                 ^
SyntaxError: unexpected character after line continuation character
  File "main.py", line 2
    print "golu"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("golu")?
"Hello, Python!"
golu
golu
golu
```

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 evaluate?

None. Its effect is enough - `print()` does not evaluate anything.

The `print()` function - instructions

You already know that this program contains one function invocation. In turn, the function invocation is one of the possible kinds of Python **instruction**. Ergo, this program consists of just one instruction.

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)

```
1 print("The itsy bitsy spider climbed up the waterspout.")
2 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.
  File "main.py", line 2
    print("Down came the rain and washed the spider out.")
                                                         ^
SyntaxError: invalid syntax
  File "main.py", line 3

                                                         ^
SyntaxError: unexpected EOF while parsing
  File "main.py", line 3

                                                         ^
SyntaxError: unexpected EOF while parsing
```

The `print()` function - instructions

We've changed the example a bit - we've added one **empty** `print()` function invocation. We call it empty because we haven't delivered any arguments to the function.

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

What happens?

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

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

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

output

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

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

```
1 print("The itsy bitsy spider climbed up the waterspout.")
2 print() # encounter a new line
3 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

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

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

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

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

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

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

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

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

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

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

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

output

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

```
1 print("The itsy bitsy spider\nclimbed up the waterspout.")
2 print()
3 print("Down came the rain\nand 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.

```
1 print("The itsy bitsy spider\nclimbed up the waterspout.")
2 print()
3 print("Down came the rain\nand washed the spider out.")
4 print
5 print("the escape nd\\newline characters")
```

Console >_

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

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

Down came the rain
and washed the spider out.
the escape nd,
\newline characters
The itsy bitsy spider
climbed up the waterspout.

Down came the rain
and washed the spider out.
the escape nd\newline characters
```

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.

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

Console >_

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

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.

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

Console >_

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

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"`.

```
print("My name is", "Python.", end=" ")
print("Monty Python.")
print("the keyword arguments", end="\n")
print("Monty Python")
```

Console >_

```
My name is Python. Monty Python.
My name is Python. Monty Python.
the keyword arguments
Monty Python
```

The `print()` function - the keyword arguments

And now it's time to try something more difficult.

If you look carefully, you'll see that we've used the `end` argument, but the string assigned to it is empty (it contains no characters at all).

What will happen now? Run the program in the editor to find out.

As the `end` argument has been set to nothing, the `print()` function outputs nothing too, once its positional arguments have been exhausted.

The console should now be showing the following text:

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

output

Note: **no newlines have been sent to the output.**

The string assigned to the `end` keyword argument can be of any length. Experiment with it if you want.

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

Console >_

```
My name is 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="-")
print("hi", "i", "am", "a", "programmer", sep="+")
```

Console >_

```
My-name-is-Monty-Python.
My-name-is-Monty-Python.
hi+i+am+programmer
```

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")
```

Console >_

```
My_name_is*Monty*Python.*
```

LAB

Estimated time

5 minutes

Level of difficulty

Very Easy

Objectives

- becoming familiar with the `print()` function and its formatting capabilities;
- experimenting with Python code.

Scenario

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

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

Expected output

```
Programming***Essentials***in...Python
```

output

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

Console >_

```
Programming***Essentials***in...
Python
Programming***Essentials***in...Python
```

LAB

Estimated time

5-10 minutes

Level of difficulty

Easy

Objectives

- experimenting with existing Python code;
- discovering and fixing basic syntax errors;
- becoming familiar with the `print()` function and its formatting capabilities.

Scenario

We strongly encourage you to **play with the code** we've written for you, and make some (maybe even destructive) amendments. Feel free to modify any part of the code, but there is one condition - learn from your mistakes and draw your own conclusions.

Try to:

- minimize the number of `print()` function invocations by inserting the `\n` sequence into the strings
- make the arrow twice as large (but keep the proportions)
- duplicate the arrow, placing both arrows side by side; note: a string may be multiplied by using the following trick: `"string" * 2` will produce `"stringstring"` (we'll tell you more about it soon)
- remove any of the quotes, and look carefully at Python's response; pay attention to where Python sees an error - is this the place where the error really exists?
- do the same with some of the parentheses;
- change any of the `print` words into something else, differing only in case (e.g., `Print`) - what happens now?
- replace some of the quotes with apostrophes; watch what happens carefully.

```
print("    *"*2)
print("   * *"*2)
print("  *   *"*2)
print(" *     *"*2)
print('***   ***'*2)
print("  #   *"*2)
print("  *   *"*2)
print('  *****'*2)
```

Console >_

```
  *     *
  *     *


***   ***
***   ***


  #   *
  #   *


  *   *
  *   *


  *****
  *****


    *     *
   * *   * *
  *   * *   *
 *     * *     *
***   *****   ***
  #   *  #    *
  *   *  *    *
  *****  *****
```

Key takeaways

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

2. Built-in functions, contrary to user-defined functions, are always available and don't have to be imported. Python 3.7.1 comes with 69 built-in functions. You can find their full list provided in alphabetical order in the Python Standard Library.

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

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

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

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

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

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

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

Literals - the data in itself

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

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

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

Take a look at the following set of digits:

```
123
```

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

But what about this:

```
c
```

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

You cannot choose the right one without some additional knowledge.

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

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

Literals - the data in itself

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

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

Try to run it.

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

What happened? What does it mean?

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

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

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

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

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

```
1 print(type("2"))
2 print(type(2))
```

Console >_

```
2
2
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    print("2".type())
AttributeError: 'str' object has no attribute 'type'
2
<class 'int'>
<class 'str'>
<class 'int'>
```

Integers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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)
```

```
1 print(0o123)
2 print(0x123)
```

Console >_

```
83
83
291
```

Floats

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

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

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

```
2.5
-0.4
```

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

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

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

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

```
0.4
```

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

In essence, you can write the value `0.4` as:

```
.4
```

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

```
4.
```

This will change neither its type nor its value.

Ints vs. floats

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

Look at these two numbers:

```
4
4.0
```

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

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

The point is what makes a float.

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

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

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

To avoid writing out so many zeros, physics textbooks use an abbreviated form, which you have probably already seen: 3×10^8.

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

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

```
3E8
```

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

Note:

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

Coding floats

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

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

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

```
6.62607E-34
```

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

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

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

```
0.00000000000000000001
```

When you run this literal through Python:

```
print(0.00000000000000000001)
```

this is the result:

```
1e-22
```

output

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

Strings

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

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

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

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

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

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

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

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

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

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

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

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

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

This example will work too:

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

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

Coding strings

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

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

Try to print out a string containing the following message:

```
I'm Monty Python.
```

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

Check

```
print('I\'m Monty Python.')
```

or

```
print("I'm Monty Python.")
```

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

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

An empty string still remains a string:

```
''
""
```

```
print("I", "m Monty Python", sep="'")
print("I\'m Monty Python")
print("I'm Monty Python")
print('''I'm Monty Pyhton''')
print('I'm Monty Pyhton')  # it will throw error
```

Console >_

```
I'm Monty Python
I'm Monty Python
I'm Monty Python
I'm Monty Python
I'm Monty Python
I'm Monty Python
  File "main.py", line 5
    print('I'm Monty Pyhton')
              ^
SyntaxError: invalid syntax
```

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?

LAB

Estimated time

5 minutes

Level of difficulty

Easy

Objectives

- becoming familiar with the `print()` function and its formatting capabilities;
- practicing coding strings;
- experimenting with Python code.

Scenario

Write a one-line piece of code, using the `print()` function, as well as the newline and escape characters, to match the expected result outputted on three lines.

Expected output

```
"I'm"
""learning""
"""Python"""
```

output

```
print("I'm")
print('""learning""')
print("\"\"\"Python\"\"\"")
```

Console >_

```
I'm
""learning""
"Python"
  File "main.py", line 3
    print("\"\"\"Python\"\"\"")
                    ^
SyntaxError: invalid syntax
I'm
""learning""
"""Python"""
```

Key takeaways

1. **Literals** are notations for representing some fixed values in code. Python has various types of literals - for example, a literal can be a number (numeric literals, e.g., `123`), or a string (string literals, e.g., "I am a literal.").

2. The **binary system** is a system of numbers that employs *2* as the base. Therefore, a binary number is made up of 0s and 1s only. e.g., `1010` is *10* in decimal.

Octal and hexadecimal numeration systems, similarly, employ *8* and *16* as their bases respectively. The hexadecimal system uses the decimal numbers and six extra letters.

3. **Integers** (or simply **int**s) are one of the numerical types supported by Python. They are numbers written without a fractional component, e.g., `256`, or `-1` (negative integers).

4. **Floating-point** numbers (or simply **float**s) are another one of the numerical types supported by Python. They are numbers that contain (or are able to contain) a fractional component, e.g., `1.27`.

5. To encode an apostrophe or a quote inside a string you can either use the escape character, e.g., `'I\'m happy.'`, or open and close the string using an opposite set of symbols to the ones you wish to encode, e.g., `"I'm happy."` to encode an apostrophe, and `'He said "Python", not "typhoon"'` to encode a (double) quote.

6. **Boolean values** are the two constant objects `True` and `False` used to represent truth values (in numeric contexts `1` is `True`, while `0` is `False`).

EXTRA

There is one more, special literal that is used in Python: the `None` literal. This literal is a so-called `NoneType` object, and it is used to represent **the absence of a value**. We'll tell you more about it soon.

Exercise 1

What types of literals are the following two examples?

```
"Hello ", "007"
```

Check

They're both strings/string literals.

Exercise 2

What types of literals are the following four examples?

```
"1.5", 2.0, 528, False
```

Check

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

Exercise 3

What is the decimal value of the following binary number?

```
1011
```

Check

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

Python as a calculator

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

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

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

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

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

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

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

Basic operators

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

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

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

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

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

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

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

```
# print(2 + 2)
# num1 = 23
# num2 = 33
# sum = num1 + num2
# print(sum)
# print(num1 + num2)
a = input("Enter 1st number: ")
b = input("Enter 2nd number: ")
print(a + b) # note the result, and think!!!!
```

Console >_

```
4
4
56
4
56
56
4
56
56
23
Traceback (most recent call last):
  File "main.py", line 9, in <module>
    print(a + b)
NameError: name 'a' is not defined
3 5

3 5
4
Enter 1st number: 4
Enter 2nd number: 6
46
```

Arithmetic operators: exponentiation

A `**` (double asterisk) sign is an **exponentiation** (power) operator. Its left argument is the **base**, its right, the **exponent**.

Classical mathematics prefers notation with superscripts, just like this: 2^3. Pure text editors don't accept that, so Python uses `**` instead, e.g., `2 ** 3`.

Take a look at our examples in the editor window.

Note: we've surrounded the double asterisks with spaces in our examples. It's not compulsory, but it improves the **readability** of the code.

The examples show a very important feature of virtually all Python **numerical operators**.

Run the code and look carefully at the results it produces. Can you see any regularity here?

Remember: It's possible to formulate the following rules based on this result:

- when **both** `**` arguments are integers, the result is an integer, too;
- when **at least one** `**` argument is a float, the result is a float, too.

This is an important distinction to remember.

```
print(2 ** 3)
print(2 ** 3.)
print(2. ** 3)
print(2. ** 3.)
```

Console >_

```
8
8.0
8.0
8.0
```

Arithmetic operators: multiplication

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

Run the code below and check if our integer vs. float rule is still working.

```
print(2 * 3)
print(2 * 3.)
print(2. * 3)
print(2. * 3.)
```

Arithmetic operators: division

A `/` (slash) sign is a **divisional** operator.

The value in front of the slash is a **dividend**, the value behind the slash, a **divisor**.

Run the code below and analyze the results.

```
print(6 / 3)
print(6 / 3.)
print(6. / 3)
print(6. / 3.)
```

You should see that there is an exception to the rule.

The result produced by the division operator is always a float, regardless of whether or not the result seems to be a float at first glance: `1 / 2`, or if it looks like a pure integer: `2 / 1`.

Is this a problem? Yes, it is. It happens sometimes that you really need a division that provides an integer value, not a float.

Fortunately, Python can help you with that.

```
print(1//2)
print(2//1)
```

Console ?...

```
0.5
2.0
0
2
```

Arithmetic operators: integer division

A `//` (double slash) sign is an **integer divisional** operator. It differs from the standard `/` operator in two details:

- its result lacks the fractional part - it's absent (for integers), or is always equal to zero (for floats); this means that **the results are always rounded**;
- it conforms to the *integer vs. float rule*.

Run the example below and see the results:

```
print(6 // 3)
print(6 // 3.)
print(6. // 3)
print(6. // 3.)
```

As you can see, *integer by integer division* gives **an integer result**. All other cases produce floats.

Let's do some more advanced tests.

Look at the following snippet:

```
print(6 // 4)
print(6. // 4)
```

Imagine that we used `/` instead of `//` - could you predict the results?

Yes, it would be `1.5` in both cases. That's clear.

But what results should we expect with `//` division?

Run the code and see for yourself.

What we get is two ones - one integer and one float.

The result of integer division is always rounded to the nearest integer value that is less than the real (not rounded) result.

This is very important: **rounding always goes to the lesser integer**.

Look at the code below and try to predict the results once again:

```
print(-6 // 4)
print(6. // -4)
```

Note: some of the values are negative. This will obviously affect the result. But how?

The result is two negative twos. The real (not rounded) result is `-1.5` in both cases. However, the results are the subjects of rounding. The **rounding goes toward the lesser integer value**, and the lesser integer value is `-2`, hence: `-2` and `-2.0`.

NOTE

Integer division can also be called **floor division**. You will definitely come across this term in the future.

```
print(6 // 3)
print(6 // 3.)
print(6. // 3)
print(6. // 3.)

print(6 // 4)
print(6. // 4)

print(-6 // 4)
print(6. // -4)
```

Console ?...

```
2.0
-2
-2.0
1
1.0
1
2.0
2.0
2.0
1
1.0
-2
-2.0
```

Operators: remainder (modulo)

The next operator is quite a peculiar one, because it has no equivalent among traditional arithmetic operators.

Its graphical representation in Python is the `%` (percent) sign, which may look a bit confusing.

Try to think of it as of a slash (division operator) accompanied by two funny little circles.

The result of the operator is a **remainder left after the integer division.**

In other words, it's the value left over after dividing one value by another to produce an integer quotient.

Note: the operator is sometimes called **modulo** in other programming languages.

Take a look at the snippet - try to predict its result and then run it:

```
print(14 % 4)
```

As you can see, the result is two. This is why:

- `14 // 4` gives `3` — this is the integer **quotient**;
- `3 * 4` gives `12` — as a result of **quotient and divisor multiplication**;
- `14 - 12` gives `2` — this is the **remainder**.

This example is somewhat more complicated:

```
print(12 % 4.5)
```

What is the result?

Check

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

Operators: how not to divide

As you probably know, **division by zero doesn't work**.

Do not try to:

- perform a division by zero;
- perform an integer division by zero;
- find a remainder of a division by zero.

```
# print(34/0)
print(1214.5)
```

Console ?...

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    print(34/0)
ZeroDivisionError: division by zero
3.0
```

Operators: addition

The **addition** operator is the `+` (plus) sign, which is fully in line with mathematical standards.

Again, take a look at the snippet of the program below:

```
print(-4 + 4)
print(-4. + 8)
```

The result should be nothing surprising. Run the code to check it.

The subtraction operator, unary and binary operators

The **subtraction** operator is obviously the `-` (minus) sign, although you should note that this operator also has another meaning - **it can change the sign of a number**.

This is a great opportunity to present a very important distinction between **unary** and **binary** operators.

In subtracting applications, the **minus operator expects two arguments**: the left (a **minuend** in arithmetical terms) and right (a **subtrahend**).

For this reason, the subtraction operator is considered to be one of the binary operators, just like the addition, multiplication and division operators.

But the minus operator may be used in a different (unary) way - take a look at the last line of the snippet below:

```
print(-4 - 4)
print(4. - 8)
print(-1.1)
```

By the way: there is also a unary `+` operator. You can use it like this:

```
print(+2)
```

The operator preserves the sign of its only argument - the right one.

Although such a construction is syntactically correct, using it doesn't make much sense, and it would be hard to find a good rationale for doing so.

```
1 print(-4 + 4)
2 print(-4. + 8)
3 print(-4 - 4)
4 print(4. - 8)
5 print(-1.1)
```

Console >_

```
0
0
4.0
0
4.0
-8
-4.0
-1.1
```

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**.

```
1 print(2 ** 2 ** 3)
```

Console >_

```
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?

Check

Key takeaways

1. An **expression** is a combination of values (or variables, operators, calls to functions - you will learn about them soon) which evaluates to a value, e.g., `1 + 2`.

2. **Operators** are special symbols or keywords which are able to operate on the values and perform (mathematical) operations, e.g., the `*` operator multiplies two values: `x * y`.

3. Arithmetic operators in Python: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (classic division - returns a float if one of the values is of float type), `%` (modulus - divides left operand by right operand and returns the remainder of the operation, e.g., `5 % 2 = 1`), `**` (exponentiation - left operand raised to the power of right operand, e.g., `2 ** 3 = 2 * 2 * 2 = 8`), `//` (floor/integer division - returns a number resulting from division, but rounded down to the nearest whole number, e.g., `3 // 2.0 = 1.0`)

4. A **unary** operator is an operator with only one operand, e.g., `-1`, or `+3`.

5. A **binary** operator is an operator with two operands, e.g., `4 + 5`, or `12 % 5`.

6. Some operators act before others - **the hierarchy of priorities**:

- unary `+` and `-` have the highest priority
- then: `**`, then: `*`, `/`, and `%`, and then the lowest priority: binary `+` and `-`.

7. Subexpressions in **parentheses** are always calculated first, e.g., `15 - 1 * (5 * (1 + 2)) = 0`.

8. The **exponentiation** operator uses **right-sided binding**, e.g., `2 ** 2 ** 3 = 256`.

Exercise 1

What is the output of the following snippet?

```
print((2 ** 4), (2 * 4.), (2 * 4))
```

Check

Exercise 2

What is the output of the following snippet?

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

Check

Exercise 3

What is the output of the following snippet?

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

Check

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).

Correct 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 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', 'el
```

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.

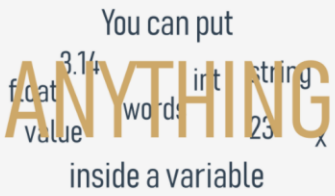
Let's talk now about two important things - **how variables are created**, and **how to put values inside them** (or rather - how to give or **pass values** to them).

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

```
1
```

Using variables

You're allowed to use as many variable declarations as you need to achieve your goal, like this:

```
var = 1
account_balance = 1000.0
client_name = 'John Doe'
print(var, account_balance, client_name)
print(var)
```

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

This example will **cause an error**:

```
var = 1
print(Var)
```

We've tried to use a variable named `Var`, which doesn't have any value (note: `var` and `Var` are different entities, and have nothing in common as far as Python's concerned).

REMEMBER

You can use the `print()` statement and combine text and variables using the `+` operator to output strings and variables, e.g.:

```
var = "3.7.1"
print("Python version: " + var)
```

Can you guess the output of the snippet above?

Check

```
Python version: 3.7.1
```

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.

Solving simple mathematical problems

Now you should be able to construct a short program solving simple mathematical problems such as the Pythagorean theorem:

The square of the hypotenuse is equal to the sum of the squares of the other two sides.

The following code evaluates the length of the hypotenuse (i.e., the longest side of a right-angled triangle, the one opposite of the right angle) using the Pythagorean theorem:

```
a = 3.0
b = 4.0
c = (a ** 2 + b ** 2) ** 0.5
print("c =", c)
```

Note: we need to make use of the `**` operator to evaluate the square root as:

$$\sqrt{(x)} = x^{(½)}$$

and

$$c = \sqrt{a^2 + b^2}$$

Can you guess the output of the code?

Check below and run the code in the editor to confirm your predictions.

Check

```
1 a = 3.0
2 b = 4.0
3 c = (a ** 2 + b ** 2) ** 0.5
4 print("c =", c)
```

Console >_

```
c = 5.0
```

Estimated time

10 minutes

Level of difficulty

Easy

Objectives

- becoming familiar with the concept of storing and working with different data types in Python;
- experimenting with Python code.

Scenario

Here is a short story:

Once upon a time in Appleland, John had three apples, Mary had five apples, and Adam had six apples. They were all very happy and lived for a long time. End of story.

Your task is to:

- create the variables: `john`, `mary`, and `adam`;
- assign values to the variables. The values must be equal to the numbers of fruit possessed by John, Mary, and Adam respectively;
- having stored the numbers in the variables, print the variables on one line, and separate each of them with a comma;
- now create a new variable named `totalApples` equal to addition of the three former variables.
- print the value stored in `totalApples` to the console;
- **experiment with your code**: create new variables, assign different values to them, and perform various arithmetic operations on them (e.g., +, -, *, /, //, etc.). Try to print a string and an integer together on one line, e.g., `"Total number of apples:"` and `totalApples`.

```
jhon = 3
mary = 5
adam = 6

print(jhon, mary, adam)
totalApple = jhon + mary + adam
print(totalApple)
```

```
Console >_...

3 5 6
14
```

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`

LAB

Estimated time

10 minutes

Level of difficulty

Easy

Objectives

- becoming familiar with the concept of, and working with, variables;
- performing basic computations and conversions;
- experimenting with Python code.

Scenario

Miles and kilometers are units of length or distance.

Bearing in mind that `1` mile is equal to approximately `1.61` kilometers, complete the program in the editor so that it converts:

- miles to kilometers;
- kilometers to miles.

Do not change anything in the existing code. Write your code in the places indicated by `###`. Test your program with the data we've provided in the source code.

Pay particular attention to what is going on inside the `print()` function. Analyze how we provide multiple arguments to the function, and how we output the expected data.

Note that some of the arguments inside the `print()` function are strings (e.g., `"miles is"`), whereas some other are variables (e.g., `miles`).

HINT

There's one more interesting thing happening there. Can you see another function inside the `print()` function? It's the `round()` function. Its job is to round the outputted result to the number of decimal places specified in the parentheses, and return a float (inside the `round()` function you can find the variable name, a comma, and the number of decimal places we're aiming for). We're going to talk about functions very soon, so don't worry that everything may not be fully clear yet. We just want to spark your curiosity.

After completing the lab, open Sandbox, and experiment more. Try to write different converters, e.g., a USD to EUR converter, a temperature converter, etc. - let your imagination fly! Try to output the results by combining strings and variables. Try to use and experiment with the `round()` function to round your results to one, two, or three decimal places. Check out what happens if you don't provide any number of digits. Remember to test your programs.

Experiment, draw conclusions, and learn. Be curious.

Expected output

```
7.00 miles is 11.55 kilometers
12.25 kilometers is 7.61 miles
```

```
kilometers = 12.25
miles = 7.00

miles_to_kilometers = (miles * 1.61)
kilometers_to_miles = (kilometers / 1.61)

print(miles, "miles is", round(miles_to_kilometers, 2), "kilometers")
print(kilometers, "kilometers is", round(kilometers_to_miles, 2), "miles")
```

```
Console >_

Traceback (most recent call last):
  File "main.py", line 5, in <module>
    kilometers_to_miles = (kilometer / 1.61)
NameError: name 'kilometer' is not defined
7.00 miles is 11.55 kilometers
12.25 kilometers is 7.61 miles
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

- becoming familiar with the concept of numbers, operators, and arithmetic operations in Python;
- performing basic calculations.

Scenario

Take a look at the code in the editor: it reads a `float` value, puts it into a variable named `x`, and prints the value of a variable named `y`. Your task is to complete the code in order to evaluate the following expression:

$3x^3 \cdot 2x^2 + 3x - 1$

The result should be assigned to `y`.

Remember that classical algebraic notation likes to omit the multiplication operator - you need to use it explicitly. Note how we change data type to make sure that `x` is of type `float`.

Keep your code clean and readable, and test it using the data we've provided, each time assigning it to the `x` variable (by hardcoding it). Don't be discouraged by any initial failures. Be persistent and inquisitive.

Test Data

Sample input

```
x = 0
x = 1
x = -1
```

Expected Output

```
y = -1.0
y = 3.0
y = -9.0
```

output

```
x = -1# hardcode your test data here
x = float(x)
# write your code here
y = ((3 * (x ** 3)) - (2 * (x ** 2)) + (3 * x) - 1)
print("y =", y)
```

Console >_

```
y = -2.0
y = -3.0
y = 1.0
y = -1.0
y = 3.0
y = -9.0
```

Key takeaways

1. A **variable** is a named location reserved to store values in the memory. A variable is created or initialized automatically when you assign a value to it for the first time. (2.1.4.1)

2. Each variable must have a unique name - an **identifier**. A legal identifier name must be a non-empty sequence of characters, must begin with the underscore(_), or a letter, and it cannot be a Python keyword. The first character may be followed by underscores, letters, and digits. Identifiers in Python are case-sensitive. (2.1.4.1)

3. Python is a **dynamically-typed** language, which means you don't need to *declare* variables in it. (2.1.4.3) To assign values to variables, you can use a simple assignment operator in the form of the equal (`=`) sign, i.e., `var = 1`.

4. You can also use **compound assignment operators** (shortcut operators) to modify values assigned to variables, e.g., `var += 1`, or `var /= 5 * 2`. (2.1.4.8)

5. You can assign new values to already existing variables using the assignment operator or one of the compound operators, e.g.: (2.1.4.5)

```
var = 2
print(var)

var = 3
print(var)

var += 1
print(var)
```

6. You can combine text and variables using the `+` operator, and use the `print()` function to output strings and variables, e.g.: (2.1.4.4)

```
var = "007"
print("Agent " + var)
```

Exercise 1

What is the output of the following snippet?

```
var = 2
var * 3
print(var)
```

Check

Exercise 2

Which of the following variable names are illegal in Python?

```
my_var
m
101
averylongvariablename
m101
m 101
Del
del
```

Check

```
my_var
m
101 # incorrect (starts with a digit)
averylongvariablename
m101
m 101 # incorrect (contains a space)
Del
del # incorrect (is a keyword)
```

Exercise 3

What is the output of the following snippet?

```
a = '1'
b = '1'
print(a + b)
```

Check

Exercise 4

What is the output of the following snippet?

```
a = 6
b = 3
a /= 2 * b
print(a)
```

Check

```
1.0
```


Key takeaways

1. A **variable** is a named location reserved to store values in the memory. A variable is created or initialized automatically when you assign a value to it for the first time. (2.1.4.1)

2. Each variable must have a unique name - an **identifier**. A legal identifier name must be a non-empty sequence of characters, must begin with the underscore(_), or a letter, and it cannot be a Python keyword. The first character may be followed by underscores, letters, and digits. Identifiers in Python are case-sensitive. (2.1.4.1)

3. Python is a **dynamically-typed** language, which means you don't need to *declare* variables in it. (2.1.4.3) To assign values to variables, you can use a simple assignment operator in the form of the equal (`=`) sign, i.e., `var = 1`.

4. You can also use **compound assignment operators** (shortcut operators) to modify values assigned to variables, e.g., `var += 1`, or `var /= 5 * 2`. (2.1.4.8)

5. You can assign new values to already existing variables using the assignment operator or one of the compound operators, e.g.: (2.1.4.5)

```
var = 2
print(var)

var = 3
print(var)

var += 1
print(var)
```

6. You can combine text and variables using the `+` operator, and use the `print()` function to output strings and variables, e.g.: (2.1.4.4)

```
var = "007"
print("Agent " + var)
```

Exercise 1

What is the output of the following snippet?

```
var = 2
var * 3
print(var)
```

Check

Exercise 2

Which of the following variable names are illegal in Python?

```
my_var
m
101
averylongvariablename
m101
m 101
Del
del
```

Check

Exercise 3

What is the output of the following snippet?

```
a = '1'
b = '1'
print(a + b)
```

Check

Exercise 4

What is the output of the following snippet?

```
a = 6
b = 3
a /= 2 * b
print(a)
```

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 `squareArea` will obviously be better than `auntJane`.

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 in Sandbox.

LAB

Estimated time

5 minutes

Level of difficulty

Very Easy

Objectives

- becoming familiar with the concept of comments in Python;
- using and not using comments;
- replacing comments with code;
- experimenting with Python code.

Scenario

The code in the editor contains comments. Try to improve it: add or remove comments where you find it appropriate (yes, sometimes removing a comment can make the code more readable), and change variable names where you think this will improve code comprehension.

NOTE

Comments are very important. They are used not only to make your programs **easier to understand**, but also to **disable those pieces of code that are currently not needed** (e.g., when you need to test some parts of your code only, and ignore other). Good programmers **describe** each important piece of code, and give **self-commenting names** to variables, as sometimes it is simply much better to leave information in the code.

It's good to use **readable** variable names, and sometimes it's better to **divide your code** into named pieces (e.g., functions). In some situations, it's a good idea to write the steps of computations in a clearer way.

One more thing: it may happen that a comment contains a wrong or incorrect piece of information - you should never do that on purpose!

```
1  #this program computes the number of seconds in a given number of hours
2
3  noOfHour = 2 # number of hours
4  noOfSeconds = 3600 # number of seconds in 1 hour
5
6  print("Hours: ", noOfHour)
7  print("Seconds in Hours: ", noOfHour * noOfSeconds)
8
9  #here we should also print "Goodbye", but a programmer didn't have time to write any code
10 #this is the end of the program that computes the number of seconds in 2 hour
```

Console >_

```
Hours:  2
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print("Seconds in Hours: ", a * noOfSeconds)
NameError: name 'a' is not defined
Hours:  2
Seconds in Hours:  7200
Hours:  2
Seconds in Hours:  7200
```

Key takeaways

1. Comments can be used to leave additional information in code. They are omitted at runtime. The information left in source code is addressed to human readers. In Python, a comment is a piece of text that begins with `#`. The comment extends to the end of line.

2. If you want to place a comment that spans several lines, you need to place `#` in front of them all. Moreover, you can use a comment to mark a piece of code that is not needed at the moment (see the last line of the snippet below), e.g.:

```
# This program prints
# an introduction to the screen.
print("Hello!")  # Invoking the print() function
# print("I'm Python.")
```

3. Whenever possible and justified, you should give **self-commenting names** to variables, e.g., if you're using two variables to store a length and width of something, the variable names `length` and `width` may be a better choice than `myvar1` and `myvar2`.

4. It's important to use comments to make programs easier to understand, and to use readable and meaningful variable names in code. However, it's equally important **not to use** variable names that are confusing, or leave comments that contain wrong or incorrect information!

5. Comments can be important when *you* are reading your own code after some time (trust us, developers do forget what their own code does), and when *others* are reading your code (can help them understand what your programs do and how they do it more quickly).

Exercise 1

What is the output of the following snippet?

```
# print("String #1")
print("String #2")
```

Check

Exercise 2

What will happen when you run the following code?

```
# This is
a multiline
comment. #

print("Hello!")
```

Check

```
SyntaxError: invalid syntax
```

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)
```

The `input()` function - prohibited operations

Look at the code in the editor. Run it, enter any number, and press *Enter*.

What happens?

Python should have given you the following output:

```
Traceback (most recent call last):
File ".main.py", line 4, in <module>
something = anything ** 2.0
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'float'
```

The last line of the sentence explains everything - you tried to apply the `**` operator to `'str'` (string) accompanied with `'float'`.

This is prohibited.

This should be obvious - can you predict the value of `"to be or not to be"` raised to the power of `2`?

We can't. Python can't either.

Have we fallen into a deadlock? Is there a solution to this issue? Of course there is.

```
1  # Testing TypeError message
2
3  anything = int(input("Enter a number: "))
4  something = anything ** 2.0
5  print(anything, "to the power of 2 is", something)
```

Console >_

```
Enter a number: 2
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    something = anything ** 2.0
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'float'
Enter a number: 4
4 to the power of 2 is 16.0
```

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.

```
1  anything = float(input("Enter a number: "))
2  something = anything ** 2.0
3  print(anything, "to the power of 2 is", something)
```

Console >_

More about `input()` and type casting

Having a team consisting of the trio `input()` + `int()` + `float()` opens up lots of new possibilities.

You'll eventually be able to write complete programs, accepting data in the form of numbers, processing them and displaying the results.

Of course, these programs will be very primitive and not very usable, as they cannot make decisions, and consequently are not able to react differently to different situations.

This is not really a problem, though; we'll show you how to overcome it soon.

Our next example refers to the earlier program to find the length of a hypotenuse. Let's rewrite it and make it able to read the lengths of the legs from the console.

Check out the editor window - this is how it looks now.

The program asks the user twice for both legs' lengths, evaluates the hypotenuse and prints the result.

Run it and try to input some negative values.

The program - unfortunately - doesn't react to this obvious error.

Let's ignore this weakness for now. We'll come back to it soon.

Note that in the program that you can see in the editor, the `hypo` variable is used for only one purpose - to save the calculated value between the execution of the adjoining line of code.

As the `print()` function accepts an expression as its argument, you can **remove the variable** from the code.

Just like this:

```
leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))
print("Hypotenuse length is", (leg_a**2 + leg_b**2) ** .5)
```

```
leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))
hypo = (leg_a**2 + leg_b**2) ** .5
print("Hypotenuse length is", round(hypo, 2))
```

Console >_

```
Input first leg length: 2
Input second leg length: 4
Hypotenuse length is 4.472135954999858
Input first leg length: 4
Input second leg length: 2
Hypotenuse length is 4.47
```

String operators - introduction

It's time to return to these two arithmetic operators: `+` and `*`.

We want to show you that they have a second function. They are able to do something more than just **add** and **multiply**.

We've seen them in action where their arguments are numbers (floats or integers, it doesn't matter).

Now we're going to show you that they can handle strings, too, albeit in a very specific way.

Concatenation

The `+` (plus) sign, when applied to two strings, becomes a **concatenation operator**:

```
string + string
```

It simply **concatenates** (glues) two strings into one. Of course, like its arithmetic sibling, it can be used more than once in one expression, and in such a context it behaves according to left-sided binding.

In contrast to its arithmetic sibling, the concatenation operator is **not commutative**, i.e., `"ab" + "ba"` is not the same as `"ba" + "ab"`.

Don't forget - if you want the `+` sign to be a **concatenator**, not an adder, you must ensure that **both its arguments are strings**.

You cannot mix types here.

This simple program shows the `+` sign in its second use:

```
fnam = input("May I have your first name, please? ")
lnam = input("May I have your last name, please? ")
print("Thank you.")
print("\nYour name is " + fnam + " " + lnam + ".")
```

Note: using `+` to concatenate strings lets you construct the output in a more precise way than with a pure `print()` function, even if enriched with the `end=` and `sep=` keyword arguments.

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

```
fnam = input("May I have your first name, please? ")
lnam = input("May I have your last name, please? ")
print("Thank you.")
print("\nYour name is " + fnam + " " + lnam + ".")
```

Console >_

```
May I have your first name, please? golu
May I have your last name, please? kumar
Thank you.

Your name is golu kumar.
```

Replication

The `*` (asterisk) sign, when applied to a string and number (or a number and string, as it remains commutative in this position) becomes a **replication operator**:

```
string * number
number * string
```

It replicates the string the same number of times specified by the number.

For example:

- `"James" * 3` gives `"JamesJamesJames"`
- `3 * "an"` gives `"ananan"`
- `5 * "2"` (or `"2" * 5`) gives `"22222"` (not `10`!)

REMEMBER

A number less than or equal to zero produces an **empty string**.

This simple program "draws" a rectangle, making use of an old operator (`*`) in a new role:

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

Note the way in which we've used the parentheses in the second line of the code.

Try practicing to create other shapes or your own artwork!

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

Console >_

```
+----------+
|          |
|          |
|          |
|          |
|          |
+----------+
```

Type conversion: `str()`

You already know how to use the `int()` and `float()` functions to convert a string into a number.

This type of conversion is not a one-way street. You can also **convert a number into a string**, which is way easier and safer - this operation is always possible.

A function capable of doing that is called `str()`:

```
str(number)
```

To be honest, it can do much more than just transform numbers into strings, but that can wait for later.

The "right-angle triangle" again

Here is our "right-angle triangle" program again:

```
leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))
print("Hypotenuse length is " + str((leg_a**2 + leg_b**2) ** .5))
```

We've modified it a bit to show you how the `str()` function works. Thanks to this, we can **pass the whole result to the `print()` function as one string**, forgetting about the commas.

You've made some serious strides on your way to Python programming.

You already know the basic data types, and a set of fundamental operators. You know how to organize the output and how to get data from the user. These are very strong foundations for Module 2. But before we move on to the next module, let's do a few labs, and recap all that you've learned in this section.

```
leg_a = float(input("Input first leg length: "))
leg_b = float(input("Input second leg length: "))
print("Hypotenuse length is " + str((leg_a**2 + leg_b**2) ** .5))

```

Console >_

```
Input first leg length: 5
Input second leg length: 2
Hypotenuse length is 5.385164807134504
```

LAB

Estimated time

5-10 minutes

Level of difficulty

Easy

Objectives

- becoming familiar with the inputting and outputting of data in Python;
- evaluating simple expressions.

Scenario

Your task is to complete the code in order to evaluate the results of four basic arithmetic operations.

The results have to be printed to the console.

You may not be able to protect the code from a user who wants to divide by zero. That's okay, don't worry about it for now.

Test your code - does it produce the results you expect?

We won't show you any test data - that would be too simple.

```
# input a float value for variable a here
# input a float value for variable b here

# output the result of addition here
# output the result of subtraction here
# output the result of multiplication here
# output the result of division here

print("\nThat's all, folks!")

a = float(input("Enter 1st number: "))
b = float(input("Enter 2nd number: "))

print("addition of these number is: ", (a + b))
print("subtraction of these nuber is: ", (a - b))
print("multiplication of these number is: ", (a * b))
print("intiger division of these number is: ", (a // b))
```

Console >_

```
That's all, folks!
Enter 1st number: 5
Enter 2nd number: 3
addition of these number is:  8.0
subtraction of these nuber is:  2.0
multiplication of these number is:  15.0
intiger division of these number is:  1.0
```

Scenario

Your task is to complete the code in order to evaluate the following expression:

$$\cfrac{1}{x+\cfrac{1}{x+\cfrac{1}{x+\cfrac{1}{x}}}}$$

The result should be assigned to `y`. Be careful - watch the operators and keep their priorities in mind. Don't hesitate to use as many parentheses as you need.

You can use additional variables to shorten the expression (but it's not necessary). Test your code carefully.

Test Data

Sample input: `1`

Expected output:

```
y = 0.6000000000000001
```

Sample input: `10`

Expected output:

```
y = 0.09901951266867294
```

Sample input: `100`

Expected output:

```
y = 0.009999000199950014
```

Sample input: `-5`

Expected output:

```
y = -0.19285202567760344
```

```
x = float(input("Enter value for x: "))

# put your code here
y = (1/(x + (1/(x + (1/(x + (1/x)))))))

print("y =", y)

```

Console >_

```
  File "main.py", line 6
    print("y =", y)
        ^
SyntaxError: invalid syntax
Enter value for x: 1
y = 0.6000000000000001
Enter value for x: 10
y = 0.09901951266867294
Enter value for x: 100
y = 0.009999000199950014
Enter value for x: -5
y = -0.19285202567760344
```

LAB

Estimated time

15-20 minutes

Level of difficulty

Easy

Objectives

- improving the ability to use numbers, operators, and arithmetic operations in Python;
- using the `print()` function's formatting capabilities;
- learning to express everyday-life phenomena in terms of programming language.

Scenario

Your task is to prepare a simple code able to evaluate the **end time** of a period of time, given as a number of minutes (it could be arbitrarily large). The start time is given as a pair of hours (0..23) and minutes (0..59). The result has to be printed to the console.

For example, if an event starts at **12:17** and lasts **59 minutes**, it will end at **13:16**.

Don't worry about any imperfections in your code - it's okay if it accepts an invalid time - the most important thing is that the code produce valid results for valid input data.

Test your code carefully. Hint: using the `%` operator may be the key to success.

Test Data

Sample input:

```
12
17
59
```

Expected output: `13:16`

Sample input:

```
23
58
642
```

Expected output: `10:40`

```
hour = int(input("Starting time (hours): "))
mins = int(input("Starting time (minutes): "))
dura = int(input("Event duration (minutes): "))

# put your code here
#
```

Console >_

Key takeaways

1. The `print()` function **sends data to the console**, while the `input()` function **gets data from the console**.

2. The `input()` function comes with an optional parameter: **the prompt string**. It allows you to write a message before the user input, e.g.:

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")
```

3. When the `input()` function is called, the program's flow is stopped, the prompt symbol keeps blinking (it prompts the user to take action when the console is switched to input mode) until the user has entered an input and/or pressed the *Enter* key.

NOTE

You can test the functionality of the `input()` function in its full scope locally on your machine. For resource optimization reasons, we have limited the maximum program execution time in Edube to a few seconds. Go to Sandbox, copy-paste the above snippet, run the program, and do nothing - just wait a few seconds to see what happens. Your program should be stopped automatically after a short moment. Now open IDLE, and run the same program there - can you see the difference?

Tip: the above-mentioned feature of the `input()` function can be used to prompt the user to end a program. Look at the code below:

```
name = input("Enter your name: ")
print("Hello, " + name + ". Nice to meet you!")

print("\nPress Enter to end the program.")
input()
print("THE END.")
```

3. The result of the `input()` function is a string. You can add strings to each other using the concatenation (`+`) operator. Check out this code:

```
num1 = input("Enter the first number: ") # Enter 12
num2 = input("Enter the second number: ") # Enter 21

print(num1 + num2) # the program returns 1221
```

4. You can also multiply (`*` - replication) strings, e.g.:

```
myInput = input("Enter something: ") # Example input: hello
print(myInput * 3) # Expected output: hellohellohello
```

Exercise 1

What is the output of the following snippet?

```
x = int(input("Enter a number: ")) # the user enters 2
print(x * "5")
```

Check

55

Exercise 2

What is the expected output of the following snippet?

```
x = input("Enter a number: ") # the user enters 2
print(type(x))
```

Check

<class 'str'>

Congratulations! You have completed Module 2.

Well done! You've reached the end of Module 2 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 2:

- the basic methods of formatting and outputting data offered by Python, together with the primary kinds of data and numerical operators, their mutual relations and bindings;
- the concept of variables and variable naming conventions;
- the assignment operator, the rules governing the building of expressions;
- the inputting and converting of data;

You are now ready to take the module quiz and attempt the final challenge: Module 2 Test, which will help you gauge what you've learned so far.

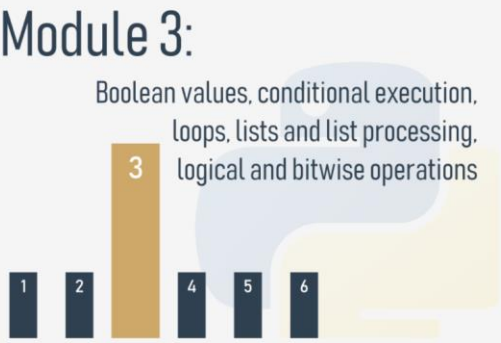
MODULE: 3

Module 3 - Boolean values, conditional execution, loops, lists and list processing, logical and bitwise operations in a new window

Programming Essentials in Python: Module 3

In this module, you will learn about:

- Boolean values;
- if-elif-else instructions;
- the while and for loops;
- flow control;
- logical and bitwise operations;
- lists and arrays.



Questions and answers

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

A computer executes the program and **provides the answers**. The program must be able to **react according to the received answers**.

Fortunately, computers know only two kinds of answers:

- yes, this is true;
- no, this is false.

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

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

Comparison: equality operator

Question: **are two values equal**?

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

Don't forget this important distinction:

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

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

Exercises

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

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

`2 == 2` Check

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

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

`2 == 2.` Check

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

`1 == 2` Check

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

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

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

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

```
var == 0
```

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

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

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

The question will be as follows:

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

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

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

```
1  var = 0 # assigning 0 to var
2  print(var == 0)
3
4  var = 1 # assigning 1 to var
5  print(var == 0)
6
7  var = 0 # assigning 0 to var
8  print(var != 0)
9
10 var = 1 # assigning 1 to var
11 print(var != 0)
```

Console >_

```
True
False
True
False
False
True
```

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

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

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

Run the code and check if you were right.

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

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

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

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

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

Run the code and check if you were right.

```
1  var = 0 # assigning 0 to var
2  print(var == 0)
3
4  var = 1 # assigning 1 to var
5  print(var == 0)
6
7  var = 0 # assigning 0 to var
8  print(var != 0)
9
10 var = 1 # assigning 1 to var
11 print(var != 0)
```

Console >_

```
True
False
True
False
False
True
```

Comparison operators: greater than

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

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

```
black_sheep > white_sheep # greater than
```

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

Comparison operators: greater than or equal to

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

There are two subsequent signs, not one.

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

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

```
centigrade_outside >= 0.0 # greater than or equal to
```

Comparison operators: less than or equal to

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

Look at this simple example:

```
current_velocity_mph < 85 # less than
current_velocity_mph <= 85 # less than or equal to
```

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

Making use of the answers

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

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

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

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

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

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

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

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

LAB

Estimated time

5 minutes

Level of difficulty

Very Easy

Objectives

- becoming familiar with the the `input()` function;
- becoming familiar with comparison operators in Python.

Scenario

Using one of the comparison operators in Python, write a simple two-line program that takes the parameter `n` as input, which is an integer, and prints `False` if `n` is less than `100`, and `True` if `n` is greater than or equal to `100`.

Don't create any `if` blocks (we're going to talk about them very soon). Test your code using the data we've provided for you.

Test Data

Sample input: `55`

Expected output: `False`

Sample input: `99`

Expected output: `False`

```
1 n = int(input("Enter a number: "))
2 print(n >= 100)
3 # print(n >= 100)
```

Console >_

```
Enter a number: 55
True
False
Enter a number: 55
True
Enter a number: 55
False
Enter a number:
Enter a number: 99
False
Enter a number: 100
True
Enter a number: -5
False
Enter a number: 123
True
```

Sample input: `100`

Expected output: `True`

Sample input: `101`

Expected output: `True`

Sample input: `-5`

Expected output: `False`

Sample input: `+123`

Expected output: `True`

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()
```

Conditonal execution: the `if` statement

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

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

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

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

Take a look at the following code:

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

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

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

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

Conditional execution: the `if-else` statement

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

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

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

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

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

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

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

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

The *if-else* execution goes as follows:

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

The `if-else` statement: more conditional execution

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

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

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

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

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

Nested `if-else` statements

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

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

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

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

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

Here are two important points:

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

The `elif` statement

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

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

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

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

Let's write the same scenario using Python:

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

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

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

Some additional attention has to be paid in this case:

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

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

Analyzing code samples

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

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

Example 1:

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

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

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

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

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

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

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

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

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

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

```
Console >_

Enter the first number: 78
Enter the second number: 56
Enter the third number: 98
The largest number is: 98
```

Example 2:

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

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

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

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

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

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

There are no other differences in the code.

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

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

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

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

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

```
Console >_

Enter the first number: 78
Enter the second number: 56
Enter the third number: 98
The largest number is: 98
```

Example 3:

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

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

Look at the code below:

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

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

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

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

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

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

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

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

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

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

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

```
Console >_

Enter the first number: 78
Enter the second number: 56
Enter the third number: 98
The largest number is: 98
```

Pseudocode and introduction to loops

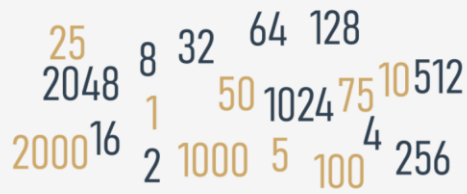
You should now be able to write a program which finds the largest of four, five, six, or even ten numbers.

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:

```
line 01 largest_number = -999999999
line 02 number = int(input())
line 03 if number == -1:
line 04     print(largest_number)
line 05     exit()
line 06 if number > largest_number:
line 07     largest_number = number
line 08 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 `largestNumber` 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.

PYTHON TIPS

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.

LAB

Estimated time

5-10 minutes

Level of difficulty

Easy

Objectives

- becoming familiar with the the `input()` function;
- becoming familiar with comparison operators in Python;
- becoming familiar with the concept of conditional execution.

Scenario

Spathiphyllum, more commonly known as a peace lily or white sail plant, is one of the most popular indoor houseplants that filters out harmful toxins from the air. Some of the toxins that it neutralizes include benzene, formaldehyde, and ammonia.

Imagine that your computer program loves these plants. Whenever it receives an input in the form of the word `Spathiphyllum`, it involuntarily shouts to the console the following string: `"Spathiphyllum is the best plant ever!"`

Write a program that utilizes the concept of conditional execution, takes a string as input, and:

- prints the sentence `"Yes - Spathiphyllum is the best plant ever!"` to the screen if the inputted string is `"Spathiphyllum"` (upper-case)
- prints `"No, I want a big Spathiphyllum!"` if the inputted string is `"spathiphyllum"` (lower-case)
- prints `"Spathiphyllum! Not [input]!"` otherwise. Note: `[input]` is the string taken as input.

```
1  plantName = input("Enter Spathiphyllum: ")
2
3  if plantName == "spathiphyllum":
4      print("No, I want a big Spathiphyllum!")
5  elif plantNmae == "Spathiphyllum":
6      print("yes - Spathiphyllum is the best plant ever!")
7  else:
8      print("Spathiphyllum! Not", plantName)
```

Console >_

```
Enter Spathiphyllum: spathiphyllum
No, I want a big Spathiphyllum
Enter Spathiphyllum: Spathiphyllum
No, I want a big Spathiphyllum
Enter Spathiphyllum: pelargonium
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    elif plantNmae.upper() == "SPATHIPHYLLUM":
NameError: name 'plantNmae' is not defined
  File "main.py", line 5
    elif plantNmae) == "Spathiphyllum":
                   ^
SyntaxError: invalid syntax
Enter Spathiphyllum: spathiphyllum
No, I want a big Spathiphyllum!
Enter Spathiphyllum: pelargonium
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    elif plantNmae == "Spathiphyllum":
NameError: name 'plantNmae' is not defined
```

Write a program that utilizes the concept of conditional execution, takes a string as input, and:

- prints the sentence `"Yes - Spathiphyllum is the best plant ever!"` to the screen if the inputted string is `"Spathiphyllum"` (upper-case)
- prints `"No, I want a big Spathiphyllum!"` if the inputted string is `"spathiphyllum"` (lower-case)
- prints `"Spathiphyllum! Not [input]!"` otherwise. Note: `[input]` is the string taken as input.

Test your code using the data we've provided for you. And get yourself a Spathiphyllum, too!

Test Data

Sample input: `spathiphyllum`

Expected output: `No, I want a big Spathiphyllum!`

Sample input: `pelargonium`

Expected output: `Spathiphyllum! Not pelargonium!`

Sample input: `Spathiphyllum`

Expected output: `Yes - Spathiphyllum is the best plant ever!`

Console >_

```
Enter Spathiphyllum: spathiphyllum
No, I want a big Spathiphyllum
Enter Spathiphyllum: Spathiphyllum
No, I want a big Spathiphyllum
Enter Spathiphyllum: pelargonium
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    elif plantNmae.upper() == "SPATHIPHYLLUM":
NameError: name 'plantNmae' is not defined
  File "main.py", line 5
    elif plantNmae) == "Spathiphyllum":
                   ^
SyntaxError: invalid syntax
Enter Spathiphyllum: spathiphyllum
No, I want a big Spathiphyllum!
Enter Spathiphyllum: pelargonium
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    elif plantNmae == "Spathiphyllum":
NameError: name 'plantNmae' is not defined
Enter Spathiphyllum: pelargonum
Spathiphyllum! Not pelargonum
Enter Spathiphyllum: Spathiphyllum
yes - Spathiphyllum is the best plant ever!
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy/Medium

Objectives

Familiarize the student with:

- using the *if-else* instruction to branch the control path;
- building a complete program that solves simple real-life problems.

Scenario

Once upon a time there was a land - a land of milk and honey, inhabited by happy and prosperous people. The people paid taxes, of course - their happiness had limits. The most important tax, called the *Personal Income Tax* (*PIT* for short) had to be paid once a year, and was evaluated using the following rule:

- if the citizen's income was not higher than 85,528 thalers, the tax was equal to 18% of the income minus 556 thalers and 2 cents (this was the so-called *tax relief*)
- if the income was higher than this amount, the tax was equal to 14,839 thalers and 2 cents, plus 32% of the surplus over 85,528 thalers.

Your task is to write a **tax calculator**.

- It should accept one floating-point value: the income.
- Next, it should print the calculated tax, rounded to full thalers. There's a function named `round()` which will do the rounding for you - you'll find it in the skeleton code in the editor.

Note: this happy country never returns money to its citizens. If the calculated tax is less than zero, it only means no tax at all (the tax is equal to zero). Take this into consideration during your calculations.

