Python 3 Syntax Guide

E-Book by Mihai Cătălin Teodosiu

~ Based on the **Python 3 Network Programming - Build 5 Network Applications** Video Course ~

~ Click Here to Access the Course ~

Contents

How to Install Python 3

Hi! Welcome to this Python 3 course! I'm really glad you're here and I'm very confident you will enjoy this training and learn a ton of useful information, concepts and skills, that will help you make the next step in your career, or, at least, improve your current job situation. Without further ado, let's start this course, by first installing Python 3.

First of all, I strongly recommend going through this training using a laptop or a PC, since we're going to work with lots of code and you're going to have to keep the pace and write code alongside me throughout the entire course. Also, for better readability of the code I'm going to write, watching this training on a smartphone or tablet would definitely not benefit you.

Secondly, since recent studies show that, on the desktop operating systems front, Windows holds over 80% of the market share, I decided to use Windows 10 as the backend operating system for this course.

However, you will find some guidelines for installing Python 3 on other platforms, as well, in the next lecture, and all the concepts discussed throughout this course are 100% applicable, regardless of the operating system you're going to use, so, you don't have to worry about that at all.

Now, let's open up the official website of the Python programming language, by going to:

https://www.python.org/

Next, hover your mouse over the Downloads tab and then simply press the Python 3.7.0 button. By the way, don't worry if, at some point, there's going to be a newer minor version, like 3.7.5 or 3.8. That won't change the validity of the content inside the course. The most important thing here is the major version, which is version 3. As long as that major version will be around, and it will, for quite a lot of years, this course will be 100% applicable and valid.

Next, let's double-click this file and run it. A new window opens up. Here, I'm going to check the *Add Python 3.7 to PATH* checkbox - we're going to

discuss what that means, later on in the course - then, I'm going to select *Install Now*. After this step, Python will run the installer for a couple of seconds and, that's it, *Setup was successful*. You can hit *Close* and now you have Python 3 installed on your Windows machine.

To check whether the installation was indeed successful, open up the Windows command line, by simply typing **cmd** in your Start menu, and type in **python** and hit Enter. You should get an output similar to this one, where you can see the version of Python currently running on your system and this special command line, which is actually called the Python interpreter.

D:\Windows\System32>python

Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD64)] on win32

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

>>>

In the next video, we're going to dive deeper into this interpreter and its role.

https://realpython.com/installing-python/

https://docs.python.org/3/using/index.html

The Python Interpreter

First of all, remember that you can run or execute your Python code in two ways. Either by writing your code inside a Python file, also called script, and then execute that file, or using the Python interpreter.

You can think of the interpreter as being a live session of Python, where each line of code you enter gets executed immediately, whereas using Python scripts, you can write your code now, save the file and run it later, whenever you choose to.

Now, focusing on the interpreter, you have two options. The first option is to use this one right here, in the Windows command line. This works perfectly, but, as you can see, it's not so eye-catching. At least for me, it isn't.

Don't worry, there's another option available. When installing Python, you also get IDLE, which is Python's IDE. IDE stands for Integrated Development Environment - that's a mouthful - and, in short, it revolves around the Python interpreter, also adding lots of features and settings, to make your programming experience better.

To access IDLE, all you have to do is go to your Start menu again and search for *idle*. Next, open it and let's have a quick overview of Python's IDE.

As you can notice already, this looks awfully similar to the output we got inside the Windows command line after typing in the **python** command, right?

Now, of course, depending on when you're watching this video, some of these version numbers may slightly differ, but that's no reason to worry, every bit of this course stays relevant nonetheless.

Another thing here that you may notice is the font size. Maybe, in your case, you have a smaller font size than mine. That's because I have already customized it to look bigger and prettier. You can do that, as well, by going to Options - Configure IDLE and then selecting the font and font size you like

better. Also, make sure that the Indentation Width is set to 4 spaces. You will understand why this setting is so important, later on in the course.

You can also go on to the Keys tab and maybe customize some keys and shortcuts, in order to further improve and optimize your IDLE experience. However, it is not my purpose to go through all the settings and options. Instead, I wanted you to know how to configure your working environment on a basic level. Remember to hit Apply and OK, and now' you're good to go. I think you can now understand why I like this version of the interpreter better than the Windows cmd one.

Also, keep in mind that you can run as many simultaneous sessions of IDLE as you need, without one interfering with the other. They will be completely separate working environments. However, the font, font size and key shortcuts are kept across all the sessions of IDLE.

Now, you have the Python interpreter ready to use and expecting your code. We will use it a lot across this course, don't worry.

Next, we're going to take a look at the other way of running Python code, using scripts.

https://docs.python.org/3/library/idle.html

Python 3 Basics - Scripts

A Python script is, basically, just a text file in which you input your Python code in order to build a larger application. Since the Python interpreter is suitable for small tasks and chunks of code only, Python scripts allow you to build programs having hundreds or thousands of lines of code, make changes to your code and run it as many times as you want.

The main thing you have to remember here is that each Python file or script must have the **.py** extension, not .txt or anything else. Furthermore, you can name your script however you like, just as with any other file on your system.

Now, in order to create and edit the text within your Python script, you must use a text editor. You are allowed to use almost any text editor on the face of the Earth, however, keep in mind that some text editors provide some additional and very useful features when dealing with code. For instance, you can use Notepad to write your Python scripts, but Notepad doesn't provide programming features like syntax highlighting, which, as its name suggests, highlights some Python-specific words using predefined colors, just to make your code easier to read.

That's why I choose to use Notepad++, which is a free text editor that you can download from https://notepad-plus-plus.org/.

Just open their website, go to *Downloads* and search for the latest version, which may be labeled as *Current Version*. Scroll down and click on the *Notepad++ Installer 64-bit x64* to download this software for Windows 10.

After the download has been completed, just install Notepad++ as you would do with any other piece of software. The installation process is pretty straightforward. Next, open up Notepad ++ and let's analyze it, a bit.

First, you should configure the font size, as we did with Python's IDLE. To do this, just go to *Settings - Style Configurator* and then choose your font and font size. Remember to hit *Save and Close* to apply your changes.

Next, let's write our first Python script.

For this, I'm going to use a Python function that does nothing more than to just print the characters in between its parentheses to the screen.

So, let's write this inside our script.

print("Hello Python")

Next, I will use *File - Save as…*, then enter the name **file1** and choose the extension from the *Save as type* list. We want to create a **.py** file, because that's the file extension we need for any Python script. So, we will choose *Python file*. Also, you should choose the location in which to save the file. I will save it to my D: partition. Great! You've just created your first Python script.

Now, it's time to run our small application. We can do that by simply opening the Windows command line again and typing in **python**, a space and then the name of the file we just created. Also, remember to add not just the file name, but also the path to that file and, very important, the **.py** extension; so, in my case, that would be *D:\file1.py*.

As soon as you hit Enter, the program will print out "Hello Python" to the screen, according to the print() function inside the script.

D:\Windows\System32>python D:\file1.py

Hello Python

Sweet! You have just built your first Python script and successfully ran it.

Congrats!

Python 3 Basics - User Input

Throughout this lecture, you are going to learn how to insert some input into a Python program. Actually, you're going to use a specific function, in order to ask the user for input, store the information he's entering at the prompt and then use that information further into the program. This is especially useful when you need to build an interactive application, usually having some sort of menu that the user needs to interact with.

Examples of such menus are **"Please enter your username: "** or **"Choose an option from the following list: "**.

Ok, let's get to work. The function I'm talking about is called **input()**.

Now, let's create a Python file on the D: drive called **file2.py** and prompt the user to enter a string that he wants to be printed out on the screen.

… create file on D: …

Next, let's open up the file and write the following code, using the input() function. Let me write this line and then we will analyze it inch by inch.

```
user_says = input("Please enter the string you want to print: ")
```

Looking at this line of code, you may ask yourself: what is this "user_says" thing? Well, that's a Python variable and, don't worry, we will talk more about variables very soon. For now, just keep in mind that by using a variable, you can *quote* store or save *unquote* the value entered by the user, for later use.

This so-called *storing* or *saving* of the user's input is accomplished using the equal sign, which is called an assignment operator, but more on that later in this section.

Following the equal sign, we have the input() function. And again, don't worry, we also will discuss functions extensively in this course. For now, let's just focus on the use case of this code.

Next, inside input()'s pair of parentheses, you have to type in a description, a phrase, which is actually a string asking the user for input. This is completely up to you to come up with an appropriate sentence.

A good practice here is to also enter a colon and a space after the text, so when the user inputs some data, it will be clearly separated from the sentence you wrote; just to make everything pretty and easy to read.

Finally, do **not** forget to enclose everything you write in between parentheses, also using either double or single quotes.

Last, but not least, in order to have our text printed out on the screen and visible to the user, we should use the print() function in Python, to print out the content of the user_says variable.

print(user_says)

Ok, that's it! Now, let's save the file and run it inside the Windows command line, cmd.

… Ctrl+S, python file1.py …

You can now see that the program nicely asks you to enter whatever you want to be displayed on the screen. So, let's try a couple of strings.

Hello Python

Python is the best

Sweet! It works like a charm!

So, keep in mind that you are able to capture user input, display this input on the screen or even store it using a variable and later re-use it across your script. All this, with the help of the input() function. See you in the next one!

https://docs.python.org/3/library/functions.html#input

Python 3 Basics – Variables

What is a variable? How to define a variable and what is it good for, in Python?

A variable is nothing more than just a reserved location in your computer's memory, used to store information; values, to be more precise. This means that when you create a variable you reserve some space in memory.

You can store different types of data using a variable; you can store a string, a number, a list or any other data type that you can think of.

Now, unlike other programming languages, in Python you don't have to explicitly declare a variable; instead, the declaration is done automatically when you assign a value to that variable, no matter what type of data you decide to assign to that memory location.

Furthermore, you can later access the value referenced by that variable and use it in other areas of your Python application.

Now let's think of how you should properly name a variable in Python.

Well, there are several rules to consider and follow for a clean and compliant code and also for avoiding any conflicts with Python's built-in names.

First, a variable name should always start with a letter, usually lowercase and never start with a number or any other symbol. However, there is **one** exception to this rule - some variable names start with underscore or double underscore, but these are Python specific structures, so let's leave them to Python.

The variable name may contain lowercase or uppercase letters, numbers and the underscore sign - but, as I said, not as the first character.

Also, do **not** include spaces or any other special characters inside variable names; this means no dollar signs, no commas, no parentheses, no question marks and so on.

And **remember**! Python names are case-sensitive, so, a variable named **my_var** is a different variable than a variable named **my_Var**, with an uppercase **V**.

Another thing about variable names is that you should keep a reasonable name length, so that it will be easier for you to remember it and reference it inside your code.

The last thing I should mention on this topic is that there are some Python reserved names, also called keywords, which you cannot use as a variable name. I have included a list of these Python keywords in the next lecture; you should keep it at hand and read it anytime you're in doubt.

Now, let's see how we can assign a value to a variable.

The rule is to use the equal sign - this should be regarded as an assignment operator, rather than the usual equal sign used in math.

In the left side of the equal sign you type the variable name, and in the right side you type the value you want to assign to that variable.

It doesn't matter if you leave a space between each side and the equal sign. Actually, I would advise you to leave a space on each side of the equal sign, for better readability of your code.

You are also able to do multiple assignment, so you can assign a value to multiple variables, at the same time. The syntax for this would be **a = b = c = 10**

Also, you can assign a different value for each variable within the same line of code, like this **a, b, c = 1, 2, 3**

I think that's really cool and may come in handy someday.

Now, let's have a brief look at how Python performs the variable assignment.

For this, we are going to use the id() built-in function, which returns the identity, the location identifier of a variable inside the computer's memory.

Let's assign the value 10 to the variable **a**.

a = 10

Ok, now let's use the id() function to see what is the address or identity of the value 10 in the computer's memory.

id(10)

Let's do the same thing on **a**.

id(a)

Notice we're getting the same value. That's because **a** is pointing to the location in memory in which the value 10 is stored.

Now, let's create variable **b**, which points to variable **a** and let's use the same id() function to see its location.

b = a

id(b)

Surprise, surprise! We got the same value as above because **b** is also pointing to the location in memory where 10 is stored. Think of this like putting two labels on the same recipient; the content of the recipient is the same, but it can be referred to in two ways, depending on which label you read.

Now, let's change the value of **b** to 20 and check its id again.

b = 20

id(b)

As you can see, **b**'s id is now different than before, because it is now pointing to the location in memory in which the value 20 is stored. However, the id of **a** is still the same, since it is still pointing to the value 10.

Now, since we are interested in the practical use of Python into the real world, I will not get into any more details regarding memory assignment and other deep concepts. Instead, I would prefer to move on and talk about which

data types you will encounter in Python and how to handle and use them in your programs.

https://www.digitalocean.com/community/tutorials/how-to-use-variables-in-python-3

Python 3 Basics - Data Types

Ok, following up on the previous lecture, let's see what types of data can be stored in memory and used inside Python applications.

The Python programming language defines many types of data for many types of operations. We will have a look at the most widely used data types starting with this lecture.

As an overview, let's enumerate the most famous and useful Python 3 data types; so, we have **strings, numbers, booleans, lists, sets, frozensets, tuples, ranges, dictionaries** and the **None** type.

All these data types are built-in into the Python programming language core and you will see that they are very clearly defined and easy to use.

One way to classify data types is by whether the objects of a data type can be modified after creation or not; this is called mutability or immutability, respectively. So, this leads to two kinds of data types: mutable data types, which can be modified after creation - here, we can mention lists, dictionaries and sets and immutable data types - here we have strings, numbers and tuples.

So, remember! If you have an object of an immutable type, you cannot change the content of that object in any way, shape or form. You will just have to create another object with the content you want.

Now, it's time to have a closer look at a data type we already talked about earlier in the course: strings.

https://docs.python.org/3/library/stdtypes.html

Python 3 Strings – Introduction

Let's define a string first - a string is a sequence of characters, meaning: letters, numbers and other characters like the dollar sign, spaces and punctuation marks enclosed by single quotes, double quotes or even triple quotes when spanning multiple lines.

So, let's define a string and assign it to a variable. I am going to create a variable named **my_string** and assign it the following value: "this is my first string", first using double quotes.

```
my_string = "this is my first string"
```

And, checking the result.

```
my_string
```

Then, using single quotes and checking the result again

```
my_string = 'this is my first string'
```

```
my_string
```

So, it is the same string, right? Ok.

Now, what do we need triple quotes for? Well, we need them whenever we want to enter a string on multiple lines; for instance, a comment in our code.

Let's see how we can do this; let's assign the same text to the **my_string** variable, this time with each word being on a separate line, so:

```
my_string = '''this
is
my
first
string'''
```

Hit *Enter* after each word to write the next one on a separate line. Finally let's end the string with the same set of triple quotes and hit *Enter* for the last time. Now, let's check our variable.

my_string

'this\nis\nmy\nfirst\nstring'

It looks like we have something new here - what's with all the **\n**s between the words? Well, **\n** is called a new line character and it signals a new line being created at that particular point in the string. In plain language, it means that we are inserting a break and a new row, a new line, in our text.

Ok, so, how we can get rid of them? It looks weird, right? Don't worry, it's simple! We just have to add a **** (backslash) wherever we want a new line to be inserted. This translates to something like: *"Go to the next row and continue the text there!"*. So, let's try this again.

my_string = '''this

is

my

first

string'''

'thisismyfirststring'

It works! No more new line characters! The insertion of backslashes is called escaping the new line characters. Remember this term, because we will use it again later in the course.

Now, let's talk about indexing, and this concept applies to other data types, as well. Python uses indexes to mark the position of an element within a sequence of elements; a string **is** a sequence of elements and the elements of a string are the characters themselves. One character, one element.

The first element of any sequence, when counting from left to right, has the index 0. Then, the second element of the sequence has the index 1, the third

element is positioned at index 2 and so on. So, when using indexes, remember to always start counting from 0!

When counting backwards, from right to left, the first index will be -1. So, the last character in a string will have index -1, when looking from right to left.

Indexes are enclosed by square brackets, when we want to access some letter of a string.

Let's see this in practice. Let's create variable **string1** and assign it the value *"Cisco Router"*.

string1 = "Cisco Router"

Now, how to extract the first character of this string? By using an index, of course! And, as stated before, that would be index 0. So, to access the element of **string1** positioned on index 0 in the string, we should type the following: the name of the variable, **string1** and then, without inserting any spaces, the index number in between brackets so **string1[0]** Hit *Enter*.

This returns the letter "C", which is correct, because this is the first character in the string.

Now, let's find the 3rd character of **string1**. What index should we use? Index 2, right? So, **string1[2]** returns "s"; correct, again!

Let's see the 6th element! We need **string1[5]**, right? And that would be the Space character between the words "Cisco" and "Router". Yes, spaces count as characters, too.

Now, for the negative indexes. Let's access the last character in the string. We will use index -1, right? So, **string1[-1]** returns the letter "r". Good!

What about **string1[-4]**? Who would that be? Well, let's count -1 is "r", -2 is "e", -3 is "t" and, finally, we have index -4. So, the letter "u" is the one located at index -4. Great!

One more thing on indexes - what if we enter an invalid index for our string?

Let's see, what do I mean by that? First, let's find out **string1**'s length. We can count how many characters are in that string visually, but what if we have a very, very, veeery long string, maybe a newspaper page? Python has a solution for this and it's called the len() function. This function is easy to use just type **len** and, without any spaces after it, add the variable name pointing to our string in between parentheses, so: **len(string1)** returns the number of characters, which is 12.

Now, back to the first question. What happens if we enter an invalid index? Well, let's try it! **string1** has 12 characters. So, starting at index 0 and counting from left to right, the last character should have the index 11, right?

So, **string1[11]** is "r". This is the same thing as string1[-1], which is correct. Now, let's see what happens if we enter **string1[20]**. Well, we got an error. Let's celebrate! It is one of the first errors we see in this course. Errors are a great tool for learning and troubleshooting our code, by the way. So, don't fear them! Now, let's read the error text. It says *IndexError* - that is the type of the error, there are many types of errors in Python and we will go through the most common ones later on in the course. Next, it says *"string index out of range"*, so Python found out that we typed in an invalid index in between the square brackets and told us that the index is indeed out of range, because a 12-characters string cannot have a character at position 20, obviously.

Now, it's time to have a look at some string operations and methods that will help us work with this data type.

