Python beginners guide pdf

File name : free-python-tutorial-for-beginners-pdf-download.pdf Languange Used: English File size : 46.6 Mb Total Download : 216 Download Now Read online description : Download free python guide for beginners Pdf Download or read Free Python Tutorial Pdf Download online books in PDF, EPUB and Mobi Format. Click on the Download or Read online button to get a free tutorial for beginners pdf download pdf now. Note:! If the content has not been found, you must refresh this page manually. Alternatively, try our book search engine. AN UNLIMITED NUMBER OF BOOKS, ALL IN ONE PLACE. FREE TO TRY FOR 30 DAYS. SUBSCRIBE TO READING OR DOWNLOADING AN E-BOOK FOR FREE. Start your free month now! To download the free Python Tutorial for beginners PDF download e-books, you need to create a free account. New to programming? Python is free and easy to learn if you know where to start! This guide will help you get started quickly. Chinese translation, new to Python? Read beginnerTheddar/Review for a brief explanation of what Python is. To receive Python Next, install the python 3 interpreter on your computer. This is the program that read Python programs and executes their instructions; you must do so before you can do Python programming. Mac and Linux distributions may include outdated versions of Python (Python 2), but you must install updated (Python 3). Check out the tutorial/Download for instructions on downloading the correct version of Python. There are also Python translator and IDE packages, such as Thonny. Other options can be found in Integrated DevelopmentImages. At some stage, you'll want to edit and save the code. Contact HowToEditPythonCode for some tips and recommendations. Study python Next, read a tutorial and try some simple experiments with your new python translator. Most tutorials suggest that you know how to run a program on your computer. If you're using Windows and need help with this, see How to start a Windows program. Some sites offer browser encoding for those who want to learn Python: Print a cheat sheet of Python's most important features and post it on your office wall until you know the basics well. After reading the tutorial, you can review python's online documentation. It includes a tutorial that can come in handy, a library reference that lists all the modules that come standard with Python, and the linguistic reference for a complete (if rather dry) explanation of python's syntax. When you're ready to write your first program, you'll need a text editor or IDE. If you do not want to use Thonny or anything more advanced, idle, which is bundled with Python and supports extensions. This python wiki also contains a page for Python One-Liners - an obscure but interesting subculture in Python. Need help? Do you need any help with that? Read beginners' hives/help beginners/help lists and newsgroups. Most python books will include an introduction to the language; see Introductory books for suggested titles. Consult beginnerGuide/Examples for small programs and small snippets of code that can help you learn. Or, if you prefer to study Python by listening to a lecture, you can attend a training course or even hire a coach to come to your company. Consult the PythonEvents page to see if training courses are planned in your area and pythonTraining page for a list of trainers. Teachers can join EDU-SIG, a mailing list to discuss the use of Python in teaching at any level ranging from K-12 to university. Full list of tutorial pages guide Quiz and exercises Python Style Checker Pythonchecker.com - Educational online tool to evaluate your python style (with dynamic computational and instilations) Looking for a specific python module or app? The first place to look is the Python Package Index. If you can't find anything relevant in the package index, try looking for python.org - you can find everything mentioned on python's website, in frequently asked questions, or in the newsgroup. More information: where to look. You can also try our outdoor guest project, pydoc.net, for advanced package and module search. Then try Google or another search engine of your choice. Searching for python and some relevant keywords will usually find something useful. Finally, you can try publishing a query for the comp.lang.python Usenet group. Do you want to contribute? Python is a product of Python Software Foundation, a not-for-profit organization that owns the copyright. PSF donations are tax deductible in the United States and you can donate by credit card or PayPal. To report a Python kernel error, use Python Bug Tracker. To help troubleshoot or another patch in python kernel, read python's developer guide for more information about python's development process. To contribute to Python's official documentation, join the SIG documentation, write to docs@python.org, or use Issue Tracker to contribute a documentary patch. To announce your module or application in the Python Community, use comp.lang.python.announce. For more information, see the Python Mailing List Guide. To suggest changes to Python's core, post your thoughts to comp.lang.python. If you have an impleme, follow python patch instructions. If you have a question not sure where to report it, see WhereDoIReportThis? Page. CategoryDo Academia.edu cumentation no longer supports Explorer.To and Academia.edu and the wider internet faster and please take a few seconds to upgrade your browser. Academia.edu uses cookies to personalize content, personalize ads, and improve the user experience. By using our site, you agree to our collection of information through the use of cookies. To learn more, see Python has become one of the fastest growing programming languages in the last few years. Not only is it widely used, but it's also great coping language if you want to enter the world of programming. This guide for beginners python allows you to learn the kernel of the language in a few hours instead of weeks. Brief information: You can download a PDF version of this guide for beginners. Are you ready to dive? Content Python was created in 1990 by Guido van Rossum in the Netherlands. One of the purposes of the language was to be available to non-programmers. Python is designed to be a second language for programmers to learn due to its low learning curve and ease of use. Python works on Mac, Linux, Windows and many other platforms. Python's: Interpretation: it can be implemented at runtime, and changes in the program are instantly noticeable. To be very technical, Python has a compiler. The difference compared to Java or C++ is how transparent and automatic it is. With Python, we should not worry about the drafting step, as is done in real time. The trade-off is that interpreted languages are usually slower than compiled ones. Seminal dynamic: you don't have to specify variable types, and there's nothing that makes you do it. Object-oriented: everything in Python is an object. But you can choose to write code in an object-oriented, procedural, or even functional way. High level: you do not have to deal with low-level machine details. Python has grown very recently in part because of its many applications in the following areas: Scripting System: it's a great tool for automating everyday repetitive tasks. Data analytics: it's a great language to experiment with and has tons of libraries and tools to process data, create models, visualize results, and even implement solutions. This is used in areas such as finance, e-commerce and research. Web development: frameworks such as Django and Flask allow the development of web applications, APIs and websites. Machine learning: Tensorflow and Pytorch are some of the libraries that allow scientists and industry to develop and deliver artificial intelligence solutions in image recognition, health, self-driving and many other areas. You can easily organize your code into modules and reuse them or share them with others. Finally, we must keep in mind that Python has broken the changes between versions 2 and 3. And since python 2 support is completed in 2020, this article is based solely on Python 3. Let's get started. Install Python 3 If you are using Mac or Linux, you already have Python installed. However, Windows does not come with Python installed by default. You may also have Python 2, and we will use Python 3. So you need to if you have Python 3 first. Type the following in the terminal. python3 -V Note the above letter V. If your result is anything like the 3.x.y', for example, Python 3.8.1, then you are ready to go. If not, follow the following instructions according to your operating system. Install Python 3 in Windows go to . Download the latest version. After downloading, double-click setup. On the first screen, select the Add python 3.x on the path check box, and then click Install Now. Wait for the installation process to complete until the next screen with the Setup message is successful. Click Close. Install Python 3 on Mac XCode from the App Store. Install the command-line tools by running the following in the terminal. xcode-select --install I recommend using Homebrew. Go to and follow the instructions on the first page to install it. After you install Homebrew, run the following preparation commands to install Python 3. Update for brew brew install python3 Homebrew already adds Python 3 to the road, so you don't have to do anything else. Install Python 3 on Linux To install via apt, available in Ubuntu and Debian, type the following: sudo apt install python3 To install using yum available in RedHat and CentOS, type the following: sudo yum install python3 Running code You can run Python code directly in the terminal as commands or you can save the code to a file with the .python extension and run the Python file. Terminal Running commands directly in the terminal is recommended when you want to run something simple. Open the command prompt and type python3 renan@mypc:~$ python3 You need to see something similar in the terminal that shows the version (in my case, Python 3.6.9), the operating