```
income = float(input("Enter the annual income: "))

#
# Put your code here.
#
if income < 85528:
    tax = (85528 - ((85528 * (18/100) - 556.2))
elif income > 85528:
    print(85528 - ((85528 * (32/100) - 14839.2))
else:
    tax = 0
tax = round(tax, 0)
print("The tax is:", tax, "thalers")
```

Console >_

Look at the code in the editor - it only reads one input value and outputs a result, so you need to complete it with some smart calculations.

Test your code using the data we've provided.

Test Data

Sample input: `10000`

Expected output: `The tax is: 1244.0 thalers`

Sample input: `100000`

Expected output: `The tax is: 19470.0 thalers`

Sample input: `1000`

Expected output: `The tax is: 0.0 thalers`

Sample input: `-100`

Expected output: `The tax is: 0.0 thalers`

Console >_

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy/Medium

Objectives

Familiarize the student with:

- using the `if-elif-else` statement;
- finding the proper implementation of verbally defined rules;
- testing code using sample input and output.

Scenario

As you surely know, due to some astronomical reasons, years may be *leap* or *common*. The former are 366 days long, while the latter are 365 days long.

Since the introduction of the Gregorian calendar (in 1582), the following rule is used to determine the kind of year:

- if the year number isn't divisible by four, it's a *common year*;
- otherwise, if the year number isn't divisible by 100, it's a *leap year*;
- otherwise, if the year number isn't divisible by 400, it's a *common year*;
- otherwise, it's a *leap year*.

Look at the code in the editor - it only reads a year number, and needs to be completed with the instructions implementing the test we've just described.

The code should output one of two possible messages, which are `Leap year` or `Common year`, depending on the value entered.

It would be good to verify if the entered year falls into the Gregorian era, and output a warning otherwise: `Not within the`

```
year = int(input("Enter a year: "))

#
# Put your code here.
#
if year % 4 == 0 or year % 400 == 0:
    print("Leap year")
elif year % 100 == 0:
    print("Common year")
elif year <= 1580:
    print("Not within the Gregorian calender peroid")
else:
    print("Common year")
```

Console >_

```
Enter a year: 2000
Leap year
Common year
Enter a year: 2015
Not within the Gregorian calender peroid
Enter a year: 2000
Leap year
Enter a year: 2015
Common year
Enter a year: 1999
Common year
Enter a year: 1996
Leap year
Enter a year: 1580
Leap year
```

The code should output one of two possible messages, which are `Leap year` or `Common year`, depending on the value entered.

It would be good to verify if the entered year falls into the Gregorian era, and output a warning otherwise: `Not within the Gregorian calendar period`. Tip: use the `!=` and `%` operators.

Test your code using the data we've provided.

Test Data

Sample input: `2000`

Expected output: `Leap year`

Sample input: `2015`

Expected output: `Common year`

Sample input: `1999`

Expected output: `Common year`

Sample input: `1996`

Expected output: `Leap year`

Sample input: `1580`

Expected output: `Not within the Gregorian calendar period`

```
#
# Put your code here.
#
if year % 4 == 0 or year % 400 == 0:
    print("Leap year")
elif year % 100 == 0:
    print("Common year")
elif year <= 1580:
    print("Not within the Gregorian calender peroid")
else:
    print("Common year")
```

Console >_...

```
Enter a year: 2000
Leap year
Common year
Enter a year: 2015
Not within the Gregorian calender peroid
Enter a year: 2000
Leap year
Enter a year: 2015
Common year
Enter a year: 1999
Common year
Enter a year: 1996
Leap year
Enter a year: 1580
Leap year
```

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 `True` 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`.

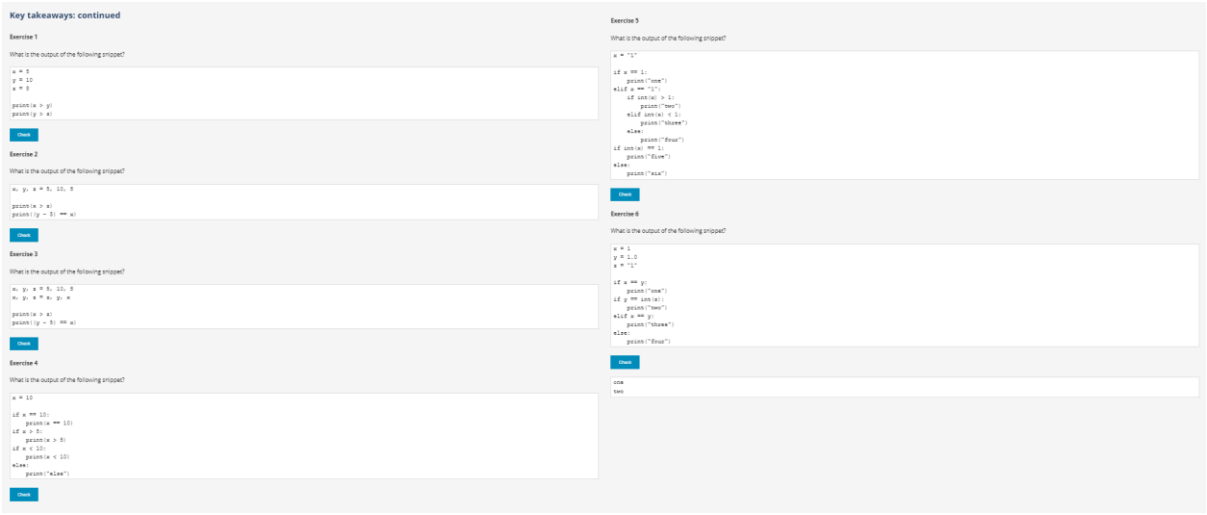
- The `if-elif-else` statement, e.g.:

```
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
x = 8

print(x > y)
print(y > x)
```

Check

Exercise 2

What is the output of the following snippet?

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

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

Check

Exercise 3

What is the output of the following snippet?

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

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

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

```
one
two
```

Looping your code with `while`

Do you agree with the statement presented below?

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

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

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

```
while conditional_expression:
    instruction
```

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

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

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

Look at the algorithm below:

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

It is now important to remember that:

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

An infinite loop

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

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

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

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

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. Locate the loop's body and find out **how the body is exited:**

```
# we will 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, we will continue
while number != -1:
    # is number larger than largest_number?
    if number > largest_number:
        # yes, update largest_number
        largest_number = number
    # input the next number
    number = int(input("Enter a number or type -1 to stop: "))

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

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

The `while` loop: more examples

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

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

odd_numbers = 0
even_numbers = 0

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

# 0 terminates execution
while number != 0:
    # check if the number is odd
    if number % 2 == 1:
        # increase the odd_numbers counter
        odd_numbers += 1
    else:
        # increase the even_numbers counter
        even_numbers += 1
    # read the next number
    number = int(input("Enter a number or type 0 to stop: "))

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

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

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

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

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

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

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

# odd_numbers = 0
# even_numbers = 0

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

# # 0 terminates execution
# while number != 0:
#     # check if the number is odd
#     if number % 2 == 1:
#         # increase the odd_numbers counter
#         odd_numbers += 1
#     else:
#         # increase the even_numbers counter
#         even_numbers += 1
#     # read the next number
#     number = int(input("Enter a number or type 0 to stop: "))

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

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

```
Console >_
Enter a number or type 0 to stop: 0
Odd numbers count: 2
Even numbers count: 3
Inside the loop. 5
Inside the loop. 4
Inside the loop. 3
Inside the loop. 2
Inside the loop. 1
Outside the loop. 0
Inside the loop. 5
Inside the loop. 4
Inside the loop. 3
Inside the loop. 2
Inside the loop. 1
Outside the loop. 0
```

Using a `counter` variable to exit a loop

Look at the snippet below:

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

This code is intended to print the string `"Inside the loop."` and the value stored in the `counter` variable during a given loop exactly five times. Once the condition has not been met (the `counter` variable has reached `0`), the loop is exited, and the message `"Outside the loop."` as well as the value stored in `counter` is printed.

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

Can you see the difference?

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

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

REMEMBER

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

```
# # 0 terminates execution
# while number != 0:
```

Console >_

```
Enter a number or type 0 to stop: 2
Enter a number or type 0 to stop: 4
Enter a number or type 0 to stop: 6
Enter a number or type 0 to stop: 5
Enter a number or type 0 to stop: 1
Enter a number or type 0 to stop: 0
Odd numbers count: 2
Even numbers count: 3
Inside the loop. 5
Inside the loop. 4
Inside the loop. 3
Inside the loop. 2
Inside the loop. 1
Outside the loop. 0
Inside the loop. 5
Inside the loop. 4
Inside the loop. 3
Inside the loop. 2
Inside the loop. 1
Outside the loop. 0
```

LAB

Estimated time

15 minutes

Level of difficulty

Easy

Objectives

Familiarize the student with:

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

Scenario

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

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

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

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

EXTRA INFO

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

```
secret_number = 777

print(
"""
+================================+
| Welcome to my game, muggle!    |
| Enter an integer number        |
| and guess what number I've     |
| picked for you.                |
| So, what is the secret number? |
+================================+
""")

user_number = int(input("Enter the number: "))

while user_number != secret_number:
    print("Ha ha! You're stuck in my loop!")
    user_number = int(input("Enter the number again: "))
print(secret_number, "Well done, muggle! You are free now.")
```

Console >_

```
| Welcome to my game, muggle!    |
| Enter an integer number        |
| and guess what number I've     |
| picked for you.                |
| So, what is the secret number? |
+================================+

Enter the number: 2323
Ha ha! You're stuck in my loop!
Enter the number again: 777
777 Well done, muggle! You are free now.
```

Looping your code with `for`

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

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

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

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

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

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

Take a look at the snippet:

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

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

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

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

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

```
for i in range(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`).

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

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

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

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

Run the program to find out if you were right.

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

```
The value of i is currently 2
The value of i is currently 5
```

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

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

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

Just like here - there will be no output:

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

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

Thus, there will be no output here, either:

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

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

```
pow = 1
for exp in range(16):
    print("2 to the power of", exp, "is", pow)
    pow *= 2
```

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

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

```
for i in range(2, 9, 3):
    print("The value of i is currently", i)

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

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

Console >_

```
The value of i is currently 2
The value of i is currently 5
The value of i is currently 2
The value of i is currently 5
```

Console >_

```
The value of i is currently 2
The value of i is currently 5
The value of i is currently 2
The value of i is currently 5
```

LAB

Estimated time

5 minutes

Level of difficulty

Very easy

Objectives

Familiarize the student with:

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

Scenario

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

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

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

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

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

Use the skeleton we've provided in the editor.

EXTRA INFO

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

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

Expected output

```
1 Mississippi
2 Mississippi
3 Mississippi
4 Mississippi
5 Mississippi
```

```
import time

for second in range(1, 6):
    print(second, "Mississippi")
    time.sleep(1)

print("Ready or not, here I come!")
```

Console >_

```
1 Mississippi
2 Mississippi
3 Mississippi
4 Mississippi
5 Mississippi
Ready or not, here I come!
```

The `break` and `continue` statements

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

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

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

These two instructions are:

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

Both these words are **keywords**.

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

```
# break - example

print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")

# continue - example

print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

Console >_

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

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

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

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

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

The `break` variant goes here:

```
largestNumber = -99999999
counter = 0

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

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

Run it, test it, and experiment with it.

```
largestNumber = -99999999
counter = 0

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

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

Console >_

```
Enter a number or type -1 to end program: 12
Enter a number or type -1 to end program: 4
Enter a number or type -1 to end program: -1
The largest number is 12
```

And now the `continue` variant:

```
largestNumber = -99999999
counter = 0

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

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

    if number > largestNumber:
        largestNumber = number
    number = int(input("Enter a number or type -1 to

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

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

```
largestNumber = -99999999
counter = 0

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

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

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

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

Console >_

```
Enter a number or type -1 to end program: 2
Enter a number or type -1 to end program: 56
Enter a number or type -1 to end program: 9
Enter a number or type -1 to end program: 34
Enter a number or type -1 to end program: -1
The largest number is 56
```

LAB

Estimated time

10 minutes

Level of difficulty

Easy

Objectives

Familiarize the student with:

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

Scenario

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

Design a program that uses a `while` loop and continuously asks the user to enter a word unless the user enters `"chupacabra"` as the secret exit word, in which case the message `"You've successfully left the loop."` should be printed to the screen, and the loop should terminate.

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

```
user_input = input("Enter a word: ")

while True:
    if user_input == "chupacabra":
        print("You've successfully left the loop.")
        break
    user_input = input("Enter a word: ")
```

Console >_

```
Enter a word: erer
Enter a word:
Enter a word: ere
Enter a word: efef
Enter a word: wew
Enter a word: ewr
Enter a word:
Enter a word: erer
Enter a word: e
Enter a word: eer
Enter a word:
Enter a word: ggh
Enter a word: hjjj
Enter a word: hhjj
Enter a word: chupacabra
You've successfully left the loop.
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

Familiarize the student with:

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

Scenario

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

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

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

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

Your program must:

```
# Prompt the user to enter a word
# and assign it to the userWord variable.
userWord = input("Enter your word: ")
userWord = userWord.upper()

for letter in userWord:
    # Complete the body of the for loop.
    if letter == "A":
        continue
    elif letter == "E":
        continue
    elif letter == "I":
        continue
    elif letter == "O":
        continue
    elif letter == "U":
        continue
    else:
        print(letter)
```

Console >_

```
Enter your word: gregory
G
R
G
R
Y
```

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

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

Test data

Sample input: `Gregory`

Expected output:

```
G
R
G
R
Y
```

Sample input: `abstemious`

Expected output:

```
B
S
T
M
S
```

Sample input: `IOUEA`

Expected output:

```
# Prompt the user to enter a word
# and assign it to the userWord variable.
userWord = input("Enter your word: ")
userWord = userWord.upper()

for letter in userWord:
    # Complete the body of the for loop.
    if letter == "A":
        continue
    elif letter == "E":
        continue
    elif letter == "I":
        continue
    elif letter == "O":
        continue
    elif letter == "U":
        continue
    else:
        print(letter)
```

Console >_

```
Enter your word: gregory
G
R
G
R
Y
R
G
R
Y
Enter your word: IOUEA
```

LAB

Estimated time

5-10 minutes

Level of difficulty

Easy

Objectives

Familiarize the student with:

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

Scenario

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

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

Your program must:

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

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

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

Test data

Sample input: `Gregory`

Expected output:

```
GRGRY
```

Sample input: `abstemious`

Expected output:

```
BSTMS
```

Sample input: `IOUEA`

Expected output:

```
wordWithoutVovels = "Gregory"
wordWithoutVovels = wordWithoutVovels.upper()

# Prompt the user to enter a word
# and assign it to the userWord variable


for letter in wordWithoutVovels:
    # Complete the body of the loop.
    if letter == 'A':
        continue
    elif letter == 'E':
        continue
    elif letter == 'I':
        continue
    elif letter == 'O':
        continue
    elif letter == 'U':
        continue
    else:
        print(letter, end="")

# Print the word assigned to wordWithoutVowels.
```

Console >_

```
    elif letter == 'U'
                   ^
SyntaxError: invalid syntax
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    for letter in userWord:
NameError: name 'userWord' is not defined
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    for letter in wordWithoutVovles:
NameError: name 'wordWithoutVovles' is not defined
GRGRY
```

```
# Prompt the user to enter a word
# and assign it to the userWord variable


for letter in wordWithoutVovels:
    # Complete the body of the loop.
    if letter == 'A':
        continue
    elif letter == 'E':
        continue
    elif letter == 'I':
        continue
    elif letter == 'O':
        continue
    elif letter == 'U':
        continue
    else:
        print(letter, end="")

# Print the word assigned to wordWithoutVowels.
```

Console >_

```
    elif letter == 'U'
                   ^
SyntaxError: invalid syntax
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    for letter in userWord:
NameError: name 'userWord' is not defined
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    for letter in wordWithoutVovles:
NameError: name 'wordWithoutVovles' is not defined
GRGRY
```

The `while` loop and the `else` branch

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

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

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

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

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

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

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

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

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

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

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

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

Can you guess the output?

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

Run the program and check its output.

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

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

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

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

```
for i in range(5):
    print(i)
else:
    print("else:", i)
```

Console >_

```
0
1
2
3
4
else: 4
```

LAB

Estimated time

20-30 minutes

Level of difficulty

Medium

Objectives

Familiarize the student with:

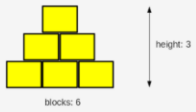
- using the `while` loop;
- finding the proper implementation of verbally defined rules;
- reflecting real-life situations in computer code.

Scenario

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

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

The figure illustrates the rule used by the builders:




```
blocks = int(input("Enter the number of blocks: "))

height = 0
inlayer = 1
while inlayer <= blocks:
    height += 1
    blocks -= inlayer
    inlayer += 1

print("The height of the pyramid:", height)
```

Console >_

```
Enter the number of blocks: 20
The height of the pyramid: 5
```

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

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

Test your code using the data we've provided.

Test Data

Sample input: `6`

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

Sample input: `20`

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

Sample input: `1000`

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

Sample input: `2`

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

Console >_

```
Enter the number of blocks: 20
The height of the pyramid: 5
Enter the number of blocks: 10000000
The height of the pyramid: 4471
```

LAB

Estimated time

20 minutes

Level of difficulty

Medium

Objectives

Familiarize the student with:

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

Scenario

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

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

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

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

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

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

Test your code using the data we've provided.

```
c0 = int(input("Enter c0: "))

if c0 > 1:
    steps = 0
    while c0 != 1:
        if c0 %2 != 0:
            cnew = 3 * c0 + 1
        else:
            cnew = c0 // 2
        print(c0)
        c0 = cnew
        steps += 1
    print("steps =",steps)
else:
    print("Bad c0 value")
```

Console >_

```
Enter c0: 5
5
16
8
4
2
steps = 5
```

Test Data

Sample input: `15`

Expected output:

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

Sample input: `16`

Expected output:

```
8
4
2
1
steps = 4
```

Sample input: `1023`

Expected output:

```
c0 = int(input("Enter c0: "))

if c0 > 1:
    steps = 0
    while c0 != 1:
        if c0 %2 != 0:
            cnew = 3 * c0 + 1
        else:
            cnew = c0 // 2
        print(c0)
        c0 = cnew
        steps += 1
    print("steps =",steps)
else:
    print("Bad c0 value")
```

Console >_

```
Enter c0: 5
5
16
8
4
2
steps = 5
```

Key takeaways

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

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

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

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

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

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

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

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

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

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

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

```
text = "pypypypyk"
for letter in text:
    if letter == "y":
        continue
    print(letter, end="")
```

3. The `while` and `for` loops can also have an `else` clause in Python. The `else` clause executes after the loop finishes its execution as long as it has not been terminated by `break`, e.g.:

```
n = 0

while n != 3:
    print(n)
    n += 1
else:
    print(n, "else")

print()

for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```

4. The `range()` function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of `range()` looks as follows: `range(start, stop, step)`, where:

- `start` is an optional parameter specifying the starting number of the sequence (0 by default)
- `stop` is an optional parameter specifying the end of the sequence generated (it is not included),
- and `step` is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

Example code:

```
for i in range(3):
    print(i, end=" ") # outputs: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ") # outputs: 6, 4, 2
```

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

Sample solution:

```
for i in range(0, 11):
    if i % 2 != 0:
        print(i)
```

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

Sample solution:

```
x = 1
while x < 11:
    if x % 2 != 0:
        print(x)
    x += 1
```

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

Sample solution:

```
for ch in "john.smith@pythoninstitute.org":
    if ch == "@":
        break
    print(ch, end="")
```

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 "01650318506510":
    if digit == "0":
        # line of code
        # line of code
    # line of code
```

Check

Sample solution:

```
for digit in "01650318506510":
    if digit == "0":
        print("x", end="")
        continue
    print(digit, end="")
```

Exercise 5

What is the output of the following code?

```
n = 3

while n > 0:
    print(n + 1)
    n -= 1
else:
    print(n)
```

Check

```
4
3
2
0
```

Exercise 6

What is the output of the following code?

```
n = range(4)

for num in n:
    print(num - 1)
else:
    print(num)
```

Check

```
-1
0
1
2
3
```

Exercise 7

What is the output of the following code?

```
for i in range(0, 6, 3):
    print(i)
```

Check

```
0
3
```

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

Logical expressions

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

```
print(var > 0)
print(not (var <= 0))
```

```
print(var != 0)
print(not (var == 0))
```

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

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

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

Let's write the same thing using Python:

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

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

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

Logical values vs. single bits

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

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

```
i = 1
j = not not i
```

Bitwise operators

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

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

Here are all of them:

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

Bitwise operations (&, |, and ^)

Arg A	Arg B	Arg B & Arg B	Arg A | Arg B	Arg A ^ Arg B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise operations (~)

Arg	~Arg
0	1
1	0

Let's make it easier:

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

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

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

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

Logical vs. bit operations: continued

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

```
i = 15
j = 22
```

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

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

The assignment is given:

```
log = i and j
```

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

```
log: True
```

Now the bitwise operation - here it is:

```
bit = i & j
```

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

`i`	`00000000000000000000000000001111`
`j`	`00000000000000000000000000010110`
`bit = i & j`	`00000000000000000000000000000110`

These bits correspond to the integer value of six.

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

```
logneg = not i
```

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

The bitwise negation goes like this:

```
bitneg = ~i
```

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

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

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

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

How do we deal with single bits?

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

```
flagRegister = 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`:

```
flagRegister = 0000000000000000000000000x000
```

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 `flagRegister` variable along with the following bit image:

```
00000000000000000000000001000
```

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

- `00000000000000000000000001000` if your bit was set to `1`
- `00000000000000000000000000000` if your bit was reset to `0`

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

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

```
theMask = 8
```

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

```
if flagRegister & theMask:
    # 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:

```
11111111111111111111111110111
```

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

```
flagRegister = flagRegister & ~theMask

flagregister &= ~theMask
```

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:

```
flagRegister = flagRegister | theMask

flagRegister |= theMask
```

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:

```
flagRegister = flagRegister ^ theMask

flagRegister ^= theMask
```

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
```

```
var = 17
varRight = var >> 1
varLeft = var << 2
print(var, varLeft, varRight)
```

Console >_

```
17 68 8
```

Note:

- `17 // 2` → `8` (shifting to the right by one bit is the same as integer division by two)
- `17 * 4` → `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	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `>>=`, `<<=`	

Console >_

```
17 68 8
```

Key takeaways

1. Python supports the following logical operators:

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

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

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

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

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

Exercise 1

What is the output of the following snippet?

```
x = 1
y = 0

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

Check

Exercise 2

What is the output of the following snippet?

```
x = 4
y = 1

a = x & y
b = x | y
c = ~x
d = x ^ 5
e = x >> 2
f = x << 2

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

Check

```
0 5 -5 1 1 16
```

Why do we need lists?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Indexing lists

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

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

```
numbers = [10, 5, 7, 2, 1]
print("Original list content:", numbers) # printing original list content

numbers[0] = 111
print("New list content: ", numbers) # current list content
```

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

```
numbers = [10, 5, 7, 2, 1]
print("Original list content:", numbers) # printing original list content

numbers[0] = 111
print("\nPrevious list content:", numbers) # printing previous list content

numbers[1] = numbers[4] # copying value of the fifth element to the second
print("New list content:", numbers) # printing current list content
```

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

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

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

```
1  numbers = [10, 5, 7, 2, 1]
2  print("List content:", numbers) # printing original list content
```

Console >_

```
List content: [10, 5, 7, 2, 1]
```

Accessing list content

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

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

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

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

```
print(numbers) # printing the whole list
```

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

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

The `len()` function

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

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

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

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

```
1  numbers = [10, 5, 7, 2, 1]
2  print("Original list content:", numbers) # printing original list content
3
4  numbers[0] = 111
5  print("\nPrevious list content:", numbers) # printing previous list content
6
7  numbers[1] = numbers[4] # copying value of the fifth element to the second
8  print("Previous list content:", numbers) # printing previous list content
9
10 print("\nList length:", len(numbers)) # printing the list's length
```

Console >_

```
Original list content: [10, 5, 7, 2, 1]

Previous list content: [111, 5, 7, 2, 1]
Previous list content: [111, 1, 7, 2, 1]

List length: 5
```

Removing elements from a list

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

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

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

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

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

```
print(numbers[4])
numbers[4] = 1
```

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

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

```
numbers = [10, 5, 7, 2, 1]
print("Original list content:", numbers) # printing original list content

numbers[0] = 111
print("\nPrevious list content:", numbers) # printing previous list content

numbers[1] = numbers[4] # copying value of the fifth element to the second
print("Previous list content:", numbers) # printing previous list content

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

###

del numbers[1] # removing the second element from the list
print("New list's length:", len(numbers)) # printing new list length
print("\nNew list content:", numbers) # printing current list content

###
```

Console >_

```
Original list content: [10, 5, 7, 2, 1]

Previous list content: [111, 5, 7, 2, 1]
Previous list content: [111, 1, 7, 2, 1]

List's length: 5
New list's length: 4

New list content: [111, 7, 2, 1]
```

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!

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

Console >_

```
1
2
```

LAB

Estimated time

5 minutes

Level of difficulty

Very easy

Objectives

Familiarize the student with:

- using basic instructions related to lists;
- creating and modifying lists.

Scenario

There once was a hat. The hat contained no rabbit, but a list of five numbers: `1`, `2`, `3`, `4`, and `5`.

Your task is to:

- write a line of code that prompts the user to replace the middle number in the list with an integer number entered by the user (step 1)
- write a line of code that removes the last element from the list (step 2)
- write a line of code that prints the length of the existing list (step 3.)

Ready for this challenge?

```
hatList = [1, 2, 3, 4, 5]  # This is an existing list of numbers hidden in the hat.
print(hatList)
# Step 1: write a line of code that prompts the user
# to replace the middle number with an integer number entered by the user.
user_input = int(input("Enter a number: "))
hatList[2] = user_input
print(hatList)

# Step 2: write a line of code here that removes the last element from the list.
del hatList[-1]
print(hatList)
# Step 3: write a line of code here that prints the length of the existing list.
print(len(hatList))
```

Console >_

```
[1, 2, 3, 4, 5]
Enter a number: 7
[1, 2, 7, 4, 5]
[1, 2, 7, 4]
4
```

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.

Adding elements to a list: `append()` and `insert()`

A new element may be *glued* to the end of the existing list:

```
list.append(value)
```

Such an operation is performed by a method named `append()`. It takes its argument's value and puts it **at the end of the list** which owns the method.

The list's length then increases by one.

The `insert()` method is a bit smarter - it can add a new element **at any place in the list**, not only at the end.

```
list.insert(location, value)
```

It takes two arguments:

- the first shows the required location of the element to be inserted; note: all the existing elements that occupy locations to the right of the new element (including the one at the indicated position) are shifted to the right, in order to make space for the new element;
- the second is the element to be inserted.

Look at the code in the editor. See how we use the `append()` and `insert()` methods. Pay attention to what happens after using `insert()`: the former first element is now the second, the second the third, and so on.

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

###

numbers.append(4)

print(len(numbers))
print(numbers)

###

numbers.insert(0, 222)
print(len(numbers))
print(numbers)

#
```

Console >_

```
4
[111, 7, 2, 1]
5
[111, 7, 2, 1, 4]
6
[222, 111, 7, 2, 1, 4]
```

Add the following snippet after the last line of code in the editor:

```
numbers.insert(1, 333)
```

Print the final list content to the screen and see what happens. The snippet above snippet inserts `333` into the list, making it the second element. The former second element becomes the third, the third the fourth, and so on.

Console >_

```
4
[111, 7, 2, 1]
5
[111, 7, 2, 1, 4]
6
[222, 111, 7, 2, 1, 4]
```

Adding elements to a list: continued

You can **start a list's life by making it empty** (this is done with an empty pair of square brackets) and then adding new elements to it as needed.

Take a look at the snippet in the editor. Try to guess its output after the `for` loop execution. Run the program to check if you were right.

It'll be a sequence of consecutive integer numbers from `1` (you then add one to all the appended values) to `5`.

We've modified the snippet a bit:

```
myList = [] # creating an empty list

for i in range(5):
    myList.insert(0, i + 1)

print(myList)
```

what will happen now? Run the program and check if this time you were right, too.

You should get the same sequence, but in **reverse order** (this is the merit of using the `insert()` method).

```
myList = [] # creating an empty list

for i in range(5):
    myList.append(i + 1)

print(myList)

myList = [] # creating an empty list

for i in range(5):
    myList.insert(0, i + 1)

print(myList)
```

Console >_

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
```

Making use of lists

The `for` loop has a very special variant that can **process lists** very effectively - let's take a look at that.

Let's assume that you want to **calculate the sum of all the values stored in the `myList` list**.

You need a variable whose sum will be stored and initially assigned a value of `0` - its name will be `total`. (Note: we're not going to name it `sum` as Python uses the same name for one of its built-in functions - `sum()`. Using the same name would generally be considered a bad practice.) Then you add to it all the elements of the list using the `for` loop. Take a look at the snippet in the editor.

Let's comment on this example:

- the list is assigned a sequence of five integer values;
- the `i` variable takes the values `0`, `1`, `2`, `3`, and `4`, and then it indexes the list, selecting the subsequent elements: the first, second, third, fourth and fifth;
- each of these elements is added together by the `+=` operator to the `total` variable, giving the final result at the end of the loop;
- note the way in which the `len()` function has been employed - it makes the code independent of any possible changes in the list's content.

```
myList = [10, 1, 8, 3, 5]
total = 0
print(sum(myList))
for i in range(len(myList)):
    total += myList[i]

print(total)
```

Console >_

```
27
27
```

The second face of the `for` loop

But the `for` loop can do much more. It can hide all the actions connected to the list's indexing, and deliver all the list's elements in a handy way.

This modified snippet shows how it works:

```
myList = [10, 1, 8, 3, 5]
total = 0

for i in myList:
    total += i

print(total)
```

Console >_

```
27
27
```

What happens here?

- the `for` instruction specifies the variable used to browse the list (`i` here) followed by the `in` keyword and the name of the list being processed (`myList` here)
- the `i` variable is assigned the values of all the subsequent list's elements, and the process occurs as many times as there are elements in the list;
- this means that you use the `i` variable as a copy of the elements' values, and you don't need to use indices;
- the `len()` function is not needed here, either.

Lists in action

Let's leave lists aside for a short moment and look at one intriguing issue.

Imagine that you need to rearrange the elements of a list, i.e., reverse the order of the elements: the first and the fifth as well as the second and fourth elements will be swapped. The third one will remain untouched.

Question: how can you swap the values of two variables?

Take a look at the snippet:

```
variable1 = 1
variable2 = 2

variable2 = variable1
variable1 = variable2
```

```
variable1 = 1
variable2 = 2

variable1, variable2 = variable2, variable1
print(variable1, variable2)
```

Console >_

```
2 1
```

If you do something like this, you would **lose the value previously stored** in `variable2`. Changing the order of the assignments will not help. You need a **third variable that serves as an auxiliary storage**.

This is how you can do it:

```
variable1 = 1
variable2 = 2

auxiliary = variable1
variable1 = variable2
variable2 = auxiliary
```

Console >_

```
2 1
```

Python offers a more convenient way of doing the swap - take a look:

```
variable1 = 1
variable2 = 2

variable1, variable2 = variable2, variable1
```

Clear, effective and elegant - isn't it?

Lists in action

Now you can easily **swap** the list's elements to **reverse their order**:

```
myList = [10, 1, 8, 3, 5]

myList[0], myList[4] = myList[4], myList[0]
myList[1], myList[3] = myList[3], myList[1]

print(myList)
```

Run the snippet. Its output should look like this:

```
[5, 3, 8, 1, 10]
```

It looks fine with five elements.

Will it still be acceptable with a list containing 100 elements? No, it won't.

Can you use the `for` loop to do the same thing automatically, irrespective of the list's length? Yes, you can.

```
1 myList = [10, 1, 8, 3, 5]
2 length = len(myList)
3
4 for i in range(length // 2):
5     myList[i], myList[length - i - 1] = myList[length - i - 1], myList[i]
6
7 print(myList)
```

Console >_

```
[5, 3, 8, 1, 10]
```

This is how we've done it:

```
myList = [10, 1, 8, 3, 5]
length = len(myList)

for i in range(length // 2):
    myList[i], myList[length - i - 1] = myList[length - i - 1], myList[i]

print(myList)
```

Note:

- we've assigned the `length` variable with the current list's length (this makes our code a bit clearer and shorter)
- we've launched the `for` loop to run through its body `length // 2` times (this works well for lists with both even and odd lengths, because when the list contains an odd number of elements, the middle one remains untouched)
- we've swapped the i^{th} element (from the beginning of the list) with the one with an index equal to `(length - i - 1)` (from the end of the list); in our example, for `i` equal to `0` the `(l - i - 1)` gives `4`; for `i` equal to `1`, it gives `3` - this is exactly what we needed.

Lists are extremely useful, and you'll encounter them very often.

```
5     myList[i], myList[length - i - 1] = myList[length - i - 1], myList[i]
6
7 print(myList)
```

Console >_

```
[5, 3, 8, 1, 10]
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

Familiarize the student with:

- creating and modifying simple lists;
- using methods to modify lists.

Scenario

The Beatles were one of the most popular music group of the 1960s, and the best-selling band in history. Some people consider them to be the most influential act of the rock era. Indeed, they were included in *Time* magazine's compilation of the 20th Century's 100 most influential people.

The band underwent many line-up changes, culminating in 1962 with the line-up of John Lennon, Paul McCartney, George Harrison, and Richard Starkey (better known as Ringo Starr).

Write a program that reflects these changes and lets you practice with the concept of lists. Your task is to:

```
1  # step 1:
2  Beatles = []
3  print("Step 1:", Beatles)
4
5  # step 2:
6
7  Beatles.append("John Lennon")
8  Beatles.append("Paul McCartney")
9  Beatles.append("George Harrison")
10 print("Step 2:", Beatles)
11
12 # step 3:
13 for members in range(2):
14     Beatles.append(input("New band member: "))
15 print("Step 3:", Beatles)
16
17 # step 4:
18 del Beatles[-1]
19 del Beatles[-1]
20 print("Step 4:", Beatles)
21
22 # step 5:
```

Console >_

```
Step 1: []
Step 2: ['John Lennon', 'Paul McCartney', 'George Harrison']
New band member: efrwerwer
New band member: fwGrgreG
Step 3: ['John Lennon', 'Paul McCartney', 'George Harrison', 'efrwerwer', 'f
Step 4: ['John Lennon', 'Paul McCartney', 'George Harrison']
Step 5: ['RingoStarr', 'John Lennon', 'Paul McCartney', 'George Harrison']
The Fab: 4
```

- step 1: create an empty list named `beatles`;
- step 2: use the `append()` method to add the following members of the band to the list: `John Lennon`, `Paul McCartney`, and `George Harrison`;
- step 3: use the `for` loop and the `append()` method to prompt the user to add the following members of the band to the list: `Stu Sutcliffe`, and `Pete Best`;
- step 4: use the `del` instruction to remove `Stu Sutcliffe` and `Pete Best` from the list;
- step 5: use the `insert()` method to add `Ringo Starr` to the beginning of the list.

By the way, are you a Beatles fan?


```
Step 1: []
Step 2: ['John Lennon', 'Paul McCartney', 'George Harrison']
New band member: efrwerwer
New band member: fwGrgreG
Step 3: ['John Lennon', 'Paul McCartney', 'George Harrison', 'efrwerwer', 'f
Step 4: ['John Lennon', 'Paul McCartney', 'George Harrison']
Step 5: ['RingoStarr', 'John Lennon', 'Paul McCartney', 'George Harrison']
The Fab: 4
```

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.:

```
myList = [1, None, True, "I am a string", 256, 0]
```

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

```
myList = [1, None, True, 'I am a string', 256, 0]
print(myList[3]) # outputs: I am a string
print(myList[-1]) # outputs: 0

myList[1] = '?'
print(myList) # outputs: [1, '?', True, 'I am a string', 256, 0]

myList.insert(0, "first")
myList.append("last")
print(myList) # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']
```

3. Lists can be **nested**, e.g.: `myList = [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.:

```
myList = [1, 2, 3, 4]
del myList[2]
print(myList) # outputs: [1, 2, 4]

del myList # 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.:

```
myList = ["white", "purple", "blue", "yellow", "green"]

for color in myList:
    print(color)
```

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

```
myList = ["white", "purple", "blue", "yellow", "green"]
print(len(myList)) # outputs 5

del myList[2]
print(len(myList)) # 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

```
[6, 2, 3, 4, 5, 1]
```

Exercise 2

What is the output of the following snippet?

```
lst = [1, 2, 3, 4, 5]
lst2 = []
add = 0

for number in lst:
    add += number
    lst2.append(add)

print(lst2)
```

Check

```
[1, 3, 6, 10, 15]
```

Exercise 3

What happens when you run the following snippet?

```
lst = []
del lst
print(lst)
```

Check

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

Exercise 4

What is the output of the following snippet?

```
lst = [1, [2, 3], 4]
print(lst[1])
print(len(lst))
```

Check

```
[2, 3]
3
```

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

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:

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.

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`.

```
myList = [8, 10, 6, 2, 4] # list to sort

for i in range(len(myList) - 1): # we need (5 - 1) comparisons
    if myList[i] > myList[i + 1]: # compare adjacent elements
        myList[i], myList[i + 1] = myList[i + 1], myList[i] # if we end up here it means that we
```

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

```
myList = [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(myList) - 1):
        if myList[i] > myList[i + 1]:
            swapped = True # swap occured!
            myList[i], myList[i + 1] = myList[i + 1], myList[i]

print(myList)
```

```
myList = [8, 10, 6, 2, 4] # list to sort
swapped = True # it's a little fake - we need it to enter the while loop
count = 0

while swapped:
    swapped = False # no swaps so far
    for i in range(len(myList) - 1):
        if myList[i] > myList[i + 1]:
            swapped = True # swap occured!
            count += 1
            myList[i], myList[i + 1] = myList[i + 1], myList[i]

print(myList, " it takes", count , "comparision to sort the list ")
```

Console >_

```
[2, 4, 6, 8, 10]  it takes 8 comparision to sort the list
```

The bubble sort - interactive version

In the editor you can see a complete program, enriched by a conversation with the user, and allowing the user to enter and to print elements from the list: **The bubble sort - final interactive version**.

Python, however, has its own sorting mechanisms. No one needs to write their own sorts, as there is a sufficient number of **ready-to-use tools**.

We explained this sorting system to you because it's important to learn how to process a list's contents, and to show you how real sorting may work.

If you want Python to sort your list, you can do it like this:

```
myList = [8, 10, 6, 2, 4]
myList.sort()
print(myList)
```

It is as simple as that.

The snippet's output is as follows:

```
[2, 4, 6, 8, 10]
```

As you can see, all the lists have a method named `sort()`, which sorts them as fast as possible. You've already learned about some of the list methods before, and you're going to learn more about others very soon.

```
myList = []
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: "))
    myList.append(val)

while swapped:
    swapped = False
    for i in range(len(myList) - 1):
        if myList[i] > myList[i + 1]:
            swapped = True
            myList[i], myList[i + 1] = myList[i + 1], myList[i]

print("\nSorted:")
print(myList)
```

Console >_

```
  File "main.py", line 7, in <module>
    myList.append(val)
NameError: name 'myList' is not defined
How many elements do you want to sort: 5
Enter a list element: 3
Enter a list element: 1
Enter a list element: 8
Enter a list element: 0
Enter a list element: 3

Sorted:
[0.0, 1.0, 3.0, 3.0, 8.0]
```

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

```
['A', 'D', 'F', 'Z']
```

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)
```

Check

```
[' ', 'C', 'B', 'A']
```

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 `list1`;
- assigns it to a new list named `list2`;
- changes the only element of `list1`;
- prints out `list2`.

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: `list2 = list1` copies the name of the array, not its contents. In effect, the two names (`list1` and `list2`) identify the same location in the computer memory. Modifying one of them affects the other, and vice versa.

How do you cope with that?

```
1 list1 = [1]
2 list2 = list1
3 list1[0] = 2
4 print(list2)
```

Console >_

```
[2]
```

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:

```
list1 = [1]
list2 = list1[:]
list1[0] = 2
print(list2)
```

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:

```
myList[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:

```
# Copying the whole list
list1 = [1]
list2 = list1[:]
list1[0] = 2
print(list2)

# Copying part of the list
myList = [10, 8, 6, 4, 2]
newList = myList[1:3]
print(newList)
```

Console >_

```
[1]
[8, 6]
```

```
myList = [10, 8, 6, 4, 2]
newList = myList[1:3]
print(newList)
```

Console >_

```
[1]
[8, 6]
```

The `newList` 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]`

Slices - negative indices

Look at the snippet below:

```
myList[start:end]
```

To repeat:

- `start` is the index of the first element **included in the slice**;
- `end` is the index of the first element **not included in the slice.**

This is how **negative indices** work with the slice:

```
myList = [10, 8, 6, 4, 2]
newList = myList[1:-1]
print(newList)
```

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

If the `start` specifies an element lying further than the one described by the `end` (from the list's beginning point of view), the slice will be **empty**:

```
myList = [10, 8, 6, 4, 2]
newList = myList[-1:1]
print(newList)
```

The snippet's output is: `[]`.

```
myList = [10, 8, 6, 4, 2]
newList = myList[1:-1]
print(newList)

myList = [10, 8, 6, 4, 2]
newList = myList[-1:1]
print(newList)
```

Console >_

```
[8, 6, 4]
[]
```

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:

```
myList[:end]
```

is a more compact equivalent of:

```
myList[0:end]
```

Look at the snippet below:

```
myList = [10, 8, 6, 4, 2]
newList = myList[:3]
print(newList)
```

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(myList)`.

In other words, the slice of this form:

```
myList[start:]
```

is a more compact equivalent of:

```
myList[start:len(myList)]
```

Look at the following snippet:

```
myList = [10, 8, 6, 4, 2]
newList = myList[3:]
print(newList)
```

```
lst = [23, 4, 78, 9, 0, 1]
print([0:(len(lst))])
```

Console >_

```
  File "main.py", line 2
    print([:len(lst)])
           ^
SyntaxError: invalid syntax
  File "main.py", line 2
    print([0:len(lst)])
            ^
SyntaxError: invalid syntax
  File "main.py", line 2
    print([0:(len(lst))])
             ^
SyntaxError: invalid syntax
```

Slices: continued

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

```
myList = [10, 8, 6, 4, 2]
newList = myList[:]
print(newList)
```

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**:

```
myList = [10, 8, 6, 4, 2]
del myList[1:3]
print(myList)
```

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:

```
myList = [10, 8, 6, 4, 2]
del myList[:]
print(myList)
```

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

Removing the slice from the code changes its meaning dramatically.

```
1 myList = [10, 8, 6, 4, 2]
2 del myList
3 print(myList)
4
```

Console >_

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print(myList)
NameError: name 'myList' is not defined
```

Take a look:

```
myList = [10, 8, 6, 4, 2]
del myList
print(myList)
```

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.


```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print(myList)
NameError: name 'myList' is not defined
```


The `in` and `not in` operators

Python offers two very powerful operators, able to **look through the list in order to check whether a specific value is stored inside the list or not**.

These operators are:

```
elem in myList
elem not in myList
```

The first of them (`in`) checks if a given element (its left argument) is currently stored somewhere inside the list (the right argument) - the operator returns `True` in this case.

The second (`not in`) checks if a given element (its left argument) is absent in a list - the operator returns `True` in this case.

Look at the code in the editor. The snippet shows both operators in action. Can you guess its output? Run the program to check if you were right.

```
1 myList = [0, 3, 12, 8, 2]
2
3 print(5 in myList)
4 print(5 not in myList)
5 print(12 in myList)
```

Console >_

```
False
True
True
```

Lists - some simple programs

Now we want to show you some simple programs utilizing lists.

The first of them tries to find the greater value in the list. Look at the code in the editor.

The concept is rather simple - we temporarily assume that the first element is the largest one, and check the hypothesis against all the remaining elements in the list.

The code outputs `17` (as expected).

The code may be rewritten to make use of the newly introduced form of the `for` loop:

```
myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = myList[0]

for i in myList:
    if i > largest:
        largest = i

print(largest)
```

The program above performs one unnecessary comparison, when the first element is compared with itself, but this isn't a problem at all.

The code outputs `17`, too (nothing unusual).

If you need to save computer power, you can use a slice:

```
1  myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
2  largest = myList[0]
3
4  for i in range(1, len(myList)):
5      if myList[i] > largest:
6          largest = myList[i]
7
8  print(largest)
9
10 myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
11 largest = myList[0]
12
13 for i in myList[1:]:
14     if i > largest:
15         largest = i
16
17 print(largest)
```

Console >_

```
17
17
17
```


If you need to save computer power, you can use a slice:

```
myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = myList[0]

for i in myList[1:]:
    if i > largest:
        largest = i

print(largest)
```


The question is: which of these two actions consumes more computer resources - just one comparison, or slicing almost all of a list's elements?

Console >_

```
17
17
17
```

Lists - some simple programs

Now let's find the location of a given element inside a list:

```
myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
toFind = 5
found = False

for i in range(len(myList)):
    found = myList[i] == toFind
    if found:
        break

if found:
    print("Element found at index", i)
else:
    print("absent")
```

Note:

- the target value is stored in the `toFind` variable;
- the current status of the search is stored in the `found` variable (`True` / `False`)
- when `found` becomes `True` , the `for` loop is exited.

Let's assume that you've chosen the following numbers in the lottery: `3` , `7` , `11` , `42` , `34` , `49` .

The numbers that have been drawn are: `5` , `11` , `9` , `42` , `3` , `49` .

The question is: how many numbers have you hit?

The program will give you the answer:

```
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
    if number in drawn:
        hits += 1

print(hits)
```

Note:

- the `drawn` list stores all the drawn numbers;
- the `bets` list stores your bets;
- the `hits` variable counts your hits.

The program output is: `4` .

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

Familiarize the student with:

- list indexing;
- utilizing the `in` and `not in` operators.

Scenario

Imagine a list - not very long, not very complicated, just a simple list containing some integer numbers. Some of these numbers may be repeated, and this is the clue. We don't want any repetitions. We want them to be removed.

Your task is to write a program which removes all the number repetitions from the list. The goal is to have a list in which all the numbers appear not more than once.

Note: assume that the source list is hard-coded inside the code - you don't have to enter it from the keyboard. Of course, you can improve the code and add a part that can carry out a conversation with the user and obtain all the data from her/him.

Hint: we encourage you to create a new list as a temporary work area - you don't need to update the list in situ.

We've provided no test data, as that would be too easy. You can use our skeleton instead.

```
myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
toFind = 5
found = False

for i in range(len(myList)):
    found = myList[i] == toFind
    if found:
        break

if found:
    print("Element found at index", i)
else:
    print("absent")

drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
    if number in drawn:
        hits += 1

print(hits)
```

```
Element found at index 4
4
```

```
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
    if number in drawn:
        hits += 1

print(hits)
```

```
Element found at index 4
4
```

```
myList = [1, 2, 4, 4, 1, 4, 2, 6, 2, 9]
newList = []
for number in myList:  # Browse all numbers from the source list.
    if number not in newList:  # If the number doesn't appear within the new list...
        newList.append(number)  # ...append it here.
myList = newList[:]  # Make a copy of newlist.
print("The list with unique elements only:")
print(myList)
```

```
  File "main.py", line 5
    myList = set(myList)
                 ^
SyntaxError: invalid syntax
The list with unique elements only:
{1, 2, 4, 6, 9}
The list with unique elements only:
[1, 2, 4, 6, 9]
```

Key takeaways

1. If you have a list `l1`, then the following assignment: `l2 = l1` does not make a copy of the `l1` list, but makes the variables `l1` and `l2` **point to one and the same list in memory**. For example:

```
vehiclesOne = ['car', 'bicycle', 'motor']
print(vehiclesOne) # outputs: ['car', 'bicycle', 'motor']

vehiclesTwo = vehiclesOne
del vehiclesOne[0] # deletes 'car'
print(vehiclesTwo) # outputs: ['bicycle', 'motor']
```

2. If you want to copy a list or part of the list, you can do it by performing **slicing**:

```
colors = ['red', 'green', 'orange']

copyWholeColors = colors[:] # copy the whole list
copyPartColors = colors[0:2] # copy part of the list
```

3. You can use **negative indices** to perform slices, too. For example:

```
sampleList = ["A", "B", "C", "D", "E"]
newList = sampleList[2:-1]
print(newList) # outputs: ['C', 'D']
```

4. The `start` and `end` parameters are **optional** when performing a slice: `list[start:end]`, e.g.:

```
myList = [1, 2, 3, 4, 5]
sliceOne = myList[2: ]
sliceTwo = myList[ :2]
sliceThree = myList[-2: ]

print(sliceOne) # outputs: [3, 4, 5]
print(sliceTwo) # outputs: [1, 2]
print(sliceThree) # outputs: [4, 5]
```

5. You can **delete slices** using the `del` instruction:

```
myList = [1, 2, 3, 4, 5]
del myList[0:2]
print(myList) # outputs: [3, 4, 5]

del myList[:]
print(myList) # deletes the list content, outputs: []
```

6. You can test if some items **exist in a list or not** using the keywords `in` and `not in`, e.g.:

```
myList = ["A", "B", 1, 2]

print("A" in myList) # outputs: True
print("C" not in myList) # outputs: True
print(2 not in myList) # outputs: False
```

Exercise 1

What is the output of the following snippet?

```
l1 = ["A", "B", "C"]
l2 = l1
l3 = l2

del l1[0]
del l2[0]

print(l3)
```

Check

```
['C']
```

Exercise 2

What is the output of the following snippet?

```
l1 = ["A", "B", "C"]
l2 = l1
l3 = l2

del l1[0]
del l2

print(l3)
```

Check

```
['B', 'C']
```

Exercise 3

What is the output of the following snippet?

```
l1 = ["A", "B", "C"]
l2 = l1
l3 = l2

del l1[0]
del l2[:]

print(l3)
```

Check

```
[]
```

Exercise 4

What is the output of the following snippet?

```
l1 = ["A", "B", "C"]
l2 = l1[:]
l3 = l2[:]

del l1[0]
del l2[0]

print(l3)
```

Check

Exercise 5

Insert `in` or `not in` instead of `???` so that the code outputs the expected result.

```
myList = [1, 2, "in", True, "ABC"]

print(1 ??? myList) # outputs True
print("A" ??? myList) # outputs True
print(3 ??? myList) # outputs True
print(False ??? myList) # outputs False
```

Check

```
myList = [1, 2, "in", True, "ABC"]

print(1 in myList) # outputs True
print("A" not in myList) # outputs True
print(3 not in myList) # outputs True
print(False in myList) # outputs False
```

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 right 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 = []

for i in range(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_PAWN for i in range(8)]
```

```
row = []

for i in range(8):
    row.append("WHITE_PAWN")


print(row)
WHITE_PAWN = 2
row = [WHITE_PAWN for i in range(8)]
print(row)


squares = [x ** 2 for x in range(10)]
print(squares)

twos = [2 ** x for x in range(10)]
print(twos)

```

Console >_

```
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

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 ** 2 for x in range(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 ** i for i in range(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 = [x for x in squares if x % 2 != 0 ]
```

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

Console >_

```
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
['WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAWN', 'WHITE_PAW
[2, 2, 2, 2, 2, 2, 2, 2]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```


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 = []

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

The inner part creates a row, and the outer part builds a list of rows.

```
board = []

for i in range(8):
    row = [i for i in range(8)]
    board.append(row)
print(board)

board = [[i for i in range(8)] for j in range(8)]
print(board)
```

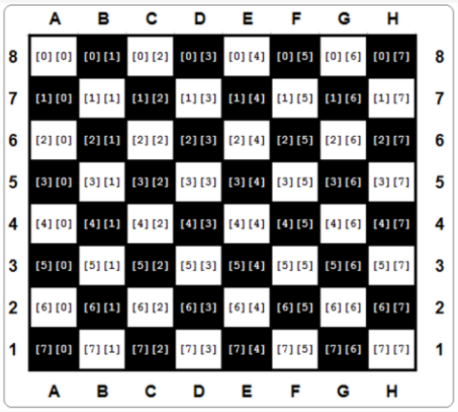
Console >_

```
[[0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4,
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    board = [[EMPTY for i in range(8)] for j in range(8)]
  File "main.py", line 8, in <listcomp>
    board = [[EMPTY for i in range(8)] for j in range(8)]
  File "main.py", line 8, in <listcomp>
    board = [[EMPTY for i in range(8)] for j in range(8)]
NameError: name 'EMPTY' is not defined
[[0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4,
[[0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4,
```

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
```

And now - experiment with the code in the editor.

```
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)
print(end="")
```

Console >_

```
[['ROOK', '-', '-', '-', '-', '-', '-', 'ROOK'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-
[['ROOK', '-', '-', '-', '-', '-', '-', 'ROOK'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-
[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '
[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '

[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '


[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '
```

Console >_

```
[['ROOK', '-', '-', '-', '-', '-', '-', 'ROOK'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-
[['ROOK', '-', '-', '-', '-', '-', '-', 'ROOK'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-
[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '
[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '

[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '


[['ROOK', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '
```

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 = [[0.0 for h in range(24)] for d in range(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:

```
for day in temps:
    total += day[11]

average = total / 31

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

temps = [[0.0 for h in range(24)] for d in range(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)
```

Console >_

```
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
Average temperature at noon: 0.0
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
Average temperature at noon: 0.0
The highest temperature was: 0.0
```

```
temps = [[0.0 for h in range(24)] for d in range(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.