Python 3 Strings – Methods

Ok, we've talked about indexing and how we can determine the length of a sequence, in our case a string, using the len() function. Now, let's see other operations on strings.

1. First, one more thing about indexes - you can find out the index of a character in a given string by using the index() method. Just remember that this method returns only the first occurrence of that particular character in the string. So, let's create variable a and assign it the string "Cisco Switch".

```
a = "Cisco Switch"
```

You can clearly see that the letter "i" appears two times in this string. So, let's find out the index of the first occurrence of "i" in this string.

```
a.index("i")
```

So, we have the name of the variable associated with the string, then a dot, then the name of the method, *index*, and then, in between parentheses, we enter the character we want to find out the index for, "i". Don't forget to enclose "i" in single or double quotes, since this letter is also of type string.

Ok, this returns 1 as being the index of "i" in the string, which is correct since it is the second letter there.

2. Another useful Python method is one that helps you find out how many times does a character appear in a string or, generally speaking, an element inside a particular sequence. This method is called count().

The syntax of count() is similar to the one of the index() method you've seen earlier. To use the count() method, just type in the name of the variable, then a dot, then the word **count**, then open and close parentheses and, finally, the letter you want to count surrounded by quotes so:

```
a.count("i")
```

Returns **2**, so, we have "i" two times inside our string, which is correct.

3. Another string method is find(). This method simply searches for a sequence of characters inside the string and, if it finds a match, then it returns the index where that sequence begins. So, let's see that. Let's type in:

a.find("sco")

So, the result indicates a match starting at index 2, which is correct, since the letter "s" is positioned at index 2 within the string.

On the other hand, if Python does **not** find a match, then it will return the value -1. So, let's test this. Let's search for the substring "xyz" inside the initial "Cisco Switch" string.

a.find("xyz")

And we see -1 being returned; that's because we don't have the substring "xyz" within the string referenced by the variable a, obviously.

4. Ok, let's see what else. We can also use some predefined Python methods to turn a string from uppercase to lowercase or vice-versa, if we want that. This can be accomplished by using the lower() and upper() methods.

So, returning to our string, let's do:

a.lower()

Now, we can see our string in lowercase only. Great!

Similarly, let's do:

a.upper()

Now all letters have been converted to uppercase. Sweet!

Keep one thing in mind here! Although we have just applied the lower() and upper() methods, the initial string is still the same. No changes have been applied.

a

So, this is a great proof that strings are indeed immutable.

5. You can also verify that a string starts or ends with a particular character or substring. We have two methods available for this. They are called startswith() and, as you might have already guessed, endswith().

Let's see them in action:

a.startswith("C")

Returns True. That's because it is true; "C" is the first character of the string.

Now, let's perform another check:

a.endswith("h")

This returns True, as well, based on the same logic.

On the other hand, let's try:

a.endswith("q")

Of course, this returns False, since our string ends in "h", not "q".

6. Three important methods which you should keep in mind, because you will use them a lot when working with strings, are the strip(), split() and join() methods. Let's test them one by one.

6.1. First, the strip() method eliminates all whitespaces from the start and the end of a string.

So, let's say we have a new string, one with 3 spaces before "Cisco" and 4 spaces after "Switch".

b = "   Cisco Switch   "

Now, let's apply the strip() method on string **b** and see the results.

b.strip()

So, b.strip() returns "Cisco Switch" without any spaces at the beginning and at the end of the string. That's nice and may be useful, so keep it in mind.

Now, consider that instead of the three spaces on each side of the string, we had three $ signs that we want to remove.

So, from a string that looks like this:

c = "$$$Cisco Switch$$$"

…we want to eliminate the leading and trailing dollar signs. For this, we should specify the character we want to remove in between strip()'s parentheses, so:

c.strip("$") will indeed return a nice and clean string.

But what if want all spaces removed from the string, including those inside the string? Then, we would use the replace() method, instead of strip().

Let's return to the string referenced by variable **b**, which has spaces at the beginning, inside and at the end of the string. You can use replace() like this, to get the string clean:

b.replace(" ", "")

This way, you are actually replacing every Space character with a so-called empty string. It works, the result is the one we expected, so great job again!

6.2. Now, let's have a look at the split() method. As its name implies, this method simply splits a string into substrings. Furthermore, you can specify a delimiter for splitting the string. The result of this method is a list. Don't worry, you will learn more about lists very soon. Let's see this method in action now.

Let's say that we have a string referenced by variable **d**, like this:

d = "Cisco,Juniper,HP,Avaya,Nortel"

The network device manufacturers in this string are delimited by commas. So, the comma will be regarded as our delimiter for the split.

What if we want to extract each provider from the string, in a nice format? Well, in this case, the split() method saves the day.

Let's type in the following:

d.split(",")

Python returns a list where each provider in the string is an element of this list and can be further used into an application. You can split by any delimiter you have in a string and get a list of elements that were previously separated by that particular delimiter.

6.3. Finally, we have the join() method for dealing with strings. Let's remember string **a**, 'Cisco Switch'. What if we want to insert a character in between every two characters of this string? So, we want to change this string to "**C_i_s_c_o_ _S_w_i_t_c_h**"

For this, we will have to use the join() method in the following way and this is a bit different than the syntax we've seen so far with the previous methods.

First, you type in the character you want to use as a separator, enclosed with double quotes. Or single quotes. It's up to you. So, this character would be the underscore in our case.

"_".join(a)

This could be read like this: "use the _ separator to join the elements of string **a**". And, following this statement, the result is the correct one, indeed.

'C_i_s_c_o_ _S_w_i_t_c_h'

Please find a comprehensive list of string methods in the link I have attached to this lecture.

https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

Python 3 Strings - Operators & Formatting

What else can we do with strings?

For instance, we can concatenate them. Concatenation means unifying two or more strings into a single string.

You can do this using the "+" operator, like you would when adding two numbers.

So, let's try this.

Let's set **x** with the value of "Cisco":

```
x = "Cisco"
```

…and **y** with the value of " Switch":

```
y = " Switch"
```

…and, finally, add them together:

```
x + y
```

"Cisco Switch"

Another thing we can do is string repetition, by using the multiplication operator, so:

```
x * 3
```

…returns *"CiscoCiscoCisco"*. Nice!

You can also verify if a character is in a string or not, using the **in** and **not in** operators. This may prove to be useful when dealing with huge strings.

So, let's look at **x** again and check if the character "o" is part of this string:

```
"o" in x
```

…returns **True**, great!

Now, let's try another one:

"b" in x

…returns False, because there is no letter "b" in **x**. But, if we use:

"b" not in x

This will result in True, because we negate the existence of letter "b" in "Cisco", which is indeed True.

Now, what's the deal with string formatting? Let's say we have some kind of string template and we want to constantly modify only **a few** words inside the text but keep the overall template the same. To see what I mean, let's assume we have the following string. Doesn't matter what it stands for, just focus on the string itself.

"Cisco model: 2600XM, 2 WAN slots, IOS 12.4"

We want to keep this string as a template and just change the model name, number of slots and IOS version a couple of times, while running our Python program. So, this would need to be a dynamic change each time.

For this, we should use the percent (**%**) operator, followed by **s** for string, **d** for digit or **f** for a floating-point number. Let's see the syntax for this:

"Cisco model: %s, %d WAN slots, IOS %f" % ("2600XM", 2, 12.4)

Now, let's translate this: the **%s** means that this is a placeholder for a string we will specify in between parentheses, at the end of this line. The **%d** operator follows the same logic, but for a number, instead of a string, and, finally, the **%f** refers to a floating-point number, a number with a decimal point. Now, moving on, the first value from within the parentheses is going to be associated with the first format operator in the string; the second value from within the parentheses is going to be associated with the second format operator in the string, and so on, for all the format operators you have in your string. Also, do **not** forget to insert the **%** sign between the string and the parentheses containing the values. This operator maps the format operators

inside the string with the values inside the parentheses. So, let's see this in action again:

"Cisco model: %s, %d WAN slots, IOS %f" % ("2691XM", 4, 12.3)

"Cisco model: %s, %d WAN slots, IOS %f" % ("7200XR", 8, 15.4)

"Cisco model: %s, %d WAN slots, IOS %f" % ("1841M", 1, 12.2)

Something you might have noticed is the addition of several decimal places when dealing with floating-point numbers. In order to control this behavior, you can easily choose the number of decimal places that you want to print out. Let's get back to our string and simply insert a dot and a value in between the percent operator and the letter **f**.

If we want just a single digit after the floating point, we should use .1, like this:

"Cisco model: %s, %d WAN slots, IOS %.1f" % ("2600XM", 2, 12.4)

For two digits, let's use .2, like this:

"Cisco model: %s, %d WAN slots, IOS %.2f" % ("2600XM", 2, 12.4)

Finally, if we want no digits at all and no floating-point, we can just enter a dot and we will get the value 12 in return:

"Cisco model: %s, %d WAN slots, IOS %.f" % ("2600XM", 2, 12.4)

That's awesome, right?

However, this is not the only way of dealing with string formatting.

Instead of formatting operators like the ones we've just seen, we can use another notation, replacing %s, %d or %f with a pair of curly braces. Also, after the string, the % operator we used for mapping the values is going to be replaced by a method called format(). Let's type this in:

"Cisco model: {}, {} WAN slots, IOS {}".format("2600XM", 2, 12.4)

So, what we did is we replaced the old formatting operators with curly braces and, right after the string we used a dot, the word format calling the format() method and then the content inside the parentheses stays the same.

Again, we're mapping each pair of curly braces inside the string template with each value inside the parentheses of the format() method. The result is the same.

Is this all there is to it? Well, no, it isn't. What else?

We can also use some sort of indexing when dealing with this type of string formatting. Why? You'll see why, in just a few moments.

Let's assign a value for each of these pairs of curly braces.

"Cisco model: {0}, {1} WAN slots, IOS {2}".format("2600XM", 2, 12.4)

Ok. If we run this line of code, the result is the same. That's because {0} is mapped to the first value in between parentheses, {1} is mapped to the second value and so on. Nothing unusual here. So, why do we need these values, these indexes, after all?

Let's say that, maybe, you want to switch the order in which the values on the right are mapped to the operators in the string. What will happen if we change our code like this?

"Cisco model: {2}, {0} WAN slots, IOS {1}".format("2600XM", 2, 12.4)

Well, we're noticing that we have easily switched the order of values inside our string template, using this simple trick. Although the mapping is still the same, meaning that {0} still corresponds to "2600XM" because it is the first element inside the parentheses, its position inside the string template has changed, according to our needs.

Another thing we can do is value repetition inside the string template. So, for instance, if we want 12.4 being printed instead of "2600XM", we can just use it twice in our string: once in its initial position and once instead of the substring at index 0. For this, just replace 0 with 2 and you get the corresponding value printed out twice inside the string.

"Cisco model: {2}, {1} WAN slots, IOS {2}".format("2600XM", 2, 12.4)

Nice, useful too. Now it's time to wrap up with strings by learning about string slices.

Python 3 Strings – Slices

Slices allow us to extract various parts of a string (or list, or other sequence of elements), leaving the initial string unchanged.

The syntax for a string slice is the following:

mystring[10: 15]

We have the name of the variable pointing to the string, followed by a pair of square brackets. In between the brackets, we have a colon. On the left side of the colon, we specify the index at which to start the slicing process. The slice will go up to, **but will not include**, the index specified on the right side of the colon. Let's see some examples to clarify this.

Let's create string1. Again, the meaning of this string is not important, but its structure and complexity will aid in our educational purposes.

string1 = "O E2 10.110.8.9 [160/5] via 10.119.254.6, 0:01:00, Ethernet2"

Let's extract the first IP address in this string, 10.110.8.9. The first character in our slice should be **"1"**. This character is located at index 5 in the string, right? Now, the last character should be **"9"**, which is located at index 14, right? Ok, so we will have to go up to index 15, as the end of the slice; however, as previously stated, the character at index 15 will **NOT** be included in the slice. Now, let's write down the slice.

string1[5:15]

…returns the IP address we wanted to extract, 10.110.8.9. If we would have gone up to index 14 only, then we would have had just 10.110.8. returned, which was an incomplete IP address and an incorrect result.

Now, what if we don't specify the second index inside the brackets? Well, then the string slice would start at the index given before the colon and would end **at the end** of the string. So, this way, we will get the rest of the string, starting from the character at index 5. Let's test this, as well.

string1[5:] …returns the slice as expected. Cool, right?

What if we only use a second index, the one after the colon, but don't specify the first index? Then, the slice would simply start at the beginning of string1 and would go up to, **but not including**, the character at the index we enter after the colon. Let's try this.

string1[:10] …returns the expected slice, starting at the beginning of the string. That is correct again.

Another question might be: what if we don't enter any indexes at all? Well, as you might expect, the entire string will be returned, with no changes whatsoever. Let's verify this, too.

string1[:] …returns the entire string, as expected. Sweet!

Now, what about negative indexes? We mentioned them in the first lecture about strings, but now, let's try extracting a couple of slices, using negative values for indexes.

string1[-1] …will return the last character in our string, right? Right. That character is "2", in our case.

string1[-2] …should return the next character, when reading from right to left, right? That's also true.

What if we want to extract a slice containing the "Ethernet" substring, this time using negative indexes? Well, we can count from right to left, starting at index -1, and then see what indexes correspond to the first and last letters of "Ethernet". So, that would be "E" and "t". Let's count. Character "t" has the index -2 and "E" has the index -9. Now, we can write the slice.

string1[-9:-1]

So, our slice starts at index -9 and goes up to, but does not include, index -1. This means that the last character we are extracting is the one positioned at index -2, which is right before -1. The final result is correct, since we got the "Ethernet" substring printed out.

Now, what if we want to obtain the last 5 characters of our string? How can we do that? So, we want our slice to start with the 5th character, when counting from right to left. This means we have to use index -5 and go all the way to the end of the string. That's why there's no need to specify a second index, just like earlier, when we were dealing with positive indexes. So:

string1[-5:] …indeed returns the last 5 characters of the string, namely *'rnet2'*.

What if we want to slice string1 starting at the beginning of the string and leave out the last 5 characters? Then, we should skip the first index since its absence means "start from the first character", and then go up to, but not including, index -5. So, the character at index -6 will be the last character we will include in our slice. Let's test this!

string1[:-5] …is indeed returning string1, minus the last 5 characters of the string, which is correct.

One more thing about slices. You can specify a third element within the square brackets, after the indexes, also separated by colon. This is called a step.

For instance, if you would like to skip every second character of the string and obtain a new string with these elements removed, you can write the following slice:

string1[::2]

So, we are not specifying any indexes, because we want to refer to the entire string, but we are inserting a step after the second colon, to skip every second element of the string. The result is the expected one. Notice that the new string consists of the elements at indexes 0, 2, 4 and so on from the original string.

The last thing I'll show you is how to print out the string in reverse order, using slices and indexes. For this, we will again use a slice and a step. But, what would be the value of that step? Well, since we want to get the string

in reverse order, we should start with the last character of the string, right? So, let's try it!

string1[::-1]

Using this slice, we are referring to the entire string, starting from the end of the string, at index -1, character by character. And this is how you can easily print a string in reversed order.

Ok, this should be more than 95% of what you will need for dealing with strings. Keep in mind that these concepts should be practiced and revisited every once in a while, since you will use strings, slices and string methods intensively inside your Python applications. So, if you feel uncertain about any concepts related to strings, feel free to review any lecture or refer to the documentation on python.org for additional string methods.

Python 3 Numbers - Math Operators

Ok, let's have a look at numbers and math operations in Python 3.

The first thing I should mention here is that Python defines three numerical types. They are: integers, floating-point numbers and complex numbers. The complex numbers are usually used in some advanced math operations and are not of great interest for our current needs. Instead, we will work a lot with integers and floating-point numbers and that's why we will focus on these two numerical types and their corresponding operators and functions.

So, let's define a variable and assign it an integer and another variable associated with a floating-point number, or a float, in short. By the way, the float type refers to real numbers having a decimal point positioned in between the integer part and the fractional part. So, let's consider the following variables and numbers.

num1 = 10

num2 = 2.5

type(num1)

<class 'int'>

type(num2)

<class 'float'>

Now, let's see what operations we are able to perform using integers and floating-point numbers.

Addition: **1 + 2**

Subtraction: **2 - 1**

Division: **5 / 2**

Integer division: **5 // 2**

Multiplication: **4 * 2**

Raising to a power: **4 ** 2**

Modulo (this means finding out the remainder after the division of one number by another): **5 % 2**

Now, it's time to have a look at the comparison operators:

Less than: **4 < 5**

Greater than: **5 > 4**

Less than or equal to: **4 <= 5**

Greater than or equal to: **5 >= 4**

Equals: **5 == 5**

Not equals: **4 != 5**

Now, let's talk a bit about the order of evaluation for these operators inside a mathematical expression. What if we have to deal with multiple operators within the same expression? Which operations have priority over others?

Well, the order is the following: firstly, the raising to a power operation has the highest priority in an expression. This means it will always be evaluated first. Then, we have the multiplication, division and modulo operations, with equal priorities and, lastly, we have addition and subtraction, also with equal priorities. Let's see an example.

100 - 5 ** 2 / 5 * 2

What would be the correct result here? Well, Python will first evaluate 5 ** 2 which returns 25. Then, since division and multiplication have the same priorities, they will be evaluated from left to right. So, 25 divided by 5 equals 5, then multiplied by 2 equals 10; and finally, subtraction having the lowest priority in this expression, we will have 100 - 10 = 90. Let's check this!

90

Now, let's look at two types of conversions. Let's see how we can convert an integer to a float and vice-versa. Well, Python has two functions available for these operations. Let's see them:

int(1.7)

The result is 1, because the int() function will round down the number in between parentheses to the nearest integer, which is 1, in this case. Next, we have:

float(2)

The result is 2.0. The float() function will add this .0, converting 2 from integer to a floating-point number.

Finally, let's have a look at a few more functions which may come in handy in the future, when working with numbers.

The abs() function returns the absolute value. This is actually the distance between the number we provide and 0.

abs(5) …returns 5, since this is the distance between 5 and 0.

abs(-5) … also returns 5, since the distance is the same as before.

Next, let's start comparing two numbers.

max(1, 2) …returns 2, the largest number in between parentheses.

min(1, 2) …returns 1, the smallest number in between parentheses.

And, finally, let's see another way of raising to a power, this time using a built-in Python function.

pow(3, 2) …where 3 is the base and 2 is the exponent or power.