system (I use Linux) and some basic commands to help you. >>> tells us we're in python's console. Python 3.6.9 (default, November 7 2019, 10:44:02) [GCC 8.3.0] on Linux help type, copyright, credits or license for more information. >>> Let's test it by running our first program to run basic math and add two numbers. >>> 2 + 2 The output is: 4 To exit the Python console just type output(). >>> exit() Running .py files If you have a complex program with many lines of code, the Python console is not the best option. The alternative is simply to open a text editor, enter the code, and save the file with the .py extension. Let's do this, create a file called second_program.py with the following content. Print ('Second Program') The print() function prints a message on the screen. The message enters the brackets with single or double quotation marks, both of which work the same. To start the program, on your terminal do the following: renan@mypc:~$ python3 second_program.py The output is: The second program Python is famous for its pure syntax. Language avoids using unnecessary characters to indicate any specificity. The Python semicolon does not use semicolons to complete the lines. A new line is sufficient to that a new command is starting. The printing method() will show something. In this example, we have two commands that will display the messages inside the single quotation marks. print('First command') print ('Second command') Output: First command Second command But the following is wrong due to semicolon at the end: print (First command); print (Second command); Indent Many languages use curly brackets to determine the range. Python's interpreter uses only indents to determine when a range ends and another starts. This means you need to be aware of the white spaces at the top of each row - they matter and can break your code if it's not available. This definition of function works: def my_function(): print ('First command') This does not work because the indentation of the second row is missing and will cause an error: def my_function(: print ('First command') Random sensitivity and python variables is case sensitive. So the names of the variables and the name are not the same thing and different values are stored. name = 'Renan' Name = 'Moura' As you can see, variables are easily created by simply assigning values to them using the symbol = . to comment on something in your code, use the hashtag #. The commented part does not affect the flow of the program. # this feature prints something def my_function(): print (First command) It is a simple review. The details of each of them will become clearer in the following chapters with examples and wider explanations. The purpose of the comments is to explain what is happening in the code. Comments are written together with the code , but do not affect the flow of the program. When you work alone, maybe the comments don't feel like something you need to write. After all, at the moment, you know why each line of code. But what if new people get on board your project in a year and the project has 3 modules, each with 10,000 lines of code? Think of people who know nothing about your app and who suddenly need to support, fix, or add new features. Remember that there is no single solution for a problem. Your way of solving things is all yours and yours. If you ask 10 people to solve the same problem, they will come up with 10 different solutions. If you want others to fully understand your reasoning, good code design is a must, but comments are an integral part of any codebase. The syntax of comments in Python is quite easy: just use the #кеш symbol in front of the text you want to be a comment. #This is a comment and will not affect my stream program You can use a comment to explain what makes some piece of code. #calculates the sum of all two numbers a + b Maybe you want to comment on something very complicated or describe how some process works in your code. In you can use multi-line comments. Yes Yes Yes this, just use one hash brand # for each line. #Everything after hash mark # is a comment #This is a comment and will not affect the program flow #Calculates of project costs given variables a and b #a is the time in the months that the project is completed #b is how much money will cost per month + b * 10 Variables In any program, you need to store and manipulate data to create a stream or some specific logic. That's what variables are for. You can have a variable to store a name, another to store a person's age, or even use a more complex type to store all this at once as a dictionary. Create also known as Variable Declaration is a basic and clear operation in Python Just select a name and attribute value for it using the = symbol. name='Bob' age = 32 You can use the print() function to display the value of a variable. Print(name) stamp (age) Bob 32 Note that in Python there is no special word to announce a variable. As soon as you assign a value, the variable is created in memory. Python also has dynamic input, which means you don't have to say it if your variable is text or a number, for example. The interpreter outputs text entry based on the specified value. If you need it, you can also reissue a variable only by changing its value. #declaring name as string name = 'Bob' #re-declarant name as int name = 32 Note that this is not recommended as variables should have meaning and context. If I have a variable name, I don't expect there to be a number in it. Naming Conventions Let's move on from the last section when I talked about meaning and context. Don't use random variable names like x or y. Say you want to save your party time, just party_time. Did you notice about tach? By convention, if you want to use a variable name that consists of two or more words, you separate them with underscores. That's called Sacks. Another option would be the use of CamelCase as in partyTime. This is very common in other languages, but not the convention in Python, as indicated above. Variables are case sensitive, so the party_time and Party_time are not the same. Also, keep in mind that the convention tells us to always use smaller cases. Remember, use names you can remember in your program easily. Poor naming can cost you a lot of time and cause annoying bugs. In summary, variable names: Are sensitive to letters: time and time are not the same Must start with underscore _ or letter (DOES NOT start with a number) Are allowed to have only numbers, letters and underscores. There are no special characters like: #, $, &, @, etc. This, for example, is not allowed: #час, 10partytime. Types In order to data in Python, you must use a variable. And each variable has its own type depending on the value of the data stored. A python has a dynamic seal that You don't have to explicitly declare your variable type, but if you want, you can. Lists, Tuples, Sets and Dictionaries are all data types and have dedicated sections later with more details, but we'll take a brief look at them here. In this way, I can show you the most important aspects and operations of each of them in my own section, while keeping this section shorter and focused on providing a broad view of the main types of data in Python. Determining the type First at all, let's learn how to determine the type of data. Just use the type () function and pass the variable of your choice as an argument, such as the example below. print(type(my_variable)) Buleva Buleva type is one of the most basic types of programming. The boolean variable type can only represent True or False. my_bool = True print (type (my_bool)) my_bool = bool(1024) print(type(my_bool)) <class 'bool'=> <class 'bool'=&gtNumbers have three types of digital types: int, float and complex. Integer my_int = 32 printing (type(my_int)) my_int = int(32) print(type(my_int)) <class 'int'=> <class 'int'=&gtFloating my_float = 32.85 printing (type (my_float)) my_float = float (32.85) printing (type(my_float)) <class 'float'=> <class 'float'=&gtComplex my_complex_number = 32+4j printing (type (my_complex_number)) my_complex_number = complex(32+4j) printing(type(my_complex_number)) <class 'complex'=> <class 'complex'=&gtThe text type is one of the most common types out there and is often called a string or in Python, just 100 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 00 my_city = New York print (type (my_city)) #Single quotation marks are exactly #the same use as double quotation marks my_city = New York print (type (my_city)) #Setting variable type explicitly my_city = Str (New York) print(type(my_city)) <class 'str'=> <class 'str'=> <class 'str'=&gtYou can use + the operator to connect strings. Concatation is when you have two or more strings and want to join them in one. word1 = New word2 = 'York' print(word1 + word2) New York The type of iasis has many built-in methods that allow us to manipulate them. I will demonstrate how some of these methods work. The Len function returns the length of a string. print(len('New York)) 8 The replacement method replaces part of the line with another. As an example, let's replace New for Old. print (New York'replace(New,New', 'Old')) Old York Upper method will return all characters as uppercase letters. print ('New York'.upper()) NEW YORK The lower method does the opposite and returns all characters as lowercase letters. Print ('New York'.lower()) New York Lists A list has its own items, and you can add the same item as many times as you want. An important detail is that the lists are muesli. To change the list after you create the list, you can change the list by adding items, removing them, or even simply changing their values. These ще бъдат показани по-нататък в раздела, посветен на списъците. my_list = [bmw,</class> </class> </class> </class> </class> </class> </class> </class> </class> </class> </class> maclaren] print(type(my_list)) my_list = list ((bmw, ferrari, maclaren)) print(type(my_list)) <class 'list'=> <class 'list'=&gtTuples A motorcade is just like a list: ordered, and allows repeat items. There is only one