```
temps = [[0.0 for h in range(24)] for d in range(31)]
print(temps)

#
# 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)

temps = [[0.0 for h in range(24)] for d in range(31)]
#
# the matrix is magically updated here
```

Console >_

```
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
Average temperature at noon: 0.0
```

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

```
temps = [[0.0 for h in range(24)] for d in range(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 °C:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# the matrix is magically updated here
#

hotDays = 0

for day in temps:
    if day[11] > 20.0:
        hotDays += 1

print(hotDays, "days were hot.")
```

```
temps = [[0.0 for h in range(24)] for d in range(31)]
print(temps)

#
# 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)

temps = [[0.0 for h in range(24)] for d in range(31)]
#
# the matrix is magically updated here
```

Console >_

```
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
Average temperature at noon: 0.0
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
Average temperature at noon: 0.0
The highest temperature was: 0.0
```

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 = [[[False for r in range(20)] for f in range(15)] for t in range(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

for roomNumber in range(20):
    if not rooms[2][14][roomNumber]:
        vacancy += 1
```

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

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]
print(rooms)

vacancy = 0

for roomNumber in range(20):
    if not rooms[2][14][roomNumber]:
        vacancy += 1
```

Console >_

```
False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False], [Fal
False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False, False], [Fal
```

Key takeaways

1. **List comprehension** allows you to create new lists from existing ones in a concise and elegant way. The syntax of a list comprehension looks as follows:

```
[expression for element in list if conditional]
```

which is actually an equivalent of the following code:

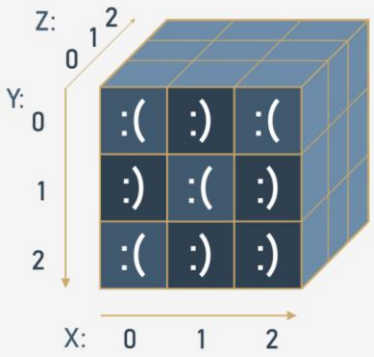
```
for element in list:
    if conditional:
        expression
```

Here's an example of a list comprehension - the code creates a five-element list filled with with the first five natural numbers raised to the power of 3:

```
cubed = [num ** 3 for num in range(5)]
print(cubed) # outputs: [0, 1, 8, 27, 64]
```

2. You can use **nested lists** in Python to create **matrices** (i.e., two-dimensional lists). For example:

3. You can nest as many lists in lists as you want, and therefore create n-dimensional lists, e.g., three-, four- or even sixty-four-dimensional arrays. For example:




```
# Cube - a three-dimensional array (3x3x3)
```

2. You can use **nested lists** in Python to create **matrices** (i.e., two-dimensional lists). For example:


```
# Cube - a three-dimensional array (3x3x3)

cube = [[[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':(', 'x', 'x']],

        [[':)', 'x', 'x'],
         [':(', 'x', 'x'],
         [':)', 'x', 'x']],

        [[':(', 'x', 'x'],
         [':)', 'x', 'x'],
         [':)', 'x', 'x']]]

print(cube)
print(cube[0][0][0]) # outputs: ':('
print(cube[2][2][0]) # outputs: ':)'
```

```
# A four-column/four-row table - a two dimensional array (4x4)

table = [[":(", ":)", ":(", ":)"],
         [":)", ":(", ":)", ":)"],
         [":(", ":)", ":)", ":("],
         [":)", ":)", ":)", ":("]]

print(table)
print(table[0][0]) # outputs: ':('
print(table[0][3]) # outputs: ':)'
```

Congratulations! You have completed Module 3.

Well done! You've reached the end of Module 3 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 3:

- Boolean values to compare different values and control the execution paths using the `if` and `if-else` instructions;
- the utilization of loops (`while` and `for`) and how to control their behavior using the `break` and `continue` instructions;
- the difference between logical and bitwise operations;
- the concept of lists and list processing, including the iteration provided by the `for` loop, and slicing;
- the idea of multi-dimensional arrays.

You are now ready to take the module quiz and attempt the final challenge: Module 3 Test, which will help you gauge what you've learned so far.

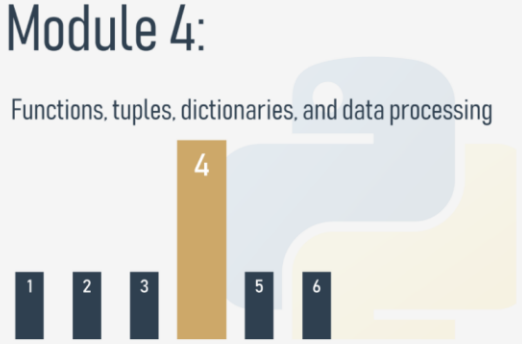
MODULE: 4

Load Module 4 - Functions, tuples, dictionaries, and data processing in a new window

Programming Essentials in Python: Module 4

In this module, you will learn about:

- defining and using functions;
- different ways of passing arguments;
- name scopes;
- tuples and dictionaries;
- data processing.



Why do we need functions?

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

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

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

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

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

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

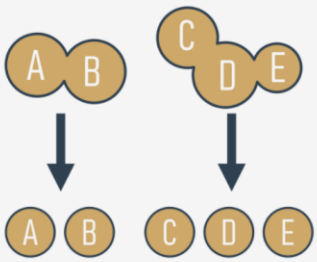
We can now define the first condition which can help you decide when to start writing your own functions: **if a particular fragment of the code begins to appear in more than one place, consider the possibility of isolating it in the form of a function** invoked from the points where the original code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

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

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

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

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

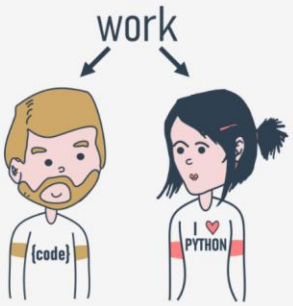


We can now state the second condition: **if a piece of code becomes so large that reading and understating it may cause a problem, consider dividing it into separate, smaller problems, and implement each of them in the form of a separate function**.

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

Decomposition

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



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

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

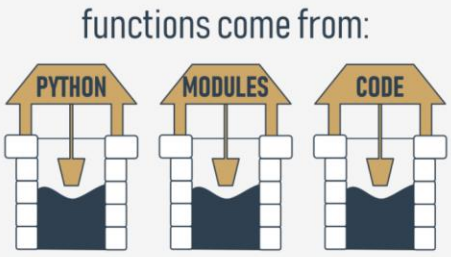
Each of them writes a clearly defined and described set of functions, which when **combined into the module** (we'll tell you about this a bit later) will give the final product.

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

Where do the functions come from?

In general, functions come from at least three places:

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



Your first function

Take a look at the snippet in the editor.

It's rather simple, but we only want it to be an example of **transforming a repeating part of a code into a function**.

The messages sent to the console by the `print()` function are always the same. Of course, there's nothing really bad in such a code, but try to imagine what you would have to do if your boss asked you to change the message to make it more polite, e.g., to start it with the phrase `"Please,"`.

It seems that you'd have to spend some time changing all the occurrences of the message (you'd use a clipboard, of course, but it wouldn't make your life much easier). It's obvious that you'd probably make some mistakes during the amendment process, and you (and your boss) would get a bit frustrated.

Is it possible to separate such a *repeatable* part of the code, name it and make it reusable? It would mean that a **change made once in one place would be propagated to all the places where it's used**.

Of course, such a code should work only when it's explicitly launched.

Yes, it's possible. This is exactly what functions are for.

```
print("Enter a value: ")
a = int(input())

print("Enter a value: ")
b = int(input())

print("Enter a value: ")
c = int(input())
```

Console >_

```
Enter a value:
1
Enter a value:
2
Enter a value:
3
```

Your first function

How do you make such a function?

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

This is what the simplest function definition looks like:

```
def functionName():
    functionBody
```

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

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

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

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

Our code contains the function definition now:

```
# print("Enter a value: ")
# a = int(input())

# print("Enter a value: ")
# b = int(input())

# print("Enter a value: ")
# c = int(input())
def message():
    return input("Enter a number: ")

print("Square of the number: ", int(message()) ** 2)
```

Console >_

```
Enter a number: 5
Traceback (most recent call last):
  File "main.py", line 13, in <module>
    print(message() ** 2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Enter a number: 6
36
```

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

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

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

When you run it, you see the following output:

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

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

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

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

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

The output looks different now:

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

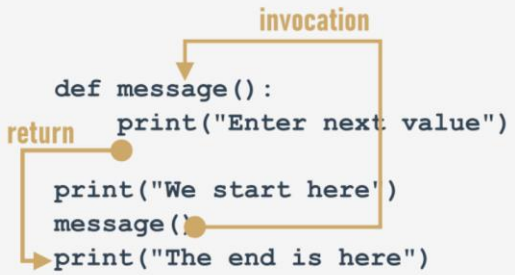
```
# print("Enter a value: ")
# a = int(input())

# print("Enter a value: ")
# b = int(input())

# print("Enter a value: ")
# c = int(input())
def message():
    return input("Enter a number: ")

print("Square of the number: ", int(message()) ** 2)
```

Console >_

```
Enter a number: 5
Traceback (most recent call last):
  File "main.py", line 13, in <module>
    print(message() ** 2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Enter a number: 6
36
Enter a number: 2
Square of the number:  4
```

How functions work

Look at the picture below:



It tries to show you the whole process:

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

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

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

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

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

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

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

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

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

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

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

The second catch sounds a little simpler:

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

The following snippet is erroneous:

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

message = 1
```

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

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

Look at the snippet:

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

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

message()

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

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

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

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

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

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

Open the sandbox, and try to do it yourself.

Key takeaways

1. A **function** is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

2. There are at least four basic types of functions in Python:

- **built-in functions** which are an integral part of Python (such as the `print()` function). You can see a complete list of Python built-in functions at https://docs.python.org/3/library/functions.html.
- the ones that come from **pre-installed modules** (you'll learn about them in *Module 5* of this course)
- **user-defined functions** which are written by users for users - you can write your own functions and use them freely in your code,
- the `lambda` functions (you'll learn about them in *Module 6* of this course.)

3. You can define your own function using the `def` keyword and the following syntax:

```
def yourFunction(optional parameters):
    # the body of the function
```

You can define a function which doesn't take any arguments, e.g.:

```
def message():    # defining a function
    print("Hello")    # body of the function

message()    # calling the function
```

Exercise 1

The `input()` function is an example of a:

a) user-defined function
b) built-in function

Check

Exercise 2

What happens when you try to invoke a function before you define it? Example:

```
hi()

def hi():
    print("hi!")
```

Check

An exception is thrown (the `NameError` exception to be more precise)

Exercise 3

What will happen when you run the code below?

```
def message():    # defining a function
    print("Hello")    # body of the function

message()    # calling the function
```

You can define a function which takes arguments, too, just like the one-parameter function below:

```
def hello(name):    # defining a function
    print("Hello,", name)    # body of the function


name = input("Enter your name: ")

hello(name)    # calling the function
```

We'll tell you more about parametrized functions in the next section. Don't worry.

Exercise 3

What will happen when you run the code below?

```
def hi():
    print("hi")


hi(5)
```

Check

An exception will be thrown (the `TypeError` exception to be more precise) - the `hi()` function doesn't take any arguments

Parametrized 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

Try to run the code in the editor.

This is what you'll see in the console:

```
TypeError: message() missing 1 required positional argument: 'number'
```

This looks better, for sure:

```
def message(number):
    print("Enter a number:", number)

message(1)
```

Moreover, it behaves better. The code will produce the following output:

```
Enter a number: 1
```

Can you see how it works? The value of the argument used during invocation (`1`) has been passed into the function, setting the initial value of the parameter named `number`.

We have to make you sensitive to one important circumstance.

It's legal, and possible, to have a **variable named the same as a function's parameter**.

The snippet illustrates the phenomenon:

```
1 def message(number):
2     print("Enter a number:", number)
3
4 message("python_function")
```

Console >_

```
Enter a number: python_function
```

```
def message(number):
    print("Enter a number:", number)

number = 1234
message(1)
print(number)
```

Console >_

```
Enter a number: python_function
Enter a number: python_function
Enter a number: 5385687
```

A situation like this activates a mechanism called **shadowing**:

- parameter `x` shadows any variable of the same name, but...
- ... only inside the function defining the parameter.

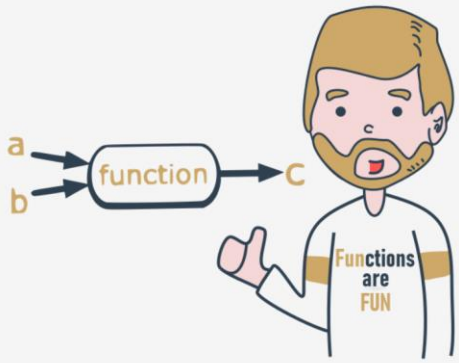
The parameter named `number` is a completely different entity from the variable named `number`.

This means that the snippet above will produce the following output:

```
Enter a number: 1
1234
```

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)
```

```
def message(what, number):
    print("Enter", what, "number", number)

# invoke function
message('@', 4)
```

Console >_

```
Enter @ number 4
```

This also means that invoking the function will require **two arguments**.

The first new parameter is intended to carry the name of the desired value.

Here it is:

```
def message(what, number):
    print("Enter", what, "number", number)

message("telephone", 11)
message("price", 5)
message("number", "number")
```

This is the output you're about to see:

```
Enter telephone number 11
Enter price number 5
Enter number number number
```

Run the code, modify it, add more parameters, and see how this affects the output.

Console >_

```
Enter @ number 4
Enter @ number 4
Enter customerNO number 101
```

Positional parameter passing

A technique which assigns the i^{th} (first, second, and so on) argument to the i^{th} (first, second, and so on) function parameter is called **positional parameter passing**, while arguments passed in this way are named **positional arguments**.

You've used it already, but Python can offer a lot more. We're going to tell you about it now.

```
def myFunction(a, b, c):
    print(a, b, c)

myFunction(1, 2, 3)
```

Note: positional parameter passing is intuitively used by people in many social occasions. For example, it may be generally accepted that when we introduce ourselves we mention our first name(s) before our last name, e.g., "My name's John Doe."

Incidentally, Hungarians do it in reverse order.

Let's implement that social custom in Python. The following function will be responsible for introducing somebody:

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction("Luke", "Skywalker")
introduction("Jesse", "Quick")
introduction("Clark", "Kent")
```

Can you guess the output? Run the code and find out if you were right.

Now imagine that the same function is being used in Hungary. In this case, the code would look like this:

```
def myFunction(a, b, c):
    print(a, b, c)

myFunction(1, 2, 3)

def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction("Luke", "Skywalker")
introduction("Jesse", "Quick")
introduction("Clark", "Kent")

def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction("Skywalker", "Luke")
introduction("Quick", "Jesse")
introduction("Kent", "Clark")
```

Console >_

```
1 2 3
1 2 3
Hello, my name is Luke Skywalker
Hello, my name is Jesse Quick
Hello, my name is Clark Kent
1 2 3
Hello, my name is Luke Skywalker
Hello, my name is Jesse Quick
Hello, my name is Clark Kent
Hello, my name is Skywalker Luke
Hello, my name is Quick Jesse
Hello, my name is Kent Clark
```

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction("Skywalker", "Luke")
introduction("Quick", "Jesse")
introduction("Kent", "Clark")
```

The output will look different. Can you guess it?

Run the code to see if you were right here, too. Are you surprised?

Can you make the function more culture-independent?

Console >_

```
1 2 3
1 2 3
Hello, my name is Luke Skywalker
Hello, my name is Jesse Quick
Hello, my name is Clark Kent
1 2 3
Hello, my name is Luke Skywalker
Hello, my name is Jesse Quick
Hello, my name is Clark Kent
Hello, my name is Skywalker Luke
Hello, my name is Quick Jesse
Hello, my name is 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:

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction(firstName = "James", lastName = "Bond")
introduction(lastName = "Skywalker", firstName = "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.

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction(firstName = "James", lastName = "Bond")
introduction(lastName = "Skywalker", firstName = "Luke")

def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction(surname="Skywalker", firstName="Luke")
```

Console >_

```
Hello, my name is James Bond
Hello, my name is Luke Skywalker
Hello, my name is James Bond
Hello, my name is Luke Skywalker
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    introduction(surname="Skywalker", firstName="Luke")
TypeError: introduction() got an unexpected keyword argument 'surname'
```

Of course, you **mustn't use a non-existent parameter name**.

The following snippet will cause a runtime error:

```
def introduction(firstName, lastName):
    print("Hello, my name is", firstName, lastName)

introduction(surname="Skywalker", firstName="Luke")
```

This is what Python will tell you:

```
TypeError: introduction() got an unexpected keyword argument 'surname'
```

Try it out yourself.

Console >_

```
Hello, my name is James Bond
Hello, my name is Luke Skywalker
Hello, my name is James Bond
Hello, my name is Luke Skywalker
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    introduction(surname="Skywalker", firstName="Luke")
TypeError: introduction() got an unexpected keyword argument 'surname'
```

Mixing positional and keyword arguments

You can mix both fashions if you want - there is only one unbreakable rule: you have to put **positional arguments before keyword arguments**.

If you think for a moment, you'll certainly guess why.

To show you how it works, we'll use the following simple three-parameter function:

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)
```

Its purpose is to evaluate and present the sum of all its arguments.

The function, when invoked in the following way:

```
adding(1, 2, 3)
```

will output:

```
1 + 2 + 3 = 6
```

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:

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)

# call the adding function here
adding(4, 3, c = 2)
```

Console >_

```
4 + 3 + 2 = 9
```

```
2 + 3 + 1 = 6
```

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
```

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)

# call the adding function here
adding(4, 3, c = 2)
```

Console >_

```
4 + 3 + 2 = 9
```

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'
```

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?

Console >_

```
4 + 3 + 2 = 9
```

Parametrized functions - more details

It happens at times that a particular parameter's values are in use more often than others. Such arguments may have their **default (predefined) values** taken into consideration when their corresponding arguments have been omitted.

They say that the most popular English last name is *Smith*. Let's try to take this into account.

The default parameter's value is set using clear and pictorial syntax:

```
def introduction(firstName, lastName="Smith"):
    print("Hello, my name is", fistName, lastName)
```

You only have to extend the parameter's name with the `=` sign, followed by the default value.

Let's invoke the function as usual:

```
introduction("James", "Doe")
```

Can you guess the output of the program? Run it and check if you were right.

And? Everything looks the same, but when you invoke the function in a way that looks a bit suspicious at first sight, like this:

```
introduction("Henry")
```

```
1  def introduction(firstName, lastName="Smith"):
2      print("Hello, my name is", firstName, lastName)
3
4  # call the function here
5  introduction(firstName="William")
6
```

Console >_

```
Hello, my name is William Smith
```

or this:

```
introduction(firstName="William")
```

there will be no error, and both invocations will succeed, while the console will show the following output:

```
Hello, my name is Henry Smith
Hello, my name is William Smith
```

Test it.

Console >_

```
Hello, my name is William Smith
```

You can go further if it's useful. Both parameters have their default values now, look at the code below:

```
def introduction(firstName="John", lastName="Smith"):
    print("Hello, my name is", firstName, lastName)
```

This makes the following invocation absolutely valid:

```
introduction()
```

And this is the expected output:

```
Hello, my name is John Smith
```

Console >_

```
Hello, my name is William Smith
```

If you use one keyword argument, the remaining one will take the default value:

```
introduction(lastName="Hopkins")
```

The output is:

```
Hello, my name is John Hopkins
```

Test it.

Congratulations - you have just learned the basic ways of communicating with functions.

Key takeaways

1. You can pass information to functions by using parameters. Your functions can have as many parameters as you need.

An example of a one-parameter function:

```
def hi(name):
    print("Hi,", name)

hi("Greg")
```

An example of a two-parameter function:

```
def hiAll(name1, name2):
    print("Hi,", name2)
    print("Hi,", name1)

hiAll("Sebastian", "Konrad")
```

An example of a three-parameter function:

```
def address(street, city, postalCode):
    print("Your address is:", street, "St.,", city, postalCode)

s = input("Street: ")
pC = input("Postal Code: ")
c = input("City: ")

address(s, c, pC)
```

2. You can pass arguments to a function using the following techniques:

- **positional argument passing** in which the order of arguments passed matters (Ex. 1),
- **keyword (named) argument passing** in which the order of arguments passed doesn't matter (Ex. 2),
- a mix of positional and keyword argument passing (Ex. 3).

```
Ex. 1
def subtra(a, b):
    print(a - b)

subtra(5, 2)    # outputs: 3
subtra(2, 5)    # outputs: -3


Ex. 2
def subtra(a, b):
    print(a - b)

subtra(a=5, b=2)    # outputs: 3
subtra(b=2, a=5)    # outputs: 3

Ex. 3
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # outputs: 3
subtra(5, 2)    # outputs: 3
```

It's important to remember that **positional arguments mustn't follow keyword arguments**. That's why if you try to run the following snippet:

```
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # outputs: 3
subtra(a=5, 2)    # Syntax Error
```

Python will not let you do it by signalling a `SyntaxError`.

3. You can use the keyword argument passing technique to **pre-define** a value for a given argument:

```
def name(firstN, lastN="Smith"):
    print(firstN, lastN)

name("Andy")    # outputs: Andy Smith
name("Betty", "Johnson")    # outputs: Betty Johnson (the keyword argument replaced by "Johnson")
```

Exercise 1

What is the output of the following snippet?

```
def intro(a="James Bond", b="Bond"):
    print("My name is", b + ".", a + ".")

intro()
```

Check

```
My name is Bond. James Bond.
```

Exercise 2

What is the output of the following snippet?

```
def intro(a="James Bond", b="Bond"):
    print("My name is", b + ".", a + ".")

intro(b="Sean Connery")
```

Check

```
My name is Sean Connery. James Bond.
```

Exercise 3

What is the output of the following snippet?

```
def intro(a, b="Bond"):
    print("My name is", b + ".", a + ".")

intro("Susan")
```

Check

```
My name is Bond. Susan.
```

Exercise 4

What is the output of the following snippet?

```
def sum(a, b=2, c):
    print(a + b + c)

sum(a=1, c=3)
```

Check

`SyntaxError` - a non-default argument (`c`) follows a default argument (`b=2`)

Effects and results: the `return` instruction

All the previously presented functions have some kind of effect - they produce some text and send it to the console.

Of course, functions - like their mathematical siblings - may have results.

To get **functions to return a value** (but not only for this purpose) you use the `return` instruction.

This word gives you a full picture of its capabilities. Note: it's a Python **keyword**.

The `return` instruction has **two different variants** - let's consider them separately.

`return` without an expression

The first consists of the keyword itself, without anything following it.

When used inside a function, it causes the **immediate termination of the function's execution, and an instant return (hence the name) to the point of invocation**.

Note: if a function is not intended to produce a result, **using the `return` instruction is not obligatory** - it will be executed implicitly at the end of the function.

Anyway, you can use it to **terminate a function's activities on demand**, before the control reaches the function's last line.

Let's consider the following function:

```
def happy_new_year(wishes = True):
    print("Three...")
    print("Two...")
    print("One...")
    if not wishes:
        return

    print("Happy New Year!")
```

When invoked without any arguments:

```
happy_new_year()
```

The function causes a little noise - the output will look like this:

`return` with an expression

The second `return` variant is **extended with an expression**:

```
def function():
    return expression
```

There are two consequences of using it:

- it causes the **immediate termination of the function's execution** (nothing new compared to the first variant)
- moreover, the function will **evaluate the expression's value and will return (hence the name once again) it as the function's result**.

Yes, we already know - this example isn't really sophisticated:

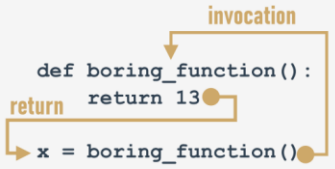
```
def boring_function():
    return 123

x = boring_function()

print("The boring_function has returned its result. It's:", x)
```

The snippet writes the following text to the console:

```
The boring_function has returned its result. It's: 123
```

Let's investigate it for a while.

Analyze the figure below:



```
Three...
Two...
One...
Happy New Year!
```

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...
```

The `return` instruction, enriched with the expression (the expression is very simple here), "transports" the expression's value to the place where the function has been invoked.

The result may be freely used here, e.g., to be assigned to a variable.

It may also be completely ignored and lost without a trace.

Note, we're not being too polite here - the function returns a value, and we ignore it (we don't use it in any way):

```
def boring_function():
    print("'Boredom Mode' ON.")
    return 123

print("This lesson is interesting!)
boring_function()
print("This lesson is boring...")
```

The program produces the following output:

```
This lesson is interesting!
'Boredom Mode' ON.
This lesson is boring...
```

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.

A few words about `None`

Let us introduce you to a very curious value (to be honest, a none value) named `None`.

Its data doesn't represent any reasonable value - actually, it's not a value at all; hence, it **mustn't take part in any expressions**.

For example, a snippet like this:

```
print(None + 2)
```

will cause a runtime error, described by the following diagnostic message:

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Note: `None` is a **keyword**.

There are only two kinds of circumstances when `None` can be safely used:

- when you **assign it to a variable** (or return it as a **function's result**)
- when you **compare it with a variable** to diagnose its internal state.

Just like here:

```
value = None
if value is None:
    print("Sorry, you don't carry any value")
```

Don't forget this: if a function doesn't return a certain value using a `return` expression clause, it is assumed that it **implicitly returns** `None`.

Let's test it.

```
value = None
if value is None:
    print("Sorry, you don't carry any value")
```

Console >_

```
Sorry, you don't carry any value
```

A few words about `None`: continued

Take a look at the code in the editor.

It's obvious that the `strangeFunction` function returns `True` when its argument is even.

What does it return otherwise?

We can use the following code to check it:

```
print(strangeFunction(2))
print(strangeFunction(1))
```

This is what we see in the console:

```
True
None
```

Don't be surprised next time you see `None` as a function result - it may be the symptom of a subtle mistake inside the function.

```
def strangeFunction(n):
    if(n % 2 == 0):
        return True

print(strangeFunction(2))
print(strangeFunction(1))
```

Console >_

```
True
None
```

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:

```
def list_sum(lst):
    s = 0

    for elem in lst:
        s += elem

    return s
```

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))
```

Python's response will be unequivocal:

```
def list_sum(lst):
    s = 0

    for elem in lst:
        s += elem

    return s

print(list_sum([5, 4, 3]))
print(list_sum(5))
```

Console >_

```
12
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    print(list_sum(5))
  File "main.py", line 4, in list_sum
    for elem in lst:
TypeError: 'int' object is not iterable
```

Effects and results: lists and functions - continued

The second question is: **may a list be a function result?**

Yes, of course! Any entity recognizable by Python can be a function result.

Look at the code in the editor. The program's output will be like this:

```
[4, 3, 2, 1, 0]
```

Now you can write functions with and without results.

Let's dive a little deeper into the issues connected with variables in functions. This is essential for creating effective and safe functions.

```
def strangeListFunction(n):
    strangeList = []

    for i in range(0, n):
        strangeList.insert(0, i)

    return strangeList

print(strangeListFunction(5))
```

Console >_

```
[4, 3, 2, 1, 0]
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

Familiarize the student with:

- projecting and writing parameterized functions;
- utilizing the `return` statement;
- testing the functions.

Scenario

Your task is to write and test a function which takes one argument (a year) and returns `True` if the year is a *leap year*, or `False` otherwise.

The seed of the function is already sown in the skeleton code in the editor.

Note: we've also prepared a short testing code, which you can use to test your function.

The code uses two lists - one with the test data, and the other containing the expected results. The code will tell you if any of your results are invalid.

```
def isYearLeap(year):
    #
    # put your code here
    #
    if year % 4 == 0 and year % 100 == 0 and year % 400 == 0:
        return True
    else:
        return False

testData = [1900, 2000, 2016, 1987]
testResults = [False, True, True, False]
for i in range(len(testData)):
    yr = testData[i]
    print(yr,"->",end="")
    result = isYearLeap(yr)
    if result == testResults[i]:
        print("OK")
    else:
        print("Failed")
```

Console >_

```
1900 ->OK
2000 ->OK
2016 ->Failed
1987 ->OK
```

LAB

Estimated time

15-20 minutes

Level of difficulty

Medium

Prerequisites

LAB 4.1.3.6

Objectives

Familiarize the student with:

- projecting and writing parameterized functions;
- utilizing the `return` statement;
- utilizing the student's own functions.

Scenario

Your task is to write and test a function which takes two arguments (a year and a month) and returns the number of days for the given month/year pair (while only February is sensitive to the `year` value, your function should be universal).

The initial part of the function is ready. Now, convince the function to return `None` if its arguments don't make sense.

Of course, you can (and should) use the previously written and tested function (LAB 4.1.3.6). It may be very helpful. We encourage you to use a list filled with the months' lengths. You can create it inside the function - this trick will significantly shorten the code.

We've prepared a testing code. Expand it to include more test cases.

```
def isYearLeap(year):
    if year % 4 != 0:
        return False
    elif year % 100 != 0:
        return True
    elif year % 400 != 0:
        return False
    else:
        return True

def daysInMonth(year,month):
    if year < 1582 or month < 1 or month > 12:
        return None
    days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    res  = days[month - 1]
    if month == 2 and isYearLeap(year):
        res = 29
    return res

testyears = [1900, 2000, 2016, 1987]
testmonths = [ 2, 2, 1, 11]
testresults = [28, 29, 31, 30]
for i in range(len(testyears)):
    yr = testyears[i]
    mo = testmonths[i]
    print(yr,mo,"-> ",end="")
    result = daysInMonth(yr, mo)
    if result == testresults[i]:
        print("OK")
    else:
```

Console >_

```
1900 2 -> OK
2000 2 -> OK
2016 1 -> OK
1987 11 -> OK
```

LAB

Estimated time

20-30 minutes

Level of difficulty

Medium

Prerequisites

LAB 4.1.3.6
LAB 4.1.3.7

Objectives

Familiarize the student with:

- projecting and writing parameterized functions;
- utilizing the `return` statement;
- building a set of utility functions;
- utilizing the student's own functions.

Scenario

Your task is to write and test a function which takes three arguments (a year, a month, and a day of the month) and returns the corresponding day of the year, or returns `None` if any of the arguments is invalid.

Use the previously written and tested functions. Add some test cases to the code. This test is only a beginning.

```
def isYearLeap(year):
    if year % 4 != 0:
        return False
    elif year % 100 != 0:
        return True
    elif year % 400 != 0:
        return False
    else:
        return True

def daysInMonth(year, month):
    if year < 1582 or month < 1 or month > 12:
        return None
    days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    res  = days[month - 1]
    if month == 2 and isYearLeap(year):
        res = 29
    return res

def dayOfYear(year, month, day):
    days = 0
    for m in range(1, month):
        md = daysInMonth(year, m)
        if md == None:
            return None
        days += md
    md = daysInMonth(year, month)
    if day >= 1 and day <= md:
        return days + day
    else:
        return None

print(dayOfYear(2000, 12, 31))
```

Console >_

```
366
```

LAB

Estimated time

15-20 minutes

Level of difficulty

Medium

Objectives

- familiarizing the student with classic notions and algorithms;
- improving the student's skills in defining and using functions.

Scenario

*A natural number is **prime** if it is greater than 1 and has no divisors other than 1 and itself.*

Complicated? Not at all. For example, 8 isn't a prime number, as you can divide it by 2 and 4 (we can't use divisors equal to 1 and 8, as the definition prohibits this).

On the other hand, 7 is a prime number, as we can't find any legal divisors for it.

Your task is to write a function checking whether a number is prime or not.

The function:

- is called `isPrime`;
- takes one argument (the value to check)
- returns `True` if the argument is a prime number, and `False` otherwise.

Hint: try to divide the argument by all subsequent values (starting from 2) and check the remainder - if it's zero, your number cannot be a prime; think carefully about when you should stop the process.

```
def isPrime(num):
    for i in range(2, int(1 + num ** 0.5)):
        if num % i == 0:
            return False
    return True

for i in range(1, 20):
    if isPrime(i + 1):
        print(i + 1, end=" ")
print()
```

Console >_

```
2 3 5 7 11 13 17 19
```

If you need to know the square root of any value, you can utilize the `**` operator. Remember: the square root of x is the same as $x^{0.5}$

Complete the code in the editor.

Run your code and check whether your output is the same as ours.

Test Data

Expected output:

```
2 3 5 7 11 13 17 19
```

Console >_

```
2 3 5 7 11 13 17 19
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

- improving the student's skills in defining, using and testing functions.

Scenario

A car's fuel consumption may be expressed in many different ways. For example, in Europe, it is shown as the amount of fuel consumed per 100 kilometers.

In the USA, it is shown as the number of miles traveled by a car using one gallon of fuel.

Your task is to write a pair of functions converting l/100km into mpg, and vice versa.

The functions:

- are named `l100kmtompg` and `mpgtol100km` respectively;
- take one argument (the value corresponding to their names)

```
# 1 American mile = 1609.344 metres
# 1 American gallon = 3.785411784 litres

def l100kmtompg(litres):
    gallons = litres / 3.785411784
    miles = 100 * 1000 / 1609.344
    return miles / gallons

def mpgtol100km(miles):
    km100 = miles * 1609.344 / 1000 / 100
    litres = 3.785411784
    return litres / km100

print(l100kmtompg(3.9))
print(l100kmtompg(7.5))
print(l100kmtompg(10.))
print(mpgtol100km(60.3))
print(mpgtol100km(31.4))
print(mpgtol100km(23.5))
```

Console >_

```
60.311431623931624
31.361944444444444
23.521458333333333
3.9007393587617467
7.490910297239916
10.009131205673757
```

Complete the code in the editor.

Run your code and check whether your output is the same as ours.

Here is some information to help you:

- 1 American mile = 1609.344 metres;
- 1 American gallon = 3.785411784 litres.

Test Data

Expected output:

```
60.31143162393162
31.36194444444444
23.52145833333333
3.9007393587617467
7.490910297239916
10.009131205673757
```

Console >_

```
60.31143162393162
31.36194444444444
23.52145833333333
3.9007393587617467
7.490910297239916
10.009131205673757
```

Key takeaways

1. You can use the `return` keyword to tell a function to return some value. The `return` statement exits the function, e.g.:

```
def multiply(a, b):
    return a * b

print(multiply(3, 4))    # outputs: 12

def multiply(a, b):
    return

print(multiply(3, 4))    # outputs: None
```

2. The result of a function can be easily assigned to a variable, e.g.:

```
def wishes():
    return "Happy Birthday!"

w = wishes()

print(w)    # outputs: Happy Birthday!
```

Look at the difference in output in the following two examples:

```
# Example 1
def wishes():
    print("My Wishes")
    return "Happy Birthday"

wishes()    # outputs: My Wishes


# Example 2
def wishes():
    print("My Wishes")
    return "Happy Birthday"

print(wishes())    # outputs: My Wishes
                   #          Happy Birthday
```

Exercise 1

What is the output of the following snippet?

```
def hi():
    return
    print("Hi!")

hi()
```

Check

the function will return an implicit `None` value

Exercise 2

What is the output of the following snippet?

```
def isInt(data):
    if type(data) == int:
        return True
    elif type(data) == float:
        return False

print(isInt(5))
print(isInt(5.0))
print(isInt("5"))
```

Check

```
True
False
None
```

Exercise 3

What is the output of the following snippet?

3. You can use a list as a function's argument, e.g.:

```
def hiEverybody(myList):
    for name in myList:
        print("Hi,", name)

hiEverybody(["Adam", "John", "Lucy"])
```

4. A list can be a function result, too, e.g.:

```
def createList(n):
    myList = []
    for i in range(n):
        myList.append(i)
    return myList

print(createList(5))
```

```
def evenNumLst(ran):
    lst = []
    for num in range(ran):
        if num % 2 == 0:
            lst.append(num)
    return lst

print(evenNumLst(11))
```

Check

```
[0, 2, 4, 6, 8, 10]
```

Exercise 4

What is the output of the following snippet?

```
def listUpdater(lst):
    updList = []
    for elem in lst:
        elem **= 2
        updList.append(elem)
    return updList

l = [1, 2, 3, 4, 5]
print(listUpdater(l))
```

Check

```
[1, 4, 9, 16, 25]
```

Functions and scopes

Let's start with a definition:

The **scope of a name** (e.g., a variable name) is the part of a code where the name is properly recognizable.

For example, the scope of a function's parameter is the function itself. The parameter is inaccessible outside the function.

Let's check it. Look at the code in the editor. What will happen when you run it?

The program will fail when run. The error message will read:

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

This is to be expected.

We're going to conduct some experiments with you to show you how Python constructs scopes, and how you can use its habits to your benefit.

```
1 def scopeTest():
2     x = 123
3
4 scopeTest()
5 print(x)
```

Console >_

```
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    print(x)
NameError: name 'x' is not defined
```

Functions and scopes: continued

Let's start by checking whether or not a variable created outside any function is visible inside the functions. In other words, does a variable's name propagate into a function's body?

Look at the code in the editor. Our guinea pig is there.

The result of the test is positive - the code outputs:

```
Do I know that variable? 1
1
```

The answer is: **a variable existing outside a function has a scope inside the functions' bodies**.

This rule has a very important exception. Let's try to find it.

Let's make a small change to the code:

```
def myFunction():
    var = 2
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)
```

The result has changed, too - the code produces a slightly different output now:

```
Do I know that variable? 2
1
```

```
def myFunction():
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)
```

Console >_

```
Do I know that variable? 1
1
```

What's happened?

- the `var` variable created inside the function is not the same as when defined outside it - it seems that there two different variables of the same name;
- moreover, the function's variable shadows the variable coming from the outside world.

We can make the previous rule more precise and adequate:

A variable existing outside a function has a scope inside the functions' bodies, excluding those of them which define a variable of the same name.

It also means that the **scope of a variable existing outside a function is supported only when getting its value** (reading). Assigning a value forces the creation of the function's own variable.

Make sure you understand this well and carry out your own experiments.

Console >_

```
Do I know that variable? 1
1
```

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`:

```
global name
global name1, 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.

The code now outputs:

```
Do I know that variable? 2
2
```

This should be sufficient evidence to show that the `global` keyword does what it promises.

```
def myFunction():
    global var
    var = 2
    print("Do I know that variable?", var)

var = 1
myFunction()
print(var)
```

Console >_

```
Do I know that variable? 2
2
```

How the function interacts with its arguments

Now let's find out how the function interacts with its arguments.

The code in the editor should teach you something. As you can see, the function changes the value of its parameter. Does the change affect the argument?

Run the program and check.

The code's output is:

```
I got 1
I have 2
1
```

The conclusion is obvious - **changing the parameter's value doesn't propagate outside the function** (in any case, not when the variable is a scalar, like in the example).

This also means that a function receives the **argument's value**, not the argument itself. This is true for scalars.

Is it worth checking how it works with lists (do you recall the peculiarities of assigning list slices versus assigning lists as a whole?).

The following example will shed some light on the issue:

```
def myFunction(myList1):
    print(myList1)
    myList1 = [0, 1]

myList2 = [2, 3]
myFunction(myList2)
print(myList2)
```

The code's output is:

```
[2, 3]
[2, 3]
```

```
1  def myFunction(n):
2      print("I got", n)
3      n += 1
4      print("I have", n)
5
6  var = 1
7  myFunction(var)
8  print(var)
9
10
```

Console >_

```
I got 1
I have 2
1
```

It seems that the former rule still works.

Finally, can you see the difference in the example below:

```
def myFunction(myList1):
    print(myList1)
    del myList1[0]

myList2 = [2, 3]
myFunction(myList2)
print(myList2)
```

We don't change the value of the parameter `myList1` (we already know it will not affect the argument), but instead modify the list identified by it.

The output may be surprising. Run the code and check:

```
[2, 3]
[3]
```

Can you explain it?

Let's try:

- if the argument is a list, then changing the value of the corresponding parameter doesn't affect the list (remember: variables containing lists are stored in a different way than scalars)
- but if you change a list identified by the parameter (note: the list, not the parameter!), the list will reflect the change.

It's time to write some example functions. You'll do that in the next section.

```
6  var = 1
7  myFunction(var)
8  print(var)
9
10
```

Console >_

```
I got 1
I have 2
1
```

Key takeaways

1. A variable that exists outside a function has a scope inside the function body (Example 1) unless the function defines a variable of the same name (Example 2, and Example 3), e.g.:

Example 1:

```
var = 2

def multByVar(x):
    return x * var

print(multByVar(7))     # outputs: 14
```

Example 2:

```
def mult(x):
    var = 5
    return x * var

print(mult(7))     # outputs: 35
```

Example 3:

```
def multip(x):
    var = 7
    return x * var

var = 3
print(multip(7))     # outputs: 49
```

Exercise 1

What will happen when you try to run the following code?

```
def message():
    alt = 1
    print("Hello, World!")

print(alt)
```

Check

The `NameError` exception will be thrown (`NameError: name 'alt' is not defined`)

Exercise 2

What is the output of the following snippet?

```
a = 1

def fun():
    a = 2
    print(a)

fun()
print(a)
```

Check

2. A variable that exists inside a function has a scope inside the function body (Example 4), e.g.:

Example 4:

```
def adding(x):
    var = 7
    return x + var

print(adding(4))    # outputs: 11

print(var)    # NameError
```

3. You can use the `global` keyword followed by a variable name to make the variable's scope global, e.g.:

```
var = 2
print(var)    # outputs: 2

def retVar():
    global var
    var = 5
    return var

print(retVar())    # outputs: 5

print(var)    # outputs: 5
```

```
2
1
```

Exercise 3

What is the output of the following snippet?

```
a = 1

def fun():
    global a
    a = 2
    print(a)

fun()
a = 3
print(a)
```

Check

```
2
3
```

Exercise 4

What is the output of the following snippet?

```
a = 1

def fun():
    global a
    a = 2
    print(a)

a = 3
fun()
print(a)
```

Check

```
2
2
```

Some simple functions: evaluating the BMI

Let's get started on a function to evaluate the Body Mass Index (BMI).

As you can see, the formula gets two values:

- weight (originally in kilograms)
- height (originally in meters)

It seems that this new function will have **two parameters**. Its name will be `bmi`, but if you prefer any other name, use it instead.

Let's code the function.

```
1 def bmi(weight, height):
2     return weight / height ** 2
3
4 print(bmi(52.5, 1.65))
```

Console >_

```
19.283746556473833
```

The function is complete below (and in the editor window):

```
def bmi(weight, height):
    return weight / height ** 2

print(bmi(52.5, 1.65))
```

The result produced by the sample invocation looks as follows:

```
19.283746556473833
```

The function fulfils our expectations, but it's a bit simple - it assumes that the values of both parameters are always meaningful. It's definitely worth checking if they're trustworthy.

Let's check them both and return `None` if any of them looks suspicious.

Console >_

```
19.283746556473833
```

Some simple functions: evaluating BMI and converting imperial units to metric units

Look at the code in the editor. There are two things we need to pay attention to.

First, the test invocation ensures that the **protection** works properly - the output is:

```
None
```

Second, take a look at the way the **backslash** (`\`) symbol is used. If you use it in Python code and end a line with it, it will tell Python to continue the line of code in the next line of code.

It can be particularly useful when you have to deal with long lines of code and you'd like to improve code readability.

Okay, but there's something we omitted too easily - the imperial measurements. This function is not too useful for people accustomed to pounds, feet and inches.

What can be done for them?

We can write two simple functions to **convert imperial units to metric ones**. Let's start with pounds.

It is a well-known fact that `1 lb = 0.45359237 kg`. We'll use this in our new function.

This is our helper function, named `lbtokg`:

```
def lbtokg(lb):
    return lb * 0.45359237

print(lbtokg(1))
```

The result of the test invocation looks good:

```
0.45359237
```

```
def bmi(weight, height):
    if height < 1.0 or height > 2.5 or \
    weight < 20 or weight > 200:
        return None

    return weight / height ** 2

print(bmi(352.5, 1.65))
```

Console >_

And now it's time for feet and inches: `1 ft = 0.3048 m`, and `1 in = 2.54 cm = 0.0254 m`.

The function we've written is named `ftintom`:

```
def ftintom(ft, inch):
    return ft * 0.3048 + inch * 0.0254

print(ftintom(1, 1))
```

The result of a quick test is:

```
0.3302
```

It looks as expected.

Note: we wanted to name the second parameter just `in`, not `inch`, but we couldn't. Do you know why?

`in` is a Python **keyword** - it cannot be used as a name.

Let's convert six feet into meters:

```
print(ftintom(6, 0))
```

And this is the output:

```
1.8288000000000002
```

```
def bmi(weight, height):
    if height < 1.0 or height > 2.5 or \
    weight < 20 or weight > 200:
        return None

    return weight / height ** 2

print(bmi(352.5, 1.65))
```

Console >_

```
None
```

It's quite possible that sometimes you may want to use just feet without inches. Will Python help you? Of course it will.

We've modified the code a bit:

```
def ftintom(ft, inch = 0.0):
    return ft * 0.3048 + inch * 0.0254

print(ftintom(6))
```

Now the `inch` parameter has its default value equal to `0.0`.

The code produces the following output - this is what is expected:

```
1.8288000000000002
```

Finally, the code is able to answer the question: what is the BMI of a person 5'7" tall and weighing 176 lbs?

This is the code we have built:

```
def ftintom(ft, inch = 0.0):
    return ft * 0.3048 + inch * 0.0254


def lbstokg(lb):
    return lb * 0.45359237


def bmi(weight, height):
    if height < 1.0 or height > 2.5 or \
    weight < 20 or weight > 200:
        return None

    return weight / height ** 2


print(bmi(weight = lbstokg(176), height = ftintom(5, 7)))
```

```
        return None

    return weight / height ** 2

print(bmi(352.5, 1.65))

def ftintom(ft, inch = 0.0):
    return ft * 0.3048 + inch * 0.0254


def lbstokg(lb):
    return lb * 0.45359237

def bmi(weight, height):
    if height < 1.0 or height > 2.5 or \
    weight < 20 or weight > 200:
        return None

    return weight / height ** 2


print(bmi(weight = lbstokg(176), height = ftintom(5, 7)))
```

Console >_

```
None
None
27.565214082533313
```

And the answer is:

```
27.565214082533313
```

Run the code and test it.

None
None
27.565214082533313

Some simple functions: continued

Let's play with triangles now. We'll start with a function to check whether three sides of given lengths can build a triangle.

We know from school that *the sum of two arbitrary sides has to be longer than the third side.*

It won't be a hard challenge. The function will have **three parameters** - one for each side.

It will return `True` if the sides can build a triangle, and `False` otherwise. In this case, `isItATriangle` is a good name for such a function.

Look at the code in the editor. You can find our function there. Run the program.

It seems that it works well - these are the results:

```
True
False
```

```
def isItATriangle(a, b, c):
    if a + b <= c:
        return False
    if b + c <= a:
        return False
    if c + a <= b:
        return False
    return True

print(isItATriangle(1, 1, 1))
print(isItATriangle(1, 1, 3))
```

Console >_

```
True
False
```

Can we make it more compact? It looks a bit wordy.

This is a more compact version:

```
def isItATriangle(a, b, c):
    if a + b <= c or b + c <= a or \
    c + a <= b:
        return False
    return True

print(isItATriangle(1, 1, 1))
print(isItATriangle(1, 1, 3))
```

Can we compact it even more?

Yes, we can - look:

```
def isItATriangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

print(isItATriangle(1, 1, 1))
print(isItATriangle(1, 1, 3))
```

We've negated the condition (reversed the relational operators and replaced `or`s with `and`s, receiving a **universal expression for testing triangles**).

Let's install the function in a larger program. It'll ask the user for three values and make use of the function.

```
print(isItATriangle(1, 1, 1))
print(isItATriangle(1, 1, 3))
```

Console >_

```
True
False
```

Some simple functions: triangles and the Pythagorean theorem

Look at the code in the editor. It asks the user for three values. Then it makes use of the `isItATriangle` function. The code is ready to run.

In the second step, we'll try to ensure that a certain triangle is a **right-angle triangle**.

We will need to make use of the **Pythagorean theorem**:

$c^2 = a^2 + b^2$

How do we recognize which of the three sides is the hypotenuse?

The hypotenuse is the longest side.

Here is the code:

```
def isItATriangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

def isItRightTriangle(a, b, c):
    if not isItATriangle(a, b, c):
        return False
    if c > a and c > b:
        return c ** 2 == a ** 2 + b ** 2
    if a > b and a > c:
        return a ** 2 == b ** 2 + c ** 2

print(isItRightTriangle(5, 3, 4))
print(isItRightTriangle(1, 3, 4))
```

Look at how we test the relationship between the hypotenuse and the remaining sides - we choose the longest side, and apply the **Pythagorean theorem** to check if everything is right. This requires three checks in total.

```
def isItATriangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

a = float(input("Enter the first side's length: "))
b = float(input("Enter the second side's length: "))
c = float(input("Enter the third side's length: "))

if isItATriangle(a, b, c):
    print("Congratulations - it can be a triangle.")
else:
    print("Sorry, it won't be a triangle.")

def isItATriangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

def isItRightTriangle(a, b, c):
    if not isItATriangle(a, b, c):
        return False
    if c > a and c > b:
        return c ** 2 == a ** 2 + b ** 2
    if a > b and a > c:
        return a ** 2 == b ** 2 + c ** 2

print(isItRightTriangle(5, 3, 4))
print(isItRightTriangle(1, 3, 4))
```

Console >_

```
Enter the first side's length: 4
Enter the second side's length: 5
Enter the third side's length: 6
Congratulations - it can be a triangle.
Enter the first side's length: 1
Enter the second side's length: 1
Enter the third side's length: 1
Congratulations - it can be a triangle.
True
False
```

Some simple functions: evaluating a triangle's field

We can also evaluate a triangle's field. **Heron's formula** will be handy here:

$$s = \frac{a+b+c}{2} \qquad A = \sqrt{s(s-a)(s-b)(s-c)}$$

We're going use the exponentiation operator to find the square root - it may seem strange, but it works:

$$\sqrt{x} = x^{\frac{1}{2}}$$

This is the resulting code:

```
def isItATriangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

def heron(a, b, c):
    p = (a + b + c) / 2
    return (p * (p - a) * (p - b) * (p - c)) ** 0.5

def fieldOfTriangle(a, b, c):
    if not isItATriangle(a, b, c):
        return None
    return heron(a, b, c)

print(fieldOfTriangle(1., 1., 2. ** .5))
```

```
#     return a + b > c and b + c > a and c + a > b

# a = float(input("Enter the first side's length: "))
# b = float(input("Enter the second side's length: "))
# c = float(input("Enter the third side's length: "))

# if isItATriangle(a, b, c):
#     print("Congratulations - it can be a triangle.")
# else:
#     print("Sorry, it won't be a triangle.")

def isItATriangle(a, b, c):
    return a + b > c and b + c > a and c + a > b

def heron(a, b, c):
    p = (a + b + c) / 2
    return (p * (p - a) * (p - b) * (p - c)) ** 0.5

def fieldOfTriangle(a, b, c):
    if not isItATriangle(a, b, c):
        return None
    return heron(a, b, c)

print(fieldOfTriangle(1., 1., 2. ** .5))
```

Console >_

```
Enter the first side's length: 4
Enter the second side's length: 5
Enter the third side's length: 6
Congratulations - it can be a triangle.
0.4999999999999983
```

We try it with a right-angle triangle as a half of a square with one side equal to 1. This means that its field should be equal to 0.5.

It's odd - the code produces the following output:

```
0.4999999999999983
```

It's very close to 0.5, but it isn't exactly 0.5. What does it mean? Is it an error?

No, it isn't. This is **the specifics of floating-point calculations**. We'll tell you more about it soon.

Console >_

```
Enter the first side's length: 4
Enter the second side's length: 5
Enter the third side's length: 6
Congratulations - it can be a triangle.
0.4999999999999983
```

Some simple functions: factorials

Another function we're about to write is **factorials**. Do you remember how a factorial is defined?