The result is, of course, 9, which is correct. I think this should be more than enough math for now, so, let's move on to Booleans.

https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Python 3 Booleans - Logical Operators

As a short definition, we can say that a boolean data type defines only two possible values: **True** and **False**. The name comes from George Boole. He was a 19th century English mathematician and philosopher.

Now, leaving history aside and returning to Python, you can think of these two values as being equivalent to 1 and 0.

In Python, **True** is written with a capital T and **False** with a capital F, keep that in mind!

We have already seen True and False in some examples during the previous lectures, so they are not completely new to you at this point in the course.

Basically, they are used to evaluate whether an expression is true or false and can be further used in conditional or loop structures, as we will see later in the course.

For now, let's evaluate some basic expressions and see how Python evaluates each of them.

1 == 1

True

1 == 2

False

"python" == "python"

True

"python" == "Python"

False

3 <= 4

True

Ok. You get the idea. These were some pretty basic evaluations we did.

Now, there are three main boolean operations, each of them having a specific operator.

The first operator is "and", the second operator is "or", the third operator is "not". Let's analyze each of them.

AND means that both operands should be **True**, in order to have the entire expression evaluated as **True**.

So, let's see this in practice. Let's do an AND operation between two true expressions.

(1 == 1) and (2 == 2)

By the way, you can skip the parentheses. I'm just using them for better readability and I advise you to do the same when dealing with longer expressions.

So, both expressions being evaluated as True, the final result is True.

Next, when performing an AND operation between two False expressions the result will be False.

(1 == 2) and (2 == 3)

False

The third case is when one expression is evaluated as True and the other one is evaluated as False. This will also return False.

(1 == 1) and (2 == 3)

False

The conclusion here is that, when using the AND operator, if both expressions are True, then the result will be also True. On the other hand, if

at least one expression is evaluated as False, then the result will be False, as well.

Now, let's study the **OR** operator. OR works like this: if at least one of the expressions evaluates to **True**, then the final result is **True**. If they are both False, then the final result will be False, as well. So, when using OR, it is enough if only one expression is True, in order to have True as the final result. Let's see this in practice.

(1 == 1) or (2 == 2)

True

(1 == 1) or (2 == 3)

True

(1 == 2) or (2 == 3)

False

Finally, using the **NOT** operator means simply denying an expression. If that expression is True, then denying it will result in False, right? And vice-versa, of course. Let's verify this.

not(1 == 1)

False

not(1 == 2)

True

Ok. One more thing to keep in mind here: some Python values always evaluate to False. They are:

None; 0; 0.0; 0j; empty string ""; empty list [ ]; empty set set(); empty tuple (); empty dictionary {}

Python provides the **bool()** function to help us evaluate values and expressions as True or False. So, let's use this function to check the always-False values.

bool(None) *False*

bool(0) *False*

bool(0.0) *False*

All other values in Python are considered to be True, so:

bool(1) *True*

bool(2.2) *True*

bool("hello") *True*

bool([ ]) *True*

Booleans and logical operators are very useful in Python, especially when dealing with if / elif conditionals and while loops. But more on that, later.

https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not

Python 3 Lists – Introduction

Ok, let's talk about lists; big topic ahead! So, what exactly is a list?

A list is, actually, a sequence consisting of elements separated by comma. The sequence of elements is enclosed by square brackets. You can have any data types as elements of a list - strings, numbers, tuples or even other lists; and a list may have any number of elements.

Similarly to strings, which can also be regarded as sequences of characters, lists are indexed, meaning each element has a certain position inside the list, starting at index 0.

You can also use the len() function to see the number of elements in the list and slices to extract only a portion of a list.

As opposed to strings, lists are mutable, meaning that you **can** modify a list by adding or removing elements and we will see and use that a lot throughout this course.

Let's create our first list in this course. We will name it list1 and will be initially empty.

```
list1 = [ ]
```

So, in order to have an empty list, you just have to type the square brackets and that's it, nothing else is needed.

To check that this is indeed a list, let's use the old type() function on list1.

```
type(list1)
```

Now, let's add some elements to our list.

```
list1 = ["Cisco", "Juniper", "Avaya", 10, 10.5, -11]
```

So, now we have a list with some strings and numbers as elements. Good.

Let's remember the len() function from the Strings section and use it on our list.

len(list1)

So, our list has 6 elements. That is correct.

Now, let's see how we can access elements inside a list. Well, the same way as we did with characters in a string, using indexes. So:

list1[0]

'Cisco'

list1[1]

'Juniper'

list1[-1]

-11

list1[-2]

10.5

As with strings, if we enter an invalid index we will get an Index Error in return, stating that the list index is out of range.

list1[100]

Indexerror: list index out of range

To check that lists are indeed mutable, let's try to replace an element in the list.

So, we still have list1 in memory. **list1[2]** will return the **"Avaya"** element, right? Now, let's replace it with another vendor, let's say **"HP"**.

We just have to use the equal sign to modify this.

list1[2]

'Avaya'

list1[2] = "HP"

list1

['Cisco', 'Juniper', 'HP', 10, 10.5, -11]

So, plain and simple, this is how you can update a list. Just type in the name of the variable and, in between square brackets, you have to insert the index at which you want the replacement to be made. Finally, following the equal sign, just enter the new element and that's it, you're good to go.

Now, as a short review, the first element of a list is always positioned at index 0 and the last element corresponds to index -1. Also, remember that lists are mutable, unlike strings or numbers. I hope this is clear and now we can move on to list methods.

Python 3 Lists – Methods

Now, it's time to see how to handle lists and list elements and what tools does Python provide for this.

We've already seen the len() function being used on a list. But, what if you want to find out the maximum or minimum value within a list? Well, you have the max() and min() functions available for that.

So, let's consider list2.

list2 = [-11, 2, 12]

min(list2)

-11 …which seems about right, since it is the only negative value in the list.

Next, we have:

max(list2) …which returns 12. Again, that's correct.

Now, what about a list of strings?

list3 = ["a", "b", "c"]

In this case:

min(list3) …will return a.

max(list3) …will, of course, return "c".

However, if we have a list with various types of elements, say numbers and strings mixed together, like **list1** is, how does Python compare a string with an integer and return the maximum value in the list? Well, in that case, **max(list1)** will return a TypeError, saying that Python 3 cannot compare integers with strings, or, in fact, generally speaking, comparison operators are not supported between different data types. This **was** possible in Python version 2, where the max() function would've returned the value Juniper,

considering strings as being greater than integers. However, this has changed in Python 3 and the result is an error.

max(list1)

Typeerror: '>' not supported between instances of 'int' and 'str'

Now, let's check the available list methods we have at hand.

First, we should learn how to append a new element to a list. It's simple enough, we have the same list1.

list1

['Cisco', 'Juniper', 'HP', 10, 10.5, -11]

To append an element to this list, just use the append() method, like this:

list1.append(100)

list1

['Cisco', 'Juniper', 'HP', 10, 10.5, -11, 100]

Now, let's remove an element from our list. We have three options this time.

First, we can use the following command:

del list1[4] …where 4 is the index of the element we want to remove. In our case, this would be 10.5. Now, checking list1 again, we see that 10.5 is no longer a member of list1.

list1

['Cisco', 'Juniper', 'HP', 10, -11, 100]

Another way to remove an element by its index is using the pop() method like this:

list1.pop(0) …will remove the first element in the list, right?

'Cisco' …now, if we check list1 again, we can spot the absence of "Cisco".

list1

['Juniper', 'HP', 10, -11, 100]

The third way is to actually remove an element by specifying the element itself, using the remove() method.

list1.remove("HP")

list1

['Juniper', 10, -11, 100] …and the element is now gone. Great!

Now, let's see how we can insert an element at a particular index in the list. This is easily accomplished by using the insert() method.

list1.insert(2, "Nortel") …where 2 is the index of the location where we want this new element to be inserted, and "Nortel" is the new element. Checking list1 now:

list1

['Juniper', 10, 'Nortel', -11, 100]

Another interesting list operation is appending a list to another list.

So, let's say we have:

list2 = [9, 99, 999]

To add the elements of list2 to list1, we can use the extend() method. Let's see both lists before that.

list1

['Juniper', 10, 'Nortel', -11, 100]

list2

[9, 99, 999]

And now, we can just extend list1 with the content of list2 like this:

list1.extend(list2)

And here's our new list, list1:

list1

['Juniper', 10, 'Nortel', -11, 100, 9, 99, 999]

Now, remember the index() and count() functions from strings? Python makes them available for lists, as well. So, let's find out the index of an element in our list and how to count the occurrences of an element in the list.

list1

['Juniper', 10, 'Nortel', -11, 100, 9, 99, 999]

list1.index(-11) …returns the index of -11 in the list.

3

Now, let's append the value 10, thus having this value twice in the list.

list1.append(10)

list1

['Juniper', 10, 'Nortel', -11, 100, 9, 99, 999, 10]

list1.count(10)

2 …and, indeed, we have 2 returned, meaning that the element 10 appears twice inside our list, which is correct.

Now, let's have a look at ways of sorting the elements in a list.

First, we can use the sort() method.

So, returning to list2, let's add a couple of elements first.

list2.append(1)

list2.append(25)

list2.append(500)

list2

[9, 99, 999, 1, 25, 500]

Ok. Now, let's say we want to have them sorted in ascending order. We will simply apply the sort() method on list 2.

list2.sort()

list2

[1, 9, 25, 99, 500, 999] …and now we have the elements of list2 sorted in ascending order. Great!

What if we want the elements sorted in reverse or descending order? Well, we have the reverse() method for this. By the way, as you might have noticed already, the names of these methods are pretty intuitive and straightforward and that's one of the reasons Python is such a beginner-friendly programming language. Now, back to our list, let's apply the reverse() method and check the results.

list2.reverse()

list2

[999, 500, 99, 25, 9, 1] …awesome, just what we were looking to achieve.

The two methods you've just seen are modifying the list **in place**, meaning that after you apply the method there is no other list created; you have the same list2, only that the elements are displayed in a specific order.

To sort the elements of a list and create a new list in memory at the same time you have the sorted() function available. Let's see it in action now; let's sort the elements of list2 in ascending order again.

sorted(list2)

[1, 9, 25, 99, 500, 999]

Ok. Now, if you want to use the same function to reverse the order, just add an argument inside the parentheses. This argument is called **reverse** and it must have the value of True assigned to it. The argument is passed to the sorted() function right inside its parentheses, after the name of our list. So:

sorted(list2, reverse = True)

[999, 500, 99, 25, 9, 1] …and the result is correct.

By the way, you may ask me: where did the **reverse = True** argument came from? Well, I knew it because I used it a couple of times, but, if you want to check the syntax or available arguments for a function, you should use the help() command inside the Python interpreter, passing the name of the function you want to know more about, as an argument.

help(sorted)

Two more things worth mentioning here. You can concatenate or repeat a list, as you did with strings, using the plus and multiplication operators, so:

list1 + list2 …adds the elements of both lists together, creating a single list, also called concatenation.

[10, 999, 99, 9, 100, -11, 'Nortel', 10, 'Juniper', 1, 9, 25, 99, 500, 999]

list2 * 3 …repeats list2 as many times as we want, in this case three times, in order to create a larger list.

[1, 9, 25, 99, 500, 999, 1, 9, 25, 99, 500, 999, 1, 9, 25, 99, 500, 999]

Notice here that duplicate elements are allowed, so don't worry about that. Next, we are going to discuss list slicing.

Python 3 Lists – Slices

Remember, list slices allow us to extract various parts of a sequence. We already used them to slice strings and, in many ways, all the rules applying to string slicing also apply to list slicing.

The general syntax is:

var1[5: 10]

The name of the variable pointing to that list, followed by square brackets; next, in between the brackets, we have a colon; on the left side of the colon, we can specify the index at which to start the slicing; the slice will go up to, **but not including**, the index specified on the right side of the colon.

Having that said, let's create a new list and do a couple of list slices:

list3 = [1, 2, 3, "a", "b", "c"]

list3

[1, 2, 3, 'a', 'b', 'c']

What if we want to extract the first three elements of list3? These would be the elements positioned at indexes 0, 1 and 2, right? So, the slice would start at index 0 and go up to, but not including, index 3.

list3[0:3]

[1, 2, 3]

Another way to extract the first 3 elements would be - remember this from strings? - …not specifying the first index in between the brackets, so:

list3[:3]

[1, 2, 3]

What about extracting a slice containing the following elements: 3, a and "b"? This means we'll have to start at index 2, corresponding to element 3 and go

up to, but not including index 5, which is the index of element "c", right? Let's verify this.

list3[2:5]

[3, 'a', 'b']

Now, let's extract all the elements of the list, starting with 3 and up to the end of the list. In this case, there's no need to enter the second index in between the brackets, so:

list3[2:]

[3,'a', 'b', 'c']

In order to select the entire list using a list slice, do not include any indexes inside the brackets, just the colon, like we did for strings.

list3[:]

[1, 2, 3, 'a', 'b', 'c']

Now, remember we can also use negative indexes to slice a sequence and negative means we are counting indexes from right to left, starting at the end of the string, so:

list3[-1]

'c'

list3[-2]

'b' …and both results are correct. Great!

Now, to extract a slice containing the elements 3, "a" and "b" using negative indexes, we would have to start the slice at "3" 's negative index, which is -4, and go up to, but not include "c" 's index, which is -1.

Let's see this in practice.

list3[-4:-1]

[3, 'a', 'b'] …which is just what we expected. Sweet!

Now, what about extracting the last three elements of list3, using negative indexes? We should start at … what index?

Well, index -3, right? Because index -3 represents the third element when counting from right to left, starting at the end of the string. What would be the second index, then? You guessed it - there isn't one. We go all the way to the end of the list and that's it, so:

list3[-3:]

['a', 'b', 'c']

What about a slice containing all the elements of the list, minus the last three of them? Well, we just have to move index -3 on the right side of the colon and that should do it. Let's check this.

list3[:-3]

[1, 2, 3]

That's cool!

The last two things I want to mention about list slices - first, using a step for the slicing process; like, for example, if you want to skip every second element of the list, then you will use two colons inside the square brackets and then the step you want to consider, in our case this is 2, so:

list3[::2]

[1, 3, 'b']

Secondly, if you want to see the list in reversed order using slices, you can also use a step. This time, its value will be -1, because you want to start with the last element in the list, right? So:

list3[::-1]

['c', 'b', 'a', 3, 2, 1]

https://docs.python.org/3/tutorial/datastructures.html

Python 3 Sets – Introduction

What is a set, you may ask yourself? Well, a set is basically an **unordered** collection of **unique** elements. Generally speaking, you may regard sets as being lists that have no duplicate elements.

Let's see the way to create a set. In fact, there are two ways.

The first one is by using the set() function, which is a Python built-in function.

To also prove that sets do not allow duplicates, let's create a list with duplicate elements and apply the set() function on this list.

list4 = [1, 2, 3, 4, 5, 2, 3]

list4

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

set(list4)

{1, 2, 3, 4, 5}

So, you see that the set() function removed the duplicate elements in list4, which is a very useful feature to have at hand.

You can also directly create a set by passing a raw sequence to the set() function, like a string or list, and referencing the result using a variable.

set1 = set([11, 12, 13, 14, 15, 15, 15, 11])

set1

{11, 12, 13, 14, 15}

type(set1)

<class 'set'>

The second way to create a set is to use curly braces. This method of creating a set is available in versions of Python starting with 2.7, so ours is included as well, according to **python.org**.

set2 = {11, 12, 13, 14, 15, 15, 15, 11}

set2

{11, 12, 13, 14, 15}

type(set2)

<class 'set'>

We can also find out the number of elements in a set, using the same len() function, as we did with strings and lists.

len(set2)

5

Next, checking whether an element is or is not a member of a set is also possible using the **in** and **not in** keywords:

11 in set2

True

10 in set2

False

10 not in set2

True

Going further, we have to remember that sets are mutable, so we can add or remove elements from a set, in the following manner:

set2

{11, 12, 13, 14, 15}

set2.add(16)

set2

{11, 12, 13, 14, 15, 16}

set2.remove(11)

set2

{12, 13, 14, 15, 16}

Notice that if I try to add an element which already exists in the set, Python will not agree with me, although no error is returned. It just doesn't add it.

set2

{12, 13, 14, 15, 16}

set2.add(16)

set2

{12, 13, 14, 15, 16}

Now, let's see some operations and methods that can be applied on sets.

Python 3 Sets - Methods

To better understand set methods and operations, let's create two sets first.

set1 = {1, 2, 3, 4}

set2 = {3, 5, 8}

Python defines some methods for identifying the similarities or differences between two sets of elements, but also other methods to better make use of this data type. Let's see them in action.

First, let's see how we can identify the common elements of the two sets we defined.

To do this, we can use the intersection() method, like this:

set1.intersection(set2)

{3}

And this is correct, of course, because 3 is the only element which resides in both sets. This operation can be done the other way around, too.

set2.intersection(set1)

{3}

Now, let's see which elements does set1 have and set2 doesn't, by using the difference() method. Looking at the two sets, we notice that set1 has the elements 1, 2 and 4, but set2 does not, so this should be the result, right?

set1.difference(set2)

{4, 1, 2} …great!

Now let's check this vice-versa, as well.

set2.difference(set1)

{8, 5} …also correct, since 8 and 5 are not members of set1.

To unify the two sets, you can use the union() method and the result, also being a set, a collection of **unique** elements, will include element 3 just once. So, do not confuse the union of two sets with concatenation.

set1.union(set2)

{5, 2, 3, 8, 4, 1}

Another thing you can do is remove a random element in the set using the pop() method.

set1

{1, 2, 3, 4}

set1.pop()

1

set1

{2, 3, 4}

So, element 1 has been removed and also displayed by the pop() method.

Finally, we can clear a set using the clear() method.

set1.clear()

set1

set() …and you can see now that set1 is just an empty set. Great!

There are other methods for working with sets in Python, but the chances you will need them are not that high, so we won't discuss them in this course.

Just in case, you can find a link to Python's official documentation regarding sets, attached to this lecture.

https://docs.python.org/3/library/stdtypes.html#set

Python 3 Sets - Frozensets

We've discussed sets and the corresponding set methods, but what is a frozenset? Well, remember I said that sets are mutable, meaning they can be modified by adding or removing an element, as we've seen earlier?