difference: the motorcade is unchanged. Immutability means that you cannot change a motorcade after its creation. If you try to add an item or update one, for example, python's intepreter will show you an error. I will show that these errors appear later in the section dedicated to Tuples. my_tuple = (bmw, ferrari, maclaren) print (type(my_tuple)) my_tuple = motorcade ((bmw, ferrari, maclaren)) printed kits (type(my_tuple)) <class 'tuple'=> <class 'tuple'=&gtsets do not guarantee the order of items and are not indexed. Key point when using sets: They do not allow item recurrence. my_set = {BMW, ferrari, maclaren} printing(type(my_set)) my_set = set ((bmw, ferrari, maclaren)) printing(type(my_set)) <class 'set'=> <class 'set'=&gtDictionaries Dictionary does not guarantee the order of elements and is softer. One important feature in dictionaries is that you can set your own access keys for each item. my_dict = {country: France, worldcups: 2} printing (type (my_dict)) my_dict = dict(country=France, worldcups =2) printing(type(my_dict)) <class 'dict'=> <class 'dict'=&gtTipcasting tiptingka allows you to convert between different types. Thus, you can make an int, or become scary, for example, in the int. Explicit conversion To throw a variable into a string, just use the str() function. # It's just a simple intimat my_str = str(32') print(my_str) # int to str my_str = str(32) print(my_str) # float for ul my_str = str(32.0) print(my_str) 32 32 32.0 To shift a variable to an integer, just use the int() function # It's just a simple intilization my_int = int(32) print(my_int) # float to the wind: circle down to 3 my_int = int(3.2) print #my_int) # to int my_int = int('32') print(my_int) 32 3 32 To infhole variable to floating, just use the float() # function it is explicit intilization my_float = float(3.2) print (my_float) # int to float my_float = float(32) print(my_float) # str to float my_float = float(32') print(my_float) 3.2 32.0 32.0 This , which I did above, is explicitly

called conversion type. In some cases, you don't need to do the conversion explicitly, as Python can do it yourself. Indirect conversions The example below shows an implicit conversion when adding an incoming and float card. Notice that the my_sum is floating. Python uses float to avoid data loss because the int type cannot represent decimal places. my_int = 32 my_float = 3.2 my_sum = my_int + my_float printing (my_sum) printing(type(my_sum)) 35.2 <class 'float'=>On the other hand, in this когато добавите int и str, Python няма да може да направи безличен преобразуване, и изричното преобразуване, и изричното преобразуване</class> </class> </class> </class> </class> </class> </class> </class> </class> </class> Necessary. my_int = 32 my_str = '32' # apparent conversion works my_sum = my_int + int(my_str) print (my_sum) #implicit #implicit causes error my_sum = my_int + my_str 64 Traceback (last call: file<stdin>, line 1, in <module>TypeError: unsupported operand type for +: int and str The same error is discarded when you try to add floating and p. my_float = 3.2 my_str = '32' # obvious conversion works my_sum = my_float + float(my_str) print(my_sum) #implicit #implicit you convert to error my_sum = my_float + my_str 35.2 Traceback (last file call):<stdin>, line 1, in <module>TypeError: unsupported operand type for +: float and str User login If you need to interact with a user when you run your program at a command prompt (for example, to request information), you can use the input() () function. country = login (What is your country?) #user enter Brazil stamp (country) Brazil The captured value is always a string. Just remember that you may need to convert it using typecasting. age = input (how old are you?) #user introduces 29 printing (age) stamp (type(age)) age = int(age) printing (type(age)) The output for each stamp() is: 29 <class 'str'=> <class 'int'=> gt;Note that age 29 is captured as a string and then explicitly converted to int. Program language operators, operators are special characters that you can apply to variables and values to perform operations such as arithmetic/mathematical and comparisons. Python has many operators that you can apply to your variables and I will demonstrate to you the most used. Arithmetic operators are the most common operators and also the most recognizable. They allow you to perform simple mathematical operations. They are: +: Collection -: subtraction *: Multiplication /: Division **: Exponential //: Floor division, rounded down as a result of division %: Module, gives you the remainder of division Let's see a program that shows how each of them is used. print (Appendix:, 5+2) printing ('subtraction:', 5-2) print (Multiplication:, 5*2) print ('Division:', 5/ 2) printing (Flooring:, 5// 2) printing ('Exponential:', 5**2) printing (Modulus:, 5% 2) Appendix: 7 Subtraction: 3 Multiplication: 10 Division: 2, 5 Floor: 2 Degree of exposure: 25 Module: 1 The connection is when you have two or more strings and want to join them to one. This is useful when you have information in multiple variables and want to combine them. For example, in the following example, I combine two variables that contain my name and last name, respectively, to have my full name. The + operator is also used to connect. first_name = 'Renan' last_name = 'Moura' print(first_name + last_name) Renan Mura because the binding is applied низове, за свързване на низове</class> </class> </module> </stdin> </module> </stdin> </stdin> other types, you must make an explicit tipcast using str(). You must enter an int value of 30 to string with str() to connect it to the rest of the string. Age = I have + str(30) + years printing (age) I have 30 years comparison operators use comparison operators to compare two values. These operators return true or false. They are: ==: Equal!=: Not equal >: Greater than <: Less than >=: Greater than or equal to <=: Less than or equal to See a program that shows how each program is used. print ('Raven:', 5 == 2) printing (Not equal:, 5!= 2) printing ('Greater than:', 5 > 2) print (Less than:, 5 < 2) print ('Greater than or equal to: ', 5 > = 2) print ('Less than or equal to:', 5 <= 2) Equal: False Not equal: True Greater than or equal to: True less than or equal to: Misappropriation operators As the name suggests, these operators are used to assign values to variables. x = 7 in the first example is direct assignment, storing the number 7 in the variable x. The assignment operation takes the value to the right and assigns it to the variable on the left. Other operators are simple wall for arithmetic operators. In the second example x starts with 7 and x += 2 is just another way to write x = x + 2. This means that the previous x value is added from 2 and transferred to x, which is now equal to 9. x = 7 print(x) 7 +=: addition and distribution x = 7 x == 2 print(x) 9 -= subtraction and distribution x = 7 x -= 2 printing (x) 5 *= multiplication and distribution x = 7 x *= 2 print (x) 14 /=: split and distribution x = 7 x /= 2 print(x) 3,0 5 %=: module and assignment x = 7 x %= 2 printing(x) 1 //=: floor division and assignment x = 7 x //= 2 printing(x) 3 **=: exposure and assignment x = 7 x x = 2 printing(x) 49 Logical operators Logical operators are used to combine reports applying boolean algebra. They are: and: True only when both statements are true or: False only when both x and y are not untrue: Not the operator just reverses the entrance, True was true and vice versa. Let's see a program that shows how each one is used. x = 5 Y = 2 printing (x == 5 and y > 3) printing (x == 5 or Y > 3) printing (not x == 5)) False operators of fake membership These operators provide an easy way to check if a particular object is present in a series: string, list, motorcade, set and dictionary. They are: c: returns true if the object is not present in: returns true if the object is not available Let's see a program that shows how each one is used. number_list = [1, 2, 4, 5, 6] print(1 in number_list) printing (5 no number_list) printing (3 no number_list) True False True Conditionals Conditionals are one of the cornerstones of any programming language. They allow you to control the program flow according to the specific conditions you can check. If The way you apply conditionally is by The general form of if it is: if expression: expression expression contains some logic that returns boolean and statement is executed only if the return is true. A simple example: bob_age = 32 sarah_age = 29 if bob_age > sarah_age: print ('Bob is older than Sarah') Bob is older than Sarah We have two variables showing the age of Bob and Sarah. The condition in plain English reads: If Bob's age is greater than Sarah's age, then print the phrase Bob is older than Sarah. Because the condition returns True, the phrase will be printed on the console. If other and elif claims In our latest example, the program does something only if the condition returns True. But we also want to do something if you return False or even check the second or third condition if the first is not met. In this example, we replaced bob and sarah's age. The first condition will return False, as Sarah is older now, and then the program will print the phrase after another instead. bob_age = 29 sarah_age = 32 if bob_age > sarah_age: print ('Bob is older than Sarah') other: seal ('Bob is younger than Sarah') Bob is younger than Sarah Now, consider the example below with Elifa. bob_age = 32 sarah_age = 32 if bob_age > sarah_age: print ('Bob is older than Sarah') elif bob_age == sarah_age: print ('Bob and Sarah have the same age') other: seal ('Bob is younger than Sarah') Bob and Sarah have the same age Elf is to provide a new condition to check before it is implemented. We changed their age again and now they're both 32 years old. As such, the condition in elif is fulfilled. Since both have the same age, the program will print Bob and Sarah have the same age. Note that you can have as many elifi as you like, just put them in sequence. bob_age = 32 sarah_age = 32 if bob_age > sarah_age: print ('Bob is older than Sarah') elif bob_age < sarah_age: print ('Bob is younger than Sarah') elif bob_age == sarah_age: print ('Bob and Sarah have the same age') other: seal ('This is never fulfilled' and Sarah have the same age in this example, otherwise it never runs because all possibilities are covered in the previous conditions and can thus be eliminated. Nested conditionalities You may need to check more than one conditional for something to happen. In this scenario, you can insert if you are inserted. For example, the second phrase Bob is the oldest is printed only if both passes. bob_age = 32 sarah_age = 28 mary_age = 25 if bob_age > sarah_age: print ('Bob is older than Sarah'), if bob_age > mary_age: print ('Bob is the oldest') Bob is older than Sarah Bob is the oldest or, depending on logic, simplify it with Boolean Alge. Thus, your code is smaller, more readable and easier to maintain. bob_age = 32 sarah_age = 28 mary_age = 25 if bob_age > sarah_age and bob_age > mary_age: print ('Bob is the oldest) Bob is Тригерни оператори Троичният оператор е едноличнa, ако е отчетна. Много е удобно за прости условия. Ето как изглежда: <expression>ако друго е възможно да се разгледа следния <condition> <expression>python код: a = 25 b = 50 x = 0 = 0 = 1 резултат = x x ако > b else y print(result) 1 Тук използваме четири променливи, a и b са за условието, докато x и y представляват изразите. a и b са стойностите, които проверяваме един срещу друг, за да оценим дадено условие. В този случай проверяваме дали е по-голямо от b. Ако изразът е вярно, т.е. a е по-голяма от b, тогава стойността на x ще бъде приписана на резултата, който ще бъде равен на 0. Ако обаче е по-малко от b, тогава ние имаме стойността на y, зададена за резултат, и резултатът ще задържи стойността 1. Тъй като е по-малко < 50, result will have 1 as final value from y. The easy way to remember how the condition is evaluated is to read it in plain English. Our example would read: result will be x if a is greater than b otherwise y. Lists As promised in the Types section, this section and the next three about Tuples, Sets, and Dictionaries will have more in depth explanations of each of them since they are very important and broadly used structures in Python to organize and deal with data. A list has its items ordered and you can add the same item as many times as you want. An important detail is that lists are mutable. Mutability means you can change a list after its creation by adding items, removing them, or even just changing their values. Initialization Empty List people = [] List with initial values people = ['Bob', 'Mary'] Adding in a List To add an item in the end of a list, use append(). people = ['Bob', 'Mary'] people.append('Sarah') print(people) ['Bob', 'Mary', 'Sarah'] To specify the position for the new item, use the insert() method. people = ['Bob', 'Mary'] people.insert(0, 'Sarah') print(people) ['Sarah', 'Bob', 'Mary'] Updating in a List Specify the position of the item to update and set the new value people = ['Bob', 'Mary'] people[1] = 'Sarah' print(people) ['Bob', 'Sarah'] Deleting in a List Use the remove() method to delete the item given as an argument. people = ['Bob', 'Mary'] people.remove('Bob') print(people) ['Mary'] To delete everybody, use the clear() method: people = ['Bob', 'Mary'] people.clear() Retrieving in a List Use the index to reference the item. Remember that the index starts at 0. So to access the second item use the index 1. people = ['Bob', 'Mary'] print(people[1]) Mary Check if a given item already exists in a List people = ['Bob', 'Mary'] if 'Bob' in people: print('Bob exists!') else: print('There is no Bob!') Tuples A tuple is similar to a list: it's ordered, and allows repetition of items. There is just one difference: a tuple is immutable. Immutability, if you remember, means you can't change a tuple after its creation. If you try add an item 50,= result= will= have= 1= as= final= value= from= y.= the= easy= way= to= remember= how= the= condition= is= evaluated= is= to= read= it= in= plain= english.= our= example= would= read:= result= will= be= x= if= a= is= greater= than= b= otherwise= y.= lists= as= promised= in= the= types= section,= this= section= and= the= next= three= about= tuples,= sets,= and= dictionaries= will= have= more= in= depth= explanations= of= each= of= them= since= they= are= very= important= and= broadly= used= structures= in= python= to= organize= and= deal= with= data.= a= list= has= its= items= ordered= and= you= can= add= the= same= item= as= many= times= as= you= want.= an= important= detail= is= that= lists= are= mutable.= mutability= means= you= can= change= a= list= after= its= creation= by= adding= items ,= removing= them,= or= even= just= changing= their= values.= initialization= empty= list= people=[] list= with= initial= values= people=['Bob', 'mary']= adding= in= a= list= to= add= an= item= in= the= end= of= a= list,= use= append().= people=['Bob', 'mary']= people.append('sarah')= print(people)= ['bob',= 'mary',= 'sarah']= to= specify= the= position= for= the= new= item,= use= the= insert()= method.= people=['Bob', 'mary']= people.insert(0,= 'sarah')= print(people)= ['sarah',= 'bob',= 'mary']= updating= in= a= list= specify= the= position= of= the= item= to= update= and= set= the= new= value= people=['Bob', 'mary']= people[1]='Sarah' print(people)= ['bob',= 'sarah']= deleting= in= a= list= use= the= remove()= method= to= delete= the= item= the= item= given= as= an= argument.= people=['Bob', 'mary']= people.remove('bob')= print(people)= ['mary']= to= delete= everybody ,= use= the= clear()= method:= people=['Bob', 'mary']= people.clear()= retrieving= in= a= list= use= the= index= to= reference= the= item.= remember= that= the= index= starts= at= 0.= so= to= access= the= second= item= use= the= index= 1.= people=['Bob', 'mary']= print(people[1])= mary= check= if= a= given= item= already= exists= in= a= list= people=['Bob', 'mary']= if= 'bob'= in= people:= print('bob= exists!') = else:= print('there= is= no= bob!') = tuples= a= tuple= is= similar= to= a= list:= it's= ordered,= and= allows= repetition= of= items.= there= is= just= one= difference:= a= tuple= is= immutable.= immutability,= if= you= remember,= means= you= can't= change= a= tuple= after= its= creation.= if= you= try= to= add= an= item=></ 50, result will have 1 as final value from y. The easy way to remember how the condition is evaluated is to read it in plain English. Our example would read: result will be x if a is greater than b otherwise y. Lists As promised in the Types section, this section and the next three about Tuples, Sets, and Dictionaries will have more in depth explanations of each of them since they are very important and broadly used structures in Python to organize and with data. A list has its items ordered and you can add the same item as many times as you want. An important detail is that lists are mutable. Mutability means you can change a list after its creation by adding items, removing them, or even just changing their values. Initialization Empty List people = [] List with initial values people = ['Bob', 'Mary'] Adding in a List To add an item in the end of a list, use append(). people = ['Bob', 'Mary'] people.append('Sarah') print(people) ['Bob', 'Mary', 'Sarah'] To specify the position for the new item, use the insert() method. people = ['Bob', 'Mary'] people.insert(0, 'Sarah') print(people) ['Sarah', 'Bob', 'Mary'] Updating in a List Specify the position of the item to update and set the new value people = ['Bob', 'Mary'] people[1] = 'Sarah' print(people) ['Bob', 'Sarah'] Deleting in a List Use the remove() method to delete the item given as an argument. people = ['Bob', 'Mary'] people.remove('Bob') print(people) ['Mary'] To delete everybody, use the clear() method: people = ['Bob', 'Mary'] people.clear() Retrieving in a List Use the index to reference the item. Remember that the index starts at 0. So to access the second item use the index 1. people = ['Bob', 'Mary'] print(people[1]) Mary Check if a given item already exists in a List people = ['Bob', 'Mary'] if 'Bob' in people: print('Bob exists!') else: print('There is no Bob!') Tuples A tuple is similar to a list: it's ordered, and allows repetition of items. There is just one difference: a tuple is immutable. Immutability, if you remember, means you can't change a tuple after its creation. If you try to add an item > от b, 25</expression> </condition> </expression> </expression> update one, for example, the Python interpreter will show you an error. Initialization Empty Cortege people = () Tuple with initial values people = ('Bob', 'Maria') Adding to Tuple Tuples are immemorable. This means that if you try to add an item, you will see an error. People = (Bob, Maria) people[2] = 'Sarah' Traceback (last call last):<stdin>, line 1, in <module>TypeError: Tuple object does not support assigning the Update job in Tuple Update Item will also return an error. But there is a trick: you can convert to a list, change the item, and then convert it back to tuple. People = ('Bob', 'Mary') people_list = List(People) people_list[1] = Sarah People = Motorcade(people_list) Print (People) (Bob, Sarah) Delete in Cortege For the same reason you can't add an item, you also can't delete an item as they are immutable. Check out in Tuple using the index to reference the item. People = (Bob, Mary) print(people[1]) Mary Check if a position already exists in people with a motorcade = (Bob, Mary in humans: printing ('Bob'))elsewhere: printing('No Bob!') Sets do not guarantee the order of items and are not indexed. Key point when using sets: They do not allow item recurrence. Initialization Empty Set of People = set() Set with starting values people = {'Bob', Mary'} Add in a setting method Use add() to add one item. People.add('Sarah') Use the update method() to add multiple items at once. people.update (['Carol', 'Susan']] Remember, Sets do not allow repetition, so if you add Mary again, nothing changes. people = {'Bob', 'Mary'} people.add (Mary' ' 