```
0! = 1 (yes! it's true)
1! = 1
2! = 1 * 2
3! = 1 * 2 * 3
4! = 1 * 2 * 3 * 4
:
:
n! = 1 * 2 ** 3 * 4 * ... * n-1 * n
```

It's marked with an **exclamation mark**, and is equal to the **product** of all natural numbers from one up to its argument.

Let's write our code. We'll create a function and call it `factorialFun`. Here is the code:

```
def factorialFun(n):
    if n < 0:
        return None
    if n < 2:
        return 1

    product = 1
    for i in range(2, n + 1):
        product *= i
    return product

for n in range(1, 6): # testing
    print(n, factorialFun(n))
```

```
def factorialFun(n):
    if n < 0:
        return None
    if n < 2:
        return 1

    product = 1
    for i in range(2, n + 1):
        product *= i
    return product

for n in range(1, 6): # testing
    print(n, factorialFun(n))
```

Console >_

```
1 1
2 2
3 6
4 24
5 120
```

Notice how we mirror step by step the mathematical definition, and how we use the `for` loop to **find the product**.

We add a simple testing code, and these are the results we get:

```
1 1
2 2
3 6
4 24
5 120
```

Some simple functions: Fibonacci numbers

Are you familiar with **Fibonacci** numbers?

They are a **sequence of integer numbers** built using a very simple rule:

- the first element of the sequence is equal to one (**$Fib_1 = 1$**)
- the second is also equal to one (**$Fib_2 = 1$**)
- every subsequent number is the sum of the two preceding numbers (**$Fib_i = Fib_{i-1} + Fib_{i-2}$**)

Here are some of the first Fibonacci numbers:

```
fib1 = 1
fib2 = 1
fib3 = 1 + 1 = 2
fib4 = 1 + 2 = 3
fib5 = 2 + 3 = 5
fib6 = 3 + 5 = 8
fib7 = 5 + 8 = 13
```

What do you think about **implementing it as a function**?

Let's create our `fib` function and test it. Here it is:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1

    elem1 = elem2 = 1
    sum = 0
    for i in range(3, n + 1):
        sum = elem1 + elem2
        elem1, elem2 = elem2, sum
    return sum
```

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1

    elem1 = elem2 = 1
    sum = 0
    for i in range(3, n + 1):
        sum = elem1 + elem2
        elem1, elem2 = elem2, sum
    return sum

for n in range(1, 10): # testing
    print(n, "->", fib(n))

```

Console >_

```
1 -> 1
2 -> 1
3 -> 2
4 -> 3
5 -> 5
6 -> 8
7 -> 13
8 -> 21
9 -> 34
```

Analyze the `for` loop body carefully, and find out how we **move the `elem1` and `elem2` variables through the subsequent Fibonacci numbers**.

The test part of the code produces the following output:

```
1 -> 1
2 -> 1
3 -> 2
4 -> 3
5 -> 5
6 -> 8
7 -> 13
8 -> 21
9 -> 34
```

Console >_

```
1 -> 1
2 -> 1
3 -> 2
4 -> 3
5 -> 5
6 -> 8
7 -> 13
8 -> 21
9 -> 34
```

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:

$Fib_i = Fib_{i-1} + Fib_{i-2}$

The definition of the i^{th} 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:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1
    return fib(n - 1) + fib(n - 2)
```

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1

    elem1 = elem2 = 1
    sum = 0
    for i in range(3, n + 1):
        sum = elem1 + elem2
        elem1, elem2 = elem2, sum
    return sum

for n in range(1, 10):
    print(n, "->", fib(n))
```

Console >_

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:

```
def factorialFun(n):
    if n < 0:
        return None
    if n < 2:
        return 1
    return n * factorialFun(n - 1)
```

Does it work? Yes, it does. Try it for yourself.

Our short *functional* journey is almost over. The next section will take care of two curious Python data types: tuples and dictionaries.

```
# def fib(n):
#     if n < 1:
#         return None
#     if n < 3:
#         return 1
#
#     elem1 = elem2 = 1
#     sum = 0
#     for i in range(3, n + 1):
#         sum = elem1 + elem2
#         elem1, elem2 = elem2, sum
#     return sum
#
# for n in range(1, 10):
#     print(n, "->", fib(n))

def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1
    return fib(n - 1) + fib(n - 2)

print(fib(5))
```

Console >_

```
5
```

Key takeaways

1. A function can call other functions or even itself. When a function calls itself, this situation is known as **recursion**, and the function which calls itself and contains a specified termination condition (i.e., the base case - a condition which doesn't tell the function to make any further calls to that function) is called a **recursive** function.

2. You can use recursive functions in Python to write **clean, elegant code, and divide it into smaller, organized chunks**. On the other hand, you need to be very careful as it might be **easy to make a mistake and create a function which never terminates**. You also need to remember that **recursive calls consume a lot of memory**, and therefore may sometimes be inefficient.

When using recursion, you need to take all its advantages and disadvantages into consideration.

The factorial function is a classic example of how the concept of recursion can be put in practice:

```
# Recursive implementation of the factorial function
def factorial(n):
    if n == 1:    # the base case (termination condition)
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(4)) # 4 * 3 * 2 * 1 = 24
```

Exercise 1

What will happen when you attempt to run the following snippet and why?

```
def factorial(n):
    return n * factorial(n - 1)

print(factorial(4))
```

Check

The factorial function has no termination condition (no base case) so Python will throw an exception (`RecursionError: maximum recursion depth exceeded`)

Exercise 2

What is the output of the following snippet?

```
def fun(a):
    if a > 30:
        return 3
    else:
        return a + fun(a + 3)

print(fun(25))
```

Check

```
56
```

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.

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:

```
tuple1 = (1, 2, 4, 8)
tuple2 = 1., .5, .25, .125
```

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

Let's print them:

```
print(tuple1)
print(tuple2)
```

This is what you should see in the console:

```
(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125)
```

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:

```
emptyTuple = ()
```

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:

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.

```
oneElementTuple1 = (1, )
oneElementTuple2 = 1.,
```

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

How to use a tuple?

If you want to get the elements of a tuple in order to read them over, you can use the same conventions to which you're accustomed while using lists.

Take a look at the code in the editor.

The program should produce the following output - run it and check:

```
1
1000
(10, 100, 1000)
(1, 10)
1
10
100
1000
```

The similarities may be misleading - **don't try to modify a tuple's contents**! It's not a list!

All of these instructions (except the topmost one) will cause a runtime error:

```
myTuple = (1, 10, 100, 1000)

myTuple.append(10000)
del myTuple[0]
myTuple[1] = -10
```

This is the message that Python will give you in the console window:

```
AttributeError: 'tuple' object has no attribute 'append'
```

```
myTuple = (1, 10, 100, 1000)

print(myTuple[0])
print(myTuple[-1])
print(myTuple[1:])
print(myTuple[:-2])

for elem in myTuple:
    print(elem)
```

Console >_

```
1
1000
(10, 100, 1000)
(1, 10)
1
10
100
1000
```

How to use a tuple: continued

What else can tuples do for you?

- the `len()` function accepts tuples, and returns the number of elements contained inside;
- the `+` operator can join tuples together (we've shown you this already)
- the `*` operator can multiply tuples, just like lists;
- the `in` and `not in` operators work in the same way as in lists.

The snippet in the editor presents them all.

The output should look as follows:

```
9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
```

One of the most useful tuple properties is their ability to **appear on the left side of the assignment operator**. You saw this phenomenon some time ago, when it was necessary to find an elegant tool to swap two variables' values.

Take a look at the snippet below:

```
myTuple = (1, 10, 100)

t1 = myTuple + (1000, 10000)
t2 = myTuple * 3

print(len(t2))
print(t1)
print(t2)
print(10 in myTuple)
print(-10 not in myTuple)
```

Console >_

```
9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
```

```
var = 123

t1 = (1, )
t2 = (2, )
t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)
```

Console >_

```
9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
```

It shows three tuples interacting - in effect, the values stored in them "circulate" - `t1` becomes `t2`, `t2` becomes `t3`, and `t3` becomes `t1`.

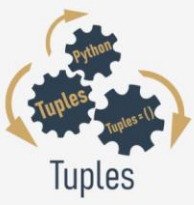
Note: the example presents one more important fact: a **tuple's elements can be variables**, not only literals. Moreover, they can be expressions if they're on the right side of the assignment operator.

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`.

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}
{}
```

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 **data of any type** (except list): it may be a number (integer or float), or even a string;
- 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.

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?

If you want to get any of the values, you have to deliver a valid key value:

```
print(dictionary['cat'])
print(phone_numbers['Suzy'])
```

Getting a dictionary's value resembles indexing, especially thanks to the brackets surrounding the key's value.

Note:

- if the key is a string, you have to specify it as a string;
- **keys are case-sensitive**: `'Suzy'` is something different from `'suzy'`.

The snippet outputs two lines of text:

```
chat
5557654321
```

And now the most important news: you **mustn't use a non-existent key**. Trying something like this:

```
print(phone_numbers['president'])
```

will cause a runtime error. Try to do it.

```
dictionary = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}
phone_numbers = {'boss' : 5551234567, 'Suzy' : 22657854310}
empty_dictionary = {}

###
dictionary = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}
words = ['cat', 'lion', 'horse']

for word in words:
    if word in dictionary:
        print(word, "->", dictionary[word])
    else:
        print(word, "is not in dictionary")
```

Console >_

```
cat -> chat
lion is not in dictionary
horse -> cheval
```

Fortunately, there's a simple way to avoid such a situation. The `in` operator, together with its companion, `not in`, can salvage this situation.

The following code safely searches for some French words:

```
dictionary = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}
words = ['cat', 'lion', 'horse']

for word in words:
    if word in dictionary:
        print(word, "->", dictionary[word])
    else:
        print(word, "is not in dictionary")
```

The code's output looks as follows:

```
cat -> chat
lion is not in dictionary
horse -> cheval
```

Console >_

```
cat -> chat
lion is not in dictionary
horse -> cheval
```

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
```

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

for key in dictionary.keys():
    print(dictionary[key])

```

Console >_

```
cat -> chat
dog -> chien
horse -> cheval
cat
dog
horse
chat
chien
cheval
```

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
```

```
cheval
```

How to use a dictionary: The `items()` and `values()` methods

Another way is based on using a dictionary's method named `items()`. The method **returns tuples** (this is the first example where tuples are something more than just an example of themselves) **where each tuple is a key-value pair**.

This is how it works:

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

for english, french in dictionary.items():
    print(english, "->", french)
```

Note the way in which the tuple has been used as a `for` loop variable.

The example prints:

```
cat -> chat
dog -> chien
horse -> cheval
```

There is also a method named `values()`, which works similarly to `keys()`, but **returns values**.

Here is a simple example:

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

for french in dictionary.values():
    print(french)
```

As the dictionary is not able to automatically find a key for a given value, the role of this method is rather limited.

Here is the expected output:

```
cheval
chien
chat
```

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

for english, french in dictionary.items():
    print(english, "->", french)
```

Console >_

```
cat -> chat
dog -> chien
horse -> cheval
```

How to use a dictionary: modifying and adding values

Assigning a new value to an existing key is simple - as dictionaries are fully **mutable**, there are no obstacles to modifying them.

We're going to replace the value `"chat"` with `"minou"`, which is not very accurate, but it will work well with our example.

Look:

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

dictionary['cat'] = 'minou'
print(dictionary)
```

The output is:

```
{'dog': 'chien', 'horse': 'cheval', 'cat': 'minou'}
```

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

dictionary['cat'] = 'minou'
print(dictionary)
```

Console >_

```
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```

Adding a new key

Adding a new key-value pair to a dictionary is as simple as changing a value - you only have to assign a value to a new, **previously non-existent key**.

Note: this is very different behavior compared to lists, which don't allow you to assign values to non-existing indices.

Let's add a new pair of words to the dictionary - a bit weird, but still valid:

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

dictionary['swan'] = 'cygne'
print(dictionary)
```

The example outputs:

```
{'swan': 'cygne', 'horse': 'cheval', 'dog': 'chien', 'cat': 'chat'}
```

EXTRA

You can also insert an item to a dictionary by using the `update()` method, e.g.:

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

dictionary.update({"duck" : "canard"})
print(dictionary)
```

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

dictionary['cat'] = 'minou'
print(dictionary)
```

Console >_

```
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```

Removing a key

Can you guess how to remove a key from a dictionary?

Note: removing a key will always cause the **removal of the associated value**. **Values cannot exist without their keys**.

This is done with the `del` instruction.

Here's the example:

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

del dictionary['dog']
print(dictionary)
```

Note: **removing a non-existing key causes an error**.

The example outputs:

```
{'cat': 'chat', 'horse': 'cheval'}
```

EXTRA

To remove the last item in a dictionary, you can use the `popitem()` method:

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

dictionary.popitem()
print(dictionary)    # outputs: {'cat' : 'chat', 'dog' : 'chien'}
```

In the older versions of Python, i.e., before 3.6.7, the `popitem()` method removes a random item from a dictionary.

```
print(dictionary)
```

Console >_

```
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```

Tuples and dictionaries can work together

We've prepared a simple example, showing how tuples and dictionaries can work together.

Let's imagine the following problem:

- you need a program to evaluate the students' average scores;
- the program should ask for the student's name, followed by her/his single score;
- the names may be entered in any order;
- entering an empty name finishes the inputting of the data;
- a list of all names, together with the evaluated average score, should be then emitted.

Look at the code in the editor. This how to do it.

Now, let's analyze it line by line:

- **line 1**: create an empty dictionary for the input data; the student's name is used as a key, while all the associated scores are stored in a tuple (the tuple may be a dictionary value - that's not a problem at all)
- **line 3**: enter an "infinite" loop (don't worry, it'll break at the right moment)
- **line 4**: read the student's name here;
- **line 5-6**: if the name is `exit`, leave the loop;
- **line 8**: ask for one of the student's scores (an integer from the range 1-10)
- **line 10-11**: if the student's name is already in the dictionary, lengthen the associated tuple with the new score (note the += operator)
- **line 12-13**: if this is a new student (unknown to the dictionary), create a new entry - its value is a one-element tuple containing the entered score;
- **line 15**: iterate through the sorted students' names;
- **line 16-17**: initialize the data needed to evaluate the average (sum and counter)
- **line 18-20**: we iterate through the tuple, taking all the subsequent scores and updating the sum, together with the counter;
- **line 21**: evaluate and print the student's name and average score.

```
1  school_class = {}
2
3  while True:
4      name = input("Enter the student's name (or type exit to stop): ")
5      if name == 'exit':
6          break
7
8      score = int(input("Enter the student's score (0-10): "))
9
10     if name in school_class:
11         school_class[name] += (score,)
12     else:
13         school_class[name] = (score,)
14
15 for name in sorted(school_class.keys()):
16     adding = 0
17     counter = 0
18     for score in school_class[name]:
19         adding += score
20         counter += 1
21     print(name, ":", adding / counter)
22
```

Console >_

This is a record of a conversation we had with our program:

```
Enter the student's name and press Enter to stop: Bob
Enter the student's score (0-10): 7
Enter the student's name (or type exit to stop): Andy
Enter the student's score (0-10): 3
Enter the student's name (or type exit to stop): Bob
Enter the student's score (0-10): 2
Enter the student's name (or type exit to stop): Andy
Enter the student's score (0-10): 10
Enter the student's name (or type exit to stop): Andy
Enter the student's score (0-10): 3
Enter the student's name (or type exit to stop): Bob
Enter the student's score (0-10): 9
Enter the student's name (or type exit to stop):
Andy : 5.333333333333333
Bob : 6.0
```

Console >_

Key takeaways: tuples

1. **Tuples** are ordered and unchangeable (immutable) collections of data. They can be thought of as immutable lists. They are written in round brackets:

```
myTuple = (1, 2, True, "a string", (3, 4), [5, 6], None)
print(myTuple)

myList = [1, 2, True, "a string", (3, 4), [5, 6], None]
print(myList)
```

Each tuple element may be of a different type (i.e., integers, strings, booleans, etc.). What is more, tuples can contain other tuples or lists (and the other way round).

2. You can create an empty tuple like this:

```
emptyTuple = ()
print(type(emptyTuple))     # outputs: <class 'tuple'>
```

3. A one-element tuple may be created as follows:

```
oneElemTup1 = ("one", )     # brackets and a comma
oneElemTup2 = "one",        # no brackets, just a comma
```

If you remove the comma, you will tell Python to create a variable, not a tuple:

```
myTup1 = 1,
print(type(myTup1))     # outputs: <class 'tuple'>

myTup2 = 1
print(type(myTup2))     # outputs: <class 'int'>
```

4. You can access tuple elements by indexing them:

```
myTuple = (1, 2.0, "string", [3, 4], (5, ), True)
print(myTuple[3])     # outputs: [3, 4]
```

5. Tuples are immutable, which means you cannot change their elements (you cannot append tuples, or modify, or remove tuple elements). The following snippet will cause an exception:

```
myTuple = (1, 2.0, "string", [3, 4], (5, ), True)
myTuple[2] = "guitar"     # a TypeError exception will be raised
```

However, you can delete a tuple as a whole:

```
myTuple = 1, 2, 3,
del myTuple
print(myTuple)     # NameError: name 'myTuple' is not defined
```

6. You can loop through a tuple elements (Example 1), check if a specific element is (not)present in a tuple (Example 2), use the `len()` function to check how many elements there are in a tuple (Example 3), or even join/multiply tuples (Example 4):

```
# Example 1
t1 = (1, 2, 3)
for elem in t1:
    print(elem)

# Example 2
t2 = (1, 2, 3, 4)
print(5 in t2)
print(5 not in t2)

# Example 3
t3 = (1, 2, 3, 5)
print(len(t3))

# Example 4
t4 = t1 + t2
t5 = t3 * 2

print(t4)
print(t5)
```

EXTRA

You can also create a tuple using a Python built-in function called `tuple()`. This is particularly useful when you want to convert a certain iterable (e.g., a list, range, string, etc.) to a tuple:

```
myTup = tuple((1, 2, "string"))
print(myTup)

lst = [2, 4, 6]
print(lst)     # outputs: [2, 4, 6]
print(type(lst))     # outputs: <class 'list'>
tup = tuple(lst)
print(tup)     # outputs: (2, 4, 6)
print(type(tup))     # outputs: <class 'tuple'>
```

By the same fashion, when you want to convert an iterable to a list, you can use a Python built-in function called `list()`:

```
tup = 1, 2, 3,
lst = list(tup)
print(type(lst))     # outputs: <class 'list'>
```

Key takeaways: dictionaries

1. Dictionaries are unordered*, changeable (mutable), and indexed collections of data. (*In Python 3.6x dictionaries have become ordered by default.

Each dictionary is a set of *key : value* pairs. You can create it by using the following syntax:

```
myDictionary = {
    key1 : value1,
    key2 : value2,
    key3 : value3,
    }
```

2. If you want to access a dictionary item, you can do so by making a reference to its key inside a pair of square brackets (ex. 1) or by using the `get()` method (ex. 2):

```
polEngDict = {
    "kwiat" : "flower",
    "woda"  : "water",
    "gleba" : "soil"
    }

item1 = polEngDict["gleba"]    # ex. 1
print(item1)    # outputs: soil

item2 = polEngDict.get("woda")
print(item2)    # outputs: water
```

3. If you want to change the value associated with a specific key, you can do so by referring to the item's key name in the following way:

```
polEngDict = {
    "zamek" : "castle",
    "woda"  : "water",
    "gleba" : "soil"
    }

polEngDict["zamek"] = "lock"
item = polEngDict["zamek"]    # outputs: lock
```

4. To add or remove a key (and the associated value), use the following syntax:

```
myPhonebook = {}    # an empty dictionary

myPhonebook["Adam"] = 3456783958    # create/add a key-value pair
print(myPhonebook)    # outputs: {'Adam': 3456783958}

del myPhonebook["Adam"]
print(myPhonebook)    # outputs: {}
```

You can also insert an item to a dictionary by using the `update()` method, and remove the last element by using the `popitem()` method, e.g.:

```
polEngDict = {"kwiat" : "flower"}

polEngDict = update("gleba" : "soil")
print(polEngDict)    # outputs: {'kwiat' : 'flower', 'gleba' : 'soil'}

polEngDict.popitem()
print(polEngDict)    # outputs: {'kwiat' : 'flower'}
```

5. You can use the `for` loop to loop through a dictionary, e.g.:

```
polEngDict = {
    "zamek" : "castle",
    "woda"  : "water",
    "gleba" : "soil"
    }

for item in polEngDict:
    print(item)    # outputs: zamek
                   #          woda
                   #          gleba
```

6. If you want to loop through a dictionary's keys and values, you can use the `items()` method, e.g.:

```
polEngDict = {
    "zamek" : "castle",
    "woda"  : "water",
    "gleba" : "soil"
    }

for key, value in polEngDict.items():
    print("Pol/Eng ->", key, ":", value)
```

7. To check if a given key exists in a dictionary, you can use the `in` keyword:

```
polEngDict = {
    "zamek" : "castle",
    "woda"  : "water",
    "gleba" : "soil"
    }

if "zamek" in polEngDict:
    print("Yes")
else:
    print("No")
```

8. You can use the `del` keyword to remove a specific item, or delete a dictionary. To remove all the dictionary's items, you need to use the `clear()` method:

```
polEngDict = {
    "zamek" : "castle",
    "woda"  : "water",
    "gleba" : "soil"
    }

print(len(polEngDict))    # outputs: 3
del polEngDict["zamek"]    # remove an item
print(len(polEngDict))    # outputs: 2

polEngDict.clear()    # removes all the items
print(len(polEngDict))    # outputs: 0

del polEngDict    # removes the dictionary
```

9. To copy a dictionary, use the `copy()` method:

```
polEngDict = {
    "zamek" : "castle",
    "woda"  : "water",
    "gleba" : "soil"
    }

copyDict = polEngDict.copy()
```

Key takeaways: tuples and dictionaries

Exercise 1

What happens when you attempt to run the following snippet?

```
myTup = (1, 2, 3)
print(myTup[2])
```

Check

The program will print `3` to the screen.

Exercise 2

What is the output of the following snippet?

```
tup = 1, 2, 3
a, b, c = tup

print(a * b * c)
```

Check

The program will print `6` to the screen. The `tup` tuple elements have been "unpacked" in the `a`, `b`, and `c` variables.

Exercise 5

Write a program that will convert the `l` list to a tuple.

```
l = ["car", "Ford", "flower", "Tulip"]

t = # your code
print(t)
```

Check

Sample solution:

```
l = ["car", "Ford", "flower", "Tulip"]

t = tuple(l)
print(t)
```

Exercise 6

Write a program that will convert the `colors` tuple to a dictionary.

```
colors = (("green", "#008000"), ("blue", "#0000FF"))

# your code

print(colDict)
```

Exercise 3

Complete the code to correctly use the `count()` method to find the number of duplicates of `2` in the following tuple.

```
tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9
duplicates = # your code

print(duplicates)     # outputs: 4
```

Check

```
tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9
duplicates = tup.count(2)

print(duplicates)     # outputs: 4
```

Exercise 4

Write a program that will "glue" the two dictionaries (`d1` and `d2`) together and create a new one (`d3`).

```
d1 = {'Adam Smith':'A', 'Judy Paxton':'B+'}
d2 = {'Mary Louis':'A', 'Patrick White':'C'}
d3 = {}

for item in (d1, d2):
    # your code

print(d3)
```

Check

Sample solution:

```
d1 = {'Adam Smith':'A', 'Judy Paxton':'B+'}
d2 = {'Mary Louis':'A', 'Patrick White':'C'}
d3 = {}

for item in (d1, d2):
    d3.update(item)

print(d3)
```

Check

Sample solution:

```
colors = (("green", "#008000"), ("blue", "#0000FF"))

colDict = dict(colors)
print(colDict)
```

Exercise 7

What will happen when you run the following code?

```
myDict = {"A":1, "B":2}
copyMyDict = myDict.copy()
myDict.clear()

print(copyMyDict)
```

Check

The program will print `{'A': 1, 'B': 2}` to the screen

Exercise 8

What is the output of the following program?

```
colors = {
    "white" : (255, 255, 255),
    "grey"  : (128, 128, 128),
    "red"   : (255, 0, 0),
    "green" : (0, 128, 0)
    }

for col, rgb in colors.items():
    print(col, ":", rgb)
```

Check

```
white : (255, 255, 255)
grey : (128, 128, 128)
red : (255, 0, 0)
green : (0, 128, 0)
```

PROJECT

Estimated time

30-60 minutes

Level of difficulty

Medium/Hard

Objectives

- perfecting the student's skills in using Python for solving complex problems,
- integration of programming techniques in one program consisting of many various parts.

Scenario

Your task is to write **a simple program which pretends to play *tic-tac-toe* with the user**. To make it all easier for you, we've decided to simplify the game. Here are our assumptions:

- the computer (i.e., your program) should play the game using `'X'` s;
- the user (e.g., you) should play the game using `'O'` s;
- the first move belongs to the computer - it always puts its first `'X'` in the middle of the board;
- all the squares are numbered row by row starting with `1` (see the example session below for reference)
- the user inputs their move by entering the number of the square they choose - the number must be valid, i.e., it must be an integer, it must be greater than `0` and less than `10`, and it cannot point to a field which is already occupied;
- the program checks if the game is over - there are four possible verdicts: the game should continue, or the game ends with a tie, your win, or the computer's win;
- the computer responds with its move and the check is repeated;
- don't implement any form of artificial intelligence - a random field choice made by the computer is good enough for the game.

The example session with the program may look as follows:

```
from random import randrange

def DisplayBoard(board):
    print("+-------" * 3,"+",sep="")
    for row in range(3):
        print("|       " * 3,"|",sep="")
        for col in range(3):
            print("|   " + str(board[row][col]) + "   ",end="")
        print("|")
        print("|       " * 3,"|",sep="")
        print("+-------" * 3,"+",sep="")

def EnterMove(board):
    ok = False  # fake assumption - we need it to enter the loop
    while not ok:
        move = input("Enter your move: ")
        ok = len(move) == 1 and move >= '1' and move <= '9' # is user's input valid?
        if not ok:
            print("Bad move - repeat your input!") # no, it isn't - do the input again
            continue
        move = int(move) - 1    # cell's number from 0 to 8
        row = move // 3      # cell's row
        col = move % 3       # cell's column
        sign = board[row][col]  # check the selected square
        ok = sign not in ['O','X']
        if not ok:  # it's occupied - to the input again
            print("Field already occupied - repeat your input!")
            continue
    board[row][col] = 'O'   # set 'O' at the selected square
```

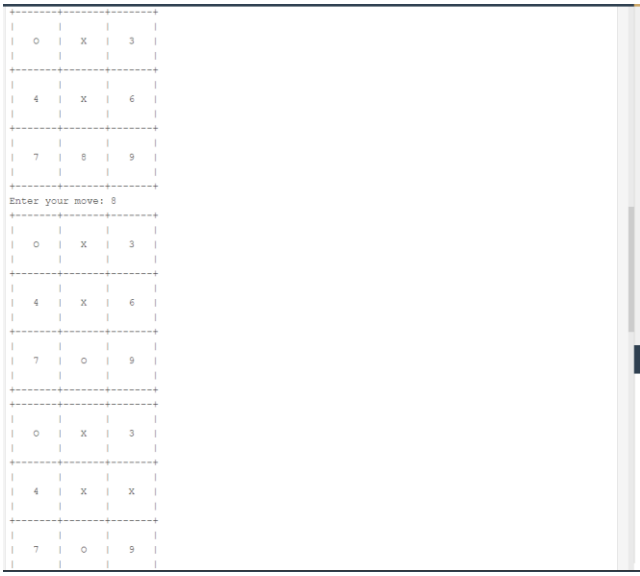
Console >_

The example session with the program may look as follows:

```
+-------+-------+-------+
|       |       |       |
|   1   |   2   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
Enter your move: 1
+-------+-------+-------+
|       |       |       |
|   O   |   2   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
```

```
def MakeListOfFreeFields(board):
    free = []   # the list is empty initially
    for row in range(3): # iterate through rows
        for col in range(3): # iterate through columns
            if board[row][col] not in ['O','X']: # is the cell free?
                free.append((row,col)) # yes, it is - append new tuple to the list
    return free


def VictoryFor(board,sgn):
    if sgn == "X":  # are we looking for X?
        who = 'me'  # yes - it's computer's side
    elif sgn == "O": # ... or for O?
        who = 'you' # yes - it's our side
    else:
        who = None  # we should not fall here!
    cross1 = cross2 = True  # for diagonals
    for rc in range(3):
        if board[rc][0] == sgn and board[rc][1] == sgn and board[rc][2] == sgn: # check
            return who
        if board[0][rc] == sgn and board[1][rc] == sgn and board[2][rc] == sgn: # check
            return who
        if board[rc][rc] != sgn: # check 1st diagonal
            cross1 = False
        if board[2 - rc][2 - rc] != sgn: # check 2nd diagonal
            cross2 = False
    if cross1 or cross2:
        return who
    return None
```

Console >_



```
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
Enter your move: 8
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
```

```
61 def DrawMove(board):
62     free = MakeListOfFreeFields(board) # make a list of free fields
63     cnt = len(free)
64     if cnt > 0: # if the list is not empty, choose a place for 'X' and set it
65         this = randrange(cnt)
66         row, col = free[this]
67         board[row][col] = 'X'
68
69 board = [ [3 * j + i + 1 for i in range(3)] for j in range(3) ] # make an empty board
70 board[1][1] = 'X' # set first 'X' in the middle
71 free = MakeListOfFreeFields(board)
72 humanturn = True # which turn is it now?
73 while len(free):
74     DisplayBoard(board)
75     if humanturn:
76         EnterMove(board)
77         victor = VictoryFor(board,'O')
78     else:
79         DrawMove(board)
80         victor = VictoryFor(board,'X')
81     if victor != None:
82         break
83     humanturn = not humanturn
84     free = MakeListOfFreeFields(board)
85
86 DisplayBoard(board)
87 if victor == 'you':
88     print("You won!")
89 elif victor == 'me':
```

Console >_

```
+-------+-------+-------+
Enter your move: 4
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
Enter your move: 7
```

```
65         this = randrange(cnt)
66         row, col = free[this]
67         board[row][col] = 'X'
68
69 board = [ [3 * j + i + 1 for i in range(3)] for j in range(3) ] # make an empty board
70 board[1][1] = 'X' # set first 'X' in the middle
71 free = MakeListOfFreeFields(board)
72 humanturn = True # which turn is it now?
73 while len(free):
74     DisplayBoard(board)
75     if humanturn:
76         EnterMove(board)
77         victor = VictoryFor(board,'O')
78     else:
79         DrawMove(board)
80         victor = VictoryFor(board,'X')
81     if victor != None:
82         break
83     humanturn = not humanturn
84     free = MakeListOfFreeFields(board)
85
86 DisplayBoard(board)
87 if victor == 'you':
88     print("You won!")
89 elif victor == 'me':
90     print("I won")
91 else:
92     print("Tie!")
```

Console >_

```
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
You won!
```

Requirements

Implement the following features:

- the board should be stored as a three-element list, while each element is another three-element list (the inner lists represent rows) so that all of the squares may be accessed using the following syntax:

```
board[row][column]
```

- each of the inner list's elements can contain `'O'`, `'X'`, or a digit representing the square's number (such a square is considered free)
- the board's appearance should be exactly the same as the one presented in the example.
- implement the functions defined for you in the editor.

Drawing a random integer number can be done by utilizing a Python function called `randrange()`. The example program below shows how to use it (the program prints ten random numbers from 0 to 8).

Note: the `from-import` instruction provides an access to the `randrange` function defined within an external Python module called `random`.

```
from random import randrange

for i in range(10):
    print(randrange(8))
```

```
    if humanturn:
        EnterMove(board)
        victor = VictoryFor(board,'O')
    else:
        DrawMove(board)
        victor = VictoryFor(board,'X')
    if victor != None:
        break
    humanturn = not humanturn
    free = MakeListOfFreeFields(board)

DisplayBoard(board)
if victor == 'you':
    print("You won!")
elif victor == 'me':
    print("I won")
else:
    print("Tie!")
```

Console >_

Congratulations! You have completed Module 4.

Well done! You've reached the end of Module 4 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 4:

- the defining and using of functions - their rationale, purpose, conventions, and traps;
- the concept of passing arguments in different ways and setting their default values, along with the mechanisms of returning the function's results;
- name scope issues;
- new data aggregates: tuples and dictionaries, and their role in data processing.

You are now ready to take the module quiz and attempt the final challenge: Module 4 Test, which will help you gauge what you've learned so far.

MODULE: 5

Load Module 5 - Modules, packages, string and list methods, and exceptions in a new window

Welcome to Programming Essentials in Python - Part 2

Module 5

Modules, packages, string and list methods, and exceptions

Module 6

The Object-Oriented Approach: classes, methods, objects and the standard objective features; exception handling, and working with files

Programming Essentials in Python

Part 2

developed by PYTHON INSTITUTE Open Education & Development Group

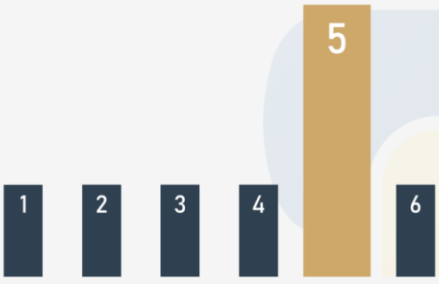
Programming Essentials in Python: Module 5

In this module, you will learn about:

- Python modules: their rationale, function, how to import them in different ways, and present the content of some standard modules provided by Python;
- the way in which modules are coupled together to make packages.
- the concept of an exception and Python's implementation of it, including the try-except instruction, with its applications, and the raise instruction.
- strings and their specific methods, together with their similarities and differences compared to lists.

Module 5:

Modules, packages, string and list methods, and exceptions



What is a module?

Computer code has a tendency to grow. We can say that code that doesn't grow is probably completely unusable or abandoned. A real, wanted, and widely used code develops continuously, as both users' demands and users' expectations develop in their own rhythms.

A code which is not able to respond to users' needs will be forgotten quickly, and instantly replaced with a new, better, and more flexible code. Be prepared for this, and never think that any of your programs is eventually completed. The completion is a transition state and usually passes quickly, after the first bug report. Python itself is a good example how the rule acts.

Growing code is in fact a growing problem. A larger code always means tougher maintenance. Searching for bugs is always easier where the code is smaller (just as finding a mechanical breakage is simpler when the machinery is simpler and smaller).

Moreover, when the code being created is expected to be really big (you can use a total number of source lines as a useful, but not very accurate, measure of a code's size) you may want (or rather, you will be forced) to divide it into many parts, implemented in parallel by a few, a dozen, several dozen, or even several hundred individual developers.

Of course, this cannot be done using one large source file, which is edited by all programmers at the same time. This will surely lead to a spectacular disaster.

If you want such a software project to be completed successfully, you have to have the means allowing you to:

- divide all the tasks among the developers;
- join all the created parts into one working whole.

For example, a certain project can be divided into two main parts:

- the user interface (the part that communicates with the user using widgets and a graphical screen)
- the logic (the part processing data and producing results)

Each of these parts can be (most likely) divided into smaller ones, and so on. Such a process is often called **decomposition**.

For example, if you were asked to arrange a wedding, you wouldn't do everything yourself - you would find a number of professionals and split the task between them all.

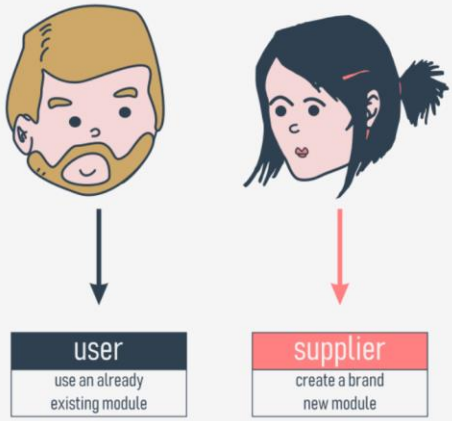
How do you divide a piece of software into separate but cooperating parts? This is the question. **Modules** are the answer.

How to make use of a module?

The handling of modules consists of two different issues:

- the first (probably the most common) happens when you want to use an already existing module, written by someone else, or created by yourself during your work on some complex project - in this case you are the module's **user**;
- the second occurs when you want to create a brand new module, either for your own use, or to make other programmers' lives easier - you are the module's **supplier**.

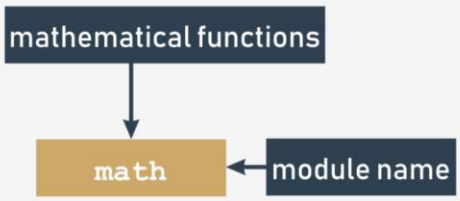
Let's discuss them separately.



First of all, a module is identified by its **name**. If you want to use any module, you need to know the name. A (rather large) number of modules is delivered together with Python itself. You can think of them as a kind of "Python extra equipment".

All these modules, along with the built-in functions, form the **Python standard library** - a special sort of library where modules play the roles of books (we can even say that folders play the roles of shelves). If you want to take a look at the full list of all "volumes" collected in that library, you can find it here: https://docs.python.org/3/library/index.html.

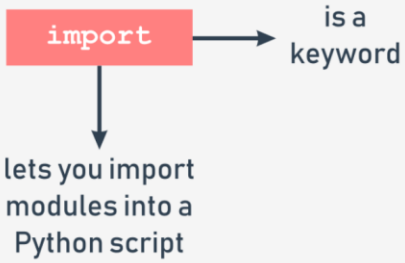
Each module consists of entities (like a book consists of chapters). These entities can be functions, variables, constants, classes, and objects. If you know how to access a particular module, you can make use of any of the entities it stores.



Let's start the discussion with one of the most frequently used modules, named `math`. Its name speaks for itself - the module contains a rich collection of entities (not only functions) which enable a programmer to effectively implement calculations demanding the use of mathematical functions, like *sin()* or *log()*.

Importing a module

To make a module usable, you must **import** it (think of it like of taking a book off the shelf). Importing a module is done by an instruction named `import`. Note: `import` is also a keyword (with all the consequences of this fact).



Let's assume that you want to use two entities provided by the `math` module:

- a symbol (constant) representing a precise (as precise as possible using double floating-point arithmetic) value of π (although using a Greek letter to name a variable is fully possible in Python, the symbol is named **pi** - it's a more convenient solution, especially for that part of the world which neither has nor is going to use a Greek keyboard)
- a function named `sin()` (the computer equivalent of the mathematical *sine* function)

Both these entities are available through the `math` module, but the way in which you can use them strongly depends on how the import has been done.

The simplest way to import a particular module is to use the import instruction as follows:

```
import math
```

The clause contains:

- the `import` keyword;
- the **name of the module** which is subject to import.

The instruction may be located anywhere in your code, but it must be placed **before the first use of any of the module's entities**.

If you want to (or have to) import more than one module, you can do it by repeating the `import` clause, or by listing the modules after the `import` keyword, like here:

```
import math, sys
```

The instruction imports two modules, first the one named math and then the second named `sys`.

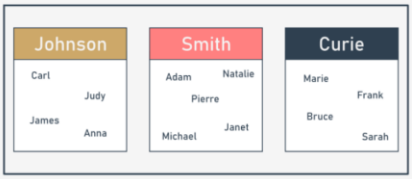
The modules' list may be arbitrarily long.

Importing a module: continued

To continue, you need to become familiar with an important term: **namespace**.

Don't worry, we won't go into great detail - this explanation is going to be as short as possible.

A **namespace** is a space (understood in a non-physical context) in which some names exist and the names don't conflict with each other (i.e., there are not two different objects of the same name). We can say that each social group is a namespace - the group tends to name each of its members in a unique way (e.g., parents won't give their children the same first names).



This uniqueness may be achieved in many ways, e.g., by using nicknames along with the first names (it will work inside a small group like a class in a school) or by assigning special identifiers to all members of the group (the US Social Security Number is a good example of such practice).

Inside a certain namespace, each name must remain unique. This may mean that some names may disappear when any other entity of an already known name enters the namespace. We'll show you how it works and how to control it, but first, let's return to imports.

If the module of a specified name **exists and is accessible** (a module is in fact a **Python source file**), Python imports its contents, i.e., **all the names defined in the module become known**, but they don't enter your code's namespace.

This means that you can have your own entities named `sin` or `pi` and they won't be affected by the import in any way.

At this point, you may be wondering how to access the `pi` coming from the `math` module.

To do this, you have to qualify the `pi` with the name of its original module.

Importing a module: continued

Look at the snippet below, this is the way in which you qualify the names of `pi` and `sin` with the name of its originating module:

```
math.pi
math.sin
```

It's simple, you put:

- the **name of the module** (`math` here)
- a **dot**;
- the **name of the entity** (`pi` here)

Such a form clearly indicates the namespace in which the name exists.

Note: using this qualification is **compulsory** if a module has been imported by the `import` module instruction. It doesn't matter if any of the names from your code and from the module's namespace are in conflict or not.

This first example won't be very advanced - we just want to print the value of **sin(1/2π)**.

Look at the code in the editor. This is how we test it.

The code outputs the expected value: `1.0` .

Note: removing any of the two qualifications will make the code erroneous. There is no other way to enter `math` 's namespace if you did the following:

```
import math
```

```
1  import math
2  print(math.sin(math.pi/2))
```

Console >_

```
1.0
```

Importing a module: continued

Now we're going to show you how the two namespaces (yours and the module's one) can coexist.

Take a look at the example in the editor window.

We've defined our own `pi` and `sin` here.

Run the program. The code should produce the following output:

```
0.99999999
1.0
```

As you can see, the entities don't affect each other.

```
1  import math
2
3  def sin(x):
4      if 2 * x == pi:
5          return 0.99999999
6      else:
7          return None
8
9  pi = 3.14
10
11 print(sin(pi/2))
12 print(math.sin(math.pi/2))
```

Console >_

```
0.99999999
1.0
```

Importing a module: continued

In the second method, the `import` 's syntax precisely points out which module's entity (or entities) are acceptable in the code:

```
from math import pi
```

The instruction consists of the following elements:

- the `from` keyword;
- the **name of the module** to be (selectively) imported;
- the `import` keyword;
- the **name or list of names of the entity/entities** which are being imported into the namespace.

The instruction has this effect:

- the listed entities (and only those ones) are **imported from the indicated module**;
- the names of the imported entities are **accessible without qualification**.

Note: no other entities are imported. Moreover, you cannot import additional entities using a qualification - a line like this one:

```
print(math.e)
```

will cause an error (`e` is Euler's number: 2.71828...)

Let's rewrite the previous script to incorporate the new technique.

Here it is:

```
from math import sin, pi

print(sin(pi/2))
```

The output should be the same as previously, as in fact we've used the same entities as before: `1.0` . Copy the code, paste it in the editor, and run the program.

Does the code look simpler? Maybe, but the look is not the only effect of this kind of import. Let's show you that.

```
1  print(math.e*2)
2  from math import sin, pi
3
4  print(sin(pi/2))
```

Console >_

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    print(math.e)
NameError: name 'math' is not defined
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    print(math.e*2)
NameError: name 'math' is not defined
```

Importing a module: continued

Look at the code in the editor. Analyze it carefully:

- line 01: carry out the selective import;
- line 03: make use of the imported entities and get the expected result (`1.0`)
- lines 05 through 11: redefine the meaning of `pi` and `sin` - in effect, they supersede the original (imported) definitions within the code's namespace;
- line 13: get `0.99999999` , which confirms our conclusions.

Let's do another test. Look at the code below:

```
pi = 3.14       # line 01

def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None     # line 07

print(sin(pi/2))        # line 09


from math import sin, pi        # line 12

print(sin(pi/2))        # line 14
```

Here, we've reversed the sequence of the code's operations:

- lines 01 through 07: define our own `pi` and `sin` ;
- line 09: make use of them (0.99999999 appears on screen)
- line 12: carry out the import - the imported symbols supersede their previous definitions within the namespace;
- line 14: get `1.0` as a result.

```
# from math import sin, pi

# print(sin(pi/2))

# pi = 3.14

# def sin(x):
#     if 2 * x == pi:
#         return 0.99999999
#     else:
#         return None

# print(sin(pi/2))
pi = 3.14   # line 01

def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None # line 07

print(sin(pi/2))    # line 09


from math import sin, pi    # line 12

print(sin(pi/2))    # line 14
```

Console >_

```
1.0
0.99999999
0.99999999
1.0
```

Importing a module: *

In the third method, the `import` 's syntax is a more aggressive form of the previously presented one:

```
from module import *
```

As you can see, the name of an entity (or the list of entities' names) is replaced with a single asterisk (`*`).

Such an instruction **imports all entities from the indicated module**.

Is it convenient? Yes, it is, as it relieves you of the duty of enumerating all the names you need.

Is it unsafe? Yes, it is - unless you know all the names provided by the module, **you may not be able to avoid name conflicts**. Treat this as a temporary solution, and try not to use it in regular code.

Importing a module: the `as` keyword

If you use the import module variant and you don't like a particular module's name (e.g., it's the same as one of your already defined entities, so qualification becomes troublesome) you can give it any name you like - this is called **aliasing**.

Aliasing causes the module to be identified under a different name than the original. This may shorten the qualified names, too.

Creating an alias is done together with importing the module, and demands the following form of the import instruction:

```
import module as alias
```

The "module" identifies the original module's name while the "alias" is the name you wish to use instead of the original.

Note: `as` is a keyword.

```
from module import *
```

Console >_

Importing a module: continued

If you need to change the word `math` , you can introduce your own name, just like in the example:

```
import math as m
print(m.sin(m.pi/2))
```

Note: after successful execution of an aliased import, the **original module name becomes inaccessible** and must not be used.

In turn, when you use the `from module import name` variant and you need to change the entity's name, you make an alias for the entity. This will cause the name to be replaced by the alias you choose.

This is how it can be done:

```
from module import name as alias
```

As previously, the original (unaliased) name becomes inaccessible.

The phrase `name as alias` can be repeated - use commas to separate the multiplied phrases, like this:

```
from module import n as a, m as b, o as c
```

The example may look a bit weird, but it works:

```
from math import pi as PI, sin as sine

print(sine(PI/2))
```

Now you're familiar with the basics of using modules. Let us show you some modules and some of their useful entities.

```
# import math as m

# print(m.sin(m.pi/2))
from math import pi as PI, sin as sine

print(sine(PI/2))
```

Console >_

```
1.0
1.0
```

Working with standard modules

Before we start going through some standard Python modules, we want to introduce the `dir()` function to you. It has nothing to do with the `dir` command you know from Windows and Unix consoles, as `dir()` doesn't show the contents of a disk directory/folder, but there is no denying that it does something really similar - it is able to reveal all the names provided through a particular module.

There is one condition: the module has to have been previously imported as a whole (i.e., using the `import module` instruction - `from module` is not enough).

The function returns an **alphabetically sorted list** containing all entities' names available in the module identified by a name passed to the function as an argument:

```
dir(module)
```

Note: if the module's name has been aliased, you must use the alias, not the original name.

Using the function inside a regular script doesn't make much sense, but it is still possible.

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

```
import math

for name in dir(math):
    print(name, end="\t")
```

The example code should produce the following output:

```
os      acosh   asin    asinh   atan    atan2   atanh   ceil    copysign        cos     cosh
```

Have you noticed these strange names beginning with `__` at the top of the list? We'll tell you more about them when we talk about the issues related to writing your own modules.

Some of the names might bring back memories from math lessons, and you probably won't have any problems guessing their meanings.

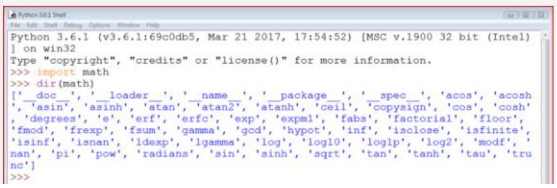
Using the `dir()` function inside a code may not seem very useful - usually you want to know a particular module's contents before you write and run the code.

Fortunately, you can execute the function **directly in the Python console** (IDLE), without needing to write and run a separate script.

This is how it can be done:

```
import math
dir(math)
```

You should see something similar to this:



Selected functions from the `math` module

Let's start with a quick preview of some of the functions provided by the `math` module.

We've chosen them arbitrarily, but that doesn't mean that the functions we haven't mentioned here are any less significant. Dive into the modules' depths yourself - we don't have the space or the time to talk about everything in detail here.

The first group of the `math`'s functions are connected with **trigonometry**:

- `sin(x)` → the sine of x;
- `cos(x)` → the cosine of x;
- `tan(x)` → the tangent of x.

All these functions take one argument (an angle measurement expressed in radians) and return the appropriate result (be careful with `tan()` - not all arguments are accepted).

Of course, there are also their inversed versions:

- `asin(x)` → the arcsine of x;
- `acos(x)` → the arccosine of x;
- `atan(x)` → the arctangent of x.

These functions take one argument (mind the domains) and return a measure of an angle in radians.

To effectively operate on angle measurements, the `math` module provides you with the following entities:

```
1  from math import pi, radians, degrees, sin, cos, tan, asin
2
3  ad = 90
4  ar = radians(ad)
5  ad = degrees(ar)
6
7  print(ad == 90.)
8  print(ar == pi / 2.)
9  print(sin(ar) / cos(ar) == tan(ar))
10 print(asin(sin(ar)) == ar)
```

```
Console >_
True
True
True
True
```

To effectively operate on angle measurements, the `math` module provides you with the following entities:

- `pi` → a constant with a value that is an approximation of π;
- `radians(x)` → a function that converts x from degrees to radians;
- `degrees(x)` → acting in the other direction (from radians to degrees)

Now look at the code in the editor. The example program isn't very sophisticated, but can you predict its results?

Apart from the circular functions (listed above) the `math` module also contains a set of their **hyperbolic analogues**:

- `sinh(x)` → the hyperbolic sine;
- `cosh(x)` → the hyperbolic cosine;
- `tanh(x)` → the hyperbolic tangent;
- `asinh(x)` → the hyperbolic arcsine;
- `acosh(x)` → the hyperbolic arccosine;
- `atanh(x)` → the hyperbolic arctangent.

```
Console >_
True
True
True
True
```

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 e^x;
- `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 x^y (mind the domains)

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

Look at the code in the editor. Can you predict its output?

```
1  from math import e, exp, log
2
3  print(pow(e, 1) == exp(log(e)))
4  print(pow(2, 2) == exp(2 * log(2)))
5  print(log(e, e) == exp(0))
6  print(exp(0))
```

```
Console >_
False
True
True
False
True
True
148.4131591025766
False
True
True
1.0
```

Selected functions from the `math` module: continued

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()`.

Run the program and check its output.

```
from math import ceil, floor, trunc

x = 1.4
y = 2.6

print(floor(x), floor(y))
print(floor(-x), floor(-y))
print(ceil(x), ceil(y))
print(ceil(-x), ceil(-y))
print(trunc(x), trunc(y))
print(trunc(-x), trunc(-y))
```

```
Console >_
1 2
-2 -3
2 3
-1 -2
1 2
-1 -2
```

Is there real randomness in computers?

Another module worth mentioning is the one named `random`.

It delivers some mechanisms allowing you to operate with **pseudorandom numbers**.

Note the prefix **pseudo** - the numbers generated by the modules may look random in the sense that you cannot predict their subsequent values, but don't forget that they all are calculated using very refined algorithms.

The algorithms aren't random - they are deterministic and predictable. Only those physical processes which run completely out of our control (like the intensity of cosmic radiation) may be used as a source of actual random data. Data produced by deterministic computers cannot be random in any way.

A random number generator takes a value called a **seed**, treats it as an input value, calculates a "random" number based on it (the method depends on a chosen algorithm) and produces a **new seed value**.

The length of a cycle in which all seed values are unique may be very long, but it isn't infinite - sooner or later the seed values will start repeating, and the generating values will repeat, too. This is normal. It's a feature, not a mistake, or a bug.

The initial seed value, set during the program start, determines the order in which the generated values will appear.

The random factor of the process may be **augmented by setting the seed with a number taken from the current time** - this may ensure that each program launch will start from a different seed value (ergo, it will use different random numbers).

Fortunately, such an initialization is done by Python during module import.

Selected functions from the `random` module

The most general function named `random()` (not to be confused with the module's name) **produces a float number** `x` **coming from the range** `(0.0, 1.0)` - in other words: (0.0 <= x < 1.0).

The example program in the editor will produce five pseudorandom values - as their values are determined by the current (rather unpredictable) seed value, you can't guess them. Run the program.

The `seed()` function is able to directly **set the generator's seed**. We'll show you two of its variants:

- `seed()` - sets the seed with the current time;
- `seed(int_value)` - sets the seed with the integer value `int_value`.

We've modified the previous program - in effect, we've removed any trace of randomness from the code:

```
from random import random, seed

seed(0)

for i in range(5):
    print(random())
```

Due to the fact that the seed is always set with the same value, the sequence of generated values always looks the same.

Run the program. This is what we've got:

```
0.844421851525
0.75795440294
0.420571580831
0.258916750293
0.511274721369
```

And you?

Note: your values may be slightly different than ours if your system uses more precise or less precise floating-point arithmetic, but the difference will be seen quite far from the decimal point.

```
from random import random

for i in range(5):
    print(random())
```

```
Console >_
0.057914953076585896
0.1818163850408122
0.7475555423310879
0.5520067057440364
0.696132188024846
0.6003009224873603
0.1511184759868257
0.17375961436939436
0.30772953890722143
0.4518700845424697
0.8972268246956586
0.7709212204712819
0.5166308824119851
0.8828240143560865
0.8758153980291448
0.07092877868591363
0.09869367688861563
0.5174533024342531
0.11657753581598418
0.7352614827013063
```

Selected functions from the `random` module: continued

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.