Frozensets are nothing more than immutable sets. The elements of a frozenset remain the same after creation. Due to this feature, frozensets can be used as elements of another set or as a key in a dictionary (and, don't worry if you don't know what a Python dictionary is, yet! We will discuss dictionaries pretty soon, in the course).

To prove the immutability of frozensets, let's create one and see what we are allowed to do with it and what is forbidden.

Let's consider list1 and list2, first.

list1 = [1, 2, 3, 4]

list2 = [3, 4, 7]

Now, let's create our frozensets, by passing these lists to the frozenset() function.

fs1 = frozenset(list1)

fs2 = frozenset(list2)

Finally, let's check if we have indeed created two frozensets.

fs1

frozenset({1, 2, 3, 4})

type(fs1)

<class 'frozenset'>

fs2

frozenset({3, 4, 7})

type(fs2)

<class 'frozenset'>

Ok, frozensets - confirmed.

Now, let's try to add or remove an element from fs1, for instance.

fs1.add(10)

AttributeError: 'frozenset' object has no attribute 'add'

fs1.remove(1)

AttributeError: **'frozenset' object has no attribute 'remove'**

fs1.pop()

AttributeError: 'frozenset' object has no attribute 'pop'

fs1.clear()

AttributeError: 'frozenset' object has no attribute 'clear'

fs1

frozenset({1, 2, 3, 4})

Notice that each time we try to add or remove an element from this frozenset, Python tells us that each of the attributes we were using with sets are no longer available with frozensets, thus, the contents of fs1 remain the same.

Of course, operations like difference, intersection and union are still available, since they're not attempting to modify the content or structure of frozensets. Let's check this real quick.

fs1.intersection(fs2)

frozenset({3, 4})

fs1.difference(fs2)

frozenset({1, 2})

fs1.union(fs2)

frozenset({1, 2, 3, 4, 7})

Ok, it works. That's about it, regarding sets and frozensets, two data types that will definitely come in handy someday.

https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

Python 3 Tuples - Introduction

Tuples are another type of sequences in Python.

You can consider tuples as being immutable lists, meaning their contents cannot be changed by adding, removing or replacing elements.

Tuples may prove to be useful when you want to store some data in the form of a sequence and keep that data untouchable. However, unlike sets, tuples are **ordered** collections of **non-unique** elements, meaning indexes and duplicates are allowed.

Ok, enough with the theory, let's start to practice and create our first tuple.

Tuples are enclosed by parentheses and their elements are separated by commas.

```
my_tuple = ()

type(my_tuple)
```

<class 'tuple'> …this is an empty tuple and Python confirms this, too.

If you want to create a tuple with a single element you have to use a trick, meaning that, although you have only one element inside the tuple, you have to write a comma after it; otherwise, it will not be regarded as a tuple. Let's see this in practice.

```
my_tuple = (9)

type(my_tuple)
```

<class 'int'>

```
my_tuple = (9,)

type(my_tuple)
```

<class 'tuple'>

Now you have a tuple set up. You should remember this when creating tuples having only one element.

Next, let's populate our tuple with more elements.

my_tuple = (1, 2, 3, 4, 5)

my_tuple

(1, 2, 3, 4, 5)

Just like strings and lists, tuples support indexing, so if you want to access an element within the tuple, the indexing rules we've seen before are still applicable.

my_tuple[0]

1

my_tuple[1]

2

my_tuple[-1]

5

my_tuple[-2]

4

Since tuples are immutable, you cannot add or modify an element of a tuple. Let's check this.

my_tuple[1] = 10

TypeError: 'tuple' object does not support item assignment

Also, removing elements is not permitted when working with tuples.

del my_tuple[1]

TypeError: 'tuple' object doesn't support item deletion

Another interesting thing you can do with tuples is tuple assignment. This means you assign a tuple of variables to a tuple of values and map each variable in the first tuple to the corresponding value in the second tuple.

Let's see this, as well. We will define tuple1 with the following elements.

tuple1 = ("Cisco", "2600", "12.4")

And now, let's assign a tuple of variables to tuple1.

(vendor, model, ios) = tuple1

Finally, let's check if the assignment and variable-to-value mapping have been properly performed.

vendor

'Cisco'

model

'2600'

ios

'12.4'

This is also called tuple packing and unpacking and you can see it as a kind of mapping between elements of two different tuples.

An important thing to remember here is that both tuples should have the same number of elements. Otherwise, if you have different number of elements, a ValueError will be returned.

tuple2 = (1, 2, 3, 4)

(x, y, z) = tuple2

ValueError: too many values to unpack (expected 3)

So, you see that tuple2 has 4 elements, but the tuple we're trying to map it with has only 3 elements. This will generate this error message right here.

You can also assign values in a tuple to variables in another tuple within a single statement, which is even more convenient.

(a, b, c) = (10, 20, 30)

a

10

b

20

c

30

Ok, the result is correct, the mapping has been done successfully.

Next, let's see some operations and methods for working with tuples.

Python 3 Tuples - Methods

As with strings and lists, we can perform some basic operations on tuples, too.

So, we can use the len() function to find out the number of elements of a tuple.

tuple2

(1, 2, 3, 4)

len(tuple2)

4

Also, we have the min() and max() functions available for finding the lowest and greatest value inside a tuple.

min(tuple2)

1

max(tuple2)

4

We can also concatenate and multiply a tuple, using the same old plus and multiplication operators.

tuple2

(1, 2, 3, 4)

tuple2 + (5, 6, 7)

(1, 2, 3, 4, 5, 6, 7)

tuple2 * 2

(1, 2, 3, 4, 1, 2, 3, 4)

Since indexing applies to tuples as it does to strings and lists, slicing is also possible with tuples.

Let's see a couple of examples, without entering into the details about slicing again, since the rules are basically identical.

tuple2[0:2]

(1, 2)

tuple2[:2]

(1, 2)

tuple2[1:]

(2, 3, 4)

Now, let's have the entire string returned.

tuple2[:]

(1, 2, 3, 4)

Next, let's also use negative indexes.

tuple2[:-2]

(1, 2)

tuple2[-2:]

(3, 4)

And, finally, let's wrap up tuple indexing up by inserting a step for our slices.

tuple2[::-1]

(4, 3, 2, 1)

tuple2[::2]

(1, 3)

Another thing you can do with tuples is you can check if an element is a member of a tuple or not, using **in** and **not in**. Let's see this.

tuple2

(1, 2, 3, 4)

3 in tuple2

True

3 not in tuple2

False

5 in tuple2

False

Last thing on tuples - we can use the **del** command to delete the entire tuple.

del tuple2

tuple2

NameError: name 'tuple2' is not defined

Tuples will be very useful in your Python programming adventure; maybe you won't use them as much as lists, but you should definitely keep them in mind.

https://docs.python.org/3.3/tutorial/datastructures.html#tuples-and-sequences

Python 3 Ranges - Introduction

In this lecture and the next one, we will have a brief look over the **range** data type and, to make things even more interesting, we will start with a quick comparison with the meaning of **range** in the previous major version of Python, namely Python 2.

For this, I will open up codeskupltor.org, which is an online Python 2 interpreter and I will use the range() function, by passing the value 10 as an argument. In Python 2, what range() does is it generates a list of integers, starting at 0 and going up to, but **not** including, the value 10.

```
print range(10)
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

As you can see, the range() function has a default starting point at 0 and a default step of 1, since we get these consecutive integers returned.

But, don't worry, these can be customized, as well. For example, let's start our list at 5, instead of 0. To do this, we just have to enter a new argument, also called **start**, in between parentheses.

```
print range(5, 10)
```

[5, 6, 7, 8, 9]

Next, let's assume that we don't want consecutive numbers being generated; instead, we want to also have a **step** of 2. All you have to do is to enter the step you need, as the third argument inside parentheses.

```
print range(0, 10, 2)
```

[0, 2, 4, 6, 8]

Finally, it's worth mentioning that we can also use negative values for the start, stop and step arguments.

```
print range(-2, -10, -2)
```

[-2, -4, -6, -8] …and the result was, I think, pretty easy to guess.

Furthermore, in Python version 2, we also had the **xrange()** function available. When given an argument, let's say 10, both range() and xrange() ultimately return the same values, 0 to 9. However, the difference laid in what data type does each of these functions return. The range() function returned a list, as we've just seen. On the other hand, the xrange() function returned an iterator. We will talk more about iterators in another chapter of this course, but, for now, just keep in mind that the range() function generates all the elements from 0 to 9 as soon as it is executed, whilst xrange() produces an iterator, that pops out each value, one at a time, only when we tell it to do so.

Why is that so important? Well, forget range(10) and imagine you have **range(1000000000000)**. That would generate a gigantic list, starting at 0 and up to this huge value, am I right? More elements means more memory being used and, usually, you want to keep your programs as optimal as possible.

That's why, sometimes, the xrange() function saves the day. It generates the same numbers, but only when we iterate over the object it generates.

In conclusion, you should keep in mind that all this was valid in Python version 2 only. In the meantime, Python 3 renamed the xrange() function to range() and the original range() function was deprecated. But, more on that, in the next video.

Python 3 Ranges - Methods

Ok, following up on the previous lecture, let's get into the Python 3 interpreter and see how does range() behave in Python version 3.

Let's consider a basic implementation of **range**.

r = range(10)

r

range(0, 10)

type(r)

<class 'range'>

You can easily notice that, unlike in Python version 2, where we would've got a list returned, in Python 3 **range** gets its own data type, *class 'range'*.

However, if you want a list instead of a range object, you can simply apply the list() function to this range.

list(r)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

What else can you do with this range?

You can use indexes, as we did with strings or lists, to extract various elements from this range. For instance:

r[0]

0

r[-1]

9

You can also verify if a certain value is a member of the range, by using the **in** and **not in** keywords.

10 in r

False

7 not in r

False

7 in r

True

Moreover, you can apply, for instance, the index() method to find out the index of a certain element of the range.

r.index(4)

4

Of course, the index() method is not that useful when working on a basic range, starting at 0 and advancing with a step of 1, since the index of any element is equal to the value of the element itself. However, when having more complex ranges, this method can prove to be pretty useful.

Finally, keep in mind that you cannot slice a range, but what you can do is you can convert the range to a list, using the list() function, and then apply your slice. For instance:

list(r)[2:5]

[2, 3, 4] …and there's your slice. It's a simple trick, but it works.

Again, don't worry if you don't yet understand what iterators are all about. We will talk more about iterators later in the course, after we cover **for** loops and other several useful concepts, to have a better understanding of this concept, overall.

https://docs.python.org/3/library/stdtypes.html#ranges

Python 3 Dictionaries – Introduction

Ok, let's study another very important data type, that you'll need for your Python adventure - dictionaries.

A dictionary is an **unordered** set of **key-value pairs**, separated by comma and enclosed by curly braces. They are very useful for storing information in a specific format. For instance, considering a router, we can store data about the device in the following format: **Vendor: Cisco, Model: 2600, IOS: 12.4, Ports: 4**.

Dictionaries are mutable, which means we can modify their contents using dictionary-specific procedures. Why do I say dictionary-specific? Because, unlike strings or lists, dictionaries are **not indexed by the position of each element**, like we previously had 0 for the first element, 1 for the next and so on.

Dictionaries are **indexed by key**. The key is the value on the left side of the colon of each key-value pair. We will see this in practice, don't worry.

For now, let's just create an empty dictionary.

```
dict1 = { }
```

```
dict1
```

{ }

```
type(dict1)
```

<class 'dict'>

This is how you create a dictionary; now, let's add some data to it.

```
dict1 = {"Vendor": "Cisco", "Model": "2600", "IOS": "12.4", "Ports": "4"}
```

```
dict1
```

{'IOS': '12.4', 'Model': '2600', 'Vendor': 'Cisco', 'Ports': '4'}

Each dictionary element consists of a key, a colon and a value, followed by comma.

Now, let's notice a few things here.

First, because the keys in our dictionary are actually strings, each key is enclosed by quotes. This may be the most widely spread data type used for a dictionary key. You may also use a number as a key, in order to have some kind of a numbering system, like this:

```
d1 = {1: "First element", 2: "Second element"}
```

```
d1
```

{1: 'First element', 2: 'Second element'}

Ok, let's get back to our **dict1** dictionary.

```
dict1
```

{'IOS': '12.4', 'Model': '2600', 'Vendor': 'Cisco', 'Ports': '4'}

Another thing to notice here is that although when we created the dictionary we entered the **"Vendor": "Cisco"** pair first, then the Model key-value pair, the IOS pair and lastly the Ports pair, when we're printing the dictionary, they are ordered differently, randomly. This is what I meant when I said that a dictionary is **an unordered set of key-value pairs**. Keep this in mind!

Don't worry, dictionaries are not so chaotic, though. Python helps us keep things smooth and clean when using dictionaries. We will see how it does that, in the next lecture.

A key thing to remember is that **each key in the dictionary must be unique and** should be of an **immutable** type; this means you can have strings, numbers or tuples as keys, but **not** lists.

On the other hand, **values don't have to be unique** and can be either of a mutable or an immutable data type. See you in the next lecture!

Python 3 Dictionaries – Methods

Let's return to the dict1 dictionary from the previous lecture and see how we can work with it.

dict1

{'IOS': '12.4', 'Model': '2600', 'Vendor': 'Cisco', 'Ports': '4'}

First, let's extract the corresponding value for a specified key. This can be done similarly to accessing elements inside a list, only that we don't specify an index, we specify the associated key for the value we want to extract. So:

dict1["IOS"]

'12.4'

dict1["Vendor"]

'Cisco' …ok, it works!

In the previous lecture, I said that dictionaries are mutable. So, let's try to add a new key-value pair to our dictionary. This is done by simply assigning a new value to the new key.

dict1["RAM"] = "128"

dict1

{'IOS': '12.4', 'Model': '2600', 'RAM': '128', 'Vendor': 'Cisco', 'Ports': '4'}

Notice that the pair we just added is not appended to the end of the dictionary, because dictionaries are **unordered**, remember? Ok.

Now, maybe we want to modify the value for the IOS data of this device. Don't worry, it's pretty simple; just use the same syntax as we did for adding a new pair.

dict1["IOS"] = "12.3"

dict1

{'IOS': '12.3', 'Model': '2600', 'RAM': '128', 'Vendor': 'Cisco', 'Ports': '4'}

We can also delete a pair from the dictionary, using the **del** command.

del dict1["Ports"]

dict1

{'IOS': '12.3', 'Model': '2600', 'RAM': '128', 'Vendor': 'Cisco'}

Now, the Ports pair is gone. Sweet!

Next, remember the **len()** function from strings, lists and tuples? We can use it here, as well, to find out the number of key-value pairs inside a dictionary.

dict1

{'IOS': '12.3', 'Model': '2600', 'RAM': '128', 'Vendor': 'Cisco'}

len(dict1)

4

Of course, you can verify if a certain string is a key in a dictionary or not, like this:

"IOS" in dict1

True

"IOS2" in dict1

False

"IOS2" not in dict1

True

Now, there are three important Python methods to deal with keys and values, in a dictionary.

The first one is the **keys()** method. This method is used to obtain a list having the keys in a dictionary as elements. A similar method is **values()** to get a list having the values in a dictionary as elements.

dict1.keys()

['IOS', 'Model', 'RAM', 'Vendor']

dict1.values()

['12.3', '2600', '128', 'Cisco']

The third main method here is the items() method, which returns a **list of tuples**, each tuple containing the key and value of each dictionary pair. Let's check this out.

dict1.items()

[('IOS', '12.3'), ('Model', '2600'), ('RAM', '128'), ('Vendor', 'Cisco')]

Now, we should expand a bit on the order of elements in a dictionary. Even though a dictionary stores the key-value pairs in a different order than the order you specified them in initially, when using the keys(), values() and items() methods, Python keeps track of the order in which it stored the pairs and displays the keys, values or tuples in the same order, thus keeping a level of consistency for later reference inside the code. Let me show you what I mean by this.

So, let's check dict1 again to see the order of the pairs inside the dictionary.

dict1

{'IOS': '12.3', 'Model': '2600', 'RAM': '128', 'Vendor': 'Cisco'}

Now, let's check the order of the elements in the lists generated by the three dictionary-specific methods.

dict1.keys()

['IOS', 'Model', 'RAM', 'Vendor']

dict1.values()

['12.3', '2600', '128', 'Cisco']

dict1.items()

[('IOS', '12.3'), ('Model', '2600'), ('RAM', '128'), ('Vendor', 'Cisco')]

So, you can see that Python helps us a bit when dealing with dictionaries by keeping an internal indexing system when using dictionary methods.

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

Python 3 - Conversions Between Data Types

The last thing I think we should cover on the data types topic is: conversions between data types.

This means that you will learn how to convert from one data type, a number for example, to another data type, like a string. There are specific functions that accomplish these tasks. Let's see some of them in action.

First, let's try to convert an integer or floating-point number to a string. This can be achieved by using the str() function.

```
num = 2

f = 2.5

type(num)
```

<class 'int'>

```
type(f)
```

<class 'float'>

Converting a number to a string.

```
num2 = str(num)

type(num2)
```

<class 'str'>

```
f2 = str(f)

type(f2)
```

<class 'str'>

Now, let's try this backwards and convert a string to an integer, using the int() function.