'Mary') print(people) {'Bob', 'Maria'} Updating into set items set are not mutable. You must add or delete an item. Delete in set To remove bob from dictionary: people = {'Bob', 'Mary'}people.remove('Bob') print (people) {'Mary'} To delete all: people.clear() Check if an item already exists in certain people = {'Bob', 'Mary'} if 'Bob' in humans: print ('Bob exists!') other: print('No Bob!') Dictionaries Dictionary does not guarantee the order of items and is mutt. One important feature of dictionaries is that you can set your custom access keys for each item. Initialization of people with blank dictionary = {} dictionary with initial values of people = {'Bob':30, 'Mary':25} Adding to dictionary If the key does not yet exist, it is added to the dictionary. People ['Sarah']=32 Update Dictionary If the key already exists, the value is updated. #Bob age is 28 now people ['Bob']=28 Note that the code is almost the same. Delete in dictionary To remove bob from dictionary: people.pop('Bob') To delete all: people.clear() Extract to Dictionary bob_age = People['Bob'] print(bob_age) 30 Check a key already exists in the dictionary if Bob in</module> </stdin> </stdin> print ('Bob exists!' other): Print ('No Bob!'), while Loops Loops Loops are used when you need to repeat a block of code several times or apply the same logic to each item in a collection. There are two types of loops: for and during. You will learn about loops in the next section. Basic syntax The main syntax for some time is below. while condition: Sentence Cycle will continue until the condition is true. The square of a number is the example below takes each value of a number and calculates its square value. number = 1 1, while the number < = 5: print (number, square is, number**2) number = number + 1 1 square is 1 2 square is 4 3 square is 9 4 square is 16 5 square is 25 You can use any variable name, but I chose a number because it makes sense in context. A common generic choice would be just me. The cycle will continue until the number (initialized with 1) is less than or equal to 5. Note that after the print() command, the variable increases by 1 to take the next value. If you do not make the increase, you will have an endless loop, since the number will never reach a value greater than 5. This is a very important detail! otherwise block When the condition returns False, otherwise the block will be called. number = 1, while number < = 5: print (number, square is, number**2) number = number + 1 else: print (No numbers left!) 1 square is 1 2 square is 4 3 square is 9 4 square is 16 5 square is 25 No numbers left! Note that the phrase No numbers left!, is printed after the end of the cycle, that is, after the condition number < = 5 is estimated to False. How to break the line for some time in Python Just use the keyword for interruption, and the cycle will stop its execution. number = 1, while the number < = 5: print(number, 'squared is', number**2) number = number + 1 if number == 4: break 1 square is 1 2 square is 4 3 square is 4 Line running normally, and when a number reaches 4 if the statement is estimated correct and the break command is called. This completes the loop before the square value of the numbers 4 and 5 are calculated. How to skip an item in a time cycle Continuation will do this for you. I had to reverse the order of if the statement and print() to show how it works properly. number = 0, while the number < 5: number = number + 1 if = = 4: continue printing (number, square, number**2) 1 square is 1 2 square is 4 3 square is 9 5 square is 25 The program always verifies that 4 is the current value of the number. If so, the square of 4 will not be calculated and continues to skip to the next iteration when the value of the number is 5. * For loops for loops are similar to loops in the sense that they are used to repeat blocks of code. The most important difference is that you can easily iter on successive types. Basic syntax The main syntax of the cycle is both below. for a collection item: Outline above list to up on a list or other collection, just continue as shown in the example below. cars = ['BMW', 'Ferrari', 'McLaren'] for cars: print(car) BMW Ferrari McLaren The car list contains three elements. For contour will be repeated on the list and store each item in the car variable, and then execute a statement, in this case printing (car) to print each car in the console. range () function The range function is widely used in for noseds because it gives you an easy way to list numbers. This code will be promeded through the numbers from 0 to 5 and print each of them. for number in a range (5): print (number) 0 1 2 3 4 Unlike, without the function range() we would do something similar. numbers = [0, 1, 2, 3, 4] for number in numbers: print (number) 0 1 2 3 4 You can also define start and stop using a range(). We start in five hours and stop in 10. The number you set to stop is not included. for number in a range (5, 10): print (number) 5 6 7 8 9 Finally it is possible to determine a step. Here we start at 10 and increase by 2 to 20, since 20 is the stop, not included. for number in a range (10, 20, 2): Print (number) 10 12 14 16 18 another block When the items in the list are above, other block will be called. cars = ('BMW', 'Ferrari', 'McLaren'] for cars: print (car) other: print (No cars left!) BMW Ferrari McLaren no cars left! How to get out of for a cycle in Python Just use the keyword to interrupt, and the cycle will stop its execution. cars = ['BMW', Ferrari, McLaren] for cars: print(car) if the car == Ferrari: Break the BMW Ferrari Cycle will be recolored in the list and print each car. In this case, once the cycle reaches Ferrari, the pause is called and McLaren will not print. How to skip an item in for a loop In this section, we will learn how this can continue for you. I had to reverse the order of if a statement and continue to show how it works properly. Note that I always check that Ferrari is the current item. If so, Ferrari will not print and continue to move on to the next McLaren item. cars = ['BMW', Ferrari, McLaren] for cars: if the car is == 'Ferrari': continue printing(car) BMW McLaren Loop over one cycle: Nested Loops Sometimes you have more complex collections, such as a list of lists. To iterate over these lists, it must be invested for loops. In this case, I have three lists: one of bmw models, another of Ferrari models, and finally one with McLaren models. The first loop is broadcast in the list of each brand, and the second will appear on the models of each brand. car_models = ['BMW I8', BMW X3, BMW X1, ['Ferrari 812', Ferrari F8, Ferrari GTC4], ['McLaren 570S, McLaren 570GT, McLaren 720S], for brand name in car_models: for brand model: print(model) BMW I8 BMW X1 Ferrari 812 Ferrari F8 Ferrari GTC McLaren McLaren 570S McLaren McLaren 720S Contour over other data structures the same logic that applies to cycles above a list can be extended to other data structures: motorcade, set and dictionary. In short, I will demonstrate how to make a basic cycle over each of them. Tie the people of Tuttle = (Bob, Mary for people: seal (face) Bob Mary Loup over certain people = {'Bob', Mary} for person in the people: print (person) Bob Mary Loop through dictionary To print keys: people = {'Bob'30, Mary':25} for person in people: print(face) Bob Mary To print values: people = {'Bob':30, 'Maria':25} for person in people : print(people[person]) 30 25 Functions , as the code grows complexity also grows. And features help organize the code. Features are a convenient way to create blocks of code that you can reuse. Definition and Calling In Python use the defecation keyword to define a function. Give it a name and use brackets to inform 0 or more arguments. In the line after the beginning of the declaration, be sure to indent the code block. Here is an example of a feature called print_first_function() that only prints the phrase My First Function!. To call the function, simply use its name as defined. def print_first_function(): print (My first function!) print_first_function() My first function! Use the value return keyword to return a value from the feature. In this example, the second_function() returns the my second function!. Note that printing() is a built-in function and our feature is called on its inside. The string returned by second_function() is passed as an argument of the print() function. def second_function(): Return My second function! print(second_function()) My second function! returns multiple values Functions can also return multiple values at once. return_numbers() returns two values at the same time. def return_numbers(): return 10, 2 printed (return_numbers()) (10, 2) Arguments You can define parameters between brackets. When calling a function with parameters must pass arguments according to the specified parameters. Previous examples did not have parameters, so there was no need for arguments. Brackets remain empty when functions are called. One argument for setting a parameter, just define it inside the parentheses. In this example, my_number function expects a number as an argument defined by parameter num. Then the argument value is available inside the function to be used. def my_number(num): return My number is: + str(num) print(my_number(10)) My number is: 10 Two or more arguments for determining more parameters, just use a comma to separate them. Here we have a function that adds two numbers called add. It expects two arguments first_num and second_num. Arguments are added by the operator + and the result is returned from the profit. second_num): return first_num + second_num (add(10,2)) 12 This example is very to the last one. The only difference is that we have 3 parameters instead of 2. def add(first_num, second_num, third_num): returning first_num + second_num + third_num print(add(10,2,3)) 15 This logic of defining parameters and passing arguments is the same for any number of parameters. It is important to note that arguments must be conveyed in the same order as the parameters are defined. Default