```
from random import randrange, randint

print(randrange(1), end=' ')
print(randrange(0, 1), end=' ')
print(randrange(0, 1, 1), end=' ')
print(randint(0, 1))
```

```
Console >_
0 0 0 0
0 0 0 1
0 0 0 1
0 0 0 0
0 0 0 1
0 0 0 0
0 0 0 1
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 1
```

Selected functions from the `random` module: continued

The previous functions have one important disadvantage - they may produce repeating values even if the number of subsequent invocations is not greater than the width of the specified range.

Look at the code in the editor. The program very likely outputs a set of numbers in which some elements are not unique.

This is what we got in one of the launches:

```
9,4,5,4,5,8,9,4,8,4,
```

As you can see, this is not a good tool for generating numbers in a lottery. Fortunately, there is a better solution than writing your own code to check the uniqueness of the "drawn" numbers.

It's a function named in a very suggestive way - `choice`:

- `choice(sequence)`
- `sample(sequence, elements_to_choose=1)`

The first variant chooses a "random" element from the input sequence and returns it.

```
from random import randint

for i in range(10):
    print(randint(1, 10), end=',')
```

Console >_

```
2,3,4,9,10,9,9,10,8,4,
10,3,4,3,7,10,5,9,5,1,
```

The second one builds a list (a sample) consisting of the `elements_to_choose` element (which defaults to `1`) "drawn" from the input sequence.

In other words, the function chooses some of the input elements, returning a list with the choice. The elements in the sample are placed in random order. Note: the `elements_to_choose` must not be greater than the length of the input sequence.

Look at the code below:

```
from random import choice, sample

lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(choice(lst))
print(sample(lst, 5))
print(sample(lst, 10))
```

Again, the output of the program is not predictable. Our results looked like this:

```
4
[3, 1, 8, 9, 10]
[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]
```

Console >_

```
2,3,4,9,10,9,9,10,8,4,
10,3,4,3,7,10,5,9,5,1,
```

How to know where you are?

Sometimes, it may be necessary to find out information unrelated to Python. For example, you may need to know the location of your program within the greater environment of the computer.

Imagine your program's environment as a pyramid consisting of a number of layers or platforms.

The layers are:

- your (running) code is located at the top of it;
- Python (more precisely - its runtime environment) lies directly below it;
- the next layer of the pyramid is filled with the OS (operating system) - Python's environment provides some of its functionalities using the operating system's services; Python, although very powerful, isn't omnipotent - it's forced to use many helpers if it's going to process files or communicate with physical devices;
- the bottom-most layer is hardware - the processor (or processors), network interfaces, human interface devices (mice, keyboards, etc.) and all other machinery needed to make the computer run; the OS knows how to drive it, and uses lots of tricks to conduct all parts in a consistent rhythm.

This means than some of your (or rather your program's) actions have to travel a long way to be successfully performed - imagine that:

- **your code** wants to create a file, so it invokes one of Python's functions;
- **Python** accepts the order, rearranges it to meet local OS requirements (it's like putting the stamp "approved" on your request) and sends it down (this may remind you of a chain of command)
- the **OS** checks if the request is reasonable and valid (e.g., whether the file name conforms to some syntax rules) and tries to create the file; such an operation, seemingly very simple, isn't atomic - it consists of many minor steps taken by...
- the **hardware**, which is responsible for activating storage devices (hard disk, solid state devices, etc.) to satisfy the OS's needs.

Usually, you're not aware of all that fuss - you want the file to be created and that's that.

But sometimes you want to know more - for example, the name of the OS which hosts Python, and some characteristics describing the hardware that hosts the OS.

There is a module providing some means to allow you to know where you are and what components work for you. The module is named platform. We'll show you some of the functions it provides to you.

Selected functions from the `platform` module

The `platform` module lets you access the underlying platform's data, i.e., hardware, operating system, and interpreter version information.

There is a function that can show you all the underlying layers in one glance, named `platform`, too. It just returns a string describing the environment; thus, its output is rather addressed to humans than to automated processing (you'll see it soon).

This is how you can invoke it: `platform(aliased = False, terse = False)`

And now:

- `aliased` → when set to `True` (or any non-zero value) it may cause the function to present the alternative underlying layer names instead of the common ones;
- `terse` → when set to `True` (or any non-zero value) it may convince the function to present a briefer form of the result (if possible)

We ran our sample program using three different platforms - this is what we got:

Intel x86 + Windows ® Vista (32 bit):

```
Windows-Vista-6.0.6002-SP2
Windows-Vista-6.0.6002-SP2
Windows-Vista
```

```
from platform import platform

print(platform())
print(platform(1))
print(platform(0, 1))
```

Intel x86 + Gentoo Linux (64 bit):

```
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_0_2.20GHz-with-gentoo-2.3
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_0_2.20GHz-with-gentoo-2.3
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_CPU_0_2.20GHz-with-glibc2.3.4
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-with-glibc2.9
```

You can also run the sample program in IDLE on your local machine to check what output you will have.

Console >_

```
Linux-4.4.0-176-generic-x86_64-with
Linux-4.4.0-176-generic-x86_64-with
Linux-4.4.0-176-generic-x86_64-with
```

Selected functions from the `platform` module: continued

Sometimes, you may just want to know the generic name of the processor which runs your OS together with Python and your code - a function named `machine()` will tell you that. As previously, the function returns a string.

Again, we ran the sample program on three different platforms:

Intel x86 + Windows ® Vista (32 bit):

```
x86
```

Intel x86 + Gentoo Linux (64 bit):

```
x86_64
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
armv7l
```

```
1 from platform import machine
2
3 print(machine())
```

Console >_

```
x86_64
```

Selected functions from the `platform` module: continued

The `processor()` function returns a string filled with the real processor name (if possible).

Once again, we ran the sample program on three different platforms:

Intel x86 + Windows ® Vista (32 bit):

```
x86
```

Intel x86 + Gentoo Linux (64 bit):

```
Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
armv7l
```

```
1 from platform import processor
2
3 print(processor())
```

Console >_

Selected functions from the `platform` module: continued

A function named `system()` returns the generic OS name as a string.

Our example platforms presented themselves like this:

Intel x86 + Windows ® Vista (32 bit):

```
Windows
```

Intel x86 + Gentoo Linux (64 bit):

```
Linux
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
Linux
```

```
1 from platform import system
2
3 print(system())
```

Console >_

```
Linux
```

Selected functions from the `platform` module: continued

The OS version is provided as a string by the `version()` function.

Run the code and check its output. This is what we got:

Intel x86 + Windows ® Vista (32 bit):

```
6.0.6002
```

Intel x86 + Gentoo Linux (64 bit):

```
#1 SMP PREEMPT Fri Jul 21 22:44:37 CEST 2017
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
#1 SMP Debian 4.4.6-1+rpi14 (2016-05-05)
```

```
1 from platform import version
2
3 print(version())
```

Console >_

```
#206-Ubuntu SMP Fri Feb 28 05:02:04 UTC 2020
```

Selected functions from the `platform` module: continued

If you need to know what version of Python is running your code, you can check it using a number of dedicated functions - here are two of them:

- `python_implementation()` → returns a string denoting the Python implementation (expect `CPython` here, unless you decide to use any non-canonical Python branch)
- `python_version_tuple()` → returns a three-element tuple filled with:
  - the **major** part of Python's version;
  - the **minor** part;
  - the **patch** level number.

Our example program produced the following output:

```
CPython
3
6
4
```

It's very likely that your version of Python will be different.

```
1 from platform import python_implementation, python_version_tuple
2
3 print(python_implementation())
4
5 for atr in python_version_tuple():
6     print(atr)
```

Console >_...

```
CPython
3
7
7
```

Python Module Index

We have only covered the basics of Python modules here. Python's modules make up their own universe, in which Python itself is only a galaxy, and we would venture to say that exploring the depths of these modules can take significantly more time than getting acquainted with "pure" Python.

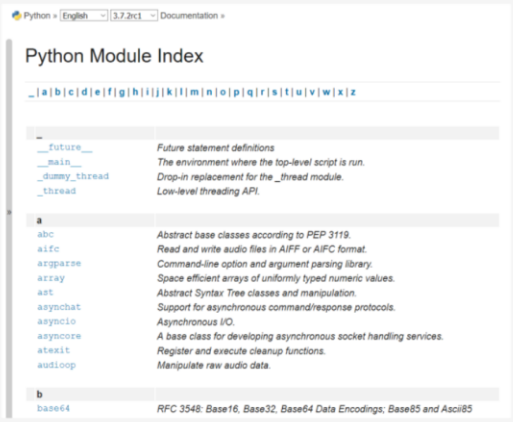
Moreover, the Python community all over the world creates and maintains hundreds of additional modules used in very niche applications like genetics, psychology, or even astrology.

These modules aren't (and won't be) distributed along with Python, or through official channels, which makes the Python universe broader - almost infinite.

You can read about all standard Python modules here: https://docs.python.org/3/py-modindex.html.

Don't worry - you won't need all these modules. Many of them are very specific.

All you need to do is find the modules you want, and teach yourself how to use them. It's easy.

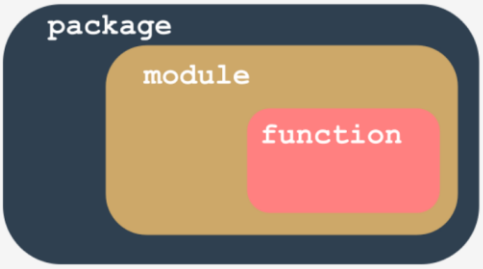


In the next section we'll take a look at something else. We're going to show you how to write your own module.

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

In this section you're going to be working locally on your machine. Let's start from scratch, just like this:

module.py

```
```

You need two files to repeat these experiments. One of them will be the module itself. It's empty now. Don't worry, you're going to fill it with actual code.

We've named the file `module.py`. Not very creative, but simple and clear.

The second file contains the code using the new module. Its name is `main.py`.

Its content is very brief so far:

main.py

```
import module
```

Note: **both files have to be located in the same folder**. We strongly encourage you to create an empty, new folder for both files. Some things will be easier then.

Launch IDLE and run the `main.py` file. What do you see?

You should see nothing. This means that Python has successfully imported the contents of the `module.py` file. It doesn't matter that the module is empty for now. The very first step has been done, but before you take the next step, we want you to take a look into the folder in which both files exist.

Do you notice something interesting?

A new subfolder has appeared - can you see it? Its name is `__pycache__`. Take a look inside. What do you see?

There is a file named (more or less) `module.cpython-xy.pyc` where `x` and `y` are digits derived from your version of Python (e.g., they will be `3` and `4` if you use Python 3.4).

The name of the file is the same as your module's name (module here). The part after the first dot says which Python implementation has created the file (`CPython` here) and its version number. The last part (`pyc`) comes from the words *Python* and *compiled*.

You can look inside the file - the content is completely unreadable to humans. It has to be like that, as the file is intended for Python's use only.

When Python imports a module for the first time, it **translates its contents into a somewhat compiled shape**. The file doesn't contain machine code - it's internal Python **semi-compiled code**, ready to be executed by Python's interpreter. As such a file doesn't require lots of the checks needed for a pure source file, the execution starts faster, and runs faster, too.

Thanks to that, every subsequent import will go quicker than interpreting the source text from scratch.

Python is able to check if the module's source file has been modified (in this case, the `pyc` file will be rebuilt) or not (when the `pyc` file may be run at once). As this process is fully automatic and transparent, you don't have to keep it in mind.

Your first module: continued

Now we've put a little something into the module file:

module.py

```
print("I like to be a module.")
```

Can you notice any differences between a module and an ordinary script? There are none so far.

It's possible to run this file like any other script. Try it for yourself.

What happens? You should see the following line inside your console:

```
I like to be a module.
```

Let's go back to the `main.py` file:

Python can do much more. It also creates a variable called `__name__`.

Moreover, each source file uses its own, separate version of the variable - it isn't shared between modules.

We'll show you how to use it. Modify the module a bit:

module.py

```
print("I like to be a module.")
print(__name__)
```

See code in Sandbox

Now run the `module.py` file. You should see the following lines:

```
I like to be a module
__main__
```

Now run the `main.py` file. And? Do you see the same as us?

main.py

```
import module
```

Run it. What do you see? Hopefully, you see something like this:

```
I like to be a module.
```

What does it actually mean?

When a module is imported, its content is **implicitly executed by Python**. It gives the module the chance to initialize some of its internal aspects (e.g., it may assign some variables with useful values). Note: the **initialization takes place only once**, when the first import occurs, so the assignments done by the module aren't repeated unnecessarily.

```
I like to be a module
module
```

We can say that:

- when you run a file directly, its `__name__` variable is set to `__main__`;
- when a file is imported as a module, its `__name__` variable is set to the file's name (excluding `.py`)

This is how you can make use of the `__main__` variable in order to detect the context in which your code has been activated:

module.py

```
if __name__ == "__main__":
    print("I prefer to be a module")
else:
    print("I like to be a module")
```

- there is a module named `mod1`;
- there is a module named `mod2` which contains the `import mod1` instruction;
- there is a main file containing the `import mod1` and `import mod2` instructions.

At first glance, you may think that `mod1` will be imported twice - fortunately, **only the first import occurs**. Python remembers the imported modules and silently omits all subsequent imports.

There's a cleverer way to utilize the variable, however. If you write a module filled with a number of complex functions, you can use it to place a series of tests to check if the functions work properly.

Each time you modify any of these functions, you can simply run the module to make sure that your amendments didn't spoil the code. These tests will be omitted when the code is imported as a module.

Your first module: continued

This module will contain two simple functions, and if you want to know how many times the functions have been invoked, you need a counter initialized to zero when the module is being imported. You can do it this way:

module.py

```
counter = 0

if __name__ == "__main__":
    print("I prefer to be a module")
else:
    print("I like to be a module")
```

See code in Sandbox

Introducing such a variable is absolutely correct, but may cause important **side effects** that you must be aware of.

Take a look at the modified `main.py` file:

The module is ready:

module.py

```
#!/usr/bin/env python3

""" module.py - an example of Python module """

__counter = 0

def suml(list):
    global __counter
    __counter += 1
    sum = 0
    for el in list:
        sum += el
    return sum

def prodl(list):
    global __counter
    __counter += 1
    prod = 1
    for el in list:
        prod *= el
    return prod

if __name__ == "__main__":
    print("I prefer to be a module, but I can do some tests for you")
    l = [i+1 for i in range(5)]
    print(suml(l) == 15)
    print(prodl(l) == 120)
```

See code in Sandbox

main.py

```
import module
print(module.counter)
```

See code in Sandbox

As you can see, the main file tries to access the module's counter variable. Is this legal? Yes, it is. Is it usable? It may be very usable. Is it safe? That depends - if you trust your module's users, there's no problem; however, you may not want the rest of the world to see your **personal/private variable**.

Unlike many other programming languages, Python has no means of allowing you to hide such variables from the eyes of the module's users. You can only inform your users that this is your variable, that they may read it, but that they should not modify it under any circumstances.

A few elements need some explanation, we think:

- the line starting with `#!` has many names - it may be called *shabang, shebang, hashbang, poundbang* or even *hashpling* (don't ask us why). The name itself means nothing here - its role is more important. From Python's point of view, it's just a **comment** as it starts with `#`. For Unix and Unix-like OSs (including MacOS) such a line **instructs the OS how to execute the contents of the file** (in other words, what program needs to be launched to interpret the text). In some environments (especially those connected with web servers) the absence of that line will cause trouble;
- a string (maybe a multiline) placed before any module instructions (including imports) is called the **doc-string**, and should briefly explain the purpose and contents of the module;
- the functions defined inside the module (`suml()` and `prodl()`) are available for import;
- we've used the `__name__` variable to detect when the file is run stand-alone, and seized this opportunity to perform some simple tests.

This is done by preceding the variable's name with `_` (one underscore) or `__` (two underscores), but remember, it's only a **convention**. Your module's users may obey it or they may not.

Of course, we'll follow the convention. Now let's put two functions into the module - they'll evaluate the sum and product of the numbers collected in a list.

In addition, let's add some ornaments there and remove any superfluous remnants.

Now it's possible to use the new module - this is one way:

main.py

```
from module import suml, prodl

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(suml(zeroes))
print(prodl(ones))
```

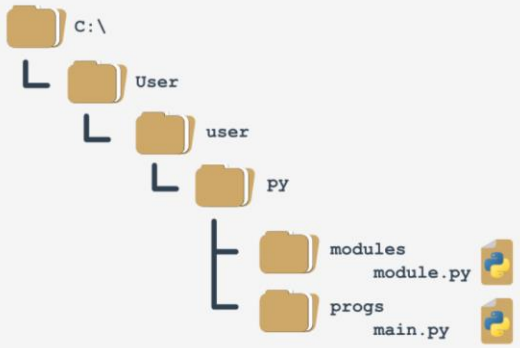
See code in Sandbox

Your first module: continued

It's time to make this example more complicated - we've assumed here 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?

Note: the folder in which the execution starts is listed in the first path's element.

Note once again: there is a zip file listed as one of the path's elements - it's not an error. Python is able to treat zip files as ordinary folders - this can save lots of storage.

Can you figure out how we can solve the problem?

You can solve it by adding a folder containing the module to the path variable (it's fully modifiable).

One of several possible solutions looks like this:

main.py

```
from sys import path

path.append('..\\modules')

import module

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(module.suml(zeroes))
print(module.prodl(ones))
```

See code in Sandbox

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:

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

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

Note:

- we've doubled the `\` inside folder name - do you know why?

  Check

  Because a backslash is used to escape other characters - if you want to get just a backslash, you have to escape it.

- we've used the relative name of the folder - this will work if you start the `main.py` file directly from its home folder, and won't work if the current directory doesn't fit the relative path; you can always use an absolute path, like this:

  ```
  path.append('C:\\Users\\user\\py\\modules')
  ```

- we've used the `append()` method - in effect, the new path will occupy the last element in the path list; if you don't like the idea, you can use `insert()` instead.

Your first package

Imagine that in the not-so-distant future you and your associates write a large number of Python functions.

Your team decides to group the functions in separate modules, and this is the final result of the ordering:

alpha.py

```
#! /usr/bin/env python3

""" module: alpha """

def funA():
 return "Alpha"

if __name__ == "__main__":
 print("I prefer to be a module")
```

beta.py

```
def funB(): ...
```

iota.py

```
def funI(): ...
```

sigma.py

```
def funS(): ...
```

tau.py

```
def funT(): ...
```

psi.py

```
def funP(): ...
```

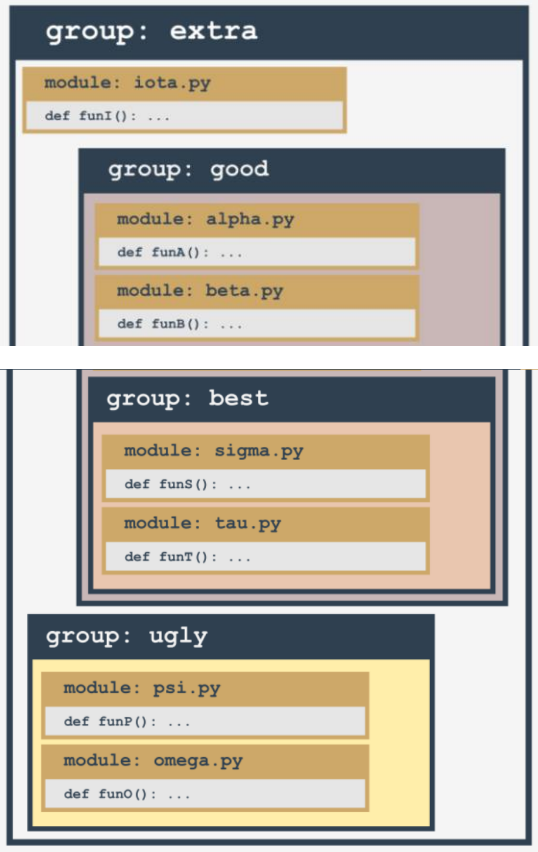
omega.py

```
def funO(): ...
```

Note: we've presented the whole content for the omega module only - assume that all the modules look similar (they contain one function named `funX`, where `X` is the first letter of the module's name).

Suddenly, somebody notices that these modules form their own hierarchy, so putting them all in a flat structure won't be a good idea.

After some discussion, the team comes to the conclusion that the modules have to be grouped. All participants agree that the following tree structure perfectly reflects the mutual relationships between the modules:



Let's review this from the bottom up:

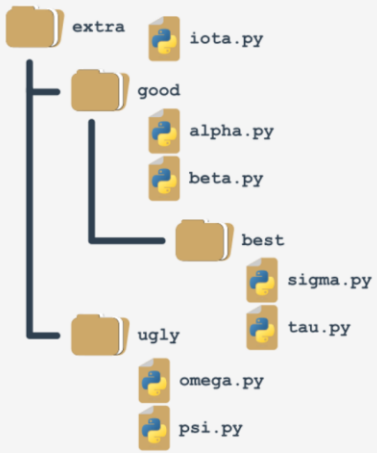
- the `ugly` group contains two modules: `psi` and `omega`;
- the `best` group contains two modules: `sigma` and `tau`;
- the `good` group contains two modules (`alpha` and `beta`) and one subgroup (`best`)
- the `extra` group contains two subgroups (`good` and `bad`) and one module (`iota`)

Does it look bad? Not at all - analyze the structure carefully. It resembles something, doesn't it?

It looks like a directory structure.

Your first package: continued

This is how the tree currently looks:



There are two questions to answer:

- how do you transform such a tree (actually, a subtree) into a real Python package (in other words, how do you convince Python that such a tree is not just a bunch of junk files, but a set of modules)?
- where do you put the subtree to make it accessible to Python?

The first question has a surprising answer: **packages, like modules, may require initialization**.

The initialization of a module is done by an unbound code (not a part of any function) located inside the module's file. As a package is not a file, this technique is useless for initializing packages.

You need to use a different trick instead - Python expects that there is a file with a very unique name inside the package's folder: `__init__.py`.

The content of the file is executed when any of the package's modules is imported. If you don't want any special initializations, you can leave the file empty, but you mustn't omit it.

Such a structure is almost a package (in the Python sense). It lacks the fine detail to be both functional and operative. We'll complete it in a moment.

If you assume that `extra` is the name of a **newly created package** (think of it as the **package's root**), it will impose a naming rule which allows you to clearly name every entity from the tree.

For example:

- the location of a function named `funT()` from the `tau` package may be described as:

```
extra.good.best.tau.funT()
```

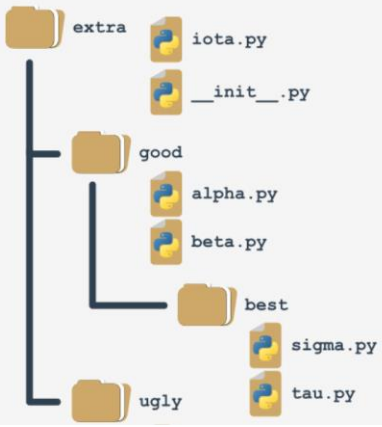
- a function marked as:

```
extra.ugly.psi.funP()
```

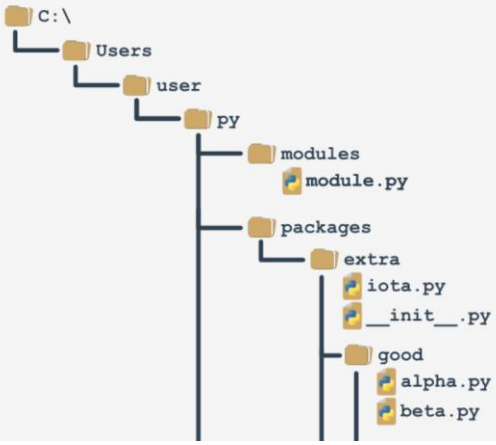
comes from the `psi` module being stored in the `ugly` subpackage of the `extra` package.

Your first package: continued

The presence of the `__init.py__` file finally makes up the package:



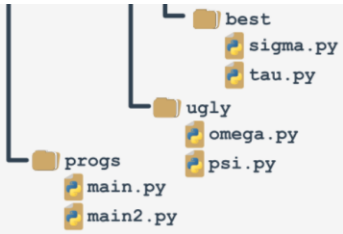
Let's assume that the working environment looks as follows:



Note: it's not only the *root* folder that can contain `__init.py__` file - you can put it inside any of its subfolders (subpackages) too. It may be useful if some of the subpackages require individual treatment and special kinds of initialization.

Now it's time to answer the second question - the answer is simple: **anywhere**. You only have to ensure that Python is aware of the package's location. You already know how to do that.

You're ready to make use of your first package.



We've prepared a zip file containing all the files from the packages branch. You can download it and use it for your own experiments, but remember to unpack it in the folder presented in the scheme, otherwise, it won't be accessible to the code from the main file.

DOWNLOAD Modules and Packages ZIP file

You'll be continuing your experiments using the `main2.py` file.

Your first package: continued

We are going to access the `funI()` function from the `iota` module from the top of the `extra` package. It forces us to use qualified package names (associate this with naming folders and subfolders - the conventions are very similar).

This is how to do it:

```
main2.py

from sys import path

path.append('..\\packages')

import extra.iota

print(extra.iota.funI())
```

See code in Sandbox

Note:

- we've modified the `path` variable to make it accessible to Python;
- the `import` doesn't point directly to the module, but specifies the fully qualified path from the top of the package;

replacing `import extra.iota` with `import iota` will cause an error.

The following variant is valid too:

```
main2.py

from sys import path

path.append('..\\packages')

from extra.iota import funI

print(funI())
```

See code in Sandbox

Note the qualified name of the `iota` module.

Your first package: continued

Now let's reach all the way to the bottom of the tree - this is how to get access to the `sigma` and `tau` modules.

```
main2.py

from sys import path

path.append('..\\packages')

import extra.good.best.sigma
from extra.good.best.tau import funT

print(extra.good.best.sigma.funS())
print(funT())
```

See code in Sandbox

You can make your life easier by using aliasing:

Let's assume that we've zipped the whole subdirectory, starting from the `extra` folder (including it), and let's get a file named `extrapack.zip`. Next, we put the file inside the `packages` folder.

Now we are able to use the zip file in a role of packages:

```
main2.py

from sys import path

path.append('..\\packages')

import extra.good.best.sigma as sig
import extra.good.alpha as alp

print(sig.funS())
print(alp.funA())
```

See code in Sandbox

main2.py

```
from sys import path

path.append('..\\packages')

import extra.good.best.sigma as sig
import extra.good.alpha as alp

print(sig.funS())
print(alp.funA())
```

See code in Sandbox

If you want to conduct your own experiments with the package we've created, you can download it below. We encourage you to do so.

DOWNLOAD Extrapack ZIP file

Now you can create modules and combine them into packages. It's time to start a completely different discussion - about errors, failures and crashes.

Errors, failures, and other plagues

Anything that can go wrong, will go wrong.

This is Murphy's law, and it works everywhere and always. Your code's execution can go wrong, too. If it can, it will.

Look the code in the editor. There are at least two possible ways it can "go wrong". Can you see them?

- As a user is able to enter a completely arbitrary string of characters, **there is no guarantee that the string can be converted into a float value** - this is the first vulnerability of the code;
- the second is that the `sqrt()` **function fails if it gets a negative argument**.

You may get one of the following error messages.

Something like this:

```
Enter x: Abracadabra

Traceback (most recent call last):

  File "sqrt.py", line 3, in

    x = float(input("Enter x: "))

ValueError: could not convert string to float: 'Abracadabra'
```

Or something like this:

```
Enter x: -1

Traceback (most recent call last):

  File "sqrt.py", line 4, in

    y = math.sqrt(x)

ValueError: math domain error
```

Can you protect yourself from such surprises? Of course you can. Moreover, you have to do it in order to be considered a good programmer.

```
1 import math
2
3 x = float(input("Enter x: "))
4 y = math.sqrt(x)
5
6 print("The square root of", x, "equals to", y)
```

Console >_

```
Enter x:
Enter x: kh
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    x = float(input("Enter x: "))
ValueError: could not convert string to float: 'kh'
Enter x: -9
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    y = math.sqrt(x)
ValueError: math domain error
```

Console >_

```
Enter x:
Enter x: kh
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    x = float(input("Enter x: "))
ValueError: could not convert string to float: 'kh'
Enter x: -9
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    y = math.sqrt(x)
ValueError: math domain error
```

Exceptions

Each time your code tries to do something wrong/foolish/irresponsible/crazy/unenforceable, Python does two things:

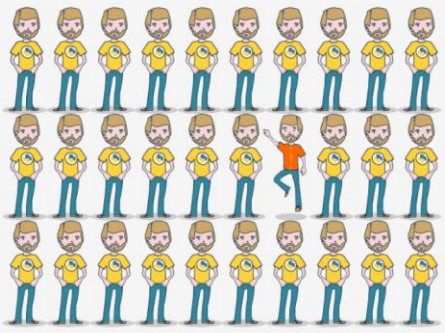
- it **stops your program**;
- it creates a special kind of data, called an **exception**.

Both of these activities are called **raising an exception**. We can say that Python always raises an exception (or that an **exception has been raised**) when it has no idea what do to with your code.

What happens next?

- the raised exception expects somebody or something to notice it and take care of it;
- if nothing happens to take care of the raised exception, the program will be **forcibly terminated**, and you will see an **error message** sent to the console by Python;
- otherwise, if the exception is taken care of and **handled** properly, the suspended program can be resumed and its execution can continue.

Python provides effective tools that allow you to **observe exceptions, identify them and handle them** efficiently. This is possible due to the fact that all potential exceptions have their unambiguous names, so you can categorize them and react appropriately.



You know some exception names already.

Take a look at the following diagnostic message:

```
ValueError: math domain error
```

The word in red is just the **exception name**. Let's get familiar with some other exceptions.

Exceptions: continued

Look at the code in the editor. Run the (obviously incorrect) program.

You will see the following message in reply:

```
Traceback (most recent call last):
File "div.py", line 2, in
value /= 0
ZeroDivisionError: division by zero
```

This exception error is called `ZeroDivisionError`.

```
value = 1
value /= 0
```

Console >_

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    value /= 0
ZeroDivisionError: division by zero
```

Exceptions: continued

Look at the code in the editor. What will happen when you run it? Check.

You will see the following message in reply:

```
Traceback (most recent call last):
File "lst.py", line 2, in
x = list[0]
IndexError: list index out of range
```

This is the `IndexError`.

```

list = []
x = list[0]
```

Console >_

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    x = list[0]
IndexError: list index out of range
```

Exceptions: continued

How do you **handle** exceptions? The word `try` is key to the solution.

What's more, it's a keyword, too.

The recipe for success is as follows:

- first, you have to ***try* to do something**;
- next, you have to **check whether everything went well**.

But wouldn't it be better to check all circumstances first and then do something only if it's safe?

Just like the example in the editor.

Admittedly, this way may seem to be the most natural and understandable, but in reality, this method doesn't make programming any easier. All these checks can make **your code bloated and illegible**.

Python prefers a completely different approach.

```
firstNumber = int(input("Enter the first number: "))
secondNumber = int(input("Enter the second number: "))

if secondNumber != 0:
    print(firstNumber / secondNumber)
else:
    print("This operation cannot be done.")

print("THE END.")
```

Console >_

```
Enter the first number: 2
Enter the second number: 4
0.5
THE END.
Enter the first number: 3
Enter the second number: 0
This operation cannot be done.
THE END.
```

Exceptions: continued

Look at the code in the editor. This is the favorite Python approach.

Note:

- the `try` keyword **begins a block of the code** which may or may not be performing correctly;
- next, Python tries to perform the risky action; if it fails, an exception is raised and Python starts to look for a solution;
- the `except` keyword starts a piece of code which will be **executed if anything inside the** `try` **block goes wrong** - if an exception is raised inside a previous `try` block, **it will fail here**, so the code located after the except keyword should provide an **adequate reaction** to the raised exception;
- returning to the previous nesting level ends the **try-except** section.

Run the code and test its behavior.

Let's summarize this:

```
try:
    :
    :
except:
    :
    :
```

- in the first step, Python tries to perform all instructions placed between the `try:` and `except:` statements;
- if nothing is wrong with the execution and all instructions are performed successfully, the execution jumps to the point after the last line of the `except:` block, and the block's execution is considered complete;
- if anything goes wrong inside the `try:` and `except:` block, the execution immediately jumps out of the block and into the first instruction located after the `except:` keyword; this means that some of the instructions from the block may be silently omitted.

```
firstNumber = int(input("Enter the first number: "))
secondNumber = int(input("Enter the second number: "))

try:
    print(firstNumber / secondNumber)
except:
    print("This operation cannot be done.")

print("THE END.")
```

Console >_

```
Enter the first number: 23
Enter the second number: 24
0.9583333333333334
THE END.
Enter the first number: 3
Enter the second number: 0
This operation cannot be done.
THE END.
```

Exceptions: continued

Look at the code in the editor. It will help you understand this mechanism.

This is the output it produces:

```
1
Oh dear, something went wrong...
3
```

Note: the `print("2")` instruction was lost in the process.

```
try:
    print("1")
    x = 1 / 0
    print("2")
except:
    print("Oh dear, something went wrong...")

print("3")
```

Console >_

```
1
Oh dear, something went wrong...
3
```

Exceptions: continued

This approach has one important disadvantage - if there is a possibility that more than one exception may skip into an `except:` branch, you may have **trouble figuring out what actually happened**.

Just like in our code in the editor. Run it and see what happens.

The message: `Oh dear, something went wrong...` appearing in the console says nothing about the reason, while there are two possible causes of the exception:

- non-integer data entered by the user;
- an integer value equal to `0` assigned to the `x` variable.

Technically, there are two ways to solve the issue:

- build two consecutive `try-except` blocks, one for each possible exception reason (easy, but will cause unfavorable code growth)
- use a more advanced variant of the instruction.

It looks like this:

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :
```

This is how it works:

- if the `try` branch raises the `exc1` exception, it will be handled by the `except exc1:` block;
- similarly, if the `try` branch raises the `exc2` exception, it will be handled by the `except exc2:` block;
- if the `try` branch raises any other exception, it will be handled by the unnamed `except` block.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
except:
    print("Oh dear, something went wrong...")

print("THE END.")
```

Console >_

```
Enter a number: 9
THE END.
Enter a number: 0
Oh dear, something went wrong...
THE END.
```

Exceptions: continued

Look at the code in the editor. Our solution is there.

The code, when run, produces one of the following four variants of output:

- if you enter a valid, non-zero integer value (e.g., `5`) it says:

```
0.2
THE END.
```

- if you enter `0`, it says:

```
You cannot divide by zero, sorry.
THE END.
```

- if you enter any non-integer string, you see:

```
You must enter an integer value.
THE END.
```

- (locally on your machine) if you press Ctrl-C while the program is waiting for the user's input (which causes an exception named `KeyboardInterrupt`), the program says:

```
Oh dear, something went wrong...
THE END.
```

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ZeroDivisionError:
    print("You cannot divide by zero, sorry.")
except ValueError:
    print("You must enter an integer value.")
except:
    print("Oh dear, something went wrong...")

print("THE END.")
```

Console >_

```
7
Enter a number:
Enter a number: 65
0.015384615384615385
THE END.
Enter a number: 8.6
You must enter an integer value.
THE END.
Enter a number: 0
You cannot divide by zero, sorry.
THE END.
```

Exceptions: continued

Don't forget that:

- the `except` branches are searched in the same order in which they appear in the code;
- you must not use more than one except branch with a certain exception name;
- the number of different `except` branches is arbitrary - the only condition is that if you use `try`, you must put at least one `except` (named or not) after it;
- the `except` keyword must not be used without a preceding `try`;
- if any of the `except` branches is executed, no other branches will be visited;
- if none of the specified `except` branches matches the raised exception, the exception remains unhandled (we'll discuss it soon)
- if an unnamed `except` branch exists (one without an exception name), it has to be specified as the last.

```
try:
    :
except exc1:
    :
except exc2:
    :
except:
    :
```

Let's continue the experiments now.

Look at the code in the editor. We've modified the previous program - we've removed the `ZeroDivisionError` branch.

What happens now if the user enters `0` as an input?

As there are **no dedicated branches** for division by zero, the raised exception falls into the **general (unnamed) branch**; this means that in this case, the program will say:

```
Oh dear, something went wrong...
THE END.
```

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ValueError:
    print("You must enter an integer value.")
except:
    print("Oh dear, something went wrong...")

print("THE END.")
```

Console >_

```
Enter a number: 45
0.022222222222222223
THE END.
Enter a number: 0
Oh dear, something went wrong...
THE END.
```

Exceptions: continued

Let's spoil the code once again.

Look at the program in the editor. This time, we've removed the unnamed branch.

The user enters `0` once again and:

- the exception raised won't be handled by `ValueError` - it has nothing to do with it;
- as there's no other branch, you should to see this message:

```
Traceback (most recent call last):
File "exc.py", line 3, in
y = 1 / x
ZeroDivisionError: division by zero
```

You've learned a lot about exception handling in Python. In the next section, we will focus on Python built-in exceptions and their hierarchies.

```
try:
    x = int(input("Enter a number: "))
    y = 1 / x
    print(y)
except ValueError:
    print("You must enter an integer value.")

print("THE END.")
```

Console >_

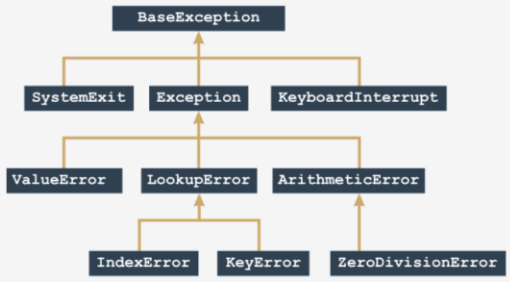
```
Enter a number: 9
0.1111111111111111
THE END.
Enter a number: 0
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    y = 1 / x
ZeroDivisionError: division by zero
Enter a number: 9.34243
You must enter an integer value.
THE END.
```

Exceptions

Python 3 defines **63 built-in exceptions**, and all of them form a **tree-shaped hierarchy**, although the tree is a bit weird as its root is located on top.

Some of the built-in exceptions are more general (they include other exceptions) while others are completely concrete (they represent themselves only). We can say that **the closer to the root an exception is located, the more general (abstract) it is**. In turn, the exceptions located at the branches' ends (we can call them **leaves**) are concrete.

Take a look at the figure:



It shows a small section of the complete exception tree. Let's begin examining the tree from the `ZeroDivisionError` leaf.

Note:

- `ZeroDivisionError` is a special case of more a general exception class named `ArithmeticError`;
- `ArithmeticError` is a special case of a more general exception class named just `Exception`;
- `Exception` is a special case of a more general class named `BaseException`;

We can describe it in the following way (note the direction of the arrows - they always point to the more general entity):

BaseException
↑
Exception
↑
ArithmeticError
↑
ZeroDivisionError

We're going to show you how this generalization works. Let's start with some really simple code.

Exceptions: continued

Look at the code in the editor. It is a simple example to start with. Run it.

The output we expect to see looks like this:

```
Oooppsss...
THE END.
```

Now look at the code below:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Oooppsss...")

print("THE END.")
```

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Oooppsss...")

print("THE END.")
```

Console >_

```
Oooppsss...
THE END.
THE END.
```

Something has changed in it - we've replaced `ZeroDivisionError` with `ArithmeticError`.

You already know that `ArithmeticError` is a general class including (among others) the `ZeroDivisionError` exception.

Thus, the code's output remains unchanged. Test it.

This also means that replacing the exception's name with either `Exception` or `BaseException` won't change the program's behavior.

Let's summarize:

- each exception raised **falls into the first matching branch**;
- the matching branch doesn't have to specify the same exception exactly - it's enough that the exception is **more general** (more abstract) than the raised one.

```
Oooppsss...
THE END.
THE END.
```

Exceptions: continued

Look at the code in the editor. What will happen here?

The first matching branch is the one containing `ZeroDivisionError`. It means that the console will show:

```
Zero division!
THE END.
```

Will it change anything if we swap the two `except` branches around? Just like here below:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Arithmetic problem!")
except ZeroDivisionError:
    print("Zero Division!")

print("THE END.")
```

The change is radical - the code's output is now:

```
try:
    y = 1 / 0
except ZeroDivisionError:
    print("Zero Division!")
except ArithmeticError:
    print("Arithmetic problem!")

print("THE END.")
```

Console >_

```
Zero Division!
THE END.
THE END.
```

The change is radical - the code's output is now:

```
Arithmetic problem!
THE END.
```

Why, if the exception raised is the same as previously?

The exception is the same, but the more general exception is now listed first - it will catch all zero divisions too. It also means that there's no chance that any exception hits the `ZeroDivisionError` branch. This branch is now completely unreachable.

Remember:

- the order of the branches matters!
- don't put more general exceptions before more concrete ones;
- this will make the latter one unreachable and useless;
- moreover, it will make your code messy and inconsistent;
- Python won't generate any error messages regarding this issue.

Exceptions: continued

If you want to **handle two or more exceptions** in the same way, you can use the following syntax:

```
try:
    :
except (exc1, exc2):
    :
```

You simply have to put all the engaged exception names into a comma-separated list and not to forget the parentheses.

If an **exception is raised inside a function**, it can be handled:

- inside the function;
- outside the function;

Let's start with the first variant - look at the code in the editor.

```
def badFun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("Arithmetic Problem!")
    return None

badFun(0)

print("THE END.")
```

Console >_

```
Arithmetic Problem!
THE END.
```

The `ZeroDivisionError` exception (being a concrete case of the `ArithmeticError` exception class) is raised inside the `badfun()` function, and it doesn't leave the function - the function itself takes care of it.

The program outputs:

```
Arithmetic problem!
THE END.
```

It's also possible to let the exception propagate **outside the function**. Let's test it now.

Look at the code below:

```
def badFun(n):
    return 1 / n

try:
    badFun(0)
except ArithmeticError:
    print("What happened? An exception was raised!")

print("THE END.")
```

```
def badFun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("Arithmetic Problem!")
    return None

badFun(0)

print("THE END.")
```

Console >_

```
Arithmetic Problem!
THE END.
```

The problem has to be solved by the invoker (or by the invoker's invoker, and so on).

The program outputs:

```
What happened? An exception was raised!
THE END.
```

Note: the **exception raised can cross function and module boundaries**, and travel through the invocation chain looking for a matching `except` clause able to handle it.

If there is no such clause, the exception remains unhandled, and Python solves the problem in its standard way - **by terminating your code and emitting a diagnostic message**.

Now we're going to suspend this discussion, as we want to introduce you to a brand new Python instruction.

Console >_

```
Arithmetic Problem!
THE END.
```

Exceptions: continued

The `raise` instruction raises the specified exception named `exc` as if it was raised in a normal (natural) way:

```
raise exc
```

Note: `raise` is a keyword.

The instruction enables you to:

- **simulate raising actual exceptions** (e.g., to test your handling strategy)
- partially **handle an exception** and make another part of the code responsible for completing the handling (separation of concerns).

Look at the code in the editor. This is how you can use it in practice.

The program's output remains unchanged.

In this way, you can **test your exception handling routine** without forcing the code to do stupid things.

```
def badFun(n):
    raise ZeroDivisionError

try:
    badFun(0)
except ArithmeticError:
    print("What happened? An error?")

print("THE END.")
```

Console >_

```
What happened? An error?
THE END.
What happened? An error?
THE END.
```

Exceptions: continued

The `raise` instruction may also be utilized in the following way (note the absence of the exception's name):

```
raise
```

There is one serious restriction: this kind of `raise` instruction may be used **inside the** `except` **branch** only; using it in any other context causes an error.

The instruction will immediately re-raise the same exception as currently handled.

Thanks to this, you can distribute the exception handling among different parts of the code.

Look at the code in the editor. Run it - we'll see it in action.

The `ZeroDivisionError` is raised twice:

- first, inside the `try` part of the code (this is caused by actual zero division)
- second, inside the `except` part by the `raise` instruction.

In effect, the code outputs:

```
I did it again!
I see!
THE END.
```

```
def badFun(n):
    try:
        return n / 0
    except:
        print("I did it again!")
        raise

try:
    badFun(0)
except ArithmeticError:
    print("I see!")

print("THE END.")
```

Console >_

```
I did it again!
I see!
THE END.
```

Exceptions: continued

Now is a good moment to show you another Python instruction, named `assert`. This is a keyword.

```
assert expression
```

How does it work?

- It evaluates the expression;
- if the expression evaluates to `True`, or a non-zero numerical value, or a non-empty string, or any other value different than `None`, it won't do anything else;
- otherwise, it automatically and immediately raises an exception named `AssertionError` (in this case, we say that the assertion has failed)

How it can be used?

- you may want to put it into your code where you want to be **absolutely safe from evidently wrong data**, and where you aren't absolutely sure that the data has been carefully examined before (e.g., inside a function used by someone else)
- raising an `AssertionError` exception secures your code from producing invalid results, and clearly shows the nature of the failure;
- **assertions don't supersede exceptions or validate the data** - they are their supplements.

```
import math

x = float(input("Enter a number: "))
assert x >= 0.0

x = math.sqrt(x)

print(x)
```

Console >_

```
Enter a number: 6
2.449489742783178
```

If exceptions and data validation are like careful driving, assertion can play the role of an airbag.

Let's see the `assert` instruction in action. Look at the code in the editor. Run it.

The program runs flawlessly if you enter a valid numerical value greater than or equal to zero; otherwise, it stops and emits the following message:

```
Traceback (most recent call last):
  File ".main.py", line 4, in
    assert x >= 0.0
AssertionError
```

Console >_

```
Enter a number: 6
2.449489742783178
```

Built-in exceptions

We're going to show you a short list of the most useful exceptions. While it may sound strange to call "useful" a thing or a phenomenon which is a visible sign of failure or setback, as you know, to err is human and if anything can go wrong, it will go wrong.

Exceptions are as routine and normal as any other aspect of a programmer's life.

For each exception, we'll show you:

- its name;
- its location in the exception tree;
- a short description;
- a concise snippet of code showing the circumstances in which the exception may be raised.

There are lots of other exceptions to explore - we simply don't have the space to go through them all here.

ArithmeticError

Location:

`BaseException ← Exception ← ArithmeticError`

Description:

an abstract exception including all exceptions caused by arithmetic operations like zero division or an argument's invalid domain

AssertionError

Location:

`BaseException ← Exception ← AssertionError`

Description:

a concrete exception raised by the assert instruction when its argument evaluates to `False`, `None`, `0`, or an empty string

Code:

```
from math import tan, radians
angle = int(input('Enter integral angle in degrees: '))

# we must be sure that angle != 90 + k * 180
assert angle % 180 != 90
print(tan(radians(angle)))
```

BaseException

Location:

`BaseException`

Description:

the most general (abstract) of all Python exceptions - all other exceptions are included in this one; it can be said that the following two `except` branches are equivalent: `except:` and `except BaseException:`.

IndexError

Location:

`BaseException ← Exception ← LookupError ← IndexError`

Description:

a concrete exception raised when you try to access a non-existent sequence's element (e.g., a list's element)

Code:

```
# the code shows an extravagant way
# of leaving the loop

list = [1, 2, 3, 4, 5]
ix = 0
doit = True

while doit:
    try:
        print(list[ix])
        ix += 1
    except IndexError:
        doit = False

print('Done')
```

KeyboardInterrupt

Location:

`BaseException ← KeyboardInterrupt`

Description:

a concrete exception raised when the user uses a keyboard shortcut designed to terminate a program's execution (*Ctrl-C* in most OSs); if handling this exception doesn't lead to program termination, the program continues its execution. Note: this exception is not derived from the `Exception` class. Run the program in IDLE.

Code:

```
# this code cannot be terminated
# by pressing Ctrl-C

from time import sleep
seconds = 0
while True:
    try:
        print(seconds)
        seconds += 1
        sleep(1)
    except KeyboardInterrupt:
        print("Don't do that!")
```

MemoryError

Location:

`BaseException ← Exception ← MemoryError`

Description:

a concrete exception raised when an operation cannot be completed due to a lack of free memory

Code:

```
# this code causes the MemoryError exception
# warning: executing this code may be crucial
# for your OS
# don't run it in production environments!

string = 'x'
try:
    while True:
        string = string + string
        print(len(string))
except MemoryError:
    print('This is not funny!')
```

OverflowError

Location:

`BaseException ← Exception ← ArithmeticError ← OverflowError`

Description:

a concrete exception raised when an operation produces a number too big to be successfully stored

Code:

```
# the code prints subsequent
# values of exp(k), k = 1, 2, 4, 8, 16, ...

from math import exp
ex = 1
try:
    while True:
        print(exp(ex))
        ex *= 2
except OverflowError:
    print('The number is too big.')
```

LookupError

Location:

`BaseException ← Exception ← LookupError`

Description:

an abstract exception including all exceptions caused by errors resulting from invalid references to different collections (lists, dictionaries, tuples, etc.)

https://docs.python.org/3.6/library/exceptions.html.

ImportError

Location:

```
BaseException ← Exception ← StandardError ← ImportError
```

Description:

a concrete exception raised when an import operation fails

Code:

```
# one of this imports will fail - which one?

try:
    import math
    import time
    import abracadabra

except:
    print('One of your imports has failed.')
```

KeyError

Location:

```
BaseException ← Exception ← LookupError ← KeyError
```

Description:

a concrete exception raised when you try to access a collection's non-existent element (e.g., a dictionary's element)

Code:

```
# how to abuse the dictionary
# and how to deal with it

dict = { 'a' : 'b', 'b' : 'c', 'c' : 'd' }
ch = 'a'
try:
    while True:
        ch = dict[ch]
        print(ch)
except KeyError:
    print('No such key:', ch)
```

We are done with exceptions for now, but they'll return when we discuss object-oriented programming in Python. You can use them to protect your code from bad accidents, but you also have to learn how to dive into them, exploring the information they carry.

Exceptions are in fact objects - however, we can tell you nothing about this aspect until we present you with classes, objects, and the like.

For the time being, if you'd like to learn more about exceptions on your own, you look into Standard Python Library at https://docs.python.org/3.6/library/exceptions.html.

LAB

Estimated time

15-25 minutes

Level of difficulty

Medium

Objectives

- improving the student's skills in defining functions;
- using exceptions in order to provide a safe input environment.

Scenario

Your task is to write a **function able to input integer values and to check if they are within a specified range**.

The function should:

- accept three arguments: a prompt, a low acceptable limit, and a high acceptable limit;
- if the user enters a string that is not an integer value, the function should emit the message `Error: wrong input`, and ask the user to input the value again;
- if the user enters a number which falls outside the specified range, the function should emit the message `Error: the value is not within permitted range (min..max)` and ask the user to input the value again;
- if the input value is valid, return it as a result.

```
def readint(prompt, min, max):
    ok = False
    while not ok:
        try:
            value = int(input(prompt))
            ok = True
        except ValueError:
            print("Error: wrong input")
        if ok:
            ok = value >= min and value <= max
        if not ok:
            print("Error: the value is not within permitted range ("
    return value;

v = readint("Enter a number from -10 to 10: ", -10, 10)

print("The number is:", v)
```

Console >_

```
Enter a number from -10 to 10: 78
Error: the value is not within permitted range (-10..10)
Enter a number from -10 to 10:
Enter a number from -10 to 10: qwr
Error: wrong input
Error: the value is not within permitted range (-10..10)
Enter a number from -10 to 10: 1
The number is: 1
```

Test data

Test your code carefully.

This is how the function should react to the user's input:

```
Enter a number from -10 to 10: 100
Error: the value is not within permitted range (-10..10)
Enter a number from -10 to 10: asd
Error: wrong input
Enter number from -10 to 10: 1
The number is: 1
```


```
Enter a number from -10 to 10: 78
Error: the value is not within permitted range (-10..10)
Enter a number from -10 to 10:
Enter a number from -10 to 10: qwr
Error: wrong input
Error: the value is not within permitted range (-10..10)
Enter a number from -10 to 10: 1
The number is: 1
```

How computers understand single characters

You've written some interesting programs since you've started this course, but all of them have processed only one kind of data - numbers. As you know (you can see this everywhere around you) lots of computer data are not numbers: first names, last names, addresses, titles, poems, scientific papers, emails, court judgements, love confessions, and much, much more.

All these data must be stored, input, output, searched, and transformed by contemporary computers just like any other data, no matter if they are single characters or multi-volume encyclopedias.

How is it possible?

How can you do it in Python? This is what we'll discuss now. Let's start with how computers understand single characters.

Computers store characters as numbers. Every character used by a computer corresponds to a unique number, and vice versa. This assignment must include more characters than you might expect. Many of them are invisible to humans, but essential to computers.

Some of these characters are called **whitespaces**, while others are named **control characters**, because their purpose is to control input/output devices.

Some of these characters are called **whitespaces**, while others are named **control characters**, because their purpose is to control input/output devices.