```
str1 = "5"
```

type(str1)

<class 'str'>

int1 = int(str1)

type(int1)

<class 'int'>

You can also convert integers to floating-point numbers, using the float() function, and vice-versa, using the same int() function we've just seen.

num2

2

type(num2)

<class 'int'>

f = float(num2)

type(f)

<class 'float'>

Now, the other way around, from float to integer, using int().

f

2.5

int1 = int(flt)

type(int1)

<class 'int'>

By the way, notice that applying the int() function on a float will return the integer part of that number.

Now, moving to sequences, let's convert a tuple into a list, using the list() function.

tup1 = (1,2,3)

type(tup1)

<class 'tuple'>

list1 = list(tup1)

type(list1)

<class 'list'>

We can also convert a list into a tuple, using the tuple() function.

list1 = [1, 2, 3]

tup = tuple(list1)

type(tup)

<class 'tuple'>

We have also seen how the set() function works for turning a list into a set.

list1

[1, 2, 3]

set1 = set(list1)

type(set1)

<class 'set'>

The last thing I'll show you here is how to convert between different numerical representations of numbers and I am referring to decimal, binary and hex notations, so base-10, base-2 and base-16 numbers. For this, we will need the bin(), hex() and int() functions.

num = 10

num_bin = bin(num)

num_bin

'0b1010'

num_hex = hex(num)

num_hex

'0xa'

Now, to convert from binary and hex format back to decimal notation, we will use the int() function.

bin_to_num = int(num_bin, 2)

bin_to_num

10

hex_to_num = int(num_hex, 16)

hex_to_num

10

Keep in mind that when converting from binary and hex back to decimal, you need to specify 2 and 16, respectively, as arguments of the int() function, where 2 and 16 are the bases we are converting from, base-2 and base-16.

That's about all you need to know about data types for your Python programming adventure. Of course, there are many other functions and methods we can discuss, hundreds, maybe thousands, but you will rarely use them in your day-to-day programs or even not use them at all.

I hope you enjoyed this section and I'll see you in the next one!

Python 3 Conditionals - If / Elif / Else

In Python, we have the **if, elif, else** statements for decision making.

Using these statements, Python evaluates expressions and runs a piece of code accordingly, meaning if an expression is evaluated as True, then the code indented below the **if** statement will be executed. Otherwise, Python goes further and evaluates the **elif** or **else** statements, if any.

Unlike many other programming languages that use curly braces or other delimiters, Python uses indentation to define code blocks, meaning **if, for** and **while** blocks, functions and classes. Using indentation means that whitespaces are used as delimiters for code blocks. And when I say whitespaces, I am referring to Space characters (usually, 4 of them in a row) or the Tab key.

We are going to see that from now on in this course.

Just remember that you should stick with a convention for using indentation and not mix Spaces and Tabs within your code. We are going to use the Tab key each time we want to indent some piece of code. I think this is the safest way to do it.

Another thing to remember is that after every if / for / while statement or function or class definition, you must use a colon, so that Python will know that it should expect an indented block right after that statement. Don't worry, you will easily get used to Python's way of writing code.

Now, let's start working with **if**, **elif** and **else** statements.

First, we shall create a Python script. Let's use Notepad++ and write our code inside this file. For now, let's just save the empty file on the D: drive and call it **test1**, then pick the appropriate extension, **.py** and that's it. You can choose a location of your preference for the file; I will use my D: drive for this course.

Should you want to edit the default font settings in Notepad++, maybe increase the font size, for instance, just go to Settings - Style Configurator

and make your changes in the Font Style section. Then just hit Save and you're done.

Now, let's say we define a variable **x = 10**. And we want to make a decision, based on that variable's value. Maybe the value of **x** will change at some point during the execution of the program and we want to handle that change in some particular way and run a piece of code. Let's use the **if** statement to execute a block of code, if the expression we provide will be evaluated as True.

```
x = 10

if x > 5:

    print("x is greater than 5")
```

Now, let's stop for a moment and analyze this piece of code. First, I typed in the **if** keyword, and then the expression I want to evaluate, x > 5, followed by a colon.

Next, you can see that Notepad++ knows that we're writing Python code and recognizes Python-specific keywords, like, for example, **if**. After I finished writing my **if** statement, on the next line I had to hit the Tab key once, in order to have the next lines of code properly indented under the **if** statement. The print() function and any other code we're going to indent under the **if** statement will get executed **only** if x > 5 is evaluated as True.

As long as we don't decrease the indent manually, the next lines of code will also get automatically indented under the **if** clause. For instance, let's add another line of code under the first print() function, **print(x * 2)**, just to make this clear.

Now, let's save the script and try to guess the result. We said x was equal to 10 and then, if **x** is greater than 5, we should have **this** string and x * 2 printed out on the screen. Since 10 is greater than 5, then the x > 5 expression is evaluated as True and the print() functions get executed and we should obtain the string and the value of 20 as a result.

Let's test this by simply opening the Windows command line and running our script.

```
python D:\test1.py
```

x is greater than 5

20

Good! Now, what if **x** would have been equal to 4? Let's test this.

```
x = 4

if x > 5:

    print("x is greater than 5")

    print(x * 2)
```

Now, in the Windows cmd:

```
python D:\test1.py
```

In this case, our **if** block does not return anything, since 4 is not greater than 5 and the expression is evaluated as False.

So, bottom line, if Python evaluates the expression as True, it will execute the indented code below the **if** statement, otherwise it will skip it and move on to the rest of the code, if any.

Now, what if you want to handle multiple cases? Let's say you want something to be printed out, regardless of the True or False result of the evaluation. For this, you will use the **elif** or **else** statements.

First, let's see how to use **else** and return to our variable **x** which is still equal to 4.

Let's print *"x is greater than 5"* if **x** is indeed greater than 5, and *"x is NOT greater than 5"* in any other case. This can be accomplished in the following way:

```
x = 4
```

```
if x > 5:
    print("x is greater than 5")
else:
    print("x is NOT greater than 5")
```

The result, in the Windows command line:

X is NOT greater than 5

So, to use the **else** statement, you must hit Enter after *"print x is greater than 5"*, then return to the beginning of the line using Backspace 4 times, and then type in the **else** keyword, followed by a colon and hit Enter again. Now, use the Tab key one more time, to get to the next level of indentation and write the code to be executed in case x is not greater than 5. For every block of code that we want to execute, we must use indentation.

By the way, there are options to perform indentation automatically, including some Notepad++ plug-ins, I think. However, I strongly recommend indenting your code manually, to have complete control over your program. Some editors or plug-ins use 4 spaces for indentation, others use Tabs and issues may occur when porting, editing and running your code on various systems. That's why using manual indentation is the best choice, in my opinion.

Now, back to our code.

The **else** statement is used to cover all the other cases not covered by the **if** statement above it. So, if the expression following the **if** keyword is True, the indented code below it will be executed. Otherwise, if the expression is evaluated as False, the indented code below **else** gets executed.

But, what if we want to be more granular and specify more cases and possible outputs in our program? Then we could use an **elif** statement.

Let's say we want to print *"x is greater than 5"* if **x** is indeed greater than 5, *"x is equal to 5"* in case **x** will become equal to 5 at some point and *"x is less than 5"* for all other cases. We should then add the **elif** statement in between

if and **else**. The **else** statement should always be the last statement in an if / elif / else block.

Let's now consider **x = 5** and modify our if / else block, by adding an **elif** clause.

```
x = 5
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is 5")
else:
    print("x is NOT greater than 5")
```

The result is, of course:

x is 5

So, the syntax for **elif** is similar to the syntax of the **if** statement, meaning **elif** is also evaluating a particular expression as being True or False. Notice that when evaluating equality between two operators, we must always use two consecutive equal signs, because this is **not** an assignment operation like x = 5 at the beginning of our example.

As a rule, keep in mind that **elif** and **else** statements are completely optional, so, you can have just an **if** statement, evaluate an expression and execute some code or not, without covering other cases.

Also, you can have as many **elif** statements as you want after an **if** statement but remember that there can be only one **else** statement at the end, covering all the other possible cases not being covered by the **if** and **elif** statements. Actually, what Python does is it evaluates these clauses in order. As soon as it finds an expression being evaluated as True, it runs the code that's indented under that clause and skips the evaluation of subsequent **elif** or **else** clauses.

Another thing here is that there may be more than one evaluated expression inside an **if** or **elif** clause. Let's check this.

We have **x** equal to 5 and Python identifies x as an integer, due to this assignment. So, this means that **x == 5** and **type(x) is int** are two expressions that should be evaluated as True. Let's use them inside an **if** block.

To check if both expressions are True inside the same **if** statement, we will use the **and** logical operator we've seen earlier when discussing booleans.

```
x = 5

if x == 5 and type(x) is int:

        print("x equals 5. x is an integer")

        print(x)

elif x == 10 and type(x) is int:

        print("x is an integer, but is not equal to 5")
```

The result, after Python evaluated both expressions in the **if** clause as being True:

x equals 5. x is an integer

5

First, notice that we don't have an **else** statement. That's because it is optional.

Secondly, notice Python printed the string in the **if** block, because both expressions are evaluated as True, which leads to a final value of True. Python executes the indented code below **if** and exits, without checking any other expressions below. If the expression in the **if** statement would have been evaluated as False, then Python would have jumped to the **elif** statement and then maybe to the next one and maybe to a final **else** statement, until it would find a True expression and execute that code over there.

Let's test this by changing the value of x to 10.

```
x = 10
if x == 5 and type(x) is int:
    print("x equals 5. x is an integer")
    print(x)
elif x == 10 and type(x) is int:
    print("x is an integer, but is not equal to 5")
```

x is an integer, but is not equal to 5

In this case, the expression in the **if** statement is evaluated as False, since we have the **and** operator and one of the operands is evaluated as False, *x == 5*.

Because this is False, Python jumps to the next statement. The **elif** clause. This time, the result of evaluating the expression is True, since both *x == 10* and *type(x) is int* are True expressions.

For this reason, Python executes the indented code below the **elif** statement, as expected.

But, what if neither of the expressions in the **if** and **elif** clauses is evaluated as True and we don't have an **else** statement to cover all other cases? Let's see

```
x = 100
if x == 5 and type(x) is int:
    print("x equals 5. x is an integer")
    print(x)
elif x == 10 and type(x) is int:
    print("x is an integer, but is not equal to 5")
```

In this case, Python returns nothing. If we would've had this **if/elif** block inside a larger program, then Python would have jumped to the rest of the code, as if this block was not even present in the code.

So, remember that Python handles **if/elif/else** blocks in order, from top to bottom, until it evaluates an expression as True. As soon as it finds one, it executes the indented code below it and jumps outside the block to the rest of the code, without evaluating other **elif/else** statements.

https://docs.python.org/3/tutorial/controlflow.html

Python 3 Loops - For / For-Else

The **for** statement is used whenever you want to iterate over a sequence and execute a piece of code for all or some elements of that sequence - list or string, or whatever sequence you have.

Let's start with an example of iterating or looping over a sequence and, first, let's define a list inside a new Python file. Let's name this file for.py.

```
vendors = ["Cisco", "HP", "Nortel", "Avaya", "Juniper"]
```

Now let's see how we can work with a **for** loop.

First, you will notice that there are some similarities to the **if/elif/else** syntax, meaning that the colon is again used to signal that an indented code follows the **for** statement and, speaking about indentation, we must indent the code inside a **for** loop using the Tab key, in order to separate it from the code that follows.

We start by typing in the **for** keyword, then we enter **the iterating variable**, which an user-defined temporary variable, so you can name it however you like; then we type in the **in** keyword, to tell Python that we are going to iterate over the sequence following this keyword and, finally, we enter the sequence itself, followed by a colon. So, moving our code to the Python interpreter:

```
for each_vendor in vendors:
    print(each_vendor)
```

Cisco

HP

Nortel

Avaya

Juniper

So, I named my list in such a way that it reflects my specific need, **vendors**; then I told Python that it should assign each element in this list to the iterating variable called **each_vendor** and execute the indented code below for every element, until all the elements in the list are exhausted. So, Python did this and printed each item in the list. Of course, you can have multiple lines of code inside the **for** block, as with the **if/elif/else** statements.

We can also iterate over a string. Let's write the code and then test it in the Python interpreter. We can also run the file in the Windows command line, but let's stick to the interpreter, for now; it looks nicer.

```
my_string = "Cisco"

for letter in my_string:

        print(letter)

        print(letter * 2)

        print(letter * 3)
```

```
C

CC

CCC

i

ii

iii

s

ss

sss

c

cc
```

ccc

o

oo

ooo

So, using a **for** statement, we assign each character in the string to a temporary variable called **letter** and, each time we do this, Python executes the indented code block below the **for** statement. This means that, for each letter in "Cisco", Python prints out the character itself, the same character doubled and then tripled.

Now it's time to see how to use a **for** loop to iterate over a range. Remember we discussed the **range** data type earlier in the course, that can be used to generate an iterator over which we can iterate and then extract some values.

Let's consider a range starting at 0 and going up to, but not including, 10, with the default step of 1. That would return the integers 0 all the way to 9, in ascending order. Let's get into the Python interpreter and create this range.

```
r = range(10)
```

Now, let's use a **for** loop to iterate over this range.

```
for i in r:
    print(i * 2)
```

And, of course, the result is each integer from 0 to 9, times 2.

0

2

4

6

8

10

12

14

16

18

So, what we did is we used a **for** loop to iterate over the range created by the range() function and for each element in the range we printed out its value multiplied by 2. Nice!

Now, let's see a more common use of the range() function inside a **for** statement. What if we want to iterate over a list using list indexes? What do I mean by that? We still have the **vendors** list in memory, so:

vendors

['Cisco', 'HP', 'Nortel', 'Avaya', 'Juniper']

Now, remember the len() function from earlier? Let's apply it to our list.

len(vendors)

5

We know that range(5) returns the integers starting with 0 up to, but not including 5, right? Moreover, we can convert this range to a list, using the list() function. Let's do this.

list(range(5))

[0, 1, 2, 3, 4]

We can look at the elements of this list as being the indexes of each element of our list, **vendors**. So, the element "Cisco" would be positioned at index 0, "HP" at index 1 and so on.

This means that if we want to get a list of indexes to iterate over, using a **for** loop, we can use **range(len(vendors))** to obtain that list. I will explain this

shortly. For now, to use this in practice, let's create a **for** loop that prints out each element of the **vendors** list by its index.

```
for element_index in range(len(vendors)):
    print(vendors[element_index])
```

Cisco

HP

Nortel

Avaya

Juniper

So, what we did here is we passed the result of one function, len(), to another function, range().

The result is a range, consisting of all the indexes in the **vendors** list. We then assigned each index in this range to the **element_index** temporary variable and executed a piece of code for each index.

In translation, we told Python to check the length of the **vendors** list, then create a range using that length as an argument for the range() function. Finally, Python prints out each element, by its index: vendors[0], vendors[1] and so on, until the list is exhausted.

Another very useful way to iterate over a sequence is by using **both the index and the element value** as iterating variables. This can be achieved using the **enumerate()** function in Python.

```
for index, element in enumerate(vendors):
    print(index, element)
```

0 Cisco

1 HP

2 Nortel

3 Avaya

4 Juniper

So, using this method, Python returns both the index and the element value at the same time.

The last thing I'll add here is that you can also use an **else** statement with a **for** loop. The indented code below **else** will be executed only when the **for** loop has finished iterating over the entire sequence. Let's write this code and then paste it into the Python interpreter.

```
for element in vendors:
    print(element)
else:
    print("The end of the list has been reached")
```

Cisco

HP

Nortel

Avaya

Juniper

The end of the list has been reached

That's about all you have to know and practice when working with **for** loops. You will use them widely across your applications, so you should have a very good understanding of them, before moving to more advanced Python concepts.

https://wiki.python.org/moin/ForLoop

Python 3 Loops - While / While-Else

The second type of Python loops is **while**. But, what is the difference between **for** and **while** loops?

Well, unlike a **for** loop, which executes a code block a number of times, depending on the sequence it iterates over, a **while** loop executes a piece of code as long as a user-defined condition is evaluated as True. If the specified condition does not change, meaning it doesn't become False, then the **while** loop will continue running forever and we end up with an infinite loop.

When the condition becomes False, Python continues to execute the code following the **while** loop, if any.

Now, let's see an example of a **while** loop. For this, we will create a new file, called while.py. We will run the code inside this file by simply pasting it into the Python interpreter.

First, we should create a variable, **x**, with the value of 1.

```
x = 1
```

To create a **while** loop, you have to type in the **while** keyword, followed by the condition you want to evaluate and then a colon. Below, using the Tab key for indentation, you will specify the code to be executed as long as the condition is evaluated as True. So:

```
while x <= 10:
    print(x)
    x = x + 1
```

…you can also write this as **x += 1**, which is an equivalent expression.

The final result is this:

1

2

3

4

5

6

7

8

9

10

So, let me explain what just happened. Python takes the first value of **x**, which is 1, evaluates it against the x <= 10 condition as True, prints **x** to the screen and then increments it by 1. Now, **x** being 2, it is again compared to 10 and the result is, again, True, then it is printed out and incremented once again and so on, until **x** reaches 10. Now, having x = 10, the condition is still True, because 10 is less than or equal to 10; that's why 10 is also printed out on the screen and it is then being incremented to 11. Now, when evaluating the expression 11 <= 10, Python returns False and exits the loop without printing anything or incrementing the value of **x**.

But, what would happen if we don't specify the statement that increments **x**? Well, **x** is initially equal to 1. During the first iteration of this loop, it is compared to 10 and the expression returns True; so, it is passed to the code below the **while** statement. It is printed out on the screen, but this time, there isn't any statement to increment it, so it keeps its value of 1 for the next iteration. That's why it is again compared to 10. The condition being met once again, the same scenario will repeat over and over again, forever. That's called an infinite loop and you should always avoid ending up with such a scenario in your programs. Let's test this! Because we already know we're going to generate an infinite loop in the Python interpreter, remember to use **Ctrl+C** to interrupt the infinite loop. So:

```
while x <= 10:
```

```
    print(x)
```

111111111111…

Another way to work with **while** loops is by using an expression which always evaluates as True, in order to make Python do something over and over again, until you tell it to quit. A great example would be an interactive menu, where the user can select a value and execute a piece of code, then return to the main menu and so on. The way to do this is by simply using **while True**, which makes sure that the expression is always evaluated as True.

The syntax for this would be:

```
while True:
    do something
```

We will see such an example later in the course.

One last thing on **while** loops is that we can use the **else** statement again, this time with a different meaning: the indented code below the **else** clause will be executed only when the condition specified in the **while** statement becomes False. Let's see an example and test it in the Python interpreter.

```
x = 1
while x <= 10:
    print (x)
    x += 1
else:
    print("x is now greater than 10")
```

1

2

3

4

5

6

7

8

9

10

x is now greater than 10

So, using the same example as earlier, Python prints out **x** on the screen as long as it is less than or equal to 10. When **x** becomes 11 and the condition becomes False, the code below the **else** clause is executed.

Pretty cool, I think. Very useful stuff.

Now, often you will see **for** loops within other **for** loops; same thing with **if** and **while** statements. You will also need to use **if** statements inside **for** loops or **while** loops, depending on what you are trying to achieve in your programs. Next, we're going to take a look at this kind of structures, which are called **nested structures**.

https://wiki.python.org/moin/WhileLoop

Python 3 Nesting - If / For / While

You can use nesting with control flow statements like **if**, **for** and **while**, to enable a certain behavior and logic inside your Python program. Think of nesting as using an **if** statement inside another **if** statement, a **for** loop inside another **for** loop or a **while** loop within another **while** loop. Let's use some basic examples, to see what I mean.

First, let's refer to **if** statements and code blocks.

When nesting an **if** statement inside another **if** statement, we are actually telling Python that the indented code below the nested **if** statement should be executed only if both the inner statement and outer statement are evaluated as True. Let's see what I mean by that. I will create a new file using Notepad++, called nesting.py and then we will run the file in the Windows cmd.