value. You can set a default value for a parameter if no argument is given by using the operator = and custom value. In this feature, if no argument is given, the number 30 is assumed as the expected default value. def my_number(my_number= 30): my return Number is: + str(my_number) print(my_number(10)) print(my_number()) My number is: 10 My number is: 30 Keyword or Named Arguments When calling a function, the order of arguments must match the order of parameters. The alternative is if you use a keyword or named arguments. Set the arguments of their respective parameters directly with the parameter name and = operators. This example reverses the arguments, but the function works as expected because I tell it which value goes to which parameter by name. def my_numbers(first_number, second_number): return 'the numbers are: ' + str(first_number) + and + str(second_number) print(my_numbers(second_number=30, first_number= 10)) The numbers are: 10 and 30 Any number of arguments: *args If you do not want to set the number of parameters, just use * before the parameter. Then the function will take as many arguments as necessary. The parameter name may be something like *numbers, but there is a convention in Python that uses *args for this definition of a variable number of arguments. def my_numbers(*args): for args, such as: print(number) my_numbers(10,2,3) 10 2 3 Any number of keywords / named arguments: **kwargs Similar to *args, we can use **kwargs to pass as many key arguments as we want, as long as we use **. Again, the name may be something like ** numbers, but **kwargs is a convention. def my_numbers(**kwargs): for key, value in kwargs.items(): print (key) print (key) print (value) my_numbers(first_number=30, second_number= 10) first_number 30 second_number 10 Other types of arguments Recent examples mostly use numbers, but you can go by any type as an argument, and they will be treated as such in the function. This example accepts strings as arguments. def my_sport(sport): print (I like + sport) my_sport ('football') my_sport (swimming) I like swimming This feature takes a list as an argument. def my_numbers(numbers): for number in numbers: print (number) my_numbers([30, 10, 64, 92, 105]) 30 10 64 92 105 Scope The location where a variable is created determines its availability to be accessed and manipulated by the rest Code. This is known as range. There are two types of scope: local and global. The Global The total scope allows you to use the variable anywhere in your program. If your variable is out of function, it has a default global scope. name = 'Bob' def print_name(): Print ('My name is ' + name) print_name() My name is Bob Note that the function can use the variable name and print my name is Bob. Local scope when you declare a variable in a function, it exists only inside this feature and is not available from the outside. def print_name(): name =Bob print (My name is + name) print_name() My name is Bob The variable name is declared in the function, so the output is the same as before. But it will throw an error: def print_name(): name = 'Bob' print (My name is + name) print (name) The output of the code above is: Traceback (most recently call last): File <stdin>, line 1, in <module>NameError: Name name is not defined I tried to print the variable name outside the function, but the variable range is local and cannot be found in global scope. Merge scopes If you use the same name for variables in and out of function, the feature will use the one within its scope. So when you call the print_name(), the name = 'Bob' is used to print the phrase. On the other hand, when calling a stamp() outside the scope of the feature, name=Sarah is used because of its global reach. name = Sarah def print_name(): name = 'Bob' print (My name is + name) print_name() print(name) The output of the code above is: My name is Bob Sarah List Understandings Sometimes we want to perform some very simple operations over the list items. A list of understanding gives us a short way to work on lists as an alternative to other iteration methods, such as snouts. Basic syntax To use list understanding to replace regular for cycle, we can do: [expression for item in the list] Which is the same as this: for item in list: expression If we want some conditional to apply the expression, we have: [expression for item in a list if conditional] Which is the same as this: for item in a list: if conditional: expression Example 1 : calculation of the cube of number Regular way numbers = [1, 2, 3, 4, 5] new_list = [] for n in digits: new_list.append (n **3) print (new_list) [1, 8, 27, 64, 125] Using the comprehension list numbers = [1, 2, 3, 4, 5] new_list = [] new_list = [n**3 for n in digits] print(new_list) [1, 8 , 27, 64, 125] Example 2 : calculate the cube of a number only if it is greater than 3 Using conditional, we can filter only the values to which we want the expression to be applied. Regular number = [1, 2, 3, 4, 5] new_list = [] for n in numbers: if(n > 3): new_list.append(n**3) print(new_list) [64, 125] Using list numbers = [1, 2, 3, 4, 5] new_list = [] new_list = [n**3 for n in numbers if n > 3] print(new_list) [64, 125] Example 3: calling functions with understanding list We can also функции, като използвате</module> </stdin> </stdin> list of syntax of interpretation: numbers = [1, 3, 4, 5] new_list = [] def cube(number): return number**3 new_list = [cube(n) for n in numbers if n > 3] print(new_list) [64, 125] Lambda Functions A Pythonda function can have only one expression and can not have many rows. This should have made it easier to create some small logic in a row instead of an entire function as it usually does. Lambda's features are also anonymous, which means you don't need to name them. Basic syntax The main syntax is very simple: just use the lambda keyword, define the necessary parameters and use :to separate the parameters from the expression. Common forms are: lambda arguments: expression One parameter Example Look at this example using only one parameter cubic = lambda number: number**3 print (cubic (2)) 8 Many parameter example If you want, you can also have multiple parameters. exponential = lambda multiplier, number, exponential: multiplier * number **exponential printing (exponential (2, 2, 3)) 16 Calling lambda function directly Do not need to use variable as we did before. Instead, you can use brackets around the lambda function and another pair of brackets around the arguments. The announcement of the function and the execution will occur in the same order. (lambda multiplier, number, grade: multiplier * number**exponential) (2, 2, 3) 16 Examples using lambda features with other built-in functions Map The Map function applies the expression to each item in a list. Let's calculate a cubic of each number in the list. numbers = [2, 5, 10] cubicles = list(map(lambda number: number**3, numbers)) print(cubes) [8, 125, 1000] Filter filter function will filter the list based on the expression. Let's filter so that only the numbers greater than 5. numbers = [2, 5, 10] filtered_list = list(filter(lambda number: number > 5, numbers)) print (filtered_list) [10] Module After a while your code begins to become more complex with many functions and variables. To make it easier to organize the code, we use Modules. The well-designed module also has the advantage of being reusable, so write code once and use it again everywhere. You can write a module with all mathematical operations and other people can use it. And if you need it, you can use someone else's modules to simplify your code by speeding up your project. In other programming languages, they are also called libraries. Using the Module To use a module, we use the keyword to import. As the name suggests, we need to tell our program to import. We can then use any feature available in this module. Let's see an example using the mathematical module. First, let's see how to access Euler's constant. import math.e 2.718281828459045 In this second example, we will function that calculates the square root of a number. It is also possible to as a keyword to create an alias. import mathematics such as m.sqrt (121) m.sqrt (729) 11 27 And finally, using the keyword, we can determine exactly what to import instead of the entire module and use the function directly without the module name. This example uses the floor() function, which returns the largest integer, less than or equal to a given number. from mathematics import floor floor (9.8923) 9 Create a module Now that we know how to use modules, let's see how to create one. This will be a module with basic mathematical operations adding, subtracting, multiplying, dividing and will be called basic_operations. Create the basic_operations.py file with the four functions. def add(first_num, second_num): return first_num + second_num def subtraction (first_num, second_num): return first_num - second_num def multiply(first_num, second_num): return first_num*second_num def division(first_num, second_num): return first_num/second_num Then, just import the basic_operations module and use the features. import basic_operations basic_operations.add(10,2) basic_operations.subtracts(10,2) basic_operations.multiply(10,2) basic_operations.divide(10,2) 12 8 20 5.0 if __name__ == __main__ You are in the process of building a module with the basic mathematical operations, subtract, multiply, and split the basic_operations saved in the basic_operations.py. To ensure that everything is ok, you can do some tests. def add(first_num, second_num): first_num return first_num + second_num subtraction (first_num, second_num): second_num first_num second_num propagating(first_num, second_num): return first_num * second_num deflation(first_num, second_num): return first_num/ second_num 2)) print(subtract (10,2)) print (multiply(10,2)) print(divide(10,2)) After executing the code: renan@pro-home: ~ $ python3 basic_operations.py The output is: 12 8 20 5.0 The output for these tests is what we expected. Our code is ready and ready to share. Let's import the