An example of a whitespace that is completely invisible to the naked eye is a special code, or a pair of codes (different operating systems may treat this issue differently), which are used to mark the ends of the lines inside text files.

People do not see this sign (or these signs), but are able to observe the effect of their application where the lines are broken.

We can create virtually any number of character-number assignments, but life in a world in which every type of computer uses a different character encoding would not be very convenient. This system has led to a need to introduce a universal and widely accepted standard implemented by (almost) all computers and operating systems all over the world.

The one named **ASCII** (short for **American Standard Code for Information Interchange**) is the most widely used, and you can assume that nearly all modern devices (like computers, printers, mobile phones, tablets, etc.) use that code.

The code provides space for **256 different characters**, but we are interested only in the first 128. If you want to see how the code is constructed, look at the table below. Click the table to enlarge it. Look at it carefully - there are some interesting facts. Look at the code of the most common character - the *space*. This is *32*.

Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111
(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113
(DC2)	18	2	50	R	82	r	114
(DC3)	19	3	51	S	83	s	115
(DC4)	20	4	52	T	84	t	116
(NAK)	21	5	53	U	85	u	117
(SYN)	22	6	54	V	86	v	118
(ETB)	23	7	55	W	87	w	119
(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

Now check the code of the lower-case letter *a*. This is *97*. And now find the upper-case *A*. Its code is *65*. Now work out the difference between the code of *a* and *A*. It is equal to *32*. That's the code of a *space*. Interesting, isn't it?

Also note that the letters are arranged in the same order as in the Latin alphabet.

I18N

Of course, the Latin alphabet is not sufficient for the whole of mankind. Users of that alphabet are in the minority. It was necessary to come up with something more flexible and capacious than ASCII, something able to make all the software in the world amenable to **internationalization**, because different languages use completely different alphabets, and sometimes these alphabets are not as simple as the Latin one.

The word *internationalization* is commonly shortened to **I18N**.

I18N

INTERNATIONALIZATION

Why? Look carefully - there is an *I* at the front of the word, next there are *18* different letters, and an *N* at the end.

Despite the slightly humorous origin, the term is officially used in many documents and standards.

The **software I18N** is a standard in present times. Each program has to be written in a way that enables it to be used all around the world, among different cultures, languages and alphabets.

A classic form of ASCII code uses eight bits for each sign. Eight bits mean 256 different characters. The first 128 are used for the standard Latin alphabet (both upper-case and lower-case characters). Is it possible to push all the other national characters used around the world into the remaining 128 locations?

No. It isn't.

Code points and code pages

We need a new term now: a **code point**.

A code point is **a number which makes a character**. For example, *32* is a code point which makes a *space* in ASCII encoding. We can say that standard ASCII code consists of 128 code points.

As standard ASCII occupies 128 out of 256 possible code points, you can only make use of the remaining 128.

It's not enough for all possible languages, but it may be sufficient for one language, or for a small group of similar languages.

Can you **set the higher half of the code points differently for different languages**? Yes, you can. Such a concept is called a **code page**.

A code page is a **standard for using the upper 128 code points to store specific national characters**. For example, there are different code pages for Western Europe and Eastern Europe, Cyrillic and Greek alphabets, Arabic and Hebrew languages, and so on.

This means that the one and same code point can make different characters when used in different code pages.

For example, the code point *200* makes Č (a letter used by some Slavic languages) when utilized by the ISO/IEC 8859-2 code page, and makes Ш (a Cyrillic letter) when used by the ISO/IEC 8859-5 code page.

In consequence, to determine the meaning of a specific code point, you have to know the target code page.

In other words, the code points derived from code the page concept are ambiguous.

Unicode

Code pages helped the computer industry to solve I18N issues for some time, but it soon turned out that they would not be a permanent solution.

The concept that solved the problem in the long term was **Unicode**.

Unicode assigns unique (unambiguous) characters (letters, hyphens, ideograms, etc.) to more than a million code points. The first 128 Unicode code points are identical to ASCII, and the first 256 Unicode code points are identical to the ISO/IEC 8859-1 code page (a code page designed for western European languages).

UCS-4

The Unicode standard says nothing about how to code and store the characters in the memory and files. It only names all available characters and assigns them to planes (a group of characters of similar origin, application, or nature).

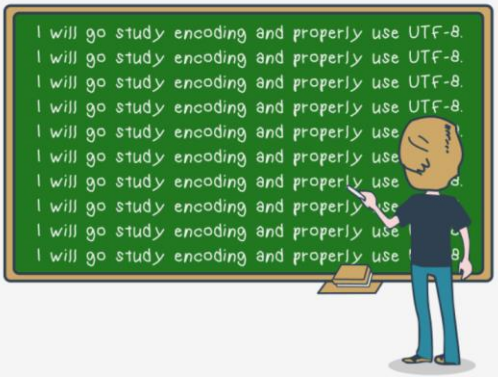
As you can see, UCS-4 is a rather wasteful standard - it increases a text's size by four times compared to standard ASCII. Fortunately, there are smarter forms of encoding Unicode texts.

UTF-8

One of the most commonly used is **UTF-8**.

The name is derived from **Unicode Transformation Format**.

The concept is very smart. **UTF-8 uses as many bits for each of the code points as it really needs to represent them**.



UCS-4

32 bits to store each character

There is more than one standard describing the techniques used to implement Unicode in actual computers and computer storage systems. The most general of them is **UCS-4**.

The name comes from **Universal Character Set**.

UCS-4 uses 32 bits (four bytes) to store each character, and the code is just the Unicode code points' unique number. A file containing UCS-4 encoded text may start with a BOM (byte order mark), an unprintable combination of bits announcing the nature of the file's contents. Some utilities may require it.

For example:

- all Latin characters (and all standard ASCII characters) occupy eight bits;
- non-Latin characters occupy 16 bits;
- CJK (China-Japan-Korea) ideographs occupy 24 bits.

Due to features of the method used by UTF-8 to store the code points, there is no need to use the BOM, but some of the tools look for it when reading the file, and many editors set it up during the save.

Python 3 fully supports Unicode and UTF-8:

- you can use Unicode/UTF-8 encoded characters to name variables and other entities;
- you can use them during all input and output.

This means that Python3 is completely I18Ned.

Strings - a brief review

Let's do a brief review of the nature of Python's strings.

First of all, Python's strings (or simply strings, as we're not going to discuss any other language's strings) are **immutable sequences**.

It's very important to note this, because it means that you should expect some familiar behavior from them.

For example, the `len()` function used for strings returns a number of characters contained by the arguments.

Take a look at **Example 1** in the editor. The snippet outputs `2`.

Any string can be empty. Its length is `0` then - just like in **Example 2**.

Don't forget that a backslash (`\`) used as an escape character is not included in the string's total length.

The code in **Example 3**, therefore, outputs `3`.

Run the three example codes and check.

```
# Example 1

word = 'by'
print(len(word))


# Example 2

empty = ''
print(len(empty))


# Example 3

i_am = 'I\'m'
print(len(i_am))
```

Console >_

```
2
0
3
```

Multiline strings

Now is a very good moment to show you another way of specifying strings inside the Python source code. Note that the syntax you already know won't let you use a string occupying more than one line of text.

For this reason, the code here is erroneous:

```
multiLine = 'Line #1
Line #2'

print(len(multiLine))
```

Fortunately, for these kinds of strings, Python offers separate, convenient, and simple syntax.

Look at the code in the editor. This is what it looks like.

As you can see, the string starts with **three apostrophes**, not one. The same tripled apostrophe is used to terminate it.

The number of text lines put inside such a string is arbitrary.

The snippet outputs `15`.

Count the characters carefully. Is this result correct or not? It looks okay at first glance, but when you count the characters, it doesn't.

```
multiLine = '''Line #1
Line #2'''

print(len(multiLine))
```

Console >_

```
15
```

`Line #1` contains seven characters. Two such lines comprise 14 characters. Did we lose a character? Where? How?

No, we didn't.

The missing character is simply invisible - it's a whitespace. It's located between the two text lines.

It's denoted as: `\n`.

Do you remember? It's a special (control) character used to **force a line feed** (hence its name: LF). You can't see it, but it counts.

The multiline strings can be delimited by **triple quotes**, too, just like here:

```
multiLine = """Line #1
Line #2"""

print(len(multiLine))
```

Choose the method that is more comfortable for you. Both work the same.

Console >_

```
15
```

Operations on strings

Like other kinds of data, strings have their own set of permissible operations, although they're rather limited compared to numbers.

In general, strings can be:

- **concatenated** (joined)
- **replicated**.

The first operation is performed by the `+` operator (note: it's not an addition) while the second by the `*` operator (note again: it's not a multiplication).

The ability to use the same operator against completely different kinds of data (like numbers vs. strings) is called **overloading** (as such an operator is overloaded with different duties).

Analyze the example:

- The `+` operator used against two or more strings produces a new string containing all the characters from its arguments (note: the order matters - this overloaded `+`, in contrast to its numerical version, is **not commutative**)
- the `*` operator needs a string and a number as arguments; in this case, the order doesn't matter - you can put the number before the string, or vice versa, the result will be the same - a new string created by the nth replication of the argument's string.

The snippet produces the following output:

```
ab
ba
aaaaa
bbbb
```

Note: shortcut variants of the above operators are also applicable for strings (`+=` and `*=`).

```
str1 = 'a'
str2 = 'b'
str1 += str2
print(str2)
print(str1 + str2)
print(str2 + str1)
print(5 * 'a')
print('b' * 4)
```

Console >_

```
b
abb
bab
aaaaa
bbbb
```

Operations on strings: `ord()`

If you want to know a specific character's ASCII/UNICODE code point value, you can use a function named `ord()` (as in *ordinal*).

The function needs a >strong>one-character string as its argument - breaching this requirement causes a `TypeError` exception, and returns a number representing the argument's code point.

Look at the code in the editor, and run it. The snippet outputs:

```
97
32
```

Now assign different values to `ch1` and `ch2`, e.g., `α` (Greek alpha), and `ę` (a letter in the Polish alphabet); then run the code and see what result it outputs. Carry out your own experiments.

```
# Demonstrating the ord() function

ch1 = 'a'
ch2 = ' ' # space

print(ord(ch1))
print(ord(ch2))
```

Console >_

```
97
32
```

Operations on strings: `chr()`

If you know the code point (number) and want to get the corresponding character, you can use a function named `chr()`.

The function **takes a code point and returns its character**.

Invoking it with an invalid argument (e.g., a negative or invalid code point) causes `ValueError` or `TypeError` exceptions.

Run the code in the editor. The example snippet outputs:

```
a
α
```

Note:

- `chr(ord(x)) == x`
- `ord(chr(x)) == x`

Again, run your own experiments.

```
# Demonstrating the chr() function

print(chr(97))
print(chr(945))
```

Console >_

```
a
α
```

Strings as sequences: indexing

We told you before that **Python's strings are sequences**. It's time to show you what that actually means.

Strings aren't lists, but **you can treat them like lists in many particular cases**.

For example, if you want to access any of a string's characters, you can do it using **indexing**, just like in the example in the editor. Run the program.

Be careful - don't try to pass a string's boundaries - it will cause an exception.

The example output is:

```
s i l l y   w a l k s
```

By the way, negative indices behave as expected, too. Check this yourself.

Strings as sequences: iterating

Iterating through the strings works, too. Look at the example below:

```
# Iterating through a string

exampleString = 'silly walks'

for ch in exampleString:
    print(ch, end=' ')

print()
```

The output is the same as previously. Check.

```
# Indexing strings

exampleString = 'silly walks'

for ix in range(len(exampleString)):
    print(exampleString[ix], end=' ')

print()
```

Console >_

```
s i l l y   w a l k s
```

Slices

Moreover, everything you know about **slices** is still usable.

We've gathered some examples showing how slices work in the string world. Look at the code in the editor, analyze it, and run it.

You won't see anything new in the example, but we want you to be sure that you can explain all the lines of the code.

The code's output is:

```
bd
efg
abd
e
e
adf
beg
```

Now do your own experiments.

```
# Slices

alpha = "abdefg"

print(alpha[1:3])
print(alpha[3:])
print(alpha[:3])
print(alpha[3:-2])
print(alpha[-3:4])
print(alpha[::2])
print(alpha[1::2])
```

Console >_

```
bd
efg
abd
e
e
adf
beg
```

The `in` and `not in` operators

The `in` operator shouldn't surprise you when applied to strings - it simply **checks if its left argument (a string) can be found anywhere within the right argument (another string)**.

The result of the check is simply `True` or `False`.

Look at the example program in the editor. This is how the `in` operator works.

The example output is:

```
True
False
False
True
False
```

As you probably suspect, the `not in` operator is also applicable here.

This is how it works:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" not in alphabet)
print("F" not in alphabet)
print("1" not in alphabet)
print("ghi" not in alphabet)
print("Xyz" not in alphabet)
```

The example output is:

```
False
True
True
False
True
```

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" in alphabet)
print("F" in alphabet)
print("1" in alphabet)
print("ghi" in alphabet)
print("Xyz" in alphabet)
```

Console >_

```
True
False
False
True
False
```

Python strings are immutable

We've also told you that Python's **strings are immutable**. This is a very important feature. What does it mean?

This primarily means that the similarity of strings and lists is limited. Not everything you can do with a list may be done with a string.

The first important difference **doesn't allow you to use the `del` instruction to remove anything from a string**.

The example here won't work:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

del alphabet[0]
```

The only thing you can do with `del` and a string is to **remove the string as a whole**. Try to do it.

Python strings **don't have the `append()` method** - you cannot expand them in any way.

The example below is erroneous:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.append("A")
```

with the absence of the `append()` method, **the `insert()` method is illegal**, too:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.insert(0, "A")
```

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

# put test code here
# del alphabet[0]
# alphabet.append("A")
alphabet.insert(0, "A")

```

Console >_

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    del alphabet[0]
TypeError: 'str' object doesn't support item deletion
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    alphabet.append("A")
AttributeError: 'str' object has no attribute 'append'
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    alphabet.insert(0, "A")
AttributeError: 'str' object has no attribute 'insert'
```

Operations on strings: continued

Don't think that a string's immutability limits your ability to operate with strings.

The only consequence is that you have to remember about it, and implement your code in a slightly different way - look at the example code in the editor.

This form of code is fully acceptable, will work without bending Python's rules, and will bring the full Latin alphabet to your screen:

```
abcdefghijklmnopqrstuvwxyz
```

You may want to ask if **creating a new copy of a string each time you modify its contents worsens the effectiveness of the code**.

Yes, it does. A bit. It's not a problem at all, though.

```
alphabet = "bcdefghijklmnopqrstuvwxy"

alphabet = "a" + alphabet
alphabet = alphabet + "z"

print(alphabet)
```

Console >_

```
abcdefghijklmnopqrstuvwxyz
```

Operations on strings: `min()`

Now that you understand that strings are sequences, we can show you some less obvious sequence capabilities. We'll present them using strings, but don't forget that lists can adopt the same tricks, too.

Let's start with a function named `min()`.

The function **finds the minimum element of the sequence passed as an argument**. There is one condition - the sequence (string, list, it doesn't matter) **cannot be empty**, or else you'll get a `ValueError` exception.

The **Example 1** program outputs:

```
A
```

Note: It's an upper-case *A*. Why? Recall the ASCII table - which letters occupy first locations - upper or lower?

We've prepared two more examples to analyze: **Examples 2 & 3**.

As you can see, they present more than just strings. The expected output looks as follows:

```
[ ]
0
```

Note: we've used the square brackets to prevent the space from being overlooked on your screen.

```
# Demonstrating min() - Example 1
print(min("aAbByYzZ"))


# Demonstrating min() - Examples 2 & 3
t = 'The Knights Who Say "Ni!"'
print('[' + min(t) + ']')

t = [0, 1, 2]
print(min(t))
```

Console >_

```
A
[ ]
0
```

Operations on strings: `max()`

Similarly, a function named `max()` **finds the maximum element of the sequence**.

Look at **Example 1** in the editor. The example program outputs:

```
z
```

Note: It's a lower-case *z*.

Now let's see the `max()` function applied to the same data as previously. Look at **Examples 2 & 3** in the editor.

The expected output is:

```
[y]
2
```

Carry out your own experiments.

```
# Demonstrating max() - Example 1
print(max("aAbByYzZ"))


# Demonstrating max() - Examples 2 & 3
t = 'The Knights Who Say "Ni!"'
print('[' + max(t) + ']')

t = [0, 1, 2]
print(max(t))
```

Console >_

```
z
[y]
2
```

Operations on strings: the `index()` method

The `index()` method (it's a method, not a function) **searches the sequence from the beginning, in order to find the first element of the value specified in its argument**.

Note: the element searched for must occur in the sequence - **its absence will cause a `ValueError` exception**.

The method returns the **index of the first occurrence of the argument** (which means that the lowest possible result is `0`, while the highest is the length of argument decremented by `1`).

Therefore, the example in the editor outputs:

```
2
7
1
```

```
# Demonstrating the index() method
print("aAbByYzZaA".index("b"))
print("aAbByYzZaA".index("Z"))
print("aAbByYzZaA".index("A"))
```

Console >_

```
2
7
1
```

https://docs.python.org/3.4/library/stdtypes.html#string-methods.

Operations on strings: the `list()` function

The `list()` function **takes its argument (a string) and creates a new list containing all the string's characters, one per list element**.

Note: it's not strictly a string function - `list()` is able to create a new list from many other entities (e.g., from tuples and dictionaries).

Take a look at the code example in the editor.

The example outputs:

```
['a', 'b', 'c', 'a', 'b', 'c']
```

Operations on strings: the `count()` method

The `count()` method **counts all occurrences of the element inside the sequence**. The absence of such elements doesn't cause any problems.

Look at the second example in the editor. Can you guess its output?

It is:

```
2
0
```

Moreover, Python strings have a significant number of methods intended exclusively for processing characters. Don't expect them to work with any other collections. The complete list of is presented here: https://docs.python.org/3.4/library/stdtypes.html#string-methods.

We're going to show you the ones we consider the most useful.

```
# Demonstrating the list() function
print(list("abcabc"))

# Demonstrating the count() method
print("abcabc".count("b"))
print('abcabc'.count("d"))
```

Console >_

```
['a', 'b', 'c', 'a', 'b', 'c']
2
0
```

The `capitalize()` method

Let's go through some standard Python string methods. We're going to go through them in alphabetical order - to be honest, any order has as many disadvantages as advantages, so the choice may as well be random.

The `capitalize()` method does exactly what it says - **it creates a new string filled with characters taken from the source string**, but it tries to modify them in the following way:

- **if the first character inside the string is a letter** (note: the first character is an element with an index equal to `0`, not just the first visible character), **it will be converted to upper-case**;
- **all remaining letters from the string will be converted to lower-case**.

Don't forget that:

- the original string (from which the method is invoked) is not changed in any way (a string's immutability must be obeyed without reservation)
- the modified (capitalized in this case) string is returned as a result - if you don't use it in any way (assign it to a variable, or pass it to a function/method) it will disappear without a trace.

Note: methods don't have to be invoked from within variables only. They can be invoked directly from within string literals. We're going to use that convention regularly - it will simplify the examples, as the most important aspects will not disappear among unnecessary assignments.

Take a look at the example in the editor. Run it.

This is what it prints:

```
Abcd
```

Try some more advanced examples and test their output:

```
print("Alpha".capitalize())
print('ALPHA'.capitalize())
print(' Alpha'.capitalize())
print('123'.capitalize())
print("αβγδ".capitalize())
```

```
# Demonstrating the capitalize() method
print('aBcD'.capitalize())


print("Alpha".capitalize())
print('ALPHA'.capitalize())
print(' Alpha'.capitalize())
print('123'.capitalize())
print("αβγδ".capitalize())
```

Console >_

```
Abcd
Abcd
Alpha
Alpha
 alpha
123
Αβγδ
```

The `center()` method

The one-parameter variant of the `center()` method makes a copy of the original string, trying to center it inside a field of a specified width.

The centering is actually done by **adding some spaces before and after the string**.

Don't expect this method to demonstrate any sophisticated skills. It's rather simple.

The example in the editor uses brackets to clearly show you where the centered string actually begins and terminates.

Its output looks as follows:

```
[  alpha   ]
```

If the target field's length is too small to fit the string, the original string is returned.

You can see the `center()` method in more examples here:

```
print('[' + 'Beta'.center(2) + ']')
print('[' + 'Beta'.center(4) + ']')
print('[' + 'Beta'.center(6) + ']')
```

Run the snippets above and check what output they produce.

The two-parameter variant of `center()` makes use of the character from the second argument, instead of a space. Analyze the example below:

```
print('[' + 'gamma'.center(20, '*') + ']')
```

This is why the output now looks like this:

```
[*******gamma********]
```

Carry out more experiments.

```
# Demonstrating the center() method
print('[' + 'alpha'.center(10) + ']')
```

Console >_

```
[  alpha   ]
```

The `endswith()` method

The `endswith()` method **checks if the given string ends with the specified argument and returns `True` or `False`**, depending on the check result.

Note: the substring must adhere to the string's last character - it cannot just be located somewhere near the end of the string.

Look at our example in the editor, analyze it, and run it. It outputs:

```
yes
```

You should now be able to predict the output of the snippet below:

```
t = "zeta"
print(t.endswith("a"))
print(t.endswith("A"))
print(t.endswith("et"))
print(t.endswith("eta"))
```

Run the code to check yor predictions.

```
# Demonstrating the endswith() method
if "epsilon".endswith("on"):
    print("yes")
else:
    print("no")

t = "zeta"
print(t.endswith("a"))
print(t.endswith("A"))
print(t.endswith("et"))
print(t.endswith("eta"))
```

Console >_

```
yes
yes
True
False
False
True
```

The `find()` method

The `find()` method is similar to `index()`, which you already know - **it looks for a substring and returns the index of first occurrence of this substring**, but:

- it's safer - it **doesn't generate an error for an argument containing a non-existent substring** (it returns `-1` then)
- it **works with strings only** - don't try to apply it to any other sequence.

Look at the code in the editor. This is how you can use it.

The example prints:

```
1
-1
```

Note: don't use `find()` if you only want to check if a single character occurs within a string - the `in`

Here is another example:

```
t = 'theta'
print(t.find('eta'))
print(t.find('et'))
print(t.find('the'))
print(t.find('ha'))
```

Can you predict the output? Run it and check your predictions.

If you want to perform the find, not from the string's beginning, but **from any position**, you can use a **two-parameter variant** of the `find()` method. Look at the example:

```
print('kappa'.find('a', 2))
```

The second argument **specifies the index at which the search will be started (it doesn't have to fit inside the string).**

```
# Demonstrating the find() method
print("Eta".find("ta"))
print("Eta".find("mma"))
```

Console >_

```
1
-1
```

Among the two *a* letters, only the second will be found. Run the snippet and check.

You can use the `find()` method to search for all the substring's occurrences, like here:

```
txt = """A variation of the ordinary lorem ipsum
text has been used in typesetting since the 1960s
or earlier, when it was popularized by advertisements
for Letraset transfer sheets. It was introduced to
the Information Age in the mid-1980s by the Aldus Corporation,
which employed it in graphics and word-processing templates
for its desktop publishing program PageMaker (from Wikipedia)"""

fnd = txt.find('the')
while fnd != -1:
    print(fnd)
    fnd = txt.find('the', fnd + 1)
```

The code prints the indices of all occurrences of the article *the*, and its output looks like this:

```
15
80
198
221
238
```

There is also a **three-parameter mutation of the `find()` method** - the third argument **points to the first index which won't be taken into consideration during the search** (it's actually the upper limit of the search).

Look at our example below:

```
print('kappa'.find('a', 1, 4))
print('kappa'.find('a', 2, 4))
```

The second argument specifies the index at which the search will be started (it doesn't have to fit inside the string).

Therefore, the modified example outputs:

```
1
-1
```

(*a* cannot be found within the given search boundaries in the second `print()`.

The `isalnum()` method

The parameterless method named `isalnum()` **checks if the string contains only digits or alphabetical characters (letters), and returns `True` or `False`** according to the result.

Look at the example in the editor and run it.

Note: any string element that is not a digit or a letter causes the method to return `False`. An empty string does, too.

The example output is:

```
True
True
True
False
False
False
```

Three more intriguing examples are here:

```
t = 'Six lambdas'
print(t.isalnum())

t = 'ĄßΠ5'
print(t.isalnum())

t = '20E1'
print(t.isalnum())
```

Run them and check their output.

Hint: the cause of the first result is a space - it's neither a digit nor a letter.

The `isalpha()` method

The `isalpha()` method is more specialized - it's interested in **letters only**.

Look at Example 1 - its output is:

```
True
False
```

The `isdigit()` method

In turn, the `isdigit()` method looks at **digits only** - anything else produces `False` as the result.

Look at Example 2 - its output is:

```
True
False
```

Carry out more experiments.

```
# Demonstrating the find() method
print("Eta".find("ta"))
print("Eta".find("mma"))


txt = """A variation of the ordinary lorem ipsum
text has been used in typesetting since the 1960s
or earlier, when it was popularized by advertisements
for Letraset transfer sheets. It was introduced to
the Information Age in the mid-1980s by the Aldus Corporation,
which employed it in graphics and word-processing templates
for its desktop publishing program PageMaker (from Wikipedia)"""

fnd = txt.find('the')
while fnd != -1:
    print(fnd)
    fnd = txt.find('the', fnd + 1)
```

Console >_

```
1
-1
1
-1
15
80
198
221
238
```

Console >_

```
1
-1
1
-1
15
80
198
221
238
```

```
# Demonstrating the isalnum() method
print('lambda30'.isalnum())
print('lambda'.isalnum())
print('30'.isalnum())
print('@'.isalnum())
print('lambda_30'.isalnum())
print(''.isalnum())
```

Console >_

```
True
True
True
False
False
False
```

```
# Example 1: Demonstrating the isapha() method
print("Moooo".isalpha())
print('Mu40'.isalpha())

# Example 2: Demonstrating the isdigit() method
print('2018'.isdigit())
print("Year2019".isdigit())
```

Console >_

```
True
False
True
False
```

The `islower()` method

The `islower()` method is a fussy variant of `isalpha()` - it accepts **lower-case letters only**.

Look at Example 1 in the editor - it outputs:

```
False
True
```

The `isspace()` method

The `isspace()` method **identifies whitespaces only** - it disregards any other character (the result is `False` then).

Look at Example 2 in the editor - the output is:

```
True
True
False
```

The `isupper()` method

The `isupper()` method is the upper-case version of `islower()` - it concentrates on **upper-case letters only**.

Again, Look at the code in the editor - Example 3 produces the following output:

```
False
False
True
```

```
# Example 1: Demonstrating the islower() method
print("Moooo".islower())
print('moooo'.islower())

# Example 2: Demonstrating the isspace() method
print(' \n '.isspace())
print(" ".isspace())
print("mooo mooo mooo".isspace())

# Example 3: Demonstrating the isupper() method
print("Moooo".isupper())
print('moooo'.isupper())
print('MOOOO'.isupper())
```

Console >_

```
False
True
True
True
False
False
False
True
```

The `join()` method

The `join()` method is rather complicated, so let us guide you step by step thorough it:

- as its name suggests, the method **performs a join** - it expects one argument as a list; it must be assured that all the list's elements are strings - the method will raise a `TypeError` exception otherwise;
- all the list's elements will be **joined into one string** but...
- ...the string from which the method has been invoked is **used as a separator**, put among the strings;
- the newly created string is returned as a result.

Take a look at the example in the editor. Let's analyze it:

- the `join()` method is invoked from within a string containing a comma (the string can be arbitrarily long, or it can be empty)
- the `join`'s argument is a list containing three strings;
- the method returns a new string.

Here it is:

```
omicron,pi,rh
```

```
# Demonstrating the join() method
print(",".join(["omicron", "pi", "rho"]))
```

Console >_

```
omicron,pi,rho
```

The `lower()` method

The `lower()` method **makes a copy of a source string, replaces all upper-case letters with their lower-case counterparts**, and returns the string as the result. Again, the source string remains untouched.

If the string doesn't contain any upper-case characters, the method returns the original string.

Note: The `lower()` method doesn't take any parameters.

The example in the editor outputs:

```
sigma=60
```

As usual, carry out your own experiments.

```
# Demonstrating the lower() method
print("SiGmA=60".lower())
```

Console >_

```
sigma=60
```

The `lstrip()` method

The parameterless `lstrip()` method **returns a newly created string formed from the original one by removing all leading whitespaces**.

Analyze the example code in the editor.

The brackets are not a part of the result - they only show the result's boundaries.

The example outputs:

```
[tau ]
```

The **one-parameter** `lstrip()` method does the same as its parameterless version, but **removes all characters enlisted in its argument** (a string), not just whitespaces:

```
print("www.cisco.com".lstrip("w."))
```

Brackets aren't needed here, as the output looks as follows:

```
cisco.com
```

Can you guess the output of the snippet below? Think carefully. Run the code and check your predictions.

```
print("pythoninstitute.org".lstrip(".org"))
```

Surprised? **Leading** characters, leading whitespaces. Again, experiment with your own examples.

```
# Demonstrating the lstrip() method
print("[" + " tau ".lstrip() + "]")

print("pythoninstitute.org".lstrip(".org"))

```

Console >_

```
[tau ]
[tau ]
pythoninstitute.org
```

The `replace()` method

The **two-parameter** `replace()` method **returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument**.

Look at the example code in the editor. Run it.

The second argument can be an empty string (replacing is actually removing, then), but the first cannot be.

The example outputs:

```
www.pyhoninstitute.org
Thare are it!
Apple
```

The **three-parameter** `replace()` variant uses the third argument (a number) to **limit the number of replacements**.

Look at the modified example code below:

```
print("This is it!".replace("is", "are", 1))
print("This is it!".replace("is", "are", 2))
```

Can you guess its output? Run the code and check your guesses.

```
# Demonstrating the replace() method
print("www.netacad.com".replace("netacad.com", "pythoninstitute.org"))
print("This is it!".replace("is", "are"))
print("Apple juice".replace("juice", ""))

print("This is it!".replace("is", "are", 1))
print("This is it!".replace("is", "are", 2))
```

Console >_

```
www.pythoninstitute.org
Thare are it!
Apple
www.pythoninstitute.org
Thare are it!
Apple
Thare is it!
Thare are it!
```

The `rfind()` method

The one-, two-, and three-parameter methods named `rfind()` do nearly the same things as their counterparts (the ones devoid of the *r* prefix), but **start their searches from the end of the string**, not the beginning (hence the prefix *r*, for *right*).

Take a look at the example code in the editor and try to predict its output. Run the code to check if you were right.

```
# Demonstrating the rfind() method
print("tau tau tau".rfind("ta"))
print("tau tau tau".rfind("ta", 9))
print("tau tau tau".rfind("ta", 3, 9))
```

Console >_

```
8
-1
```

The `rstrip()` method

Two variants of the `rstrip()` method do nearly the same as `lstrip`s, but **affect the opposite side of the string**.

Look at the code example in the editor. Can you guess its output? Run the code to check your guesses.

As usual, we encourage you to experiment with your own examples.

```
# Demonstrating the rstrip() method
print("[" + " upsilon ".rstrip() + "]")
print("cisco.com".rstrip(".com"))
```

Console >_

```
[ upsilon]
cis
```

The `split()` method

The `split()` method does what it says - it **splits the string and builds a list of all detected substrings**.

The method **assumes that the substrings are delimited by whitespaces** - the spaces don't take part in the operation, and aren't copied into the resulting list.

If the string is empty, the resulting list is empty too.

Look at the code in the editor. The example produces the following output:

```
['phi', 'chi', 'psi']
```

Note: the reverse operation can be performed by the `join()` method.

```
# Demonstrating the split() method
print("phi       chi\npsi".split())
```

Console >_

```
['phi', 'chi', 'psi']
```

The `startswith()` method

The `startswith()` method is a mirror reflection of `endswith()` - it **checks if a given string starts with the specified substring**.

Look at the example in the editor. This is the result from it:

```
False
True
```

The `strip()` method

The `trip()` method combines the effects caused by `rstrip()` and `lstrip()` - it **makes a new string lacking all the leading and trailing whitespaces**.

Look at the second example in the editor. This is the result it returns:

```
[aleph]
```

Now carry out you own experiments with the two methods.

```
# Demonstrating the startswith() method
print("omega".startswith("meg"))
print("omega".startswith("om"))

print()

# Demonstrating the strip() method
print("[" + "   aleph   ".strip() + "]")
```

Console >_

```
False
True

[aleph]
```

The `swapcase()` method

The `swapcase()` method **makes a new string by swapping the case of all letters within the source string**: lower-case characters become upper-case, and vice versa.

All other characters remain untouched.

Look at the first example in the editor. Can you guess the output? It won't look good, but you must see it:

```
i KNOW THAT i KNOW NOTHING.
```

The `title()` method

The `title()` method performs a somewhat similar function - it **changes every word's first letter to upper-case, turning all other ones to lower-case**.

Look at the second example in the editor. Can you guess its output? This is the result:

```
I Know That I Know Nothing. Part 1.
```

The `upper()` method

Last but not least, the `upper()` method **makes a copy of the source string, replaces all lower-case letters with their upper-case counterparts**, and returns the string as the result.

Look at the third example in the editor. It outputs:

```
I KNOW THAT I KNOW NOTHING. PART 2.
```

Hoooray! We've made it to the end of this section. Are you surprised with any of the string methods we've discussed so far? Take a couple of minutes to review them, and let's move on to the next part of the course where we'll show you what great things we can do with strings.

```
# Demonstrating the swapcase() method
print("I know that I know nothing.".swapcase())

print()

# Demonstrating the title() method
print("I know that I know nothing. Part 1.".title())

print()

# Demonstrating the upper() method
print("I know that I know nothing. Part 2.".upper())
```

Console >_

```
i KNOW THAT i KNOW NOTHING.

I Know That I Know Nothing. Part 1.

I KNOW THAT I KNOW NOTHING. PART 2.
```

LAB

Estimated time

20-25 minutes

Level of difficulty

Medium

Objectives

- improving the student's skills in operating with strings;
- using built-in Python string methods.

Scenario

You already know how `split()` works. Now we want you to prove it.

Your task is to **write your own function, which behaves almost exactly like the original `split()` method**, i.e.:

- it should accept exactly one argument - a string;
- it should return a list of words created from the string, divided in the places where the string contains whitespaces;
- if the string is empty, the function should return an empty list;
- its name should be `mysplit()`

Use the template in the editor. Test your code carefully.

Expected output

```
['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question']
['To', 'be', 'or', 'not', 'to', 'be,that', 'is', 'the', 'question']
[]
['abc']
[]
```

```
def mysplit(strng):
    # return [] if string is empty or contains whitespaces only
    if strng == '' or strng.isspace():
        return [ ]
    # prepare a list to return
    lst = []
    # prepare a word to build subsequent words
    word = ''
    # check if we are currently inside a word (i.e., if the string starts with a word)
    inword = not strng[0].isspace()
    # iterate through all the characters in string
    for x in strng:
        # if we are currently inside a string...
        if inword:
            # ... and current character is not a space...
            if not x.isspace():
                # ... update current word
                word = word + x
            else:
                # ... otherwise, we reached the end of the word so we need to append it
                lst.append(word)
                # ... and signal a fact that we are outside the word now
                inword = False
        else:
            # if we are outside the word and we reached a non-white character...
            if not x.isspace():
                # ... it means that a new word has begun so we need to remember it and.
                inword = True
                # ... store the first letter of the new word
                word = x
            else:
                pass
    # if we left the string and there is a non-empty string in word, we need to update
    if inword:
        lst.append(word)
    # return the list to invoker
    return lst

print(mysplit("To be or not to be, that is the question"))
print(mysplit("To be or not to be,that is the question"))
print(mysplit("   "))
print(mysplit(" abc "))
print(mysplit(""))
```

Console >_

```
['To', 'be', 'or', 'not', 'to', 'be,', 'that', 'is', 'the', 'question']
['To', 'be', 'or', 'not', 'to', 'be,that', 'is', 'the', 'question']
[]
['abc']
[]
```

Scenario

You already know how `split()` works. Now we want you to prove it.

Your task is to **write your own function, which behaves almost exactly like the original `split()` method**, i.e.:

- it should accept exactly one argument - a string;
- it should return a list of words created from the string, divided in the places where the string contains whitespaces;
- if the string is empty, the function should return an empty list;
- its name should be `mysplit()`

Comparing strings

Python's strings **can be compared using the same set of operators** which are in use in relation to numbers.

Take a look at these operators - they can all compare strings, too:

- `==`
- `!=`
- `>`
- `>=`
- `<`
- `<=`

There is one "but" - the results of such comparisons may sometimes be a bit surprising. Don't forget that Python is not aware (it cannot be in any way) of subtle linguistic issues - it just **compares code point values**, character by character.

The results you get from such an operation are sometimes astonishing. Let's start with the simplest cases.

Two strings are equal when they consist of the same characters in the same order. By the same fashion, two strings are not equal when they don't consist of the same characters in the same order.

Both comparisons give `True` as a result:

```
'alpha' == 'alpha'
'alpha' != 'Alpha'
```

The final relation between strings is determined by **comparing the first different character in both strings** (keep ASCII/UNICODE code points in mind at all times.)

```
# Test examples here
```

Console >_

When you compare two strings of different lengths and the shorter one is identical to the longer one's beginning, the **longer string is considered greater**.

Just like here:

```
'alpha' < 'alphabet'
```

The relation is `True`.

String comparison is always case-sensitive (**upper-case letters are taken as lesser than lower-case**).

The expression is `True`:

```
'beta' > 'Beta'
```

Console >_

```
True
```

Comparing strings: continued

Even **if a string contains digits only, it's still not a number**. It's interpreted as-is, like any other regular string, and its (potential) numerical aspect is not taken into consideration in any way.

Look at the examples:

```
'10' == '010'
'10' > '010'
'10' > '8'
'20' < '8'
'20' < '80'
```

They produce the following results:

```
False
True
False
True
True
```

Comparing strings against numbers is generally a bad idea.

The only comparisons you can perform with impunity are these symbolized by the `==` and `!=` operators. The former always gives `False`, while the latter always produces `True`.

Using any of the remaining comparison operators will raise a `TypeError` exception.

```
1  # Test examples here
```

Console >_

Let's check it:

```
'10' == 10
'10' != 10
'10' == 1
'10' != 1
'10' > 10
```

The results in this case are:

```
False
True
False
True
TypeError exception
```

Run all the examples, and carry out more experiments.

Console >_

Sorting

Comparing is closely related to sorting (or rather, sorting is in fact a very sophisticated case of comparing).

This is a good opportunity to show you two possible ways to **sort lists containing strings**. Such an operation is very common in the real world - any time you see a list of names, goods, titles, or cities, you expect them to be sorted.

Let's assume that you want to sort the following list:

```
greek = ['omega', 'alpha', 'pi', 'gamma']
```

In general, Python offers two different ways to sort lists.

The first is implemented as **a function named** `sorted()`.

The function takes one argument (a list) and **returns a new list**, filled with the sorted argument's elements. (Note: this description is a bit simplified compared to the actual implementation - we'll discuss it later.)

The original list remains untouched.

Look at the code in the editor, and run it. The snippet produces the following output:

```
['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']
```

The second method affects the list itself - **no new list is created**. Ordering is performed in situ by the method named `sort()`.

The output hasn't changed:

```
['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']
```

If you need an order other than non-descending, you have to convince the function/method to change its default behaviors. We'll discuss it soon.

```
1  # Demonstrating the sorted() function
2  firstGreek = ['omega', 'alpha', 'pi', 'gamma']
3  firstGreek2 = sorted(firstGreek)
4
5  print(firstGreek)
6  print(firstGreek2)
7
8  print()
9
10 # Demonstrating the sort() method
11 secondGreek = ['omega', 'alpha', 'pi', 'gamma']
12 print(secondGreek)
13
14 secondGreek.sort()
15 print(secondGreek)
```

Console >_

```
['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']

['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']
```

Strings vs. numbers

There are two additional issues that should be discussed here: **how to convert a number (an integer or a float) into a string, and vice versa**. It may be necessary to perform such a transformation. Moreover, it's a routine way to process input/output data.

The number-string conversion is simple, as it is always possible. It's done by a function named `str()`.

Just like here:

```
itg = 13
flt = 1.3
si = str(itg)
sf = str(flt)

print(si + ' ' + sf)
```

The code outputs:

```
13 1.3
```

The reverse transformation (string-number) is possible when and only when the string represents a valid number. If the condition is not met, expect a `ValueError` exception.

Use the `int()` function if you want to get an integer, and `float()` if you need a floating-point value.

Just like here:

```
si = '13'
sf = '1.3'
itg = int(si)
flt = float(sf)

print(itg + flt)
```

This is what you'll see in the console:

```
14.3
```

```
# Test code here
```

Console >_

article.

LAB

Estimated time

30 minutes

Level of difficulty

Medium

Objectives

- improving the student's skills in operating with strings;
- using strings to represent non-text data.

Scenario

You've surely seen a *seven-segment display*.

It's a device (sometimes electronic, sometimes mechanical) designed to present one decimal digit using a subset of seven segments. If you still don't know what it is, refer to the following Wikipedia article.

Your task is to write **a program which is able to simulate the work of a seven-display device**, although you're going to use single LEDs instead of segments.

Each digit is constructed from 13 LEDs (some lit, some dark, of course) - that's how we imagine it:

```
  # ### ### # # ### ### ### ### ### ###
  #   #   # # # #   #     # # # # # # #
  # ### ### ### ### ###   # ### ### # #
  # #     #   # # #   # # #   # # #
  # ### ###   # ### ###   # ### ### ###
```

```
digits = [ '1111110',   # 0
       '0110000',   # 1
       '1101101',   # 2
       '1111001',   # 3
       '0110011',   # 4
       '1011011',   # 5
       '1011111',   # 6
       '1110000',   # 7
       '1111111',   # 8
       '1111011',   # 9
       ]

def printNumber(num):
    global digits
    digs = str(num)
    lines = [ '' for l in range(5) ]
    for d in digs:
        segs = [ [' ',' ',' '] for l in range(5) ]
        ptrn = digits[ord(d) - ord('0')]
        if ptrn[0] == '1':
            segs[0][0] = segs[0][1] = segs[0][2] = '#'
        if ptrn[1] == '1':
            segs[0][2] = segs[1][2] = segs[2][2] = '#'
        if ptrn[2] == '1':
            segs[2][2] = segs[3][2] = segs[4][2] = '#'
        if ptrn[3] == '1':
            segs[4][0] = segs[4][1] = segs[4][2] = '#'
        if ptrn[4] == '1':
            segs[2][0] = segs[3][0] = segs[4][0] = '#'
        if ptrn[5] == '1':
```

Console >_

Note: the number 8 shows all the LED lights on.

Your code has to *display* any non-negative integer number entered by the user.

Tip: using a list containing patterns of all ten digits may be very helpful.

Test data

Sample input:

```
123
```

Sample output:

```
  # ### ###
  #   #   #
  # ### ###
  # #     #
  # ### ###
```

Sample input:

```
9081726354
```

Sample output:

```
### ### ###   # ### ### ### ### ### # #
# # # # # #   #   #   # #     # #   # #
### # # ###   #   # ### ### ### ### ###
  # # # # #   #   # #   # #   #   #   #
### ### ###   #   # ### ### ### ###   #
```

```
    global digits
    digs = str(num)
    lines = [ '' for l in range(5) ]
    for d in digs:
        segs = [ [' ',' ',' '] for l in range(5) ]
        ptrn = digits[ord(d) - ord('0')]
        if ptrn[0] == '1':
            segs[0][0] = segs[0][1] = segs[0][2] = '#'
        if ptrn[1] == '1':
            segs[0][2] = segs[1][2] = segs[2][2] = '#'
        if ptrn[2] == '1':
            segs[2][2] = segs[3][2] = segs[4][2] = '#'
        if ptrn[3] == '1':
            segs[4][0] = segs[4][1] = segs[4][2] = '#'
        if ptrn[4] == '1':
            segs[2][0] = segs[3][0] = segs[4][0] = '#'
        if ptrn[5] == '1':
            segs[0][0] = segs[1][0] = segs[2][0] = '#'
        if ptrn[6] == '1':
            segs[2][0] = segs[2][1] = segs[2][2] = '#'
        for l in range(5):
            lines[l] += ''.join(segs[l]) + ' '
    for l in lines:
        print(l)

printNumber(int(input("Enter the number you wish to display: ")))
```

Console >_

```
Enter the number you wish to display: 112233256699922
  #   # ### ### ### ### ### ### ### ### ### ### ### ### ###
  #   #   #   #   #   #   # #   #   #   # # # # #   #   #
  #   # ### ### ### ### ### ### ### ### ### ### ### ### ###
  #   # #   #     #   # #     # # # # #   #   #   # #   #
  #   # ### ### ### ### ### ### ### ### ### ### ### ### ###
```

The Caesar Cipher: encrypting a message

We're going to show you four simple programs in order to present some aspects of string processing in Python. They are purposefully simple, but the lab problems will be significantly more complicated.

The first problem we want to show you is called the **Caesar cipher** - more details here: https://en.wikipedia.org/wiki/Caesar_cipher.

This cipher was (probably) invented and used by Gaius Julius Caesar and his troops during the Gallic Wars. The idea is rather simple - every letter of the message is replaced by its nearest consequent (*A* becomes *B*, *B* becomes *C*, and so on). The only exception is *Z*, which becomes *A*.

The program in the editor is a very simple (but working) implementation of the algorithm.

We've written it using the following assumptions:

- it accepts Latin letters only (note: the Romans used neither whitespaces nor digits)
- all letters of the message are in upper case (note: the Romans knew only capitals)

Let's trace the code:

- line 02: ask the user to enter the open (unencrypted), one-line message;
- line 03: prepare a string for an encrypted message (empty for now)
- line 04: start the iteration through the message;
- line 05: if the current character is not alphabetic...
- line 06: ...ignore it;
- line 07: convert the letter to upper-case (it's preferable to do it blindly, rather than check whether it's needed or not)
- line 08: get the code of the letter and increment it by one;
- line 09: if the resulting code has "left" the Latin alphabet (if it's greater than the *Z* code)...
- line 10: ...change it to the *A* code;
- line 11: append the received character to the end of the encrypted message;
- line 13: print the cipher.

```
1  # Caesar cipher
2  text = input("Enter your message: ")
3  cipher = ''
4  for char in text:
5      if not char.isalpha():
6          continue
7      char = char.upper()
8      code = ord(char) + 1
9      if code > ord('Z'):
10         code = ord('A')
11     cipher += chr(code)
12
13 print(cipher)
```

```
Console >_
Enter your message:  this is code is pro
UIJTJTDPEFJTQSP
```

The code, fed with this message:

```
AVE CAESAR
```

outputs:

```
BWFDBFTBS
```

Do your own tests.

```
Console >_
Enter your message:  this is code is pro
UIJTJTDPEFJTQSP
```

The Caesar Cipher: decrypting a message

The reverse transformation should now be clear to you - let's just present you with the code as-is, without any explanations.

Look at the code in the editor. Check carefully if it works. Use the cryptogram from the previous program.

```
1  # Caesar cipher - decrypting a message
2  cipher = input('Enter your cryptogram: ')
3  text = ''
4  for char in cipher:
5      if not char.isalpha():
6          continue
7      char = char.upper()
8      code = ord(char) - 1
9      if code < ord('A'):
10         code = ord('Z')
11     text += chr(code)
12
13 print(text)
```

```
Console >_
Enter your cryptogram: hjhgjgytfhj
GIGFIFXSEGI
```

The Numbers Processor

The third program shows a simple method allowing you to input a line filled with numbers, and to process them easily. Note: the routine `input()` function, combined together with the `int()` or `float()` functions, is unsuitable for this purpose.

The processing will be extremely easy - we want the numbers to be summed.

Look at the code in the editor. Let's analyze it.

Using list comprehension may make the code slimmer. You can do that if you want.

Let's present our version:

- line 03: ask the user to enter a line filled with any number of numbers (the numbers can be floats)
- line 04: split the line receiving a list of substrings;
- line 05: initiate the total sum to zero;
- line 06: as the string-float conversion may raise an exception, it's best to continue with the protection of the try-except block;
- line 07: iterate through the list...
- line 08: ...and try to convert all its elements into float numbers; if it works, increase the sum;
- line 09: everything is good so far, so print the sum;
- line 10: the program ends here in the case of an error;
- line 11: print a diagnostic message showing the user the reason for the failure.

The code has one important weakness - it displays a bogus result when the user enters an empty line. Can you fix it?

```
1  # Numbers Processor
2
3  line = input("Enter a line of numbers - separate them with spaces: ")
4  strings = line.split()
5  total = 0
6  try:
7      for substr in strings:
8          total += float(substr)
9      print("The total is:", total)
10 except:
11     print(substr, "is not a number.")
```

```
Console >_
Enter a line of numbers - separate them with spaces: 23 4 5 6 7 8
The total is: 53.0
```

The IBAN Validator

The fourth program implements (in a slightly simplified form) an algorithm used by European banks to specify account numbers. The standard named **IBAN** (International Bank Account Number) provides a simple and fairly reliable method of validating the account numbers against simple typos that can occur during rewriting of the number e.g., from paper documents, like invoices or bills, into computers.

You can find more details here: https://en.wikipedia.org/wiki/International_Bank_Account_Number.

An IBAN-compliant account number consists of:

- a two-letter country code taken from the ISO 3166-1 standard (e.g., *FR* for France, *GB* for Great Britain, *DE* for Germany, and so on)
- two check digits used to perform the validity checks - fast and simple, but not fully reliable, tests, showing whether a number is invalid (distorted by a typo) or seems to be good;
- the actual account number (up to 30 alphanumeric characters - the length of that part depends on the country)

The standard says that validation requires the following steps (according to Wikipedia):

- (step 1) Check that the total IBAN length is correct as per the country (this program won't do that, but you can modify the code to meet this requirement if you wish; note: you have to teach the code all the lengths used in Europe)
- (step 2) Move the four initial characters to the end of the string (i.e., the country code and the check digits)
- (step 3) Replace each letter in the string with two digits, thereby expanding the string, where *A = 10, B = 11 ... Z = 35*;
- (step 4) Interpret the string as a decimal integer and compute the remainder of that number on division by 97; If the remainder is `1`, the check digit test is passed and the IBAN might be valid.

Look at the code in the editor. Let's analyze it:

```
1  # IBAN Validator
2
3  iban = input("Enter IBAN, please: ")
4  iban = iban.replace(' ','')
5  if not iban.isalnum():
6      print("You have entered invalid characters.")
7  elif len(iban) < 15:
8      print("IBAN entered is too short.")
9  elif len(iban) > 31:
10     print("IBAN entered is too long.")
11 else:
12     iban = (iban[4:] + iban[0:4]).upper()
13     iban2 = ''
14     for ch in iban:
15         if ch.isdigit():
16             iban2 += ch
17         else:
18             iban2 += str(10 + ord(ch) - ord('A'))
19     ibann = int(iban2)
20     if ibann % 97 == 1:
21         print("IBAN entered is valid.")
22     else:
23         print("IBAN entered is invalid.")
```

```
Console >_
Enter IBAN, please: 33467567890321
IBAN entered is too short.
Enter IBAN, please: FFGKEJHR5675746 676586DLHDJ 909084095DGIHGOI
IBAN entered is too long.
Enter IBAN, please:
Enter IBAN, please: UYGUGUYFUY96764640=-0=-9-090987
You have entered invalid characters.
```

Look at the code in the editor. Let's analyze it:

- line 03: ask the user to enter the IBAN (the number can contain spaces, as they significantly improve number readability...
- line 04: ...but remove them immediately)
- line 05: the entered IBAN must consist of digits and letters only - if it doesn't...
- line 06: ...output the message;
- line 07: the IBAN mustn't be shorter than 15 characters (this is the shortest variant, used in Norway)
- line 08: if it is shorter, the user is informed;
- line 09: moreover, the IBAN cannot be longer than 31 characters (this is the longest variant, used in Malta)
- line 10: if it is longer, make an announcement;
- line 11: start the actual processing;
- line 12: move the four initial characters to the number's end, and convert all letters to upper case (step 02 of the algorithm)
- line 13: this is the variable used to complete the number, created by replacing the letters with digits (according to the algorithm's step 03)
- line 14: iterate through the IBAN;
- line 15: if the character is a digit...
- line 16: just copy it;
- line 17: otherwise...
- line 18: ...convert it into two digits (note the way it's done here)
- line 19: the converted form of the IBAN is ready - make an integer out of it;
- line 20: is the remainder of the division of `iban2` by `97` equal to `1` ?
- line 21: If yes, then success;
- line 22: Otherwise...
- line 23: ...the number is invalid.

Let's add some test data (all these numbers are valid - you can invalidate them by changing any character).

- British: `GB72 HBZU 7006 7212 1253 00`
- French: `FR76 30003 03620 00020216907 50`
- German: `DE02100100100152517108`

If you are an EU resident, you can use you own account number for tests.