```
x = "Cisco"

if "i" in x:

    if len(x) > 3:

        print(x, len(x))
```

So, we have a string, referenced by variable **x**. The first **if** statement, the outer statement, checks if the letter "i" is part of this string. If this statement is evaluated as True, which it is, then Python moves further and evaluates the condition in the second **if** statement, which is also True, since the string has more than 3 characters. Finally, since both statements have been validated as being True, the code below the inner **if** statement is executed.

Now, let me save the file, run it, get the result and then I'm going to analyze the code further.

The result of this nested **if** structure is: **Cisco 5**

This code is actually the same thing as using the **and** logical operator between the two expressions.

```
if ("i" in x) and (len(x) > 3):
    print(x, len(x))
```

The result is, indeed, the same.

Cisco 5

The parentheses surrounding each condition inside the **if** statement are completely optional and their role is to provide better readability within the code. I strongly recommend using them, to have your code as clean and neat as possible.

Another thing to be careful about is indentation. When using nested structures, there are multiple levels of indentation created. So, going back to the first version of our nesting structure, the second **if** statement is indented one level, to the right. That's basically obtained using the Tab key once.

Then, the code which should run after evaluating the second **if** statement is indented two levels to the right from the first **if** and one level from the second **if**. That's the result of hitting the Tab key twice, when starting at the beginning of the line. This way we can easily use indentation as a delimiter between nested structures. This also applies to nested **for** and **while** loops. So, keep in mind that each level of indentation changes the logic of your program and for this reason, you should be careful when adding a level of indentation and also when returning to the previous indentation level.

Anyway, it is recommended that you keep a low level of nesting to have a clear and readable program. So, two-level nesting, like in our example, should be just fine.

Now, let's have a look at a nested **for** loop.

Let's assume we have two lists and we want to multiply each element of the first list by each element of the second list. For this, we should iterate over both lists at the same time, take each element into account, do the

multiplications and return the results. Let me rewrite this file and we will discuss it afterwards.

```
list1 = [4, 5, 6]

list2 = [10, 20, 30]

for i in list1:

        for j in list2:

                print(i * j)
```

Let's save the file and run it once again.

40

80

120

50

100

150

60

120

180

So, let me translate this code: dear Python, for the first element in list1 take each element in list2 and multiply them, then for the next element in list1 take each element in list2 and multiply them and so on, until list1 is exhausted. And that's how we got the result you can see on the screen.

Keep in mind that you can also go back from the second indentation level to the previous one, just by hitting the Tab key once, instead of twice, so that the new code is vertically aligned with the second **for** loop.

Let's see this. Let's say that after doing the multiplication we did for each element of list1, we want to also print that element of list1, just to have it printed out for each iteration. For this, we can go back one indentation level and just type in **print(i)** at the same level with the second **for** loop.

```
list1 = [4, 5, 6]

list2 = [10, 20, 30]

for i in list1:
        for j in list2:
                print(i * j)
        print(i)
```

40

80

120

4

50

100

150

5

60

120

180

6

And, indeed, we got each element of list1 printed out after multiplying it with the elements of list2.

So, nesting a **for** loop inside another **for** loop can be pretty useful sometimes.

Now, let's take a look at **while** loops, too and consider the same example we discussed in the lecture about **while** loops. I'm going to rewrite our file again.

```
x = 1
while x <= 10:
    print(x)
    x = x + 1
```

Ok, let's save this file and run it using cmd.

1 2 3 4 5 6 7 8 9 10

As we've already seen, this code block returns all the numbers starting at 1 up to 10, inclusively.

Now, let me write a nested **while** structure and then we will analyze it, as always.

```
x = 1
while x <= 10:
    z = 5
    x += 1
    while z <= 10:
        print(z)
        z += 1
```

Now, let me save the file and execute it using the Windows cmd. The result is 5 6 7 8 9 10 printed out 10 times.

The first **while** loop does the same thing as it did before. It checks whether **x** is less than or equal to 10 and, as long as this is expression is True, it

increments x by one unit; the new thing here is the nested **while** loop, which gets executed each time the first **while** loop is executed.

Now, if we look carefully at the second **while** loop, we notice that it basically does the same thing as the outer loop, meaning it takes the value of **z**, which is initially 5, compares it to 10, then increments it by 1 unit; it does this as long as **z** is less than or equal to 10. This means - as long as **z** is equal to 5, 6 ,7, 8, 9 or 10.

So, the final result consists of the result of the nested **while** loop, which is 5 6 7 8 9 10, printed out 10 times, due to the first **while** loop, which performs 10 iterations. This is just a basic example of nested **while** loops and I hope it has been useful to you.

The last thing I'll add here is nesting a structure inside a different structure. So, let's try nesting an **if** statement inside a **for** loop, as an example. Again, I will delete everything and rewrite the code inside the same file.

```
for number in range(10):
    if 5 <= number <= 9:
        print(number)
```

Let's save and run this Python script now and see the result.

5

6

7

8

9

In this piece of code, the **for** loop iterates over the range created by the **range()** function, meaning the consecutive integers starting at 0 up to 9. Then, the **if** statement specifies a certain condition before printing the

numbers to the screen; actually, there are two conditions. The number should be greater than or equal to 5 **and** less than or equal to 9, in order for Python to execute the indented code below the **if** statement. Finally, for each value in the range that meets these conditions, the **print()** function gets executed and the result is the one you can see in the Windows cmd.

I think you now have a very good understanding of nesting and indentation, so it's time to move on to the next lecture.

https://www.tutorialspoint.com/python3/nested_if_statements_in_python.htm

https://www.tutorialspoint.com/python/python_nested_loops.htm

Python 3 - Break, Continue, Pass

The **break** and **continue** statements are used to handle the flow of a **while** or a **for** loop in a Python program, meaning the programmer can interrupt or restart the execution of a loop structure, in certain conditions.

Let's start with the **break** statement, which is used to terminate the loop in which it resides.

To show how **break** works, we will first create a new file, called break.py and write a **for** loop.

```
for number in range(10):
    if number == 7:
        break
    print(number)
```

Let's save the file, run it, see the result and then I will explain the code.

0

1

2

3

4

5

6

Ok, what this loop does is it takes the numbers in the range generated by the range() function, so 0 up to 9, inclusively, and compares them to 7, according to the **if** statement.

Since the numbers 0 to 6 do not meet the condition in the **if** statement, the **break** statement is not executed for any of them. So, Python goes further into the program and prints them out on the screen.

When the number 7 is selected from the range, it is then passed to the **if** statement and the condition is now evaluated as True. This is where Python executes the indented code below the **if** statement, where it finds the **break** statement; **break** tells Python to stop the execution of the loop right here, ignore the **print** function below and completely quit this **for** loop.

Because Python completely exits the **for** loop, the rest of the numbers in the range, meaning 8 and 9, are also ignored and the program keeps executing other code below the **for** statement, if any.

Now, what if we have a **for** loop nested inside another **for** loop? Let's rewrite our file and test this scenario.

```
list1 = [4, 5, 6]

list2 = [10, 20, 30]

for i in list1:
        for j in list2:
                if j == 20:
                        break
                print(i * j)
        print("Outside the nested loop")
```

Let's save the file and run it and then analyze the code and the result.

40

Outside the nested loop

50

Outside the nested loop

60

Outside the nested loop

So, as we've seen in the previous lecture, in this case Python takes the first element of list1 and tries to multiply it by each element of list2; then, it takes the second element of list1 and tries to multiply it by each element of list2 and so on; but, in this case, we have the **if** statement inside the nested **for** loop and the **break** statement within this **if** block.

So, each time Python tries to multiply the elements of list1 with 20, the **break** statement tells it to stop the execution and quit this loop!

Keep in mind that the **break** statement terminates only the execution of the inner-most loop, not all the loops! So, in this case, **break** stops the execution of the nested **for** loop only, not both loops.

Notice that each time the **break** statement interrupts the multiplication process, we still get the "Outside the nested loop" string printed out on the screen. That's because the print("Outside the nested loop") statement is not part of the nested loop, but **it is** part of the outer-most **for** loop. Notice the indentation levels! The nested **for** loop and the second **print()** function are aligned at the same level of indentation. So, always be careful at indentation, since it **can** change the logic of your program!

The exact same principles apply to nested **while** loops, so we won't get into **while** loops this time.

Now let's have a look at the **continue** statement.

When Python stumbles upon a **continue** statement inside a **for** or **while** loop, it ignores the rest of the code below, for the current iteration, goes up to the top of the loop and immediately starts the next iteration.

In order to see this in practice, let's consider the same lists and for loops as earlier, but replace break with continue in our file.

list1 = [4, 5, 6]

list2 = [10, 20, 30]

```
for i in list1:
    for j in list2:
        if j == 20:
            continue
        print(i * j)
    print("Outside the nested loop")
```

Let's save the file and run it in the Windows cmd.

40

120

Outside the nested loop

50

150

Outside the nested loop

60

180

Outside the nested loop

So, let's study the result of this code to understand what happened.

Python tried to do the same thing here, as before. It took the first element of list1 and tried to multiply it with the first element of list2, and succeeded; so, we got 40 as a result. Then, Python tried to multiply 4 with 20, but since 20 satisfies the condition in the **if** statement, the program executes the **continue** statement underneath it, which ignores the rest of the code in this inner loop, **only** for the current iteration of the loop. That's why we don't see 80 among the results.

After ignoring the multiplication below **continue** for j == 20, Python goes back to the top of the nested loop and extracts the next element from list2, which is 30. And that's why we see 120 among the results.

Finally, after list2 is exhausted, Python executes the **print("Outside the nested loop") function**, which is part of the outer-most **for** loop, so it isn't affected by the **continue** statement at all. Then, the program repeats the same steps for the next two elements of list1 and there's your final result.

As with **break**, the same principles apply to **while** loops when discussing the **continue** statement.

Finally, let's talk about the **pass** statement.

pass is the equivalent of "do nothing". It is actually just a placeholder, for whenever you want to leave the addition of a piece of code for later and move on to write other segments of your program.

Let's see a short example of this, in the Python interpreter.

```
for i in range(10):
    pass
```

As I said, this code doesn't do anything, actually. However, it helps you keep the syntax of your program valid and not get an error returned when running the program. Of course, using the **pass** statement in one area of your application assumes you **do** have some other code inside your program and you want to design the behavior of this **for** loop later on.

But, what if you just don't write anything below the for statement?

Let's try it and see what happens.

```
for i in range(10):
```

IndentationError: expected an indented block

Python throws an error, because this code is invalid, since you haven't typed anything inside the **for** loop. This is the reason why we need the **pass** statement as a temporary placeholder.

I think we covered enough in this lecture, let's go further to the next one.

https://docs.python.org/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops

Python 3 – Exceptions

In Python, you may encounter two main types of errors: syntax errors and exceptions.

A **syntax error** occurs when you don't follow Python's syntax and maybe you forget to add a colon, or the indentation is not proper.

Let's try this in the Python interpreter and skip the colon following a **for** statement, on purpose.

```
for i in range(10)
```

SyntaxError: invalid syntax

So, we got a SyntaxError, as expected. This is an example of an error in Python.

Now let's talk about exceptions.

Unlike syntax errors, exceptions are raised during the execution of the program, interrupting the normal flow of the application.

Let's see a couple of examples.

First, let's use the same **for** loop to generate one of the many types of exceptions, called the NameError.

I will misspell the name of the range() function and wait for Python to notice my mistake.

```
for i in rnge(10):
    print(i)
```

NameError: name 'rnge' is not defined

What do we have here? A **NameError.** This means either that I misspelled a word in my code, in this case the name of a function, **or** the variable or

function I'm trying to use is not yet defined in my namespace. Don't worry, we will talk about namespaces soon. Now, back to exceptions.

Another type of exception is raised when trying to divide a number by 0.

Let's see this in the Python interpreter.

4 / 0

ZeroDivisionError: division by zero

Of, fair enough, we cannot divide a number by 0, so, Python raises a special exception for this particular case.

Also, please notice that Python helps us localize the cause of the exception by also displaying the line of code that generated the exception and the number of the line, as well.

There are many types of exceptions in Python. We won't analyze all of them now, because you will most likely encounter most exceptions as you start creating and troubleshooting your own real programs.

However, you can find a comprehensive list of Python 3 exceptions in one of the links I have attached to this lecture.

Ok, this has been a general introduction to exceptions in Python. In the next lecture, we are going to see how to handle exceptions inside our programs.

https://docs.python.org/3/tutorial/errors.html

https://docs.python.org/3/library/exceptions.html

Python 3 - Try / Except / Else / Finally

Ok, we've seen some Python exceptions and how they look like, but, given the fact that an exception interrupts the normal execution of Python code, it would be more convenient for us to handle an exception when it occurs, and tell Python to keep executing the rest of the code in the program.

To do this, we will use the so-called **try-except** block for exception handling.

Let's consider the ZeroDivision exception from the previous lecture and see how we can handle it properly.

First, let's define the **try** and **except** clauses.

Under the **try** clause you're going to insert the code that you think might generate an exception at a given point in the execution of the program. Of course, this code is going to be one level of indentation away from the **try** statement. As with **for**, **while** or **if** statements, you must always type in a colon after the **try** keyword. Below the **try** statement, you must input the piece of code you want to be executed.

Let's create a new file, called try.py and write the code. We will use the same **for** loop from the previous examples, so:

```
for i in range(5):
    try:
        print(i / 0)
```

Ok, up to this point you should have noticed a couple of things.

First, the indentation. The **try** statement is indented one level to the right from the **for** statement. After **try**, a colon should always be inserted. Then, we enter the code we want to check for exceptions. This code will also be indented one level from the **try** statement, to tell Python that it belongs to the **try** block.

In this case, we are 100% positive we are going to get an exception, right?

Now, let's move to the second part - the **except** clause. Below **except** we will specify what exception types should Python expect, as a consequence of running the code below the **try** clause.

You should think of this whole **try-except** procedure as if you are telling Python the following: "Try to execute the indented code below the **try** clause! If no exceptions occur while executing that code, then skip the **except** clause and move on to the rest of the program. If an exception **does** occur and it is defined in the **except** statement, then execute the indented code below that **except** statement.

So, in our case, knowing that division by zero is not allowed, we are expecting a ZeroDivision exception. Let's tell Python how to handle it.

```
for i in range(5):
    try:
        print(i / 0)
    except ZeroDivisionError as e:
        print(e, "--> Division by 0 is not allowed, sorry!")
```

Let's save the file now and run it.

division by zero --> Division by 0 is not allowed, sorry!

division by zero --> Division by 0 is not allowed, sorry!

division by zero --> Division by 0 is not allowed, sorry!

division by zero --> Division by 0 is not allowed, sorry!

division by zero --> Division by 0 is not allowed, sorry!

So, let's analyze this result. Python took each element from the range generated by the range() function, tried to divide it by 0 under the **try** clause, realized that this isn't possible, so, it raised the ZeroDivision exception in the background.

Now, because we already had that exact exception defined in the **except** clause, it didn't generate the same old output, but instead, it executed the code underneath the **except** clause, printing a customized output.

By the way, notice that in the **except** clause we should define a temporary, user-defined variable - I called it **e** - which can be further embedded into your customized output, if you wish. Otherwise, just print out whatever message you like. The use of this temporary variable is not mandatory. We could have very well written just **except ZeroDivisionError:** and the result would've been the same.

On the other hand, if we wouldn't have had the ZeroDivision exception specified in the **except** clause, or if we would've had any other exception defined instead, Python would've given us the plain old exception message.

Let's test this. Let's say that we didn't realized that a ZeroDivision exception may occur in the code under the **try** clause, but instead we wanted to cover any NameException which could have been generated by the program.

So, we defined this type of potential exception in the **except** clause. Let me write the code for this and then explain it.

```
for i in range(5):
    try:
        print(i / 0)
    except NameError:
        print("You have a name error in the code!")
```

Now, if we save the file and run it, we should still get a ZeroDivisionError, right?

ZeroDivisionError: division by zero

As you can see, this time Python returned the old exception message, because this time we didn't specify any customized message for this type of exception.

Another thing I want to clarify is that if the code inside the **try** block does not generate any exceptions, then Python doesn't even look at the **except** clause and moves on to the rest of the lines of code.

To prove this, I'm going to change the denominator for the division operation to be 1, instead of 0, in order for the operation to be allowed. Let me rewrite the code, first.

```
for i in range(5):
    try:
        print(i / 1)
    except ZeroDivisionError:
        print("Division by 0 is just wrong!")
    print("The rest of the code…")
```

Ok, now let's save the file and run it.

0.0

The rest of the code...

1.0

The rest of the code...

2.0

The rest of the code...

3.0

The rest of the code...

4.0

The rest of the code...

First of all, notice I have also added the **print("The rest of the code…")** function inside the **for** loop, at the same level of the indentation as the **try-except** block, just to show you that for each iteration, Python tries to execute the **i / 1** operation, it succeeds without throwing any exceptions, skips the **except** block and then executes the rest of the code, without even caring that **except** is standing right there. Pretty rude, shame on you, Python!

Another thing to add here is that you can have multiple **except** clauses after a **try** clause. This enables you to be super cautious and handle multiple types of exceptions that can be raised during the execution of the code under **try**.

As always, don't believe me unless we test it. Let's add a couple of **except** clauses to our existing code.

```
for i in range(5):
    try:
        print(i / 1)
    except ZeroDivisionError:
        print("Division by 0 is just wrong!")
    except NameError:
        print("Name error detected!")
    except ValueError:
        print("Wrong value!")
```

Now you may ask me: Ok, but what if I want to catch all possible exceptions that can be generated by the code located under the **try** clause?

Well, you can just use the "except" clause alone, without specifying an exception name to be expected. Just like this:

```
    try:
        print(i / 0)
```

```
except:
    print("Exception was raised!")
```

This is, however, **not** recommended and it is not a good practice, because it will catch absolutely all programming exceptions which may occur and hide the actual root cause of the exception.

I strongly recommend you create multiple **except** clauses according to your needs, specifying the name of the exception each time.

Finally, two more things about **try-except** blocks. You can add two other clauses after the **except** clause or clauses. They are called **else** and **finally**.

The code indented under an **else** clause is executed only if the code under the **try** clause raises **no** exceptions. Let me write this code.