new module to run it into the Python console. Python 3.6.9 (default, November 7 2019, 10:44:02) [GCC 8.3.0] on Linux help type, copyright, credits or license for more information. >>> import basic_operations 12 8 20 5.0 >>> When the module is imported our tests are displayed on the screen, although we did nothing but import basic_operations. To determine that we use if __name__ == '__main__' in the file basic_operations.py as this: def add(first_num, second_num): return first_num + second_num def subtraction (first_num, second_num): return first_num - second_num def(first_num, second_num): return first_num * second_num def divide(first_num, second_num): first_num second_num to return __name__ if __name__ == '__main__': print(10, 2)) print (subtraction (10,2)) stamp (multiplied(10,2)) Running the code directly on the terminal will continue to display the tests. But when you import it as a module, the tests and you can use the features the way you are Python 3.6.9 (default, November 7 2019, 10:44:02) [GCC 8.3.0] on Linux type help, copyright, credits or license for more information. >>> Import basic_operations >>> basic_operations.multiply(10,2) 20 >>> Now that you know how to use if __name__ == __main__, let's find out how it works. The status says when the interpreter treats the code as a module or as a program that runs directly. Python has this basic variable called __name__. This variable represents the name of the module that is the name of the .py file. Create a my_program.py file with the following and run it. print(__name__) The output will be: __main__ This tells us that when a program runs directly, the __name__ is defined as __main__. But when imported as a module, the value of __name__ is the module name. Python 3.6.9 (default, November 7 2019, 10:44:02) [GCC 8.3.0] on Linux help type, copyright, credits or license for more information. >>> Import my_program my_program >>> This is how the Python Interpreter distinguishes the behavior of the imported module and program run directly at the terminal. Files Create, delete, read, and many other features attached to files are an integral part of many programs. As such, it is very important to know how to organize and deal with files directly from your code. Let's see how the python files work. File create first things first, create! We will use the open() function. This feature opens a file and returns the appropriate object. The first argument is the name of the file we are processing, and the second argument refers to the operation we use. The code below creates the file people.txt, the argument x is used when we just want to create the file. If a file with the same name already exists, it will throw an exception. people_file = Open (people.txt, x) You can also use w mode to create a file. Unlike x mode, it will not throw an exception, as this mode displays the writing mode. We open a file to save data to it, and if the file does not exist, it is created. people_file = open (people.txt, w) The last is a mode that means adding. As the name suggests, you can add more data to the file, while w mode simply overwrites all existing data. When added, if the file does not exist, it also creates it. people_file = Open (people.txt, a) File record To write data to a file, just open a file with w mode. In this scenario, the object is called people_file. Then call the save() submit data function as an argument. people_file = open (people.txt, w) people_file.write (Bob) (Maria) people_file.write (Sarah) people_file.close() We use at the end of the otherwise the contents in the file will remain in the same order as BobMarySarah. Another detail is to close the file() . This is not only good practice, but also ensures that the changes are applied to the file. Remember that when using w mode, data that already exists in the file will be replaced by the new data. To add new data without losing what was already there, we need to use add mode. Add a Mode file adds new data to the file while preserving the existing one. In this example, after the first writing with w mode, we use add mode. The result is that each name will appear twice in the people.txt file. #first write people_file = open (people.txt, w) people_file.write (Bob) people_file.write (Maria) people_file.write (Sarah) people_file.close() #appending more data #keeping existing data people_file = open (people txt.), a) people_file.write (Bob) people_file.write (Mary) people_file.write (Sarah) people_file.close() File read Reading the file is also very clear: just use r mode like this. If you read the people.txt file created in the last example, you should see 6 names in the output. people_file = Open (people.txt, R) print(people_file.read()) Bob Maria Sarah Sarah read () function reading the entire file at once. If you use the Reading Line() function, you can read the file order by line. people_file = Open (people.txt, R) print(people_file.readline()) print(people_file.readline()) print(people_file.readline()) Bob Maria Sarah You can also read lines as the example below. people_file = open (people.txt, R) for a person in people_file: print (face) Bob Maria Sarah Sarah Delete file To delete a file, you also need the operating system module. Use the removal method() . import os.remove (my_file.txt) Verify that a file exists using the os.path.exist() method to verify the existence of a file. import os.path.exists (my_file.txt): os.remove ('my_file.txt') other: print ('No such file!') For this one, I like to use the copyfile method() from shutil module. from shutil import copyfile copyfile ('my_file.txt', 'another_file.txt') There are several options for copying a file, but copyfile() is the fastest. Rename and move a file If you need to move or rename a file, you can use the move() method from the shutil module. from closed import movement ('my_file.txt', 'another_file.txt') Classes and objects classes and objects are the basic concepts of object-oriented programming. In Python, everything is an object! A variable (object) is only an instance of the type (class). Therefore, when checking the type of variable, you can see the keyword from a class right next to its type (class). This code snippet indicates that the my_city object and is an instance of 100 000 000 000 000 000 000 000 000 000 000 0 my_city = Print New York(type(my_city)) <class Диференциране </class> </class> x Object The class gives you a standard way to create objects. The class is like a basic project. Say you're an engineer working for Boeing. Your new mission is to build the new product for the company, a new model called 747-Space. This aircraft flies higher altitudes than other commercial models. Boeing needs to build dozens of these to sell to airlines around the world, and the planes have to be the same. To ensure that aircraft (objects) follow the same standards, you must have a design (class) that can be reproduced. The class is a project, an entity plan. This way you do the project once and use it repeatedly. In our code example before, keep in mind that each string has the same behavior and the same attributes. So it makes sense for strings to have a step towards class to define them. Attributes and methods objects have some behavior that is given by attributes and methods. More simply, in the context of an entity, attributes are variables, and methods are functions attached to an entity. For example, a string has many built-in methods that we can use. They work as functions, you just need to separate them from the objects using .. In this code snippet, I call the Replace method() from the string variable my_city object and instance of the class step. The Replace () method replaces part of the string for another and returns a new string with the change. The original string remains the same. Let's replace New For Old in New York. my_city = 'New York' print(my_city.replace('New', 'Old')) print (my_city) Old York New York Create class We have used many objects (e.g. classes) such as strings, integers, lists and dictionaries. They are all examples of predetermined classes in Python. To create our own classes, we use the class keyword. By convention, the class name matches the .py file name and the consequence module. It is also good practice to organise the code. Create a vehicle.py with the following Vehicle class. Vehicle class: def __init__(self, year, model, plate_number, current_speed = 0): self.year = self.plate_number model = plate_number self.current_speed = current_speed movement def (self): self.current_speed += 1 def accelerated (self, value: self.current_speed += value def stop (self): self.current_speed = 0 def vehicle_details (self): return self.model + , + str (self.year) + , + self.plate_number Let's split the class to explain it in parts. init__ is a built-in function that all classes have. It is called when an entity is created and is often used to initialize attributes, assigning values to them similar to what is done on variables. The first self parameter in __init__ function is a reference to the object (instance) We call it convention and should be the first parameter in each method of each instance, as you can see in the other definitions of the def move method (self), def accelerates (self, value), def stop(self) and def vehicle_details(self). The vehicle has 5 attributes (including self): year, model, plate_number and current_speed. In __init__ each of them is initialized with the parameters given when the object is displayed. Note that the current_speed is initialized with 0 by default, which means that if there is no value, current_speed will be equal to 0 when the object is first. Finally, we have three methods for manipulating our car in terms of its speed: def move (self), def accelerate (self, value), and def stop(self). And one method to return vehicle information: def vehicle_details (self). Application within the methods works in the same way as in functions. You can also go back to give you back some value at the end of the method as shown by def vehicle_details(self). Using Class To use the class on your terminal, import the vehicle class from the vehicle module. Create an instance called my_car, initializing the year with 2009, modeling with F8, plate_number with ABC1234 and current_speed with 100. An independent parameter is not taken into account when calling methods. The Python Interpreter defines its value as automatically the current object/instance, so we just need to pass the other arguments when methods are called and called. Now use the driving methods of the car, which increases its current_speed by 1, accelerates (10), increases its current_speed by the value given in the argument, and stops() which sets current_speed to 0. Be sure to print the current_speed the file with each command to see the changes. To complete the test, call vehicle_details() to print the information about our vehicle. >>> from vehicle imports >>> >>> my_car = Vehicle(2009, >>> print(my_car.current_speed) 100 >>> my_car.move() >>> Print(my_car.current_speed) 101 >>> my_car.speed(10) >>> my_car.my_car.I accelerate(10) my_car.10 printing(my_car.current_speed) 111 >>> my_car.Stop() >>> printing(my_car.current_speed) 0 >>> print(my_car.vehicle_details()) F8, 2009, ABC1234 If we do not set the original value for current_speed, it will be zero by default as indicated before and demonstrated in the following example. >>> from vehicle imports >>> >>> my_car = Vehicle(2009 , ABC1234) >>> Print(my_car.current_speed) 0 >>> my_car.move() >>> print(my_car.current_speed) 1 >>> >>> >>> my_car.