```
1  # IBAN Validator
2
3  iban = input("Enter IBAN, please: ")
4  iban = iban.replace(' ','')
5  if not iban.isalnum():
6      print("You have entered invalid characters.")
7  elif len(iban) < 15:
8      print("IBAN entered is too short.")
9  elif len(iban) > 31:
10     print("IBAN entered is too long.")
11 else:
12     iban = (iban[4:] + iban[0:4]).upper()
13     iban2 = ''
14     for ch in iban:
15         if ch.isdigit():
16             iban2 += ch
17         else:
18             iban2 += str(10 + ord(ch) - ord('A'))
19     ibann = int(iban2)
20     if ibann % 97 == 1:
21         print("IBAN entered is valid.")
22     else:
23         print("IBAN entered is invalid.")
```

```
Console >_
Enter IBAN, please: 334675678903 21
IBAN entered is too short.
Enter IBAN, please: FFGKEJWR5675746 676586DLWDJ 909084095DGIHGOI
IBAN entered is too long.
Enter IBAN, please:
Enter IBAN, please: UYGUGUYFUY96764640=-0=-9-090987
You have entered invalid characters.
```

LAB

Estimated time

30-45 minutes

Level of difficulty

Hard

Pre-requisites

Module 5.1.11.1, Module 5.1.11.2

Objectives

- improving the student's skills in operating with strings;
- converting characters into ASCII code, and vice versa.

Scenario

You are already familiar with the Caesar cipher, and this is why we want you to improve the code we showed you recently.

The original Caesar cipher shifts each character by one: *a* becomes *b*, *z* becomes *a*, and so on. Let's make it a bit harder, and allow the shifted value to come from the range 1..25 inclusive.

Moreover, let the code preserve the letters' case (lower-case letters will remain lower-case) and all non-alphabetical characters should remain untouched.

Your task is to write a program which:

- asks the user for one line of text to encrypt;
- asks the user for a shift value (an integer number from the range 1..25 - note: you should force the user to enter a valid shift value (don't give up and don't let bad data fool you!)
- prints out the encoded text.

```
1  # input text to encrypt
2  text = input("Enter message: ")
3
4  # enter valid shift value (repeat until it succeeds)
5  shift = 0
6
7  while shift == 0:
8      try:
9          shift = int(input("Enter cipher's shift (1..25): "))
10         if shift not in range(1,26):
11             raise ValueError
12     except ValueError:
13         shift = 0
14     if shift == 0:
15         print("Bad shift value!")
16
17 cipher = ''
18
19 for char in text:
20     # is it a letter?
21     if char.isalpha():
22         # shift its code
23         code = ord(char) + shift
24         # find the code of the first letter (upper- or lower-case)
25         if char.isupper():
26             first = ord('A')
27         else:
28             first = ord('a')
29         # make correction
30         code -= first
31         code %= 26
32         # append encoded character to message
33         cipher += chr(first + code)
34     else:
35         # append original character to message
```

```
Console >_
```

Test your code using the data we've provided.

Test data

Sample input:

```
abcxyzABCxyz 123
2
```

Sample output:

```
cdezabCDEzab 123
```

Sample input:

```
The die is cast
25
```

Sample output:

```
Sgd chd hr bzrs
```

```
21     if char.isalpha():
22         # shift its code
23         code = ord(char) + shift
24         # find the code of the first letter (upper- or lower-case)
25         if char.isupper():
26             first = ord('A')
27         else:
28             first = ord('a')
29         # make correction
30         code -= first
31         code %= 26
32         # append encoded character to message
33         cipher += chr(first + code)
34     else:
35         # append original character to message
36         cipher += char
37
38 print(cipher)
```

```
Console >_
Enter message: ghjjh hii huioo
Enter cipher's shift (1..25): 7
noqqo opp obpvv
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

- improving the student's skills in operating with strings;
- encouraging the student to look for non-obvious solutions.

Scenario

Do you know what a palindrome is?

It's a word which look the same when read forward and backward. For example, "kayak" is a palindrome, while "loyal" is not.

Your task is to write a program which:

- asks the user for some text;
- checks whether the entered text is a palindrome, and prints result.

Note:

- assume that an empty string isn't a palindrome;
- treat upper- and lower-case letters as equal;
- spaces are not taken into account during the check - treat them as non-existent;
- there are more than a few correct solutions - try to find more than one.

```
text = input("Enter text: ")

# remove all spaces...
text = text.replace(' ','')

# ... and check if the word is equal to reversed itself
if len(text) > 1 and text.upper() == text[::-1].upper():
    print("It's a palindrome")
else:
    print("It's not a palindrome")
```

Console >_

```
Enter text: madam
It's a palindrome
```

Test your code using the data we've provided.

Test data

Sample input:

```
Ten animals I slam in a net
```

It's a palindrome

Sample input:

```
Eleven animals I slam in a net
```

It's not a palindrome

Console >_

```
Enter text: madam
It's a palindrome
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

- improving the student's skills in operating with strings;
- converting strings into lists, and vice versa.

Scenario

An anagram is a new word formed by rearranging the letters of a word, using all the original letters exactly once. For example, the phrases "rail safety" and "fairy tales" are anagrams, while "I am" and "You are" are not.

Your task is to write a program which:

- asks the user for two separate texts;
- checks whether, the entered texts are anagrams and prints the result.

Note:

- assume that two empty strings are not anagrams;
- treat upper- and lower-case letters as equal;
- spaces are not taken into account during the check - treat them as non-existent

Test your code using the data we've provided.

Test data

```
str1 = input("Enter the first string: ")
str2 = input("Enter the second string: ")

# this is what we're going to do with both strings:
# - remove spaces
# - change all letters to upper case
# - convert into list
# - sort the list
# - join list's elements into string
# and finally, compare both strings
# Let's do it!

strx1 = ''.join(sorted(list(str1.upper().replace(' ',''))))
strx2 = ''.join(sorted(list(str2.upper().replace(' ',''))))
if len(strx1) > 0 and strx1 == strx2:
    print("Anagrams")
else:
    print("Not anagrams")
```

Console >_

```
Enter the first string: silent
Enter the second string:
Enter the first string: listen
Enter the second string: silent
Anagrams
Enter the first string: modern
Enter the second string: old
Not anagrams
```

Sample input:

```
Listen
Silent
```

Anagrams

Sample input:

```
modern
norman
```

Not anagrams

```
Enter the first string: silent
Enter the second string:
Enter the first string: listen
Enter the second string: silent
Anagrams
Enter the first string: modern
Enter the second string: old
Not anagrams
```

LAB

Estimated time

10-15 minutes

Level of difficulty

Easy

Objectives

- improving the student's skills in operating with strings;
- converting integers into strings, and vice versa.

Scenario

Some say that the *Digit of Life* is a digit evaluated using somebody's birthday. It's simple - you just need to sum all the digits of the date. If the result contains more than one digit, you have to repeat the addition until you get exactly one digit. For example:

- 1 January 2017 = 2017 01 01
- 2 + 0 + 1 + 7 + 0 + 1 + 0 + 1 = 12
- 1 + 2 = 3

3 is the digit we searched for and found.

Your task is to write a program which:

- asks the user her/his birthday (in the format YYYYMMDD, or YYYYDDMM, or MMDDYYYY - actually, the order of the digits doesn't matter)
- outputs the *Digit of Life* for the date.

```
date = input("Enter your birthday date (in the following format: YYYYMMDD or YYYYDDMM, 8 d
if len(date) != 8 or not date.isdigit():
    print("Invalid date - sorry, we can do nothing with it.")
else:
    # while there is more than one digit in the date...
    while len(date) > 1:
        sum = 0
        # ... sum all the digits...
        for dig in date:
            sum += int(dig)
        print(date)
        # ... and store sum inside the string
        date = str(sum)
    print("Your Digit of Life is: " + date)
```

Console >_

Test data

Sample input:

```
19991229
```

Sample output:

```
6
```

Sample input:

```
20000101
```

Sample output:

```
4
```

LAB

Estimated time

15-20 minutes

Level of difficulty

Medium

Objectives

- improving the student's skills in operating with strings;
- using the `find()` method for searching strings.

Scenario

Let's play a game. We will give you two strings: one being a word (e.g., "dog") and the second being a combination of any characters.

Your task is to write a program which answers the following question: **are the characters comprising the first string hidden inside the second string?**

For example:

- If the second string is given as `"vcxzxduybfdsobywuefgas"`, the answer is `yes`;
- if the second string is `"vcxzxdcybfdstbywuefsas"`, the answer is `no` (as there are neither the letters "d", "o", or "g", in this order)

Hints:

- you should use the two-argument variants of the `pos()` functions inside your code;
- don't worry about case sensitivity.

Test your code using the data we've provided.

```
word = input("Enter the word you wish to find: ").upper()
strn = input("Enter the string you wish to search through: ").upper()

found = True
start = 0

for ch in word:
    pos = strn.find(ch, start)
    if pos < 0:
        found = False
        break
    start = pos + 1
if found:
    print("Yes")
else:
    print("No")
```

Console >_

Test data

Sample input:

```
donor
Nabucodonosor
```

Sample output:

```
Yes
```

Sample input:

```
donut
Nabucodonosor
```

Sample output:

```
No
```

Console >_

LAB

Estimated time

60 minutes

Level of difficulty

Hard

Objectives

- improving the student's skills in operating with strings and lists;
- converting strings into lists.

Scenario

As you probably know, *Sudoku* is a number-placing puzzle played on a 9x9 board. The player has to fill the board in a very specific way:

- each row of the board must contain all digits from 0 to 9 (the order doesn't matter)
- each column of the board must contain all digits from 0 to 9 (again, the order doesn't matter)
- each of the nine 3x3 "tiles" (we will name them "sub-squares") of the table must contain all digits from 0 to 9.

If you need more details, you can find them here.

Your task is to write a program which:

- reads 9 rows of the Sudoku, each containing 9 digits (check carefully if the data entered are valid)
- outputs `Yes` if the Sudoku is valid, and `No` otherwise.

Test your code using the data we've provided.

```
# this function checks whether a list passed as an argument contains
# nine digits from '1' to '9'
def checkset(digs):
    return sorted(list(digs)) == [chr(x + ord('0')) for x in range(1, 10)]



# this will be a list of rows representing the sudoku
rows = [ ]
for r in range(9):
    ok = False
    while not ok:
        row = input("Enter row #" + str(r + 1) + ": ")
        ok = len(row) == 9 or row.isdigit()
        if not ok:
            print("Incorrect row data - 9 digits required")
    rows.append(row)

ok = True

# check if all rows are good
for r in range(9):
    if not checkset(rows[r]):
        ok = False
        break

# check if all columns are good
if ok:
    for c in range(9):
        col = []
        for r in range(9):
            col.append(rows[r][c])
        if not checkset(col):
            ok = False
            break
```

Console >_

```
Enter row #1: 4252352462
Enter row #2: 1354262562
Enter row #3: 546576879890
Enter row #4: 324645758
Enter row #5: 25346676
```

Test data

Sample input:

```
295743861
431865927
876192543
387459216
612387495
549216738
763524189
928671354
154938672
```

Sample output:

```
Yes
```

Sample input:

```
195743862
431865927
876192543
387459216
612387495
549216738
763524189
928671354
254938671
```

Sample output:

```
No
```

```
ok = True

# check if all rows are good
for r in range(9):
    if not checkset(rows[r]):
        ok = False
        break

# check if all columns are good
if ok:
    for c in range(9):
        col = []
        for r in range(9):
            col.append(rows[r][c])
        if not checkset(col):
            ok = False
            break

# check if all sub-squares (3x3) are good
if ok:
    for r in range(0, 9, 3):
        for c in range(0, 9, 3):
            sqr = ''
            # make a string containing all digits from a sub-square
            for i in range(3):
                sqr += rows[r+i][c:c+3]
            if not checkset(list(sqr)):
                ok = False
                break

# print the final verdict
if ok:
    print("Yes")
else:
    print("No")
```

Console >_

MODULE: 6

The object-oriented approach: classes, methods, objects, and the standard objective features; exception handling, and working with files in a new window

Programming Essentials in Python: Module 6

In this module, you will learn about:

- the object-oriented approach - foundations;
- classes, methods, objects, and the standard objective features;
- exception handling;
- working with files.

Module 6:

The Object-Oriented Approach: classes, methods, objects and the standard objective features; exception handling, and working with files

6

1 2 3 4 5

The basic cocepts 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.

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.

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.

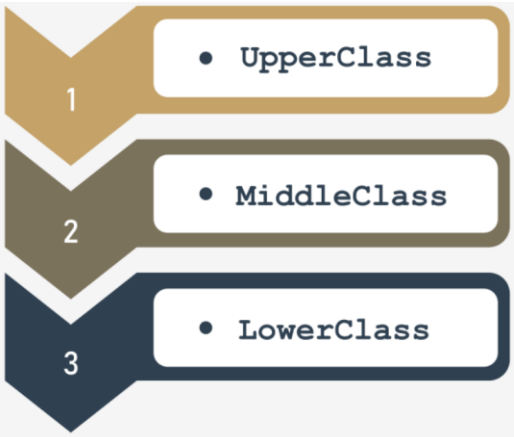
All these concepts are not as abstract as you may at first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programming: they don't create artificial life - **they reflect real facts, relationships, and circumstances**.

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).

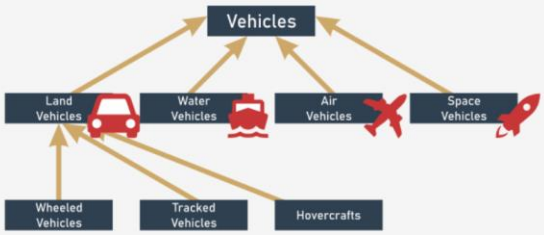
The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.



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.

Class hierarchies: continued

Another example is the hierarchy of the taxonomic kingdom of animals.

We can say that all *animals* (our top-level class) can be divided into five subclasses:

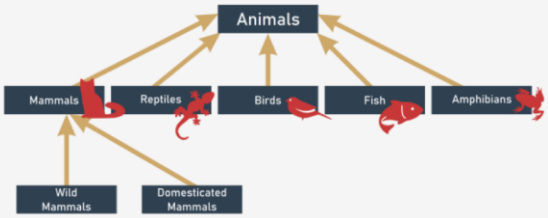
- mammals;
- reptiles;
- birds;
- fish;
- amphibians.

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.

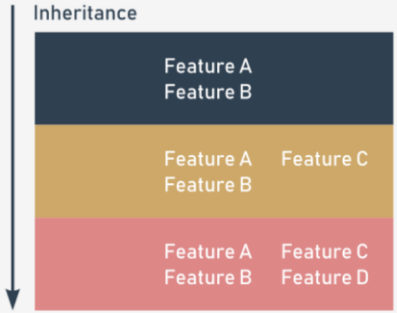
Each **subclass is more specialized** (or more specific) than its superclass. Conversely, each **superclass is more general** (more abstract) than any of its subclasses. Note that we've presumed that a class may only have one superclass - this is not always true, but we'll discuss this issue more a bit 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 superclasses.

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.

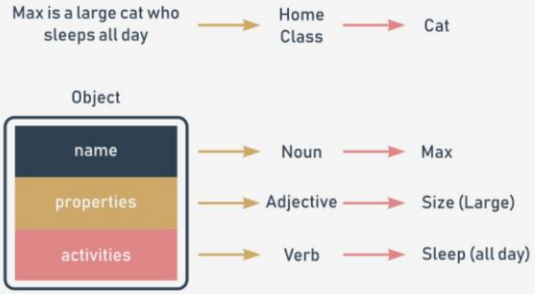
Two sample phrases should serve as a good example:

- Max is a large cat who sleeps all day.

  Object name = Max
  Home class = Cat
  Property = Size (large)
  Activity = Sleep (all day)

- A pink Cadillac went quickly.

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



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:

```
myFirstObject = 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.

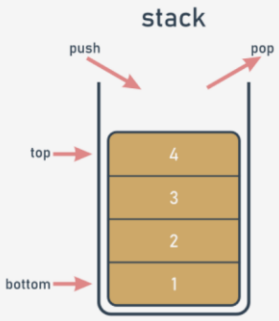
What is a stack?

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.

```
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())
```

```
Console >_
1
2
3
0
1
2
```

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
```

Test it.

```
Console >_
1
2
3
0
1
2
```

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;

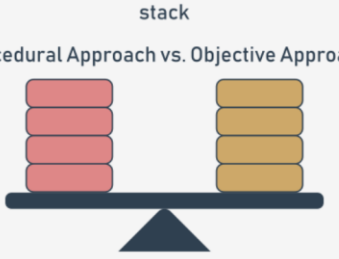
- it may also happen that you need not only `push` and `pop` functions, but also some other conveniences; you could certainly implement them, but try to imagine what would happen if you had dozens of separately implemented stacks.

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.

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:

```
class Stack:    # defining the Stack class
    def __init__(self):    # defining the constructor function
        print("Hi!")

stackObject = Stack()    # instantiating the object
```

Console >_

```
Hi!
```

- 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!")

stackObject = 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!
```

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.

```
    def __init__(self):    # defining the constructor function
        print("Hi!")

stackObject = Stack()    # instantiating the object
```

Console >_

```
Hi!
```

The stack - the object approach: continued

Any change you make inside the constructor that modifies the state of the `self` parameter will reflect 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 `stackList`.

Just like here:

```
class Stack:
    def __init__(self):
        self.stackList = []

stackObject = Stack()
print(len(stackObject.stackList))
```

Note:

```
class Stack:
    def __init__(self):
        self.stackList = []

stackObject = Stack()
print(len(stackObject.stackList))
```

Console >_

```
0
```

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 `stackList` 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:

Console >_

```
0
```

```
0
```

This is not we want from the stack. We prefer `stackList` to be **hidden from the outside world**. Is that possible?

Yes, and it's simple, but not very intuitive.

The stack - the object approach: continued

Take a look - we've added two underscores before the `stackList` name - nothing more:

```
class Stack:
    def __init__(self):
        self.__stackList = []

stackObject = Stack()
print(len(stackObject.__stackList))
```

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.

```
class Stack:
    def __init__(self):
        self.__stackList = []

stackObject = Stack()
print(len(stackObject.__stackList))
```

Console >_

```
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(len(stackObject.__stackList))
AttributeError: 'Stack' object has no attribute '__stackList'
```

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.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val
```

```
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

stackObject = Stack()

stackObject.push(3)
stackObject.push(2)
stackObject.push(1)

print(stackObject.pop())
print(stackObject.pop())
print(stackObject.pop())
```

Console >_

```
1
2
3
```

```
stackObject = Stack()

stackObject.push(3)
stackObject.push(2)
stackObject.push(1)

print(stackObject.pop())
print(stackObject.pop())
print(stackObject.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.

```
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

stackObject = Stack()

stackObject.push(3)
stackObject.push(2)
stackObject.push(1)

print(stackObject.pop())
print(stackObject.pop())
print(stackObject.pop())
```

Console >_

```
1
2
3
```

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 `__stackList` 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 `__stackList` list → `self.__stackList`;
- with `__stackList` ready to be used, you can perform the third and last step → `self.__stackList.append(val)`.

The class declaration is complete, and all its components have been listed. The class is ready for use.

```
print(stackObject.pop())
```

Console >_

```
1
2
3
```

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.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val


stackObject1 = Stack()
stackObject2 = Stack()

stackObject1.push(3)
stackObject2.push(stackObject1.pop())

print(stackObject2.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.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val


stackObject1 = Stack()
stackObject2 = Stack()

stackObject1.push(3)
stackObject2.push(stackObject1.pop())

print(stackObject2.pop())
```

Console >_

```
3
```

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.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val

littleStack = Stack()
anotherStack = Stack()
funnyStack = Stack()

littleStack.push(1)
anotherStack.push(littleStack.pop() + 1)
funnyStack.push(anotherStack.pop() - 2)

print(funnyStack.pop())
```

So, what's the result? Run the program and check if you were right.

```
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

# enter code here
littleStack = Stack()
anotherStack = Stack()
funnyStack = Stack()

littleStack.push(1)
anotherStack.push(littleStack.pop() + 1)
funnyStack.push(anotherStack.pop() - 2)

print(funnyStack.pop())
```

Console >_

```
0
```

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 after 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 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
```

Console >_

```
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 `__stackList` 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 superclass'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.

```
        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
```

Console >_

The 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
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

Console >_

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 `getSum`. Its only task will be to **return the `__sum` value**.

Here it is:

```
def getSum(self):
    return self.__sum
```

```
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

    def getSum(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
```

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.

```
    stackObject.push(i)
print(stackObject.getSum())

for i in range(5):
    print(stackObject.pop())
```

Console >_

```
10
4
3
2
1
0
```

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.

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:
  - `exampleObject1` only has the property named `first`;
  - `exampleObject2` has two properties: `first` and `second`;
  - `exampleObject3` has been enriched with a property named `third` just on the fly, outside the class's code - this is possible and fully permissible.

Here is an example:

```
class ExampleClass:
    def __init__(self, val = 1):
        self.first = val

    def setSecond(self, val):
        self.second = val


exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)

exampleObject2.setSecond(3)

exampleObject3 = ExampleClass(4)
exampleObject3.third = 5


print(exampleObject1.__dict__)
print(exampleObject2.__dict__)
print(exampleObject3.__dict__)
```

The program's output clearly shows that our assumptions are correct - here it is:

```
{'first': 1}
{'second': 3, 'first': 2}
{'third': 5, 'first': 4}
```

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.

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}
```

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.

```
1  class ExampleClass:
2      def __init__(self, val = 1):
3          self.__first = val
4
5      def setSecond(self, val = 2):
6          self.__second = val
7
8
9  exampleObject1 = ExampleClass()
10 exampleObject2 = ExampleClass(2)
11
12 exampleObject2.setSecond(3)
13
14 exampleObject3 = ExampleClass(4)
15 exampleObject3.__third = 5
16
17
18 print(exampleObject1.__dict__)
19 print(exampleObject2.__dict__)
20 print(exampleObject3.__dict__)
```

Console >_

```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

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(exampleObject1._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 an instance variable outside the class code. In this case, it'll behave like any other ordinary property.

Console >_

```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

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

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)
exampleObject3 = ExampleClass(4)

print(exampleObject1.__dict__, exampleObject1.counter)
print(exampleObject2.__dict__, exampleObject2.counter)
print(exampleObject3.__dict__, exampleObject3.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
```

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

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)
exampleObject3 = ExampleClass(4)

print(exampleObject1.__dict__, exampleObject1._ExampleClass__counter)
print(exampleObject2.__dict__, exampleObject2._ExampleClass__counter)
print(exampleObject3.__dict__, exampleObject3._ExampleClass__counter)
```

Console >_

```
{'_ExampleClass__first': 1} 3
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3
```

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:

- we define one class named `ExampleClass`;
- the class defines one class variable named `varia`;
- the class constructor sets the variable with the parameter's value;
- 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)
- 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.

```
class ExampleClass:
    varia = 1
    def __init__(self, val):
        ExampleClass.varia = val

print(ExampleClass.__dict__)
exampleObject = ExampleClass(2)

print(ExampleClass.__dict__)
print(exampleObject.__dict__)
```

Console >_

```
  File "main.py", line 1
    ,class ExampleClass:
    ^
SyntaxError: invalid syntax
{'__module__': '__main__', 'varia': 1, '__init__': <function ExampleClass.
{'__module__': '__main__', 'varia': 2, '__init__': <function ExampleClass.
{}
```

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(exampleObject.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
```

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

exampleObject = ExampleClass(1)

print(exampleObject.a)
print(exampleObject.b)
```

Console >_

```
1
Traceback (most recent call last):
  File "main.py", line 11, in <module>
    print(exampleObject.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
1
Traceback (most recent call last):
  File "main.py", line 11, in <module>
    print(exampleObject.b)
AttributeError: 'ExampleClass' object has no attribute '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

exampleObject = ExampleClass(1)
print(exampleObject.a)

if hasattr(exampleObject, 'b'):
    print(exampleObject.b)
```

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1

exampleObject = ExampleClass(1)
print(exampleObject.a)

try:
    print(exampleObject.b)
except AttributeError:
    if hasattr(exampleObject, 'b'):
        print(exampleObject.b)
```

Console >_

```
  File "main.py", line 15
    print(exampleObject.b)
        ^
IndentationError: expected an indented block
1
1
```

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

exampleObject = ExampleClass()

print(hasattr(exampleObject, 'b'))
print(hasattr(exampleObject, '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.

```
class ExampleClass:
    attr = 1

print(hasattr(ExampleClass, 'attr'))
print(hasattr(ExampleClass, 'prop'))


class ExampleClass:
    a = 1
    def __init__(self):
        self.b = 2

exampleObject = ExampleClass()

print(hasattr(exampleObject, 'b'))
print(hasattr(exampleObject, 'a'))
print(hasattr(ExampleClass, 'b'))
print(hasattr(ExampleClass, 'a'))
```

Console >_

```
True
False
True
False
True
True
False
True
```

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
```

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.

```
class Classy:
    def method(self):
        print("method")

obj = Classy()
obj.method()
```

Console >_

```
method
```

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
```

Console >_

```
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
```

```
# test examples here

class Classy:
    varia = 2
    def method(self):
        print(self.varia, self.var)

obj = Classy()
obj.var = 3
obj.method()


class Classy:
    def other(self):
        print("other")

    def method(self):
        print("method")
        self.other()

obj = Classy()
obj.method()
```

Console >_

```
2 3
```

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
```

```
class Classy:
    def other(self):
        print("other")

    def method(self):
        print("method")
        self.other()

obj = Classy()
obj.method()
```

Console >_

```
2 3
2 3
method
other
```

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
```

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 superclasses, but we'll discuss this issue later.)

```
class Classy:
    def __init__(self, value):
        self.var = value

obj1 = Classy("object")

print(obj1.var)
```

Console >_

```
object
```

Methods in detail: continued

As `__init__` is a method, and a method is a function, you can do the same tricks with constructors/methods as you do with ordinary functions.

The example in the editor shows how to define a constructor with a default argument value. Test it.

The code outputs:

```
object
None
```

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
```

Run the program, and test it.

```
class Classy:
    def __init__(self, value = None):
        self.var = value

obj1 = Classy("object")
obj2 = Classy()

print(obj1.var)
print(obj2.var)
```

Console >_

```
object
None
```

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__)
```

Console >_

```
_weakref__': <attribute '__weakref__' of 'Classy' objects>, '__doc__': None}
```

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
```

Note: a statement like this one:

`print(obj.__name__)` will cause an error.

```
class Classy:
    pass

print(Classy.__name__)
obj = Classy()
print(type(obj).__name__)
```

Console >_

```
Classy
Classy
```

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__
```

As you know, any module named `__main__` is actually not a module, but the **file currently being run**.

```
class Classy:
    pass

print(Classy.__module__)
obj = Classy()
print(obj.__module__)
```

Console >_

```
__main__
__main__
```

The inner life of classes and objects: continued

`__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 )
```

Note: **a class without explicit superclasses points to object** (a predefined Python class) as its direct ancestor.

```
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)
```

Console >_

```
( object )
( object )
( SuperOne SuperTwo )
```

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.

introspection

the ability of a program to examine the type or properties of an object at runtime

reflection

the ability of a program to manipulate the values, properties and/or functions of an object at runtime

Investigating classes

What can you find out about classes in Python? The answer is simple - everything.

Both reflection and introspection enable a programmer to do anything with every object, no matter where it comes from.

Analyze the the code in the editor.

The function named `incIntsI()` gets an object of any 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.

```
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__)
```

Console >_

```
{'a': 1, 'b': 2, 'i': 3, 'ireal': 3.5, 'integer': 4, 'z': 5}
{'a': 1, 'b': 2, 'i': 4, 'ireal': 3.5, 'integer': 5, 'z': 5}
```

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>
```

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.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

sun = Star("Sun", "Milky Way")
print(sun)
```

Console >_

```
<__main__.Star object at 0x7f658cec4d90>
```

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.

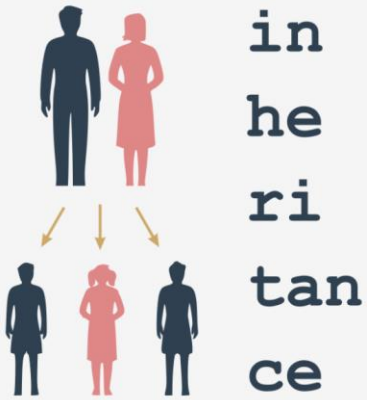
```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

    def __str__(self):
        return self.name + ' in ' + self.galaxy

sun = Star("Sun", "Milky Way")
print(sun)
```

Console >_

```
Sun in Milky Way
```

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.



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.

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 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
```

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**.

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass


for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:
    for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:
        print(issubclass(cls1, cls2), end="\t")
    print()
```

Console >_

```
True    False   False
True    True    False
True    True    True
```

Inheritance: `isinstance()`

As you already 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.

```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass


myVehicle = Vehicle()
myLandVehicle = LandVehicle()
myTrackedVehicle = TrackedVehicle()

for obj in [myVehicle, myLandVehicle, myTrackedVehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```

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
```

Let's make the result more readable once again:

↓ is an instance of →	Vehicle	LandVehicle	TrackedVehicle
myVehicle	`True`	`False`	`False`
myLandVehicle	`True`	`True`	`False`
myTrackedVehicle	`True`	`True`	`True`

Does the table confirm our expectations?

```
    pass


myVehicle = Vehicle()
myLandVehicle = LandVehicle()
myTrackedVehicle = TrackedVehicle()

for obj in [myVehicle, myLandVehicle, myTrackedVehicle]:
    for cls in [Vehicle, LandVehicle, TrackedVehicle]:
        print(isinstance(obj, cls), end="\t")
    print()
```

Console >_

```
True    False   False
True    True    False
True    True    True
```

Inheritance: the `is` operator

There is also a Python operator worth mentioning, as it refers directly to objects - here it is:

```
objectOne is objectTwo
```

The `is` operator checks whether two variables (`objectOne` and `objectTwo` 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
```

The results prove that `ob1` and `ob3` are actually the same objects, while `str1` and `str2` aren't, despite their contents being the same.

```
class SampleClass:
    def __init__(self, val):
        self.val = val

ob1 = SampleClass(0)
ob2 = SampleClass(2)
ob3 = ob1
ob3.val += 1

print(ob1 is ob2)
print(ob2 is ob3)
print(ob3 is ob1)
print(ob1.val, ob2.val, ob3.val)

str1 = "Mary had a little "
str2 = "Mary had a little lamb"
str1 += "lamb"

print(str1 == str2, str1 is str2)
```

Console >_

How Python finds properties and methods

Now we're going to look at how Python deals with inheriting methods.

Take a look at the example in the editor. Let's analyze it:

- there is a class named `Super`, which defines its own constructor used to assign the object's property, named `name`.
- the class defines the `__str__()` method, too, which makes the class able to present its identity in clear text form.
- the class is next used as a base to create a subclass named `Sub`. The `Sub` class defines its own constructor, which invokes the one from the superclass. Note how we've done it: `Super.__init__(self, name)`.
- we've explicitly named the superclass, and pointed to the method to invoke `__init__()`, providing all needed arguments.
- we've instantiated one object of class `Sub` and printed it.

The code outputs:

```
My name is Andy.
```

Note: As there is no `__str__()` method within the `Sub` class, the printed string is to be produced within the `Super` class. This means that the `__str__()` method has been inherited by the `Sub` class.

```
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__(self, name)

obj = Sub("Andy")

print(obj)
```

Console >_

```
My name is Andy.
```

How Python finds properties and methods: continued

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**.

```
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 >_

```
My name is Andy.
```

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:

```
2
1
```

```
# Testing properties: class variables
class Super:
    supVar = 1

class Sub(Super):
    subVar = 2

obj = Sub()

print(obj.subVar)
print(obj.supVar)
```

Console >_

```
2
1
```

How Python finds properties and methods: continued

The same effect can be observed with **instance variables** - take a look at the second example in the editor.

The `Sub` class constructor creates an instance variable named `subVar`, while the `Super` constructor does the same with a variable named `supVar`. As previously, both variables are accessible from within the object of class `Sub`.

The program's output is:

```
12
11
```

Note: the existence of the `supVar` variable is obviously conditioned by the `Super` class constructor invocation. Omitting it would result in the absence of the variable in the created object (try it yourself).

```
# 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)
```

Console >_

```
12
11
```

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**.

```
class Level1:
    varia1 = 100
    def __init__(self):
        self.var1 = 101

    def fun1(self):
        return 102

class Level2(Level1):
    varia2 = 200
    def __init__(self):
        super().__init__()
        self.var2 = 201

    def fun2(self):
        return 202

class Level3(Level2):
    varia3 = 300
    def __init__(self):
        super().__init__()
        self.var3 = 301

    def fun3(self):
        return 302

obj = Level3()
print(obj.varia1, obj.var1, obj.fun1())
print(obj.varia2, obj.var2, obj.fun2())
print(obj.varia3, obj.var3, obj.fun3())
```

Console >_

```
100 101 102
200 201 202
300 301 302
```

How Python finds properties and methods: continued

Multiple inheritance occurs when a class has more than one superclass.

Syntactically, such inheritance is presented as a comma-separated list of superclasses put inside parentheses after the new class name - just like here:

```
class SuperA:
    varA = 10
    def funA(self):
        return 11

class SuperB:
    varB = 20
    def funB(self):
        return 21

class Sub(SuperA, SuperB):
    pass

obj = Sub()

print(obj.varA, obj.funA())
print(obj.varB, obj.funB())
```

```
class SuperA:
    varA = 10
    def funA(self):
        return 11

class SuperB:
    varB = 20
    def funB(self):
        return 21

class Sub(SuperA, SuperB):
    pass

obj = Sub()

print(obj.varA, obj.funA())
print(obj.varB, obj.funB())
```

Console >_

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
```

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?

Console >_

```
10 11
20 21
```

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
```

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())

```

Console >_

```
200 201
```

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 `varRight` comes from the `Right` class, and `varLeft` 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
```

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
```

Do you see the same, or something different?

```
class Left:
    var = "L"
    varLeft = "LL"
    def fun(self):
        return "Left"


class Right:
    var = "R"
    varRight = "RR"
    def fun(self):
        return "Right"

class Sub(Left, Right):
    pass


obj = Sub()

print(obj.var, obj.varLeft, obj.varRight, obj.fun())
```

Console >_

```
L LL RR Left
```

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 `doit()` method.
- the `doit()` 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.

```
class One:
    def doit(self):
        print("doit from One")

    def doanything(self):
        self.doit()

class Two(One):
    def doit(self):
        print("doit from Two")

one = One()
two = Two()

one.doanything()
two.doanything()
```

Console >_

```
doit from One
doit from Two
```

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 `doit()` 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:

```
doit from One
doit from Two
```

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**.

```
one.doanything()
two.doanything()
```

Console >_

```
doit from One
doit from Two
```

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 `controltrack()` method, which will be implemented later)
- a wheeled vehicle turns when its front wheels turn (this is done by the `turnfrontwheels()` 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.

```
import time

class TrackedVehicle:
    def controltrack(left, stop):
        pass

    def turn(left):
        controltrack(left, True)
        time.sleep(0.25)
        controltrack(left, False)


class WheeledVehicle:
    def turnfrontwheels(left, on):
        pass

    def turn(left):
        turnfrontwheels(left, True)
        time.sleep(0.25)
        turnfrontwheel(left, False)
```

Success (1.75s)

Console >_

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 `changedirection()`; 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 `changedirection()` method by using the specific (concrete) method named `controltrack()`
- respectively, the subclass named `WheeledVehicle` does the same trick, but uses the `turnfrontwheel()` 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**.

```
import time

class Vehicle:
    def changedirection(left, on):
        pass

    def turn(left):
        changedirection(left, True)
        time.sleep(0.25)
        changedirection(left, False)

class TrackedVehicle(Vehicle):
    def controltrack(left, stop):
        pass

    def changedirection(left, on):
        controltrack(left, on)

class WheeledVehicle(Vehicle):
    def turnfrontwheels(left, on):
        pass

    def changedirection(left, on):
        turnfrontwheels(left, on)
```

Success (1.44s)

Console >_

How to build a hierarchy of classes: continued

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.

```
import time

class Tracks:
    def changedirection(self, left, on):
        print("tracks: ", left, on)

class Wheels:
    def changedirection(self, left, on):
        print("wheels: ", left, on)

class Vehicle:
    def __init__(self, controller):
        self.controller = controller

    def turn(self, left):
        self.controller.changedirection(left, True)
        time.sleep(0.25)
        self.controller.changedirection(left, False)

wheeled = Vehicle(Wheels())
tracked = Vehicle(Tracks())

wheeled.turn(True)
tracked.turn(False)
```

Console >_

```
wheels:  True True
wheels:  True False
tracks:  False True
tracks:  False False
```

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
```


```
tracked = Vehicle(Tracks())

wheeled.turn(True)
tracked.turn(False)
```

Console >_

```
wheels:  True True
wheels:  True False
tracks:  False True
tracks:  False 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.

https://en.wikipedia.org/wiki/Single_responsibility_principle

More about exceptions

Discussing object programming offers a very good opportunity to return to exceptions. The objective nature of Python's exceptions makes them a very flexible tool, able to fit to specific needs, even those you don't yet know about.

Before we dive into the **objective face of exceptions**, we want to show you some syntactical and semantic aspects of the way in which Python treats the try-except block, as it offers a little more than what we have presented so far.

The first feature we want discuss here is an additional, possible branch that can be placed inside (or rather, directly behind) the try-except block - it's the part of the code starting with `else` - just like in the example in the editor.

A code labelled in this way is executed when (and only when) no exception has been raised inside the `try:` part. We can say that exactly one branch can be executed after `try:` - either the one beginning with `except` (don't forget that there can be more than one branch of this kind) or the one starting with `else`.

Note: the `else:` branch has to be located after the last `except` branch.

The example code produces the following output:

```
Everything went fine
0.5
Division failed
None
```

```
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))
```

Console >_

```
Everything went fine
0.5
Division failed
None
```

More about exceptions

The try-except block can be extended in one more way - by adding a part headed by the `finally` keyword (it must be the last branch of the code designed to handle exceptions).

Note: these two variants (`else` and `finally`) aren't dependent in any way, and they can coexist or occur independently.

The `finally` block is always executed (it finalizes the try-except block execution, hence its name), no matter what happened earlier, even when raising an exception, no matter whether this has been handled or not.

Look at the code in the editor. It outputs:

```
Everything went fine
It's time to say good bye
0.5
Division failed
It's time to say good bye
None
```

```
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))
```

Console >_

```
Everything went fine
It's time to say goodbye
0.5
Division failed
It's time to say goodbye
None
```

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.

```
try:
    i = int("Hello!")
except Exception as e:
    print(e)
    print(e.__str__())
```

Console >_

```
invalid literal for int() with base 10: 'Hello!'
invalid literal for int() with base 10: 'Hello!'
```

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 `printExcTree()` 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 `printExcTree()` 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:

```
def printExcTree(thisclass, nest = 0):
    if nest > 1:
        print("    |" * (nest - 1), end="")
    if nest > 0:
        print("    +---", end="")

    print(thisclass.__name__)

    for subclass in thisclass.__subclasses__():
        printExcTree(subclass, nest + 1)

printExcTree(BaseException)
```

Console >_

```
BaseException
   +---Exception
   |    +---TypeError
   |    +---StopAsyncIteration
   |    +---StopIteration
   |    +---ImportError
   |    |    +---ModuleNotFoundError
   |    |    +---ZipImportError
```

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')
```

```
def printargs(args):
    lng = len(args)
    if lng == 0:
        print("")
    elif lng == 1:
        print(args[0])
    else:
        print(str(args))

try:
    raise Exception
except Exception as e:
    print(e, e.__str__(), sep=' : ' ,end=' : ')
    printargs(e.args)

try:
    raise Exception("my exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    printargs(e.args)

try:
    raise Exception("my", "exception")
except Exception as e:
    print(e, e.__str__(), sep=' : ', end=' : ')
    printargs(e.args)
```

Console >_

```
 :  :
my exception : my exception : my exception
('my', 'exception') : ('my', 'exception') : ('my', 'exception')
```

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:

```
class MyZeroDivisionError(ZeroDivisionError):
    pass

def doTheDivision(mine):
    if mine:
        raise MyZeroDivisionError("some worse news")
    else:
        raise ZeroDivisionError("some bad news")

for mode in [False, True]:
    try:
        doTheDivision(mode)
    except ZeroDivisionError:
        print('Division by zero')


for mode in [False, True]:
    try:
        doTheDivision(mode)
    except MyZeroDivisionError:
        print('My division by zero')
    except ZeroDivisionError:
        print('Original division by zero')
```

Console >_

- 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 `doTheDivision()` 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!)

```
    except ZeroDivisionError:
        print('Original division by zero')
```

Console >_

```
Division by zero
Division by zero
Original division by zero
My division by zero
```

How to create your own exception: continued

When you're going to build a completely new universe filled with completely new creatures that have nothing in common with all the familiar things, you may want to **build your own exception structure**.

For example, if you work on a large simulation system which is intended to model the activities of a pizza restaurant, it can be desirable to form a separate hierarchy of exceptions.

You can start building it by **defining a general exception as a new base class** for any other specialized exception. We've done in in the following way:

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(message)
        self.pizza = pizza
```

Note: we're going to collect more specific information here than a regular `Exception` does, so our constructor will take two arguments:

- one specifying a pizza as a subject of the process,
- and one containing a more or less precise description of the problem.

As you can see, we pass the second parameter to the superclass constructor, and save the first inside our own property.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese
```

Console >_

A more specific problem (like an excess of cheese) can require a more specific exception. It's possible to derive the new class from the already defined `PizzaError` class, like we've done here:

```
class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese
```

The `TooMuchCheeseError` exception needs more information than the regular `PizzaError` exception, so we add it to the constructor - the name `cheese` is then stored for further processing.

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 `makePizza()` 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, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def makePizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no such pizza on the menu")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "too much cheese")
    print("Pizza ready!")


for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        makePizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

```
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 makePizza(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:
                makePizza(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.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza, cheese, message):
        PizzaError.__init__(self, pizza, message)
        self.cheese = cheese


def makePizza(pizza, cheese):
    if pizza not in ['margherita', 'capricciosa', 'calzone']:
        raise PizzaError(pizza, "no such pizza on the menu")
    if cheese > 100:
        raise TooMuchCheeseError(pizza, cheese, "too much cheese")
    print("Pizza ready!")


for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:
    try:
        makePizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

Console >_

```
Pizza ready!
too much cheese : 110
no such pizza on the menu : mafia
```

Generators - where to find them

Generator - what do you associate this word with? Perhaps it refers to some electronic device. Or perhaps it refers to a heavy and serious machine designed to produce power, electrical or other.

A Python generator is **a piece of specialized code able to produce a series of values, and to control the iteration process**. This is why generators are very often called **iterators**, and although some may find a very subtle distinction between these two, we'll treat them as one.

You may not realize it, but you've encountered generators many, many times before. Take a look at the very simple snippet:

```
for i in range(5):
    print(i)
```

The `range()` function is, in fact, a generator, which is (in fact, again) an iterator

What is the difference?

A function returns one, well-defined value - it may be the result of a more or less complex evaluation of, e.g., a polynomial, and is invoked once - only once.

A generator **returns a series of values**, and in general, is (implicitly) invoked more than once.

In the example, the `range()` generator is invoked six times, providing five subsequent values from zero to four, and finally signaling that the series is complete.

The above process is completely transparent. Let's shed some light on it. Let's show you the **iterator protocol**.

```
for i in range(5):
    print(i)
```

Console >_

```
0
1
2
3
4
```

Generators - where to find them: continued

The **iterator protocol is a way in which an object should behave to conform to the rules imposed by the context of the `for` and `in` statements**. An object conforming to the iterator protocol is called an **iterator**.

An iterator must provide two methods:

- `__iter__()` which should **return the object itself** and which is invoked once (it's needed for Python to successfully start the iteration)
- `__next__()` which is intended to **return the next value** (first, second, and so on) of the desired series - it will be invoked by the `for` / `in` statements in order to pass through the next iteration; if there are no more values to provide, the method should **raise the `StopIteration` exception**.

Does it sound strange? Not at all. Look at the example in the editor.

We've built a class able to iterate through the first `n` values (where `n` is a constructor parameter) of the Fibonacci numbers.

Let us remind you - the Fibonacci numbers (*Fib*) are defined as follows:

$Fib_1 = 1$
$Fib_2 = 1$
$Fib_i = Fib_{i-1} + Fib_{i-2}$

In other words:

- the first two Fibonacci numbers are equal to 1;
- any other Fibonacci number is the sum of the two previous ones (e.g., $Fib_3 = 2$, $Fib_4 = 3$, $Fib_5 = 5$, and so on)

```
class Fib:
    def __init__(self, nn):
        print("__init__")
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1

    def __iter__(self):
        print("__iter__")
        return self

    def __next__(self):
        print("__next__")
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret

for i in Fib(10):
    print(i)
```

Console >_

```
__next__
8
__next__
13
__next__
21
__next__
34
__next__
```

Let's dive into the code:

- lines 2 through 6: the class constructor prints a message (we'll use this to trace the class's behavior), prepares some variables (`__n` to store the series limit, `__i` to track the current Fibonacci number to provide, and `__p1` along with `__p2` to save the two previous numbers);
- lines 8 through 10: the `__iter__` method is obliged to return the iterator object itself; its purpose may be a bit ambiguous here, but there's no mystery; try to imagine an object which is not an iterator (e.g., it's a collection of some entities), but one of its components is an iterator able to scan the collection; the `__iter__` method should **extract the iterator and entrust it with the execution of the iteration protocol**; as you can see, the method starts its action by printing a message;
- lines 12 through 21: the `__next__` method is responsible for creating the sequence; it's somewhat wordy, but this should make it more readable; first, it prints a message, then it updates the number of desired values, and if it reaches the end of the sequence, the method breaks the iteration by raising the StopIteration exception; the rest of the code is simple, and it precisely reflects the definition we showed you earlier;
- lines 23 and 24 make use of the iterator.

The code produces the following output:

```
__init__
__next__
8
__next__
13
__next__
21
__next__
34
__next__
55
__next__
```

Look:

- the iterator object is instantiated first;
- next, Python invokes the `__iter__` method to get access to the actual iterator;
- the `__next__` method is invoked eleven times - the first ten times produce useful values, while the eleventh terminates the iteration.

Generators - where to find them: continued

The previous example shows you a solution where the **iterator object is a part of a more complex class**.

The code isn't really sophisticated, but it presents the concept in a clear way.

Take a look at the code in the editor.

We've built the `Fib` iterator into another class (we can say that we've composed it into the `Class` class). It's instantiated along with `Class`'s object.

The object of the class may be used as an iterator when (and only when) it positively answers to the `__iter__` invocation - this class can do it, and if it's invoked in this way, it provides an object able to obey the iteration protocol.

This is why the output of the code is the same as previously, although the object of the `Fib` class isn't used explicitly inside the `for` loop's context.

```
class Fib:
    def __init__(self, nn):
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1

    def __iter__(self):
        print("Fib iter")
        return self

    def __next__(self):
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret

class Class:
    def __init__(self, n):
        self.__iter = Fib(n)

    def __iter__(self):
        print("Class iter")
        return self.__iter;

object = Class(8)
for i in object:
    print(i)
```

Console >_

```
Class iter
Fib iter
1
1
2
3
5
8
13
21
```

The `yield` statement

The iterator protocol isn't particularly difficult to understand and use, but it is also indisputable that the **protocol is rather inconvenient**.

The main discomfort it brings is **the need to save the state of the iteration between subsequent `__iter__` invocations**.

For example, the `Fib` iterator is forced to precisely store the place in which the last invocation has been stopped (i.e., the evaluated number and the values of the two previous elements). This makes the code larger and less comprehensible.

This is why Python offers a much more effective, convenient, and elegant way of writing iterators.

The concept is fundamentally based on a very specific and powerful mechanism provided by the `yield` keyword.

You may think of the `yield` keyword as a smarter sibling of the `return` statement, with one essential difference.

Take a look at this function:

```
def fun(n):
    for i in range(n):
        return i
```

It looks strange, doesn't it? It's clear that the `for` loop has no chance to finish its first execution, as the `return` will break it irrevocably.

We've replaced exactly one word in the code - can you see it?

```
def fun(n):
    for i in range(n):
        yield i
```

We've added `yield` instead of `return`. This little amendment **turns the function into a generator**, and executing the `yield` statement has some very interesting effects.

First of all, it provides the value of the expression specified after the `yield` keyword, just like `return`, but doesn't lose the state of the function.

All the variables' values are frozen, and wait for the next invocation, when the execution is resumed (not taken from scratch, like after `return`).

There is one important limitation: such a **function should not be invoked explicitly** as - in fact - it isn't a function anymore; **it's a generator object**.

The invocation will **return the object's identifier**, not the series we expect from the generator.

Due to the same reasons, the previous function (the one with the `return` statement) may only be invoked explicitly, and must not be used as a generator.

How to build a generator

Let us show you the new generator in action.

This is how we can use it:

Moreover, invoking the function won't change anything - the `for` loop will start from scratch and will be broken immediately.

We can say that such a function is not able to save and restore its state between subsequent invocations.

This also means that a function like this **cannot be used as a generator**.

```
def fun(n):
    for i in range(n):
        yield i

for v in fun(5):
    print(v)
```

Can you guess the output?

Check

```
0
1
2
3
4
```

How to build your own generator

What if you need a **generator to produce the first *n* powers of *2***?

Nothing easier. Just look at the code in the editor.

Can you guess the output? Run the code to check your guesses.

```
def powersOf2(n):
    pow = 1
    for i in range(n):
        yield pow
        pow *= 2

for v in powersOf2(8):
    print(v)
```

Console >_

```
1
2
4
8
16
32
64
128
```

Generators may also be used within **list comprehensions**, just like here:

```
def powersOf2(n):
    pow = 1
    for i in range(n):
        yield pow
        pow *= 2

t = [x for x in powersOf2(5)]

print(t)
```

Run the example and check the output.

The `list()` function can transform a series of subsequent generator invocations into **a real list**:

```
def powersOf2(n):
    pow = 1
    for i in range(n):
        yield pow
        pow *= 2

t = list(powersOf2(3))

print(t)
```

Again, try to predict the output and run the code to check your predictions.

Moreover, the context created by the `in` operator allows you to use a generator, too.

The example shows how to do it:

```
def powersOf2(n):
    pow = 1
    for i in range(n):
        yield pow
        pow *= 2

for i in range(20):
    if i in powersOf2(4):
        print(i)
```

What's the code's output? Run the program and check.

Now let's see a **Fibonacci number generator**, and ensure that it looks much better than the objective version based on the direct iterator protocol implementation.

Here it is:

```
def Fib(n):
    p = pp = 1
    for i in range(n):
        if i in [0, 1]:
            yield 1
        else:
            n = p + pp
            pp, p = p, n
            yield n

fibs = list(Fib(10))

print(fibs)
```

Guess the output (a list) produced by the generator, and run the code to check if you were right.

More about list comprehensions

You should be able to remember the rules governing the creation and use of a very special Python phenomenon named **list comprehension - a simple and very impressive way of creating lists and their contents**.

In case you need it, we've provided a quick reminder in the editor.

There are two parts inside the code, both creating a list containing a few of the first natural powers of ten.

The former uses a routine way of utilizing the `for` loop, while the latter makes use of the list comprehension and builds the list in situ, without needing a loop, or any other extended code.

It looks like the list is created inside itself - it's not true, of course, as Python has to perform nearly the same operations as in the first snippet, but it is indisputable that the second formalism is simply more elegant, and lets the reader avoid any unnecessary details.

The example outputs two identical lines containing the following text:

```
[1, 10, 100, 1000, 10000, 100000]
```

Run the code to check if we're right.

```
listOne = []

for ex in range(6):
    listOne.append(10 ** ex)


listTwo = [10 ** ex for ex in range(6)]

print(listOne)
print(listTwo)
```

Console >_

```
[1, 10, 100, 1000, 10000, 100000]
[1, 10, 100, 1000, 10000, 100000]
```

More about list comprehensions: continued

There is a very interesting syntax we want to show you now. Its usability is not limited to list comprehensions, but we have to admit that comprehensions are the ideal environment for it.

It's a **conditional expression - a way of selecting one of two different values based on the result of a Boolean expression**.

Look:

```
expression_one if condition else expression_two
```

It may look a bit surprising at first glance, but you have to keep in mind that it is **not a conditional instruction**. Moreover, it's not an instruction at all. It's an operator.

The value it provides is equal to `expression_one` when the condition is `True`, and `expression_two` otherwise.

A good example will tell you more. Look at the code in the editor.

The code fills a list with `1`'s and `0`s - if the index of a particular element is odd, the element is set to `0`, and to `1` otherwise.

Simple? Maybe not at first glance. Elegant? Indisputably.