```
try:
    print(4 / 2)
except NameError:
    print("Name Error!")
else:
    print("No exceptions raised by the try block!")
```

Now, let's save the file again and run it.

2

'No exceptions raised by the try block!'

The output is the expected one - the result of 4 / 2 and then the string under the **else** clause gets printed out, because the division raised no exceptions.

Now, the code inside a **finally** clause is executed whether the code inside the **try** block raises an exception or not. So, either way, the **finally** clause gets executed. Let's try this, too. I will replace the **else** clause with **finally**.

First, let's consider a code that doesn't generate any exceptions under **try**.

```
try:
    print(4 / 2)
except ZeroDivisionError:
    print("Division Error!")
finally:
    print("I don't care, I'm getting printed either way!")
```

Let's save and run the file now.

2

'I don't care, I'm getting printed either way!'

The result is 2 and then the string under the **finally** clause.

Now, using code that will generate an exception under the **try** clause:

```
try:
    print(4 / 0)
except ZeroDivisionError:
    print("Division Error!")
finally:
    print("I don't care, I'm getting printed either way!")
```

Save and run.

Division Error!

I don't care if an exception was raised or not!

Ok, so both scenarios are covered by the **finally** clause, as expected.

That's almost all you need to know about exceptions and exception handling for your future programming tasks, so, it's time to move on to the next lecture.

https://docs.python.org/3/tutorial/errors.html#handling-exceptions

Python 3 Functions - Basics

Big, big topic ahead - functions.

This is a core topic in Python and, in fact, in any other programming language. You are going to use functions a lot to build your applications.

But, first of all, what is a function?

Well, you can use a function to organize your code in blocks that can be later reused. This is helpful if you want better readability for your code, modularity and also time saving when designing and running your code.

Before we start writing code, remember that functions follow the same syntax rules as other structures that we've seen so far, but also add some features that we're going to analyze very soon.

First, a function is defined using the **def** keyword, followed by the name of the function, a pair of parentheses and then a colon. After the colon, on the next row, you will type in the code you want to store in this function, indented one level to the right, as we did with **if** or **for** statements.

Let's see this in the Python interpreter.

```
def my_first_function():
    "This is our first function!"
    print("Hello Python!")
```

So, as I said before, we have the **def** keyword, the name of the function, **my_first_function**, then the parentheses and finally the colon. This is the way you define a function. One important thing to remember here is that in between the parentheses you can specify one or several parameters for the function. We will see how to do that in a moment.

Until then, let's further study our function definition. After the colon, which signals the start of an indented code block, we type in the code we want this function to execute; indented, of course.

Notice that on the first row inside the function I typed in a string. This is called a docstring and its role is to describe the role of the function. However, it is entirely optional; maybe you can use it when you define complex functions within complex applications, to remember the role of the function and how it is connected to other segments of your program.

Inside the Python interpreter, you can access the docstring of a function by using the help() command.

```
help(my_first_function)
```

my_first_function()

This is our first function!

We basically created the help section for this function, by using a docstring.

Inside a function, all the syntax rules of Python apply as we've seen up to this point in the course. Whether you're using **for** or **while** loops, **if/elif/else** blocks, dictionaries or list indexing and slicing inside your function, the exact same syntax rules apply.

Now, I said earlier that functions are reusable blocks of code. Let's see why.

After defining a function, with or without any parameters inside the parentheses, we can call that function whenever we need to run the code inside it. In order to call a function, you just need to type in that function's name, followed by the parentheses and that's it. Let's see this in action.

So, we have the my_first_function() function already defined. Let's call it inside the Python interpreter.

```
my_first_function()
```

Hello Python!

```
my_first_function()
```

Hello Python!

```
my_first_function()
```

Hello Python!

So, each time we call that function, it will execute the code inside it. Of course, this is a very basic example of a function, but it should give you an overall view of functions.

Another advantage of functions is that you can change the code within a function and see the results changing, as well, the next time you call that function. So, we can say functions are dynamic structures. Let's redefine our function and change the code inside it. Then, on the next call, the result should reflect the update we've just made.

```
def my_first_function():
    print("Hello Java!")
my_first_function()
```

Hello Java!

Ok, now let's talk a bit about parameters and arguments and the difference between the two concepts. I will expand on that in the next lecture, when you're going to see multiple types of arguments that can be passed to a function call.

For now, let's modify our existing function and insert our first parameter into the picture. Remember, parameters are written inside the parentheses.

Let's see how they work.

```
def my_first_function(x):
    print(x)
```

So, in this case, **x** is passed as a parameter to the function and then used inside the code within the function. This means that whenever we're going to call the function and pass an argument of our choice to the function, that argument will be further passed to the code inside the function.

As a side note - one thing to keep in mind here is the terminology when using functions. Parameters are the ones written inside parentheses when **defining** the function.

Arguments are the ones written inside parentheses when **calling** the function.

Most of the time, they are used interchangeably, but you should try to follow and use this terminology, though.

We will discuss arguments in more detail in the following lecture.

For now, let's return to our function and call it by passing an argument to the function call.

```
my_first_function("Hello Python")
```

Hello Python

So, what we did is we called our function and told Python to use the string inside the parentheses as an argument. Then, the string was passed to the print() function and, finally, printed out on the screen.

You can also enter multiple parameters within a function definition, like this:

```
def my_first_function(x, y):
    print("Hello " + x)
    print("Hello " + y)
```

So, according to this piece of code, we expect that when calling the function and passing two strings as arguments inside the parentheses, they will both be printed out, preceded by the "Hello " string. Let's see if I'm right.

```
my_first_function("Python", "Java")
```

Hello Python

Hello Java

Notice that **x** was mapped to "Python" and **y** was mapped to "Java", because that was the order we used when passing the arguments. Pretty intuitive, I think.

Now, we're going to touch on another topic and that is the **return** statement. This statement is used to exit a function and return something whenever the function is called. Let's delete the print() function inside our main function and see how to use the **return** statement. Let me write this first.

```
def my_first_function(x, y):

    sum = x + y
```

Ok, so, we have created a variable inside our function, that will reference the result of adding **x** and **y**, which are the function's parameters. Let's see if our function returns something when we call it.

```
my_first_function(1, 2)
```

So, nothing is returned. That's because we haven't specified what exactly are we looking to get back from the function. The function, in its current form, does nothing more than creating a variable named **sum** and storing the result of adding 1 and 2. However, it doesn't return anything. It keeps the result secret, for now. That's why we need the **return** statement.

Let's see how to use it.

```
def my_first_function(x, y):

    sum = x + y

    return sum
```

my_first_function(1, 2)

3

So, this time we told Python to return the value stored by **sum** and we got the expected result in return.

Now, we can change the function even more and call it using another set of arguments.

Let's add another parameter, **z**, change the expression inside the function and then return the value of **sum** squared.

```
def my_first_function(x, y, z):
    sum = (x + y) * z
    return sum ** 2
```

Let's call the function again.

my_first_function(1, 2, 3)

81

The result seems correct, right?

The last thing I should add here is that if you just use the **return** statement, without specifying what exactly do you want to get out of the function, Python will return the None value. So, **return** without specifying what to return is, actually, the same thing as **return None**. Let's test this.

```
def my_first_function(x, y, z):
    sum = (x + y) * z
    return
my_first_function(1, 2, 3)
```

So, as I said, nothing is returned, this time. Now, let's get to the next lecture.

Python 3 Functions - Arguments

Ok, we've seen how to use function arguments, we went through some examples, but why do we need a special lecture dedicated to arguments? Well, that's because we've only seen one type of function arguments, but Python defines several types and we're going to analyze each of them in this lecture.

The type of arguments we've seen in the previous lecture is called **positional**, meaning that the number and position of arguments in the function call should be the same as the number of parameters in the function definition.

If the numbers are different, then Python will return a TypeError.

Let's test this. So, we will consider the same function as earlier.

```
def my_first_function(x, y, z):
    sum = (x + y) * z
    return sum
```

This function takes 3 parameters: x, y and z. However, if we're going to pass only two arguments when calling the function, Python will raise a TypeError, telling us that the number of expected arguments and the number of provided arguments do not match. Let's test this by passing only two arguments during the function call.

```
my_first_function(1, 2)
```

TypeError: my_first_function() missing 1 required positional argument: 'z'

Also, remember that, in this case, the order in which we provide the arguments when calling the function is very important, because the first argument inside the function call will be associated with the **x** parameter, the second argument is going to be mapped to **y** and the third one with **z**.

Now, let's see another type of arguments - keyword arguments.

Keyword arguments allow us to ignore the order in which we entered the parameters in the function definition or even to skip some of them when calling the function.

Let's see this in an example using the same function as earlier.

```
def my_first_function(x, y, z):
    sum = (x + y) * z
    return sum
```

Using keyword arguments means you can specify the name of each parameter and assign it a corresponding value during the function call.

```
my_first_function(x = 1, y = 2, z = 3)
```

9

This allows us to type in the arguments in a different order, as well, unlike positional arguments.

```
my_first_function(z = 3, x = 1, y = 2)
```

9

Python also allows you to mix positional and keyword arguments, but the rule when doing this is to **first** specify the positional arguments and then the keyword arguments. If you don't follow this rule, Python will return a SyntaxError:

```
my_first_function(x = 1, y = 2, 3)
```

SyntaxError: positional argument follows keyword argument

Ok, now let's try this, the right way.

```
my_first_function(1, 2, z = 3)
```

9

This way, we can call the function multiple times, keeping **z** equal to 3 each time and only changing the values of **x** and **y**.

What else?

You can also define a default value for an argument, by specifying it right in the function definition, inside the parentheses. Let's see this.

```
def my_first_function(x, y, z = 3):
    sum = (x + y) * z
    return sum
```

This time, the value of 3 is assigned to **z** by default, in case no other value is specified during the function call. If a different value is assigned to **z** when calling the function, then the new value overrides the default one.

Let's test this.

```
my_first_function(1, 2)
```

9

Ok, so, Python assumed that **z** is by default equal to 3, without us having to specify its value during the function call.

However, if we do want **z** to take another value, we can explicitly specify this.

```
my_first_function(1, 2, 4)
```

12

But, what if you don't know how many parameters you're going to need in your function and you want to specify a variable number of parameters inside a function definition? Well, Python has a solution for this, as well. In fact, two solutions - one, for positional arguments and one, for keyword arguments.

The solution for passing a list of parameters having a variable length is to use an asterisk, like this:

```
def my_first_function(x, *args):
```

```
    print(x)
```

The **args** variable is actually a tuple of potential additional parameters. The tuple is initially empty, if we don't specify additional parameters, besides **x**.

So, calling this function using a single argument would be a valid thing to do, because that is the role of **x**.

```
my_first_function("hello")
```

hello

In fact, to grasp this concept even better, let's add another **print()** function inside the main function, to also see the contents of the **args** variable.

```
def my_first_function(x, *args):
    print(x)
    print(args)
```

Now, when calling the function, we will see **x** being printed out, but we will also get an empty tuple, which corresponds to the **args** parameter.

Let's test this.

```
my_first_function("hello")
```

hello

()

Now, let's call the same function again, but this time using several arguments and see if the tuple gets populated.

```
my_first_function("hello", 100, 200)
```

hello

(100, 200)

Ok, so, we have the confirmation that the **args** variable is indeed a tuple. But, how can we print all the arguments that we pass to the function, without

having this ugly tuple returned? Well, simply by iterating over that tuple, using a **for** loop. First, let's remove the second **print()** function and write the **for** loop instead.

```
def my_first_function(x, *args):
    print(x)
    for argument in args:
        print(argument)
```

As a side note, don't forget about correctly indenting your code when using the **for** loop.

Now, let's call the function again, using three arguments.

```
my_first_function(1, 2, 3)
```

1

2

3

We got the expected result, right?

Now, if you need a variable number of parameters, but you want to use keyword arguments, instead of positional arguments, when calling the function, you can use a double asterisk, followed by **kwargs**, instead of **args**. The concept and functionality are the same as we've seen with the **args** variable, so I won't get into it, just to avoid becoming boring.

Also, remember that the names **args** and **kwargs** are just conventions; you can use any name instead, as long as you don't forget the asterisk for positional arguments and double asterisk for keyword arguments - these are mandatory.

You can find a guide to use different types of arguments in the link attached to this lecture.

https://en.wikibooks.org/wiki/Python_Programming/Functions

Python 3 Functions - Namespaces

What is a namespace, in Python? Well, as its name implies, a namespace is a space holding some names. What names are we talking about? By names, I am referring to variables, functions or classes we create.

Now, you can think of a namespace as a container holding the names we define, where each name is associated with the namespace in which it is defined.

This way, you may have the same name defined in different namespaces. We will see this in practice, in a second.

The last thing I should mention before we get to work is that there are 3 types of namespaces.

The built-in namespace contains Python's built-in functions. We've seen a lot of built-in functions up to this point, in the course. For instance, we've used the len(), max(), range() or sort() functions. They are available whenever you want to use them, whether you are trying to access them from within the Python interpreter or into a Python script.

The global namespace contains all the variables or functions **you** define or import directly into your program, whereas the local namespace refers to the names you use only inside a particular function, in your program. Let's better see this, by creating a Python file and playing around with some variables.

I'm going to create a new file on the D: drive, called namespaces.py.

Now, we have an empty file available. Let's open the Windows command line, in which we're going to run our file each time we make any changes to the file.

First, let's use a name from the built-in namespace, just to prove that it is already available to use.

So, let's insert **print(list(range(10)))** in our file. Now, let's save the file and run it.

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Ok, so, the file was successfully executed and notice that we didn't need to define the print(), list() and range() functions, prior to using them, because they are built-in functions and they belong to the built-in namespace.

Now, let's get back into the file and try to use a non-built-in name, a random variable, called **my_var**.

```
print(my_var)
```

Let's save the file again and try to run it one more time.

As soon as we run it, we get a NameError, saying that the name **my_var** is not defined. This is because **my_var** is neither part of the built-in namespace, nor previously defined within the file; so, Python doesn't know where to get it from when trying to print its value.

Now, let's get back into the file and create this variable, by assigning a value to it.

```
my_var = 10

print(my_var)
```

After running this file again, we notice that the file is successfully executed, and no exceptions are raised, because this time **my_var** was created before passing it as an argument to the print() function.

We can say that we have created **my_var** in the global namespace of our program, so, now, **my_var** is a global variable.

Now, let's modify our file and move both statements inside a function and then call that function, in order to have the value of **my_var** printed out, like we did in the previous example.

Also, let's spice things up and print out the value of **my_var** times 10, as well.

```
def my_var_func():
    my_var = 10
```

```
    print(my_var)

my_var_func()

print(my_var * 10)
```

Let's run the file again, now.

10

NameError: name 'my_var' is not defined

So, let's analyze the result. First, we got 10 printed out to the screen.

That's because inside the function, we first created **my_var** and then used it as an argument for the print() function below; we then called the my_var_func() function and that's how we got 10 printed out.

But, wait! Shouldn't the second print() function return the value of 10 times 10, so 100? Why did we get an exception saying that "name my_var is not defined"? It's right there in the function, isn't it?

Well, here's the catch - this time, **my_var** is not a global variable anymore, available to use across the entire program; this time, **my_var** is just a local variable, belonging to the local namespace of the my_var_func() function.

So, inside the my_var_func() function, the **my_var** variable is ok to use; it is accessible and usable. However, as soon as you go outside the function, **my_var** is no longer available.

To better see this, let's edit the file again, this time creating a variable outside the function, having the same name, just to prove that each name is associated only with the namespace in which it is defined. So, let's add **my_var = 20** somewhere outside the function, but before the print() function; and then let's run the file again.

10

200

This time, we got the same value, 10, as before, by calling the function and then the value 200, by printing the global variable **my_var** multiplied by 10. Makes sense, right? This example clearly separates the concepts of global namespaces and local namespaces.

Now, continuing with the same example, let's delete the second print() function.

```
def my_var_func():
    my_var = 10
    print(my_var)
my_var_func()
```

And now, let's switch the two statements inside the function.

```
def my_var_func():
    print(my_var)
    my_var = 10
```

Let's run the file once again.

UnboundLocalError: local variable 'my_var' referenced before assignment

This time, we got a different error, an UnboundLocalError exception, saying that the "**local variable 'my_var' referenced before assignment**". That's because Python reads the code lines in order, so, if you first use the **my_var** variable in a print() function and then create that variable, Python will be unaware that you created it when it tries to execute the print() function; so, when you then call the my_var_func() function, Python will return the UnboundLocalError exception. You should be very careful when you create variables and when you decide to reference them inside the code. This rule is valid both in the global and in the local namespaces.

Now, let me show you another thing. Let's return to the file and create a global variable named **my_var** before our function. We will see how to use a global variable inside the local namespace of a function.

```
my_var = 5

def my_var_func():

    print(my_var)

    my_var = 10

my_var_func()
```

If we run this code as it is, we will get the same UnboundLocalError, right? Because we still have the print() function before the variable definition inside the main function. Nothing has changed there.

But, what if you decide to use the global **my_var** variable inside the function? To do that, you need to use the **global** keyword to - let's say - import a global variable to a local namespace. So, this is how to do it. Inside the function, before any other statement that refers to this global variable you want to use, you just type in **global** and the name of the global variable you want to import.

```
global my_var
```

Let's save the file as it is and run it.

5

The result is the value referenced by the global variable, printed out to the screen using the print() function inside the main function. Magic!

The last thing I'll show you here is how you can use the **return** statement in this case.

First, remember that **return** is used to exit a function and return something when the function is called.

Let's edit our file again and return to the code we used at the beginning of this lecture, when we got the **"name my_var is not defined"** error.

```
def my_var_func():

    my_var = 10
```

```
    print(my_var)

my_var_func()

print(my_var * 10)
```

Now let's add a **return my_var** statement at the end of the function and run the file again.

10

NameError: name 'my_var' is not defined

Notice that we still get 10 returned by the function call, which is normal, but we are unable to use the **my_var** variable further into the program. In order to have this variable available outside the function, you can make use of the **return** statement we defined and then store the value returned by the function using a new variable, outside the function. Then, you can use the new variable in the final print() function.