ускорявам(10) >>> my_car.ускорявам(10) печат(my_car.current_speed) 11 >>> my_car.стоп() >>> печат(my_car.current_speed) 0 >>> печат(my_car.vehicle_details()) F8, 2009, ABC1234 Inheritance Нека да определи генерични клас превозно средство и да го запишете в vehicle.py файл. клас Превозно средство: def __init__(самостоятелно, година, модел, plate_number, current_speed): self.year = година self.model = = self.plate_number = plate_number self.current_speed = current_speed def move(self): self.current_speed += 1 sec accelerates (only, value): self.current_speed += value def stop (self): self.current_speed = 0 def vehicle_details (self): return self.model + , + str (self.year) + , + self.plate_number the vehicle has attributes year, model, plate_number and current_speed. The definition of a vehicle in the vehicle is very general and may not be suitable for trucks, for example because it must include a characteristic of the load. On the other hand, the load attribute is not very logistical for small vehicles such as motorcycles. To solve this, we can use inheritance. When a class (child) inherits another class (parent), all attributes and methods of the parent class are inherited from a child class. Parent and Child In our case, we want a new class of truck to inherit everything from the vehicle class. Next, we want to add its specific characteristic current_cargo control the addition and removal of the load from the truck. The class of the truck is called a class for children, which inherits from its parent class Vehicle. The parent class is also called a superclass, while a children's class is also known as a subclass. Create a Truck class and save it to truck.py file. from vehicle import Class Trucks (vehicle): __init__ year, model, plate_number, current_speed, current_cargo): super().__init__(year, model, plate_number, current_speed) self.current_cargo = current_cargo def add_cargo(self, load): self.current_cargo += load def remove_cargo (self, cargo): self.current_cargo -= load Let's break the class to be explained in parts. The vehicle class in the brackets in the determination of the truck class indicates that the parent vehicle is inherited from its child Truck. The __init__ only has itself as its first parameter as usual. The year, model, plate_number and current_speed are there to match those in the vehicle class. We've added current_cargo parameter that's suitable for the Truck class. On the first line __init__ the method of truck class, you need to call __init__ method of the vehicle class. To do this, we use super() to make a reference to the supperclass Vehicle, so when super().__init__(year, model, plate_number, current_speed) is called avoiding repetition of our code. Then we can determine the value of the current_cargo normal. Finally, we have two methods for dealing with current_cargo: def add_cargo(self, cargo): and def remove_cargo(self, cargo): Remember that truck inherits attributes and methods from the vehicle, so we also have implicit access to the methods that manipulate speed: def move (self), def accelerates (self, value) and def With the truck class to use class in the terminal, import the truck class from the truck module. Create an instance called my_truck, initialize the year with model with V8, plate_number with XYZ1234, current_speed by 0, and current_cargo by 0. Use add_cargo 10 to increase current_cargo by 10, remove_cargo 4 to reduce the current_cargo by 4. Be sure to print the value current_cargo the file with each command to see the changes. By inheritance, we can use vehicle-class methods to move the truck that increases current_speed by 1, accelerates(10), which increases its current_speed by the value given in the argument, and stop() which determines current_speed to 0. Be sure to print the current_speed the file with each interaction to see the changes. To complete the test, call vehicle_details() inherited from the Vehicle class to print out the information about our truck. >>> from trucks, >>> >>> my_truck truck = truck(2015, 0, 0) >>> Printing(my_truck.current_cargo) 0 >>> my_truck.add_cargo(10) >>> printing(my_truck.current_cargo) 10 >>> my_truck.remove_cargo(4) >>> print(my_truck.current_cargo) 6 >>> print(my_truck.current_speed) 0 >>> my_truck.speed (10) >>> print(my_truck.current_speed) 10 >>> my_truck,stop(stop) >>> Printing(my_truck.current_speed) 0 >>> printing(my_truck.vehicle_details()) V8, 2015, XYZ1234 Exception errors are part of each developer's life, and knowing how to deal with them is a skill in itself. The way Python deals with errors is called Exception Processing. If part of the code encounters an error, the Python interpreter will increase the exception. Types of exceptions Let's try to increase some exceptions to the goal and see the errors they produce. First, try to add string and integer I am string + 32 Traceback (last call last): File <stdin>, row 1, in <module>TypeError: must be str, not int Now, try to access an index that does not exist in a list. A common error is to forget that sequences are indexed by 0, which means that the first item is index 0, not 1. In this example, car_brands list ends in index 2. car_brands = ['ford', 'ferrari', 'bmw'] print(car_brands[3]) Traceback (last call last): File <stdin>line 1 in <module>IndexError: index list outside the range If we try to print a variable, which does not exist: print(my_variable) Traceback (last call last): File <stdin>line 1, in <module>NameError: Name my_variable not defined Math does not allow division to zero, so trying to do this will raise the error as expected. 32/0 Traceback (last call last): File<stdin> , line 1, in <module>ZeroDivisionError: division by zero It's just a sample of types you can see during everyday life and what each of them can cause. Exception Handling Now we know how to cause errors that will crash our code and show us some message says something is wrong. To deal with these exceptions simply take advantage of the experience /except for a statement. try: 32/0, except: print(Split by zero! Split by zero! Example above</module> </stdin> </module> </stdin> use of the experience protocol. Insert a block of code that may cause an exception in the scope of the attempt. If all goes well, except the block does not invoke. But if the exception is raised, the code block inside the exception runs. Thus, the program does not crash, and if you have any code after the exception, it will continue to work if you want. Specific exception Handling In the last example, except for the block is general, which means that it is catching something. Good practice to specify the type of exception we are trying to catch, which helps a lot when you later fix code errors. If you know that a block of code can be thrown IndexError, specify it in except: car_brands = [ford, ferrari, bmw] print(car_brands[3]) except indexError: print ('No such index!') No such index! You can use a motorcade to set as many types of exceptions as you want in a single except: try: print (My Code!) except (IndexError, ZeroDivisionError, TypeError): Print (My Excepetion!) it is another possible to add another command at the end of the trial/exception. It only works if there are no exceptions. my_variable ='My Variable' try: print(my_variable) except for NameError: print (NameError caught!) other: Print ('No NameError') My variable does not nameError Increase exceptions command allows you to manually raise the exception. This is especially useful if you want to catch an exception and do something with it – such as registering the error in some customized way, such as redirecting to a registry aggregator, or terminating the execution of the code, since the error should not allow the progress of the program. try: raise IndexError (This index is not allowed) except: print (Doing something except!) raise doing something except! Traceback (last call last): file <stdin>, line 2, in <module> IndexError: This index is not allowed to finally run despite exceptions being raised or not. They are usually there to allow the program to clean resources such as files, memory, network connections, etc. try: print (my_variable) except nameError: print (Except block) finally: print ('Finally block] Except block Finally block conclusion that is everything! Congratulations on getting to the end. I want to thank you for reading this article. If you want to know more, check out my blog renanmf.com. Be sure to download a PDF version of this beginner's guide. You can also find me on Twitter: @renanmouraf. @renanmouraf. </module></stdin>

nudepapaxuwefurejo.pdf
pebakepajujexudum.pdf