Can you use the same trick within a list comprehension? Yes, you can.

```
lst = []

for x in range(10):
    lst.append(1 if x % 2 == 0 else 0)

print(lst)
```

Console >_

```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

More about list comprehensions: continued

Look at the example in the editor.

Compactness and elegance - these two words come to mind when looking at the code.

So, what do they have in common, generators and list comprehensions? Is there any connection between them? Yes. A rather loose connection, but an unequivocal one.

Just one change can **turn any comprehension into a generator**.

Now look at the code below and see if you can find the detail that turns a list comprehension into a generator:

```
lst = [1 if x % 2 == 0 else 0 for x in range(10)]
genr = (1 if x % 2 == 0 else 0 for x in range(10))

for v in lst:
    print(v, end=" ")
print()

for v in genr:
    print(v, end=" ")
print()
```

It's the **parentheses**. The brackets make a comprehension, the parentheses make a generator.

The code, however, when run, produces two identical lines:

```
1 0 1 0 1 0 1 0 1 0
1 0 1 0 1 0 1 0 1 0
```

```
lst = [1 if x % 2 == 0 else 0 for x in range(10)]

print(lst)
```

Console >_

```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

How can you know that the second assignment creates a generator, not a list?

There is some proof we can show you. Apply the `len()` function to both these entities.

`len(lst)` will evaluate to `10`. Clear and predictable. `len(genr)` will raise an exception, and you will see the following message:

```
TypeError: object of type 'generator' has no len()
```

Of course, saving either the list or the generator is not necessary - you can create them exactly in the place where you need them - just like here:

```
for v in [1 if x % 2 == 0 else 0 for x in range(10)]:
    print(v, end=" ")
print()

for v in (1 if x % 2 == 0 else 0 for x in range(10)):
    print(v, end=" ")
print()
```

Note: the same appearance of the output doesn't mean that both loops work in the same way. In the first loop, the list is created (and iterated through) as a whole - it actually exists when the loop is being executed.

In the second loop, there is no list at all - there are only subsequent values produced by the generator, one by one.

Carry out your own experiments.

Console >_

```
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

The `lambda` function

The `lambda` function is a concept borrowed from mathematics, more specifically, from a part called *the Lambda calculus*, but these two phenomena are not the same.

Mathematicians use *the Lambda calculus* in many formal systems connected with logic, recursion, or theorem provability. Programmers use the `lambda` function to simplify the code, to make it clearer and easier to understand.

A `lambda` function is a function without a name (you can also call it **an anonymous function**). Of course, such a statement immediately raises the question: how do you use anything that cannot be identified?

Fortunately, it's not a problem, as you can name such a function if you really need, but, in fact, in many cases the `lambda` function can exist and work while remaining fully incognito.

The declaration of the `lambda` function doesn't resemble a normal function declaration in any way - see for yourself:

```
lambda parameters : expression
```

Such a clause **returns the value of the expression when taking into account the current value of the current `lambda` argument**.

As usual, an example will be helpful. Our example uses three `lambda` functions, but gives them names. Look at it carefully:

```
two = lambda : 2
sqr = lambda x : x * x
pwr = lambda x, y : x ** y

for a in range(-2, 3):
    print(sqr(a), end=" ")
    print(pwr(a, two()))
```

Let's analzye it:

- the first `lambda` is an anonymous **parameterless function** that always returns `2`. As we've **assigned it to a variable named** `two`, we can say that the function is not anonymous anymore, and we can use the name to invoke it.
- the second one is a **one-parameter anonymous function** that returns the value of its squared argument. We've named it as such, too.
- the third `lambda` **takes two parameters** and returns the value of the first one raised to the power of the second one. The name of the variable which carries the `lambda` speaks for itself. We don't use `pow` to avoid confusion with the built-in function of the same name and the same purpose.

The program produces the following output:

```
4 4
1 1
0 0
1 1
4 4
```

This example is clear enough to show how `lambda`s are declared and how they behave, but it says nothing about why they're necessary, and what they're used for, since they can all be replaced with routine Python functions.

Where is the benefit?

How to use lambdas and what for?

The most interesting part of using lambdas appears when you can use them in their pure form - **as anonymous parts of code intended to evaluate a result**.

Imagine that we need a function (we'll name it `printfunction`) which prints the values of a given (other) function for a set of selected arguments.

We want `printfunction` to be universal - it should accept a set of arguments put in a list and a function to be evaluated, both as arguments - we don't want to hardcode anything.

Look at the example in the editor. This is how we've implemented the idea.

Let's analyze it. The `printfunction()` function takes two parameters:

- the first, a list of arguments for which we want to print the results;
- the second, a function which should be invoked as many times as the number of values that are collected inside the first parameter.

Note: we've also defined a function named `poly()` - this is the function whose values we're going to print. The calculation the function performs isn't very sophisticated - it's the polynomial (hence its name) of a form:

$f(x) = 2x^2 - 4x + 2$

The name of the function is then passed to the `printfunction()` along with a set of five different arguments - the set is built with a list comprehension clause.

The code prints the following lines:

```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

```
def printfunction(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')

def poly(x):
    return 2 * x**2 - 4 * x + 2

printfunction([x for x in range(-2, 3)], poly)
```

Console >_

```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

Can we avoid defining the `poly()` function, as we're not going to use it more than once? Yes, we can - this is the benefit a lambda can bring.

Look at the example below. Can you see the difference?

```
def printfunction(args, fun):
        for x in args:
                print('f(', x,')=', fun(x), sep='')

printfunction([x for x in range(-2, 3)], lambda x: 2 * x**2 - 4 * x + 2)
```

The `printfunction()` has remained exactly the same, but there is no `poly()` function. We don't need it anymore, as the polynomial is now directly inside the `printfunction()` invocation in the form of a lambda defined in the following way: `lambda x: 2 * x**2 - 4 * x + 2`.

The code has become shorter, clearer, and more legible.

Let us show you another place where lambdas can be useful. We'll start with a description of `map()`, a built-in Python function. Its name isn't too descriptive, its idea is simple, and the function itself is really usable.

Console >_

```
f(-2)=18
f(-1)=8
f(0)=2
f(1)=0
f(2)=2
```

Lambdas and the `map()` function

In the simplest of all possible cases, the `map()` function takes two arguments:

- a function;
- a list.

```
map(function, list)
```

The above description is extremely simplified, as:

- the second `map()` argument may be any entity that can be iterated (e.g., a tuple, or just a generator)
- `map()` can accept more than two arguments.

The `map()` **function applies the function passed by its first argument to all its second argument's elements, and returns an iterator delivering all subsequent function results**. You can use the resulting iterator in a loop, or convert it into a list using the `list()` function.

Can you see a role for any lambda here?

Look at the code in the editor - we've used two lambdas in it.

This is the intrigue:

- build the `list1` with values from `0` to `4`;
- next, use `map` along with the first `lambda` to create a new list in which all elements have been evaluated as `2` raised to the power taken from the corresponding element from `list1`;
- `list2` is printed then;
- in the next step, use the `map()` function again to make use of the generator it returns and to directly print all the values it delivers; as you can see, we've engaged the second `lambda` here - it just squares each element from `list2`.

```
list1 = [x for x in range(5)]
list2 = list(map(lambda x: 2 ** x, list1))
print(list2)
for x in map(lambda x: x * x, list2):
    print(x, end=' ')
print()
```

Console >_

```
[1, 2, 4, 8, 16]
1 4 16 64 256
```

Lambdas and the `filter()` function

Another Python function which can be significantly beautified by the application of a lambda is `filter()`.

It expects the same kind of arguments as `map()`, but does something different - it **filters its second argument while being guided by directions flowing from the function specified as the first argument** (the function is invoked for each list element, just like in `map()`).

The elements which return `True` from the function **pass the filter** - the others are rejected.

The example in the editor shows the `filter()` function in action.

Note: we've made use of the `random` module to initialize the random number generator (not to be confused with the generators we've just talked about) with the `seed()` function, and to produce five random integer values from `-10` to `10` using the `randint()` function.

The list is then filtered, and only the numbers which are even and greater than zero are accepted.

Of course, it's not likely that you'll receive the same results, but this is what our results looked like:

```
[6, 3, 3, 2, -7]
[6, 2]
```

```
from random import seed, randint

seed()
data = [ randint(-10,10) for x in range(5) ]
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))
print(data)
print(filtered)
```

Console >_

```
[0, -1, 10, 7, -1]
[10]
```

A brief look at closures

Let's start with a definition: **closure is a technique which allows the storing of values in spite of the fact that the context in which they have been created does not exist anymore**. Intricate? A bit.

Let's analyze a simple example:

```
def outer(par):
    loc = par

var = 1
outer(var)

print(var)
print(loc)
```

The example is obviously erroneous.

The last two lines will cause a `NameError` exception - neither `par` nor `loc` is accessible outside the function. Both the variables exist when and only when the `outer()` function is being executed.

Look at the example in the editor. We've modified the code significantly.

There is a brand new element in it - a function (named `inner`) inside another function (named `outer`).

How does it work? Just like any other function except for the fact that `inner()` may be invoked only from within `outer()`. We can say that `inner()` is `outer()`'s private tool - no other part of the code can access it.

```
def outer(par):
    loc = par
    def inner():
        return loc
    return inner

var = 1
fun = outer(var)
print(fun())
```

Console >_

```
1
```

Look carefully:

- the `inner()` function returns the value of the variable accessible inside its scope, as `inner()` can use any of the entities at the disposal of `outer()`
- the `outer()` function returns the `inner()` function itself; more precisely, it returns a copy of the `inner()` function, the one which was frozen at the moment of `outer()`'s invocation; the frozen function contains its full environment, including the state of all local variables, which also means that the value of `loc` is successfully retained, although `outer()` ceased to exist a long time ago.

In effect, the code is fully valid, and outputs:

```
1
```

The function returned during the `outer()` invocation is a **closure**.

Console >_

```
1
```

A brief look at closures: continued

A closure has to be invoked in exactly the same way in which it has been declared.

In the previous example (see the code below):

```
def outer(par):
        loc = par
        def inner():
                return loc
        return inner

var = 1
fun = outer(var)
print(fun())
```

the `inner()` function was parameterless, so we had to invoke it without arguments.

Now look at the code in the editor. It is fully possible to **declare a closure equipped with an arbitrary number of parameters**, e.g., one, just like the `power()` function.

This means that the closure not only makes use of the frozen environment, but it can also **modify its behavior by using values taken from the outside**.

```
def makeclosure(par):
    loc = par
    def power(p):
        return p ** loc
    return power

fsqr = makeclosure(2)
fcub = makeclosure(3)
for i in range(5):
    print(i, fsqr(i), fcub(i))
```

Console >_

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

This example shows one more interesting circumstance - you can **create as many closures as you want using one and the same piece of code**. This is done with a function named `makeclosure()`. Note:

- the first closure obtained from `makeclosure()` defines a tool squaring its argument;
- the second one is designed to cube the argument.

This is why the code produces the following output:

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

Carry out your own tests.

Console >_

```
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

Accessing files from Python code

One of the most common issues in the developer's job is to **process data stored in files** while the files are usually physically stored using storage devices - hard, optical, network, or solid-state disks.

It's easy to imagine a program that sorts 20 numbers, and it's equally easy to imagine the user of this program entering these twenty numbers directly from the keyboard.

It's much harder to imagine the same task when there are 20,000 numbers to be sorted, and there isn't a single user who is able to enter these numbers without making a mistake.

It's much easier to imagine that these numbers are stored in the disk file which is read by the program. The program sorts the numbers and doesn't send them to the screen, but instead creates a new file and saves the sorted sequence of numbers there.

If we want to implement a simple database, the only way to store the information between program runs is to save it into a file (or files if your database is more complex).

In principle, any non-simple programming problem relies on the use of files, whether it processes images (stored in files), multiplies matrices (stored in files), or calculates wages and taxes (reading data stored in files).

You may ask why we have waited until now to show you these issues.

The answer is very simple - Python's way of accessing and processing files is implemented using a consistent set of objects. There is no better moment to talk about it.

File names

Different operating systems can treat the files in different ways. For example, Windows uses a different naming convention than the one adopted in Unix/Linux systems.

If we use the notion of a canonical file name (a name which uniquely defines the location of the file regardless of its level in the directory tree) we can realize that these names look different in Windows and in Unix/Linux:

Windows

```
C:\directory\file
```

Linux

```
/directory/files
```

As you can see, systems derived from Unix/Linux don't use the disk drive letter (e.g., `C:`) and all the directories grow from one root directory called `/`, while Windows systems recognize the root directory as `\`.

In addition, Unix/Linux system file names are case-sensitive. Windows systems store the case of letters used in the file name, but don't distinguish between their cases at all.

This means that these two strings:

```
ThisIsTheNameOfTheFile
```

and

```
thisisthenameofthefile
```

describe two different files in Unix/Linux systems, but are the same name for just one file in Windows systems.

The main and most striking difference is that you have to use **two different separators for the directory names**: `\` in Windows, and `/` in Unix/Linux.

This difference is not very important to the normal user, but is **very important when writing programs in Python**.

To understand why, try to recall the very specific role played by the `\` inside Python strings.

File names: continued

Suppose you're interested in a particular file located in the directory `dir`, and named `file`.

Suppose also that you want to assign a string containing the name of the file.

In Unix/Linux systems, it may look as follows:

```
name = "/dir/file"
```

But if you try to code it for the Windows system:

```
name = "\dir\file"
```

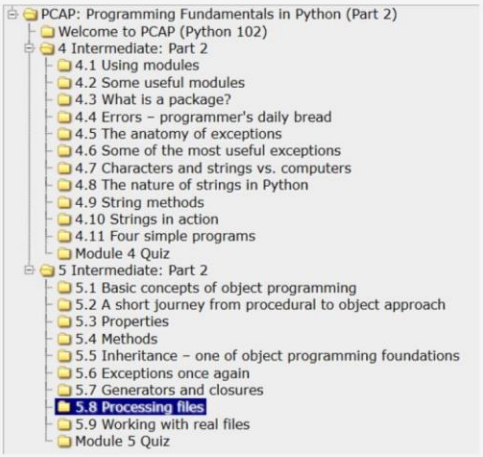
you'll get an unpleasant surprise: either Python will generate an error, or the execution of the program will behave strangely, as if the file name has been distorted in some way.

In fact, it's not strange at all, but quite obvious and natural. Python uses the `\` as an escape character (like `\n`).

This means that Windows file names must be written as follows:

```
name = "\\dir\\file"
```

Fortunately, there is also one more solution. Python is smart enough to be able to convert slashes into backslashes each time it discovers that it's required by the OS.



slashes into backslashes each time it discovers that it's required by the OS.

This means that any the following assignments:

```
name = "/dir/file"
name = "c:/dir/file"
```

will work with Windows, too.

Any program written in Python (and not only in Python, because that convention applies to virtually all programming languages) does not communicate with the files directly, but through some abstract entities that are named differently in different languages or environments - the most-used terms are **handles** or **streams** (we'll use them as synonyms here).

The programmer, having a more- or less-rich set of functions/methods, is able to perform certain operations on the stream, which affect the real files using mechanisms contained in the operating system kernel.

In this way, you can implement the process of accessing any file, even when the name of the file is unknown at the time of writing the program.

The operations performed with the abstract stream reflect the activities related to the physical file.

To connect (bind) the stream with the file, it's necessary to perform an explicit operation.

The operation of connecting the stream with a file is called **opening the file**, while disconnecting this link is named **closing the file**.

Hence, the conclusion is that the very first operation performed on the stream is always `open` and the last one is `close`. The program, in effect, is free to manipulate the stream between these two events and to handle the associated file.

This freedom is limited, of course, by the physical characteristics of the file and the way in which the file has been opened.

Let us say again that the opening of the stream can fail, and it may happen due to several reasons: the most common is the lack of a file with a specified name.

It can also happen that the physical file exists, but the program is not allowed to open it. There's also the risk that the program has opened too many streams, and the specific operating system may not allow the simultaneous opening of more than `n` files (e.g., 200).

A well-written program should detect these failed openings, and react accordingly.

File streams

The opening of the stream is not only associated with the file, but should also declare the manner in which the stream will be processed. This declaration is called an **open mode**.

If the opening is successful, the **program will be allowed to perform only the operations which are consistent with the declared open mode**.

There are two basic operations performed on the stream:

- **read** from the stream: the portions of the data are retrieved from the file and placed in a memory area managed by the program (e.g., a variable);
- **write** to the stream: the portions of the data from the memory (e.g., a variable) are transferred to the file.

There are three basic modes used to open the stream:

- **read mode**: a stream opened in this mode allows **read operations only**; trying to write to the stream will cause an exception (the exception is named `UnsupportedOperation`, which inherits `OSError` and `ValueError`, and comes from the `io` module);
- **write mode**: a stream opened in this mode allows **write operations only**; attempting to read the stream will cause the exception mentioned above;
- **update mode**: a stream opened in this mode allows **both writes and reads**.

Before we discuss how to manipulate the streams, we owe you some explanation. **The stream behaves almost like a tape recorder**.

When you read something from a stream, a virtual head moves over the stream according to the number of bytes transferred from the stream.

When you write something to the stream, the same head moves along the stream recording the data from the memory.

Whenever we talk about reading from and writing to the stream, try to imagine this analogy. The programming books refer to this mechanism as the **current file position**, and we'll also use this term.

It's necessary now to show you the object responsible for representing streams in programs.

File handles

Python assumes that **every file is hidden behind an object of an adequate class**.

Of course, it's hard not to ask how to interpret the word *adequate*.

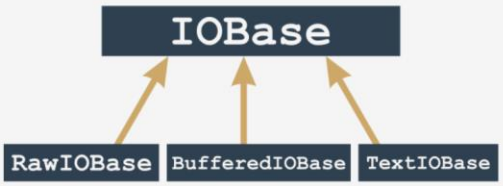
Files can be processed in many different ways - some of them depend on the file's contents, some on the programmer's intentions.

In any case, different files may require different sets of operations, and behave in different ways.

An object of an adequate class is **created when you open the file and annihilate it at the time of closing**.

Between these two events, you can use the object to specify what operations should be performed on a particular stream. The operations you're allowed to use are imposed by **the way in which you've opened the file**.

In general, the object comes from one of the classes shown here:



Note: you never use constructors to bring these objects to life. The only way you **obtain them is to invoke the function named** `open()`.

The function analyses the arguments you've provided, and automatically creates the required object.

If you want to **get rid of the object, you invoke the method named** `close()`.

The invocation will sever the connection to the object, and the file and will remove the object.

For our purposes, we'll concern ourselves only with streams represented by `BufferIOBase` and `TextIOBase` objects. You'll understand why soon.

File handles: continued

Due to the type of the stream's contents, **all the streams are divided into text and binary streams**.

The text streams ones are structured in lines; that is, they contain typographical characters (letters, digits, punctuation, etc.) arranged in rows (lines), as seen with the naked eye when you look at the contents of the file in the editor.

This file is written (or read) mostly character by character, or line by line.

The binary streams don't contain text but a sequence of bytes of any value. This sequence can be, for example, an executable program, an image, an audio or a video clip, a database file, etc.

Because these files don't contain lines, the reads and writes relate to portions of data of any size. Hence the data is read/written byte by byte, or block by block, where the size of the block usually ranges from one to an arbitrarily chosen value.

Then comes a subtle problem. In Unix/Linux systems, the line ends are marked by a single character named `LF` (ASCII code 10) designated in Python programs as `\n`.

Other operating systems, especially these derived from the prehistoric CP/M system (which applies to Windows family systems, too) use a different convention: the end of line is marked by a pair of characters, `CR` and `LF` (ASCII codes 13 and 10) which can be encoded as `\r\n`.

This ambiguity can cause various unpleasant consequences.

If you create a program responsible for processing a text file, and it is written for Windows, you can recognize the ends of the lines by finding the `\r\n` characters, but the same program running in a Unix/Linux environment will be completely useless, and vice versa: the program written for Unix/Linux systems might be useless in Windows.

Such undesirable features of the program, which prevent or hinder the use of the program in different environments, are called **non-portability**.

Similarly, the trait of the program allowing execution in different environments is called **portability**. A program endowed with such a trait is called a **portable program**.

File handles: continued

Since portability issues were (and still are) very serious, a decision was made to definitely resolve the issue in a way that doesn't engage the developer's attention.

It was done at the level of classes, which are responsible for reading and writing characters to and from the stream. It works in the following way:

- when the stream is open and it's advised that the data in the associated file will be processed as text (or there is no such advisory at all), it is **switched into text mode**;
- during reading/writing of lines from/to the associated file, nothing special occurs in the Unix environment, but when the same operations are performed in the Windows environment, a process called a **translation of newline characters** occurs: when you read a line from the file, every pair of `\r\n` characters is replaced with a single `\n` character, and vice versa; during write operations, every `\n` character is replaced with a pair of `\r\n` characters;
- the mechanism is completely **transparent** to the program, which can be written as if it was intended for processing Unix/Linux text files only; the source code run in a Windows environment will work properly, too;

Opening the streams

The **opening of the stream** is performed by a function which can be invoked in the following way:

```
stream = open(file, mode = 'r', encoding = None)
```

Let's analyze it:

- when the stream is open and it's advised to do so, its contents are taken as-is, **without any conversion** - no bytes are added or omitted.

- the name of the function (`open`) speaks for itself; if the opening is successful, the function returns a stream object; otherwise, an exception is raised (e.g., `FileNotFoundError` **if the file you're going to read doesn't exist**);
- the first parameter of the function (`file`) specifies the name of the file to be associated with the stream;
- the second parameter (`mode`) specifies the open mode used for the stream; it's a string filled with a sequence of characters, and each of them has its own special meaning (more details soon);
- the third parameter (`encoding`) specifies the encoding type (e.g., UTF-8 when working with text files)
- the opening must be the very first operation performed on the stream.

Note: the mode and encoding arguments may be omitted - their default values are assumed then. The default opening mode is reading in text mode, while the default encoding depends on the platform used.

Let us now present you with the most important and useful open modes. Ready?

Opening the streams: modes

`r` open mode: read

- the stream will be opened in **read mode**;
- the file associated with the stream **must exist** and has to be readable, otherwise the `open()` function raises an exception.

`w` open mode: write

- the stream will be opened in **write mode**;
- the file associated with the stream **doesn't need to exist**; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased); if the creation isn't possible (e.g., due to system permissions) the `open()` function raises an exception.

`a` open mode: append

- the stream will be opened in **append mode**;
- the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)

`r+` open mode: read and update

- the stream will be opened in **read and update mode**;
- the file associated with the stream **must exist and has to be writeable**, otherwise the `open()` function raises an exception;
- both read and write operations are allowed for the stream.

`w+` open mode: write and update

- the stream will be opened in **write and update** mode;
- the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; the previous content of the file remains untouched;
- both read and write operations are allowed for the stream.

Selecting text and binary modes

If there is a letter `b` at the end of the mode string it means that the stream is to be opened in the **binary mode**.

If the mode string ends with a letter `t` the stream is opened in the **text mode**.

Text mode is the default behaviour assumed when no binary/text mode specifier is used.

Finally, the successful opening of the file will set the current file position (the virtual reading/writing head) before the first byte of the file **if the mode is not** `a` and after the last byte of file **if the mode is set to** `a` .

Text mode	Binary mode	Description
`rt`	`rb`	read
`wt`	`wb`	write
`at`	`ab`	append
`r+t`	`r+b`	read and update
`w+t`	`w+b`	write and update

EXTRA

You can also open a file for its exclusive creation. You can do this using the `x` open mode. If the file already exists, the `open()` function will raise an exception.

Opening the stream for the first time

Imagine that we want to develop a program that reads content of the text file named: `C:\Users\User\Desktop\file.txt`.

How to open that file for reading? Here's the relevant snippet of the code:

```
try:
    stream = open("C:\Users\User\Desktop\file.txt", "rt")
    # processing goes here
    stream.close()
except Exception as exc:
    print("Cannot open the file:", exc)
```

What's going on here?

- we open the try-except block as we want to handle runtime errors softly;
- we use the `open()` function to try to open the specified file (note the way we've specified the file name)
- the open mode is defined as text to read (as **text is the default setting**, we can skip the `t` in mode string)
- in case of success we get an object from the `open()` function and we assign it to the stream variable;
- if `open()` fails, we handle the exception printing full error information (it's definitely good to know what exactly happened)

Pre-opened streams

The names of these streams are: `sys.stdin` , `sys.stdout` , and `sys.stderr` .

Let's analyze them:

- `sys.stdin`
  - `stdin` (as *standard input*)
  - the `stdin` stream is normally associated with the keyboard, pre-open for reading and regarded as the primary data source for the running programs;
  - the well-known `input()` function reads data from `stdin` by default.

- `sys.stdout`
  - `stdout` (as *standard output*)
  - the `stdout` stream is normally associated with the screen, pre-open for writing, regarded as the primary target for outputting data by the running program;
  - the well-known `print()` function outputs the data to the `stdout` stream.

- `sys.stderr`
  - `stderr` (as *standard error output*)
  - the `stderr` stream is normally associated with the screen, pre-open for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;
  - we haven't presented any method to send the data to this stream (we will do it soon, we promise)
  - the separation of `stdout` (useful results produced by the program) from the

Pre-opened streams

We said earlier that any stream operation must be preceded by the `open()` function invocation. There are three well-defined exceptions to the rule.

When our program starts, the three streams are already opened and don't require any extra preparations. What's more, your program can use these streams explicitly if you take care to import the `sys` module:

```
import sys
```

because that's where the declaration of the three streams is placed.

- we haven't presented any method to send the data to this stream (we will do it soon, we promise)
- the separation of `stdout` (useful results produced by the program) from the `stderr` (error messages, undeniably useful but does not provide results) gives the possibility of redirecting these two types of information to the different targets. More extensive discussion of this issue is beyond the scope of our course. The operation system handbook will provide more information on these issues.

Closing streams

The last operation performed on a stream (this doesn't include the `stdin`, `stdout`, and `stderr` streams which don't require it) should be **closing**.

That action is performed by a method invoked from within open stream object: `stream.close()`.

- the name of the function is definitely self-commenting (`close()`)
- the function expects exactly no arguments; the stream doesn't need to be opened
- the function returns nothing but raises `IOError` exception in case of error;
- most developers believe that the `close()` function always succeeds and thus there is no need to check if it's done its task properly.

  This belief is only partly justified. If the stream was opened for writing and then a series of write operations were performed, it may happen that the data sent to the stream has not been transferred to the physical device yet (due to mechanism called **caching** or **buffering**). Since the closing of the stream forces the buffers to flush them, it may be that the flushes fail and therefore the `close()` fails too.

We have already mentioned failures caused by functions operating with streams but not mentioned a word how exactly we can identify the cause of the failure.

The possibility of making a diagnosis exists and is provided by one of streams' exception component which we are going to tell you about just now.

Diagnosing stream problems

The `IOError` object is equipped with a property named `errno` (the name comes from the phrase *error number*) and you can access it as follows:

```
try:
    # some stream operations
except IOError as exc:
    print(exc.errno)
```

The value of the `errno` attribute can be compared with one of the predefined symbolic constants defined in the `errno` module.

Let's take a look at some selected **constants useful for detecting stream errors**:

`errno.EACCES` → `Permission denied`

The error occurs when you try, for example, to open a file with the *read only* attribute for writing.

`errno.EBADF` → `Bad file number`

The error occurs when you try, for example, to operate with an unopened stream.

`errno.EEXIST` → `File exists`

The error occurs when you try, for example, to rename a file with its previous name.

`errno.EFBIG` → `File too large`

The error occurs when you try to create a file that is larger than the maximum allowed by the operating system.

`errno.EISDIR` → `Is a directory`

The error occurs when you try to treat a directory name as the name of an ordinary file.

`errno.EMFILE` → `Too many open files`

The error occurs when you try to simultaneously open more streams than acceptable for your operating system.

`errno.ENOENT` → `No such file or directory`

The error occurs when you try to access a non-existent file/directory.

`errno.ENOSPC` → `No space left on device`

The error occurs when there is no free space on the media.

The complete list is much longer (it includes also some error codes not related to the stream processing.)

Diagnosing stream problems: continued

If you are a very careful programmer, you may feel the need to use the sequence of statements similar to those presented below:

```
import errno
try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # actual processing goes here
    s.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file doesn't exist.")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files.")
    else:
        printf("The error number is:", exc.errno)
```

Fortunately, there is a function that can dramatically **simplify the error handling code**. Its name is `strerror()`, and it comes from the `os` module and **expects just one argument - an error number**.

Its role is simple: you give an error number and get a string describing the meaning of the error.

Note: if you pass a non-existent error code (a number which is not bound to any actual error), the function will raise `ValueError` exception.

Now we can simplify our code in the following way:

```
from os import strerror
try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # actual processing goes here
    s.close()
except Exception as exc:
    print("The file could not be opened:", strerror(exc.errno));
```

Okay. Now it's time to deal with text files and get familiar with some basic techniques you can use to process them.

Processing text files

In this lesson we're going to prepare a simple text file with some short, simple content.

We're going to show you some basic techniques you can utilize to **read the file contents** in order to process them.

The processing will be very simple - you're going to copy the file's contents to the console, and count all the characters the program has read in.

But remember - our understanding of a text file is very strict. In our sense, it's a plain text file - it may contain only text, without any additional decorations (formatting, different fonts, etc.).

That's why you should avoid creating the file using any advanced text processor like MS Word, LibreOffice Writer, or something like this. Use the very basics your OS offers: Notepad, vim, gedit, etc.

If your text files contain some national characters not covered by the standard ASCII charset, you may need an additional step. Your `open()` function invocation may require an argument denoting specific text encoding.

For example, if you're using a Unix/Linux OS configured to use UTF-8 as a system-wide setting, the `open()` function may look as follows:

```
stream = open('file.txt', 'rt', encoding='utf-8')
```

where the encoding argument has to be set to a value which is a string representing proper text encoding (UTF-8, here).

Consult your OS documentation to find an encoding name adequate to your environment.

```
1  stream = open("tzop.txt", "rt", encoding = "utf-8")
2  # opening tzop.txt in read mode, returning it as a file object
3  print(stream.read()) # printing the content of the file
```

Console >_

```
The Zen of Python
==========


::

    Beautiful is better than ugly.
    Explicit is better than implicit.
    Simple is better than complex.
    Complex is better than complicated.
    Flat is better than nested.
    Sparse is better than dense.
    Readability counts.
    Special cases aren't special enough to break the rules.
    Although practicality beats purity.
    Errors should never pass silently.
    Unless explicitly silenced.
    In the face of ambiguity, refuse the temptation to guess.
    There should be one-- and preferably only one --obvious way to do it.
    Although that way may not be obvious at first unless you're Dutch.
    Now is better than never.
    Although never is often better than *right* now.
    If the implementation is hard to explain, it's a bad idea.
    If the implementation is easy to explain, it may be a good idea.
    Namespaces are one honking great idea -- let's do more of those!


---------------
source: https://github.com/python/peps/blob/master/pep-0020.txt
```

where the encoding argument has to be set to a value which is a string representing proper text encoding (UTF-8, here).

Consult your OS documentation to find an encoding name adequate to your environment.

INFORMATION

For the purposes of our experiments with file processing carried out in this section, we're going to use a pre-uploaded set of files (e.g., `tzop.txt`, or `text.txt` files) which you'll be able to work with. If you'd like to work with your own files locally on your machine, we strongly encourage you to do so, and to use IDLE to carry out your own tests.

Processing text files: continued

Reading a text file's contents can be performed using several different methods - none of them is any better or worse than any other. It's up to you which of them you prefer and like.

Some of them will sometimes be handier, and sometimes more troublesome. Be flexible. Don't be afraid to change your preferences.

The most basic of these methods is the one offered by the `read()` function, which you were able to see in action in the previous lesson.

If applied to a text file, the function is able to:

- read a desired number of characters (including just one) from the file, and return them as a string;
- read all the file contents, and return them as a string;
- if there is nothing more to read (the virtual reading head reaches the end of the file), the function returns an empty string.

We'll start with the simplest variant and use a file named `text.txt`. The file has the following contents:

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

```
1  from os import strerror
2
3  try:
4      cnt = 0
5      s = open('text.txt', "rt")
6      ch = s.read(1)
7      while ch != '':
8          print(ch, end='')
9          cnt += 1
10         ch = s.read(1)
11     s.close()
12     print("\n\nCharacters in file:", cnt)
13 except IOError as e:
14     print("I/O error occurred: ", strerr(e.errno))
```

Console >_

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

Characters in file: 131
```

Now look at the code in the editor, and let's analyze it.

The routine is rather simple:

- use the try-except mechanism and open the file of the predetermined name (`text.txt` in our case)
- try to read the very first character from the file (`ch = s.read(1)`)
- if you succeed (this is proven by a positive result of the `while` condition check), output the character (note the `end=` argument - it's important! You don't want to skip to a new line after every character!);
- update the counter (`cnt`), too;
- try to read the next character, and the process repeats.

Console >_

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

Characters in file: 131

Processing text files: continued

If you're absolutely sure that the file's length is safe and you can read the whole file to the memory at once, you can do it - the `read()` function, invoked without any arguments or with an argument that evaluates to `None`, will do the job for you.

Remember - **reading a terabyte-long file using this method may corrupt your OS**.

Don't expect miracles - computer memory isn't stretchable.

Look at the code in the editor. What do you think of it?

Let's analyze it:

- open the file as previously;
- read its contents by one `read()` function invocation;
- next, process the text, iterating through it with a regular `for` loop, and updating the counter value at each turn of the loop;

The result will be exactly the same as previously.

```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    content = s.read()
    for ch in content:
        print(ch, end='')
        cnt += 1
        ch = s.read(1)
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```

Console >_

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

Characters in file: 131
```

Processing text files: `readline()`

If you want to treat the file's contents **as a set of lines**, not a bunch of characters, the `readline()` method will help you with that.

The method tries to **read a complete line of text from the file**, and returns it as a string in the case of success. Otherwise, it returns an empty string.

This opens up new opportunities - now you can also count lines easily, not only characters.

Let's make use of it. Look at the code in the editor.

As you can see, the general idea is exactly the same as in both previous examples.

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    line = s.readline()
    while line != '':
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
        line = s.readline()
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:     ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Console >_

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

Characters in file: 131
Lines in file:      4
```

Processing text files: `readlines()`

Another method, which treats text file as a set of lines, not characters, is `readlines()`.

The `readlines()` method, when invoked without arguments, tries to **read all the file contents, and returns a list of strings, one element per file line**.

If you're not sure if the file size is small enough and don't want to test the OS, you can convince the `readlines()` method to read not more than a specified number of bytes at once (the returning value remains the same - it's a list of a string).

Feel free to experiment with this example code to understand how the `readlines()` method works.

The maximum accepted input buffer size is passed to the method as its argument.

You may expect that `readlines()` can process a file's contents more effectively than `readline()`, as it may need to be invoked fewer times.

Note: when there is nothing to read from the file, the method returns an empty list. Use it to detect the end of the file.

To the extent of the buffer's size, you can expect that increasing it may improve input performance, but there is no golden rule for it - try to find the optimal values yourself.

Look at the code in the editor. We've modified it to show you how to use `readlines()`.

We've decided to use a 15-byte-long buffer. Don't think it's a recommendation.

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readlines(20)
    while len(lines) != 0:
        for line in lines:
            lcnt += 1
            for ch in line:
                print(ch, end='')
                ccnt += 1
        lines = s.readlines(10)
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:     ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Console >_

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

Characters in file: 131
Lines in file:      4
```

We've decided to use a 15-byte-long buffer. Don't think it's a recommendation.

We've used such a value to avoid the situation in which the first `readlines()` invocation consumes the whole file.

We want the method to be forced to work harder, and to demonstrate its capabilities.

There are **two nested loops in the code**: the outer one uses `readlines()`'s result to iterate through it, while the inner one prints the lines character by character.

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

Characters in file: 131
Lines in file: 4

Processing text files: continued

The last example we want to present shows a very interesting trait of the object returned by the `open()` function in text mode.

We think it may surprise you - **the object is an instance of the iterable class**.

Strange? Not at all. Usable? Yes, absolutely.

The **iteration protocol defined for the file object** is very simple - its `__next__` method just **returns the next line read in from the file**.

Moreover, you can expect that the object automatically invokes `close()` when any of the file reads reaches the end of the file.

Look at the editor and see how simple and clear the code has now become.

```
from os import strerror

try:
    ccnt = lcnt = 0
    for line in open('text.txt', 'rt'):
        lcnt += 1
        for ch in line:
            print(ch, end='')
            ccnt += 1
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:     ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```

Console >_

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.

Characters in file: 131
Lines in file:      4
```

Dealing with text files: `write()`

Writing text files seems to be simpler, as in fact there is one method that can be used to perform such a task.

p>The method is named `write()` and it expects just one argument - a string that will be transferred to an open file (don't forget - the open mode should reflect the way in which the data is transferred - **writing a file opened in read mode won't succeed**).

No newline character is added to the `write()`'s argument, so you have to add it yourself if you want the file to be filled with a number of lines.

The example in the editor shows a very simple code that creates a file named `newtext.txt` (note: the open mode `w` ensures that **the file will be created from scratch**, even if it exists and contains data) and then puts ten lines into it.

The string to be recorded consists of the word line, followed by the line number. We've decided to write the string's contents character by character (this is done by the inner `for` loop) but you're not obliged to do it in this way.

We just wanted to show you that `write()` is able to operate on single characters.

The code creates a file filled with the following text:

```
from os import strerror

try:
    fo = open('newtext.txt', 'wt') # a new file (newtext.txt) is created
    for i in range(10):
        s = "line #" + str(i+1) + "\n"
        for ch in s:
            fo.write(ch)
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```

Console >_

```
line #1
line #2
line #3
line #4
line #5
line #6
line #7
line #8
line #9
line #10
```

We encourage you to test the behavior of the `write()` method locally on your machine.

Dealing with text files: continued

Look at the example in the editor. We've modified the previous code to write whole lines to the text file.

The contents of the newly created file are the same.

Note: you can use the same method to write to the `stderr` stream, but don't try to open it, as it's always open implicitly.

For example, if you want to send a message string to `stderr` to distinguish it from normal program output, it may look like this:

```
import sys
sys.stderr.write("Error message")
```

```
from os import strerror

try:
    fo = open('newtext.txt', 'wt')
    for i in range(10):
        fo.write("line #" + str(i+1) + "\n")
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```

Console >_

What is a bytearray?

Before we start talking about binary files, we have to tell you about one of the **specialized classes Python uses to store amorphous data**.

Amorphous data is data which have no specific shape or form - they are just a series of bytes.

This doesn't mean that these bytes cannot have their own meaning, or cannot represent any useful object, e.g., bitmap graphics.

The most important aspect of this is that in the place where we have contact with the data, we are not able to, or simply don't want to, know anything about it.

Amorphous data cannot be stored using any of the previously presented means - they are neither strings nor lists.

There should be a special container able to handle such data.

Python has more than one such container - one of them is **a specialized class name bytearray** - as the name suggests, it's **an array containing (amorphous) bytes**.

If you want to have such a container, e.g., in order to read in a bitmap image and process it in any way, you need to create it explicitly, using one of available constructors.

Take a look:

```
data = bytearray(10)
```

Such an invocation creates a bytearray object able to store ten bytes.

Note: such a constructor **fills the whole array with zeros**.

Bytearrays: continued

Bytearrays resemble lists in many respects. For example, they are **mutable**, they're a subject of the `len()` function, and you can access any of their elements using conventional indexing.

There is one important limitation - **you mustn't set any byte array elements with a value which is not an integer** (violating this rule will cause a `TypeError` exception) and you're **not allowed to assign a value that doesn't come from the range 0 to 255 inclusive** (unless you want to provoke a `ValueError` exception).

You can **treat any byte array elements as integer values** - just like in the example in the editor.

Note: we've used two methods to iterate the byte arrays, and made use of the `hex()` function to see the elements printed as hexadecimal values.

Now we're going to show you **how to write a byte array to a binary file** - binary, as we don't want to save its readable representation - we want to write a one-to-one copy of the physical memory content, byte by byte.

```
1  data = bytearray(10)
2
3  for i in range(len(data)):
4      data[i] = 10 - i
5
6  for b in data:
7      print(hex(b))
```

Console >_

```
0xa
0x9
0x8
0x7
0x6
0x5
0x4
0x3
0x2
0x1
```

Bytearrays: continued

So, how do we write a byte array to a binary file?

Look at the code in the editor. Let's analyze it:

- first, we initialize `bytearray` with subsequent values starting from `10`; if you want the file's contents to be clearly readable, replace `10` with something like `ord('a')` - this will produce bytes containing values corresponding to the alphabetical part of the ASCII code (don't think it will make the file a text file - it's still binary, as it was created with a `wb` flag);
- then, we create the file using the `open()` function - the only difference compared to the previous variants is the open mode containing the `b` flag;
- the `write()` method takes its argument (`bytearray`) and sends it (as a whole) to the file;
- the stream is then closed in a routine way.

The `write()` method returns a number of successfully written bytes.

If the values differ from the length of the method's arguments, it may announce some write errors.

In this case, we haven't made use of the result - this may not be appropriate in every case.

Try to run the code and analyze the contents of the newly created output file.

You're going to use it in the next step.

```
1  from os import strerror
2
3  data = bytearray(10)
4
5  for i in range(len(data)):
6      data[i] = 10 + i
7
8  try:
9      bf = open('file.bin', 'wb')
10     bf.write(data)
11     bf.close()
12 except IOError as e:
13     print("I/O error occurred:", strerr(e.errno))
14
15
16
17 # enter code that reads bytes from the stream here
```

Success (1.56s) ×

Console >_

How to read bytes from a stream

Reading from a binary file requires use of a specialized method name `readinto()`, as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file.

Note:

- the method returns the number of successfully read bytes;
- the method tries to fill the whole space available inside its argument; if there are more data in the file than space in the argument, the read operation will stop before the end of the file; otherwise, the method's result may indicate that the byte array has only been filled fragmentarily (the result will show you that, too, and the part of the array not being used by the newly read contents remains untouched)

Look at the complete code below:

```
from os import strerror

data = bytearray(10)

try:
    bf = open('file.bin', 'rb')
    bf.readinto(data)
    bf.close()

    for b in data:
        print(hex(b), end=' ')
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))


# enter code that reads bytes from the stream here
```

Success (1.56s) ×

Console >_

Let's analyze it:

- first, we open the file (the one you created using the previous code) with the mode described as `rb`;
- then, we read its contents into the byte array named `data`, of size ten bytes;
- finally, we print the byte array contents - are they the same as you expected?

Run the code and check if it's working.

How to read bytes from a stream

An alternative way of reading the contents of a binary file is offered by the method named `read()`.

Invoked without arguments, it tries to **read all the contents of the file into the memory**, making them a part of a newly created object of the bytes class.

This class has some similarities to `bytearray`, with the exception of one significant difference - it's **immutable**.

Fortunately, there are no obstacles to creating a byte array by taking its initial value directly from the bytes object, just like here:

```
from os import strerror

try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read())
    bf.close()

    for b in data:
        print(hex(b), end=' ')

except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Be careful - **don't use this kind of read if you're not sure that the file's contents will fit the available memory**.

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))


# enter code that reads bytes from the stream here
```

Console >_

How to read bytes from a stream: continued

If the `read()` method is invoked with an argument, it **specifies the maximum number of bytes to be read**.

The method tries to read the desired number of bytes from the file, and the length of the returned object can be used to determine the number of bytes actually read.

You can use the method just like here:

```
try:
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read(5))
    bf.close()

    for b in data:
        print(hex(b), end=' ')

except IOError as e:
    print("I/O error occurred:", strerr(e.errno))
```

Note: the first five bytes of the file have been read by the code - the next five are still waiting to be processed.

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerr(e.errno))


# enter code that reads bytes from the stream here
```

Console >_

Copying files - a simple and functional tool

Now you're going to amalgamate all this new knowledge, add some fresh elements to it, and use it to write a real code which is able to actually copy a file's contents.

Of course, the purpose is not to make a better replacement for commands like *copy* (MS Windows) or *cp* (Unix/Linux) but to see one possible way of creating a working tool, even if nobody wants to use it.

Look at the code in the editor. Let's analyze it:

- lines 3 through 8: ask the user for the name of the file to copy, and try to open it to read; terminate the program execution if the open fails; note: use the `exit()` function to stop program execution and to pass the completion code to the OS; any completion code other than `0` says that the program has encountered some problems; use the `errno` value to specify the nature of the issue;
- lines 9 through 15: repeat nearly the same action, but this time for the output file;
- line 17: prepare a piece of memory for transferring data from the source file to the target one; such a transfer area is often called a buffer, hence the name of the variable; the size of the buffer is arbitrary - in this case, we decided to use 64 kilobytes; technically, a larger buffer is faster at copying items, as a larger buffer means fewer I/O operations; actually, there is always a limit, the crossing of which renders no further improvements; test it yourself if you want.
- line 18: count the bytes copied - this is the counter and its initial value;
- line 20: try to fill the buffer for the very first time;
- line 21: as long as you get a non-zero number of bytes, repeat the same actions;
- line 22: write the buffer's contents to the output file (note: we've used a slice to limit the number of bytes being written, as `write()` always prefer to write the whole buffer)
- line 23: update the counter;
- line 24: read the next file chunk;
- lines 29 through 31: some final cleaning - the job is done.

```
4  try:
5      src = open(srcname, 'rb')
6  except IOError as e:
7      print("Cannot open source file: ", strerror(e.errno))
8      exit(e.errno)
9  dstname = input("Destination file name?: ")
10 try:
11     dst = open(dstname, 'wb')
12 except Exception as e:
13     print("Cannot create destination file: ", strerr(e.errno))
14     src.close()
15     exit(e.errno)
16
17 buffer = bytearray(65536)
18 total  = 0
19 try:
20     readin = src.readinto(buffer)
21     while readin > 0:
22         written = dst.write(buffer[:readin])
23         total += written
24         readin = src.readinto(buffer)
25 except IOError as e:
26     print("Cannot create destination file: ", strerr(e.errno))
27     exit(e.errno)
28
29 print(total,'byte(s) succesfully written')
30 src.close()
31 dst.close()
```

Console >_

```
Source file name?: dst
Cannot open source file:  No such file or directory
```

LAB

Estimated time

30 minutes

Level of difficulty

Medium

Objectives

- improving the student's skills in operating with files (reading)
- using data collections for counting numerous data.

Scenario

A text file contains some text (nothing unusual) but we need to know how often (or how rare) each letter appears in the text. Such an analysis may be useful in cryptography, so we want to be able to do that in reference to the Latin alphabet.

Your task is to write a program which:

- asks the user for the input file's name;
- reads the file (if possible) and counts all the Latin letters (lower- and upper-case letters are treated as equal)
- prints a simple histogram in alphabetical order (only non-zero counts should be presented)

Create a test file for the code, and check if your histogram contains valid results.

```
1
```

Console >_

Assuming that the test file contains just one line filled with:

```
aBc
```

the expected output should look as follows:

```
a -> 1
b -> 1
c -> 1
```

Console >_

Tip:

We think that a dictionary is a perfect data collection medium for storing the counts. The letters may be keys while the counters can be values.

LAB

Estimated time

15-20 minutes

Level of difficulty

Medium

Prerequisites

05_9.15.1

Objectives

- improve the student's skills in operating with files (reading/writing)
- using lambdas to change the sort order.

Scenario

The previous code needs to be improved. It's okay, but it has to be better.

Your task is to make some amendments, which generate the following results:

- the output histogram will be sorted based on the characters' frequency (the bigger counter should be presented first)
- the histogram should be sent to a file with the same name as the input one, but with the suffix '.hist' (it should be concatenated to the original name)

```
1
```

Console >_

Assuming that the input file contains just one line filled with:

```
cBabAa
```

the expected output should look as follows:

```
a -> 3
b -> 2
c -> 1
```

Console >_

Tip:

Use a `lambda` to change the sort order.

LAB

Estimated time

30 minutes

Level of difficulty

Medium

Objectives

- improve the student's skills in operating with files (reading)
- perfecting the student's abilities in defining and using self-defined exceptions and dictionaries.

Scenario

Prof. Jekyll conducts classes with students and regularly makes notes in a text file. Each line of the file contains 3 elements: the student's first name, the student's last name, and the number of point the student received during certain classes.

The elements are separated with white spaces. Each student may appear more than once inside Prof. Jekyll's file.

The file may look as follows:

```
John    Smith   5
Anna    Boleyn  4.5
John    Smith   2
Anna    Boleyn  11
Andrew  Cox     1.5
```

```
class StudentsDataException(Exception):
    pass

class BadLine(StudentsDataException):
    # put your code here

class FileEmpty(StudentsDataException):
    # put your code here
```

Console >_

Your task is to write a program which:

- asks the user for Prof. Jekyll's file name;
- reads the file contents and counts the sum of the received points for each student;
- prints a simple (but sorted) report, just like this one:

```
Andrew Cox      1.5
Anna Boleyn     15.5
John Smith      7.0
```

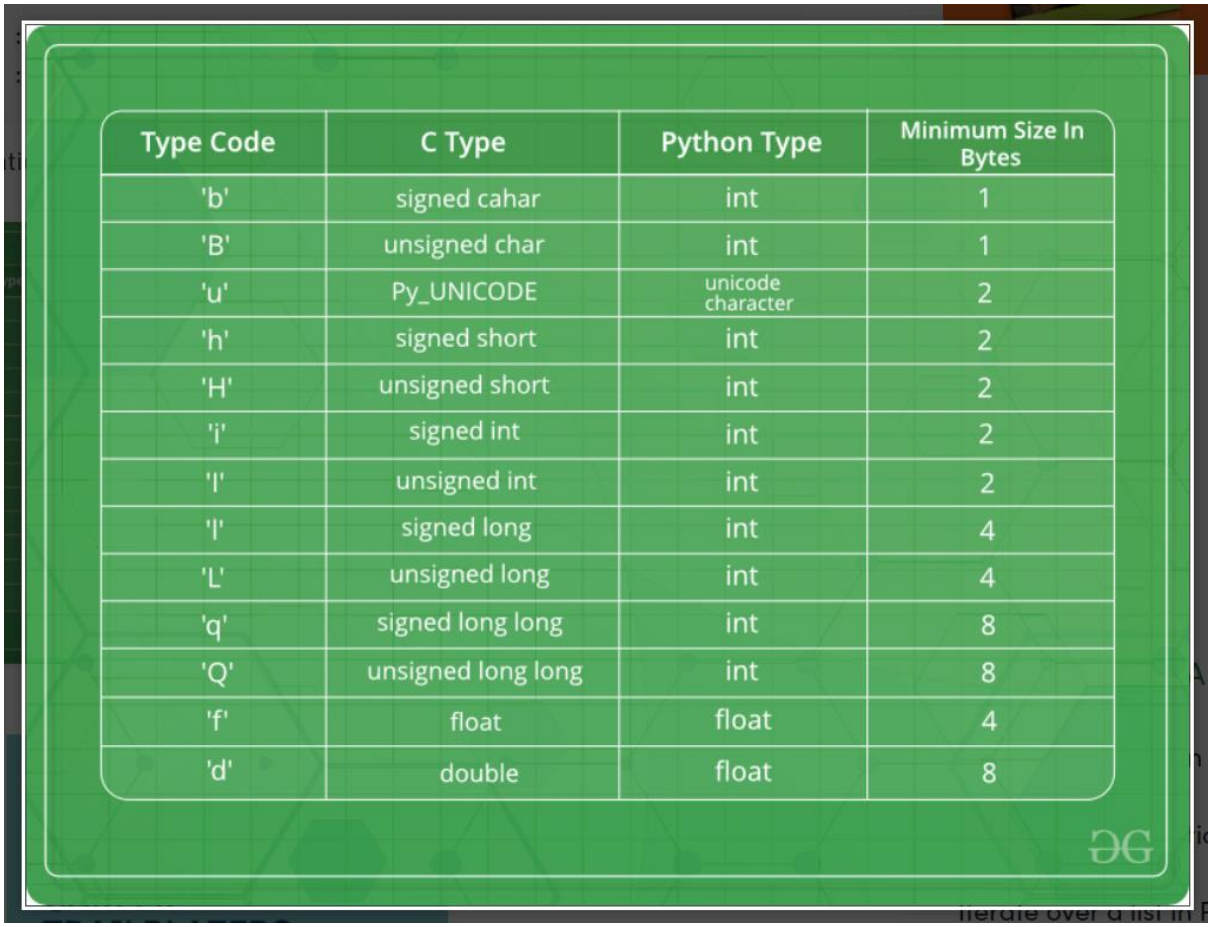
Note:

- your program must be fully protected against all possible failures: the file's non-existence, the file's emptiness, or any input data failures; encountering any data error should cause immediate program termination, and the erroneous should be presented to the user;
- implement and use your own exceptions hierarchy - we've presented it in the editor; the second exception should be raised when a bad line is detect, and the third when the source file exists but is empty.

Tip:

Use a dictionary to store the students' data.

Console >_



Type Code	C Type	Python Type	Minimum Size In Bytes
'b'	signed cahar	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8