```
def my_var_func():
    my_var = 10
    print(my_var)
    return my_var

result = my_var_func()

print(result * 5)
```

Let's see the result.

10

50

Now, we got 10 returned by the first print function and the function call itself, and then 50 from the print() function at the end of the file, that intercepts the value of my_var returned by the main function and multiplies it by 5.

So, this is the way you can propagate the value of **my_var** from within the local namespace of the function, all the way down to the print() function at the end of the program.

Next, it's time to learn about modules and ways of importing a module.

https://www.python-course.eu/python3_namespaces.php

Python 3 Modules - Importing

Technically speaking, a module is nothing more than just a Python file containing specific functions, variables or classes. Of course, these functions and other data types inside a module are not randomly grouped inside that file. They are grouped together to provide some functionality, which can then be used further inside other programs. Modules are also a great way to logically organize your code.

There are a lot of Python modules - hundreds, maybe even thousands - that you can use into your programs. Some of them are default modules, belonging to the Python built-in namespace, and others can be downloaded, unzipped and installed separately.

You can also create your own Python modules, meaning you can write some code into a file and then use that code further, inside the application you are building, by simply importing the file.

Now, let's consider the **math** module, which is a Python built-in module and see its contents, using the dir() function in the Python interpreter. Also, don't forget to import it first, using the **import** statement.

```
import math
```

```
dir(math)
```

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

We can see here a list of names which are specific to this particular module and exist only within its own namespace.

The general rule for accessing module functions and variables when using the "import" statement is typing in the name of the module, a dot and then

the name of the function or variable you need to access. Now, we can do the same thing, using the **math** module, to access the value of **pi**.

```
math.pi
```

3.141592653589793

Remember that if you forget to import the module you need and try to access one of its variables or functions, you will get a *NameError: name 'math' is not defined* exception.

Now, let's create our own module and see the three ways in which you can import data from within a module.

I will create two files on my D: drive - my_module.py and python_app.py.

```
D:
```

Let's consider my_module.py as being our own user-defined module and python_app.py as being an application we are currently working on.

Let's open each file in a separate Notepad++ window, so we can see this better.

In our python_app.py file, we are going to insert a print() function only. This is going to be a very basic Python application.

```
print("This is my app!")
```

Let's save the file and run it in the Windows command line.

```
D:\WINDOWS\system32>python D:\python_app.py
```

This is my app!

Now, let's also enter some code inside our module. We will create a variable, a function and also add a print() function at the end of the file, outside the main function.

```
my_var = 10

def my_function():
```

```
    print("This is the function inside the module!")

print("This will be executed!")
```

As you can see, the **my_var** variable is not used anywhere inside the module and the function is not called anywhere either. They are available there for you to import and use them into your program whenever you decide to do so.

Let's save the module now and see how we can use the code inside the module into our application.

As we've seen so far, we can use the **import** statement in order to make use of the code, variables and functions inside a module. An important thing to remember here is that when you import a module into your program, Python will automatically execute the code inside that module. This will happen only once, no matter how many times you import that module. Let's test this by importing my_module into python_app.py. Both files are in the same directory, so the module is automatically found by our application.

I will add the **import** statement before anything else inside the application.

```
import my_module
```

Let's run our application again, now.

```
D:\WINDOWS\system32>python D:\python_app.py
```

This will be executed!

This is my app!

As I said, when importing a module for the first time, the code within that module gets executed. So, this time, we got the string printed out by the print() function at the end of the module and then the string printed out by our application. Cool!

Now, to prove that the code inside the module gets executed just once, let's say that, for some reason, you import that module twice into your application. So, let's go ahead and enter another **import** statement.

```
import my_module
```

Now, let's run our main application again.

```
D:\WINDOWS\system32>python D:\python_app.py
```

This will be executed!

This is my app!

So, there's the proof! The string printed out by the print() function at the end of the module is displayed just once, even though we have imported the module twice inside our main application.

Another thing to add here is that now you have all the variables and functions inside my_module available in the namespace of your main program. You can reference and use them by typing in the name of the module, a dot and then the variable or function you need to use. Let's test this. I want to print out my_var from within the module and also execute my_function() inside the main application. So, let's add some code at the end of our main file.

```
print(my_module.my_var)

my_module.my_function()
```

Now, let's save the application and run it one more time.

```
D:\WINDOWS\system32>python D:\python_app.py
```

This will be executed!

This is my app!

10

This is the function inside the module!

Now, we can say that we have successfully imported a custom module into our application and then used variables and functions from within that module inside the code of the main application.

Ok, this was the first way of importing code from an external source, a module. Let's see the second way, now.

You can use the **from … import** statement to import only some variables or functions from within a module into your application's namespace, thus reducing the resource usage of your program.

Let's say that from my_module you only need to use the my_var variable into your application, not the entire module. You can do this very easily, like this.

```
from my_module import my_var

print("This is my app!")

print(my_var)

my_function()
```

Now, let's run our application one more time:

```
D:\WINDOWS\system32>python D:\python_app.py
```

This will be executed!

This is my app!

10

Traceback (most recent call last):

File "D:\python_app.py", line 4, in <module>

my_function()

NameError: name 'my_function' is not defined

This time, we got the string printed out by the module, because Python executed the code inside the module, meaning only the print() function at the end of the module. Then, we got the string printed out by our application; next, the value of **my_var** got printed out, because we have imported that variable using the **from … import** statement and, finally, we got a *NameError: name 'my_function' is not defined.*

Why? Because, this time, we only imported the **my_var** variable from within the module into the local namespace, not the entire module, as we did previously. So, now, Python sees no **my_function()** in the local namespace of our application. This is why we got a NameError at the end of the output.

Notice one more thing here - unlike the first way of importing we've seen earlier, when using the **from … import** statement you are not required to specify the module name before the variable or function name; that's why you can just use **my_var** inside the main application, instead of **my_module.my_var**.

The last way of importing is using the **from import *** method, where the asterisk means import ***all*** names from within that module. This method is generally not recommended because it overloads the local namespace of the main application you are importing to. So, if you want to import all the names from within a module, you should use the **import** statement instead.

Now that we've seen the ways to import a module into our applications, let's talk a bit about where you can store and how you can locate a module inside your file system. In our previous example, we had both the module and the main application file in the same folder. Python first checks the built-in namespace for the module; then, if the module is not a built-in one, it looks in the current folder, to check if the module you are trying to import is stored there.

Next, if the file isn't found in the same folder, Python will look for the module in each folder inside the PATH variable, which is actually a list of directories or folders. You can see the PATH variable by opening up the Python interpreter and using the sys module.

```
import sys

sys.path
```

```
['',
'D:\\Users\\teodo\\AppData\\Local\\Programs\\Python\\Python37\\Lib\\idlelib'
,
```

'D:\\Users\\teodo\\AppData\\Local\\Programs\\Python\\Python37\\python37.zip',
'D:\\Users\\teodo\\AppData\\Local\\Programs\\Python\\Python37\\DLLs',
'D:\\Users\\teodo\\AppData\\Local\\Programs\\Python\\Python37\\lib',
'D:\\Users\\teodo\\AppData\\Local\\Programs\\Python\\Python37',
'D:\\Users\\teodo\\AppData\\Local\\Programs\\Python\\Python37\\lib\\site-packages']

The last thing we are going to talk about in this lecture is how to skip the execution of some code when importing a module.

Let's return to our custom module from earlier and let's say that we don't want to execute the print() function at the end of the file when importing the module into our main application. We want to keep the print() function inside the module and skip its execution when importing that module.

For this, we are going to use a trick. Inside the module, we are going to include the code we don't want to be executed at import inside an **if** block. I will write this first and then I will translate it.

```
if __name__ == "__main__":

    print("This will be executed")
```

Now, in our main application file, let's return to the old **import** statement and add the module name as a prefix to **my_var** and **my_function()**.

```
import my_module

print("This is my app!")

print(my_module.my_var)

my_module.my_function()
```

Let's save both files before continuing.

Returning to our module, the **if** statement can be translated like this: if this file is going to be executed as a stand-alone program, then go ahead and execute the code below, in addition to the rest of the code inside the module.

On the other hand, if the file is **not** going to be executed as a stand-alone application, instead it is going to be used as a module and imported somewhere else, then do **not** execute de code below the **if** statement.

Don't believe me, let's test this!

First, let's execute our application.

D:\WINDOWS\system32>python D:\python_app.py

This is my app!

10

This is the function inside the module!

Notice that, this time, the print() function below the if statement is not executed anymore when we import the module, as it did before.

Now, let's run the module as a stand-alone application.

D:\WINDOWS\system32>python D:\my_module.py

This will be executed!

This time, the print() function is indeed executed, because the file was treated as a stand-alone program, not as a module.

If you're curious to learn more about the theory behind this trick, check out the links attached to this lecture. For now, just remember the trick itself. You can apply it whenever you decide to build your own modules.

https://docs.python.org/3/reference/import.html

https://docs.python.org/3/library/__main__.html

https://stackoverflow.com/questions/419163/what-does-if-name-main-do

Python 3 Modules - Helpful Functions: dir() and help()

Let's say that you want to use a Python module or you just heard about one particular module and you want to find out how it works and what functions are available to you inside that module.

For this purpose, Python provides you with two helpful functions in the Python interpreter, to quickly find out more information about a module's functionality and contents.

Let's see these functions in action, by opening up the Python interpreter.

So, let's say that we heard about the **sys** module and we want to find out if it can actually help us to achieve our goal.

As a side note, keep in mind that you should already have the Python module installed, if it's not a Python built-in module. You will learn how to download and install a custom, non-built-in module, later in the course, don't worry. For now, we will make use of the default modules.

Second of all, don't forget that you should first import that module into the Python interpreter, so that Python will be able to access its information.

The first function that we're going to discuss is **help()**.

So, let's apply **help()** on the sys module and see the results.

help(sys)

The function returns lots of information about the module and, if you **scroll** all the way up to the beginning of this output, you will find the module name, the reference to its documentation and, finally, a comprehensive description of each of its functions, objects and attributes.

Next, the second function is **dir()**. This one shows you what names - functions or attributes - are part of the specified module's namespace. So, now let's use **dir()** on our **sys** module and see its available names.

dir(sys)

['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__', '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames', '_debugmallocstats', '_enablelegacywindowsfsencoding', '_framework', '_getframe', '_git', '_home', '_xoptions', 'api_version', 'argv', 'base_exec_prefix', 'base_prefix', 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle', 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth', 'get_coroutine_wrapper', 'getallocatedblocks', 'getcheckinterval', 'getdefaultencoding', 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettrace', 'getwindowsversion', 'hash_info', 'hexversion', 'implementation', 'int_info', 'intern', 'is_finalizing', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'set_coroutine_wrapper', 'setcheckinterval', 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout', 'thread_info', 'version', 'version_info', 'warnoptions', 'winver']

We can see here a list of all the available functions and attributes inside this module. For instance, you can notice the *path* attribute we used earlier to get the Python PATH variable.

Keep in mind that these two functions, **help()** and **dir()** are there to help you get information about any Python module, so use them whenever you feel uncertain about a module.

https://docs.python.org/3/library/functions.html#dir

https://docs.python.org/3/library/functions.html#help

Python 3 Modules - Installing a Non-Default Module

We have used only Python's built-in modules up to this point in the course.

However, there are hundreds or thousands of other modules, custom modules, available to use in Python, most of them being focused on a specific task or area. For instance, you may encounter modules that are built for handling Excel files, or databases, or web scraping, network devices, scientific operations and so on.

This means that, whenever you want to build a specific application, you may have several non-default Python modules available to download and use, completely free. Most of them have their own GitHub pages or websites and extensive documentation may be available for most of them.

That's why it is very important for you to know how to properly download and install a Python module and have it ready to use inside your scripts and in the Python interpreter.

Don't worry, the process of installing a non-built-in module is very straightforward. Python provides, by default, the **pip** package manager. **pip** is a package management system used to install and manage software packages written in Python. **pip** will handle the entire download and installation process, including the download of any dependencies, if necessary.

Let's say we want to build an application that will handle Excel spreadsheets and we find out that there's a module named **openpyxl** that can help us.

This is a non-default Python module and, to prove this, let's get into the Python interpreter and try to import this module.

```
import openpyxl
```

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
```

import openpyxl

ModuleNotFoundError: No module named 'openpyxl'

Of course, we got a ModuleNotFound exception.

Now, to automatically download and install this module, along with any necessary dependencies, let's open up the Windows command line and use the **pip** package manager.

The command you should use is:

python -m pip install openpyxl

So, we have the **python** command, **-m** which stands for module, the name of the package manager, **pip**, the action to be performed, **install** and, finally, the name of the module we want to install. Now, after hitting Enter, the module is downloaded and installed. Of course, make sure your Internet connection is up and running, before using **pip**.

Collecting openpyxl

Downloading https://files.pythonhosted.org/packages/e5/0a/e0a095149a23cedd9c8db6cdde2af7f82105e219e14edea0c31a19aeff9e/openpyxl-2.5.8.tar.gz (1.9MB)

100% |████████████████████████████████| 2.0MB 2.1MB/s

Collecting jdcal (from openpyxl)

Downloading https://files.pythonhosted.org/packages/a0/38/dcf83532480f25284f3ef13f8ed63e03c58a65c9d3ba2a6a894ed9497207/jdcal-1.4-py2.py3-none-any.whl

Collecting et_xmlfile (from openpyxl)

Downloading https://files.pythonhosted.org/packages/22/28/a99c42aea746e18382ad9fb36f64c1c1f04216f41797f2f0fa567da11388/et_xmlfile-1.0.1.tar.gz

Installing collected packages: jdcal, et-xmlfile, openpyxl

Running setup.py install for et-xmlfile ... done

Running setup.py install for openpyxl ... done

Successfully installed et-xmlfile-1.0.1 jdcal-1.4 openpyxl-2.5.8

Great, it seems the installation was successful. Now, let's verify this in the Python interpreter, by trying to import the module once again.

```
import openpyxl
```

```
dir(openpyxl)
```

['LXML', 'NUMPY', 'PANDAS', 'Workbook', '__author__', '__author_email__', '__builtins__', '__cached__', '__doc__', '__file__', '__license__', '__loader__', '__maintainer_email__', '__name__', '__package__', '__path__', '__spec__', '__url__', '__version__', '_constants', 'cell', 'chart', 'chartsheet', 'comments', 'compat', 'constants', 'descriptors', 'drawing', 'formatting', 'formula', 'load_workbook', 'packaging', 'pivot', 'reader', 'styles', 'utils', 'workbook', 'worksheet', 'writer', 'xml']

And, indeed, this time we get no exceptions and we can already use the new module.

Also, over time, you may see that the **pip** version on your system gets outdated, as new versions are released. Notice we also got a warning at the end of this output, saying that there's a new version of **pip** available.

You are using pip version 10.0.1, however version 18.0 is available.

You should consider upgrading via the 'python -m pip install --upgrade pip' command.

We also get the command to use for updating **pip**, so, why not use it right now?

```
D:\Windows\System32>python -m pip install --upgrade pip
```

Collecting pip

Using cached https://files.pythonhosted.org/packages/5f/25/e52d3f31441505a5f3af41213346e5b6c221c9e086a166f3703d2ddaf940/pip-18.0-py2.py3-none-any.whl

Installing collected packages: pip

Found existing installation: pip 10.0.1

Uninstalling pip-10.0.1:

Successfully uninstalled pip-10.0.1

Successfully installed pip-18.0

Great, so, in this lecture you learned how to properly install a non-default Python module using the **pip** package manager and also how to upgrade **pip** itself. See you in the next lecture!

https://docs.python.org/3/installing/index.html

Python 3 Files - Opening & Reading

Python provides all the tools you need for handling files in your operating system.

You can open and read files, you can write or append to a file, and also store and use the information within a file into your application.

As you do with any file in your filesystem, whether it is a Microsoft Word document, a picture or a video, you first have to open the file, in order to read or modify its contents.

For this, Python provides you with the open() function.

You are not required to import any modules to be able to use all the file methods Python provides to the user.

Let's first create a text file on the D: drive and insert some data into the file, and then we will start playing around with it.

I have already created a file named **routers.txt**, having the names of some networking vendors as its content.

Cisco

Juniper

HP

Avaya

Nortel

Arista

Linksys

Now, it's time to create a file object and read it. This is how you can do this.

Let's open the Python interpreter and start working. First you should type in a name for the file object you create, let's say **myfile**, then the equal sign and then you will use the open() function.

```
myfile = open("D:\\routers.txt", "r")
```

So, we have: open() and, in between parentheses, you first specify the name of the file you want to open and the path to the file, enclosed by double quotes; then, separated by a comma, you enter the access mode in which you want the file to be opened, also enclosed by double quotes.

So, let's open the file for reading; for this, we will use **"r"** as the access mode.

The most common modes in which you can open a file are: **"r"** for reading - keep in mind that this is the default mode, so, if you don't type in "r" when opening a file, Python will open it for reading, by default.

The second mode is **"w"**, for writing and **"a"**, for appending to the end of the file.

You may also encounter **"b"** as a file access mode; this stands for binary format and it is useful when playing around with binary files like photos, adobe pdfs, executable files and so on; the binary mode should be used for all files that don't contain text.

Finally, we have the **"x"** mode available, which stands for exclusive creation. This clearly means that you would use this mode for exclusively creating the file and fails if the file already exists. Basically, this is the same as using the "w" access mode, only it raises an exception if the file already exists.

I will provide a complete list of file opening modes at the end of this section, so you can have it at hand whenever you are in doubt about using a certain access mode.

If you want to check the mode in which a file has been opened, you can check out an attribute named **mode** for your file, like this:

```
myfile.mode
```

'r'

So, again, we see that our file is currently open for reading.

Now, having the file open, let's see how to access and print out the data within the file.

Python provides three important methods to read data from a file.

The first method is **read()**, which returns the entire content of a file, in the form of a string.

myfile.read()

'Cisco\nJuniper\nHP\nAvaya\nNortel\nArista\nLinksys'

So, you can see that we have all the vendors listed inside a single string, separated by the newline character.

Although not very aesthetic, this may be useful for searching something inside the entire file or maybe matching some text patterns inside the file. We will talk more about patterns and regular expressions later in the course.

One more thing about the **read()** method is that you can tell Python to read only a number of bytes from the file and return that many characters, not the entire content of the file. You can do this by entering an integer in between parentheses.

Let's say you want only "Cisco" to be returned; this means the first 5 characters in the file, right?

myfile.read(5)

''

Uhm, this doesn't return what it should. Did I lie when I said it will return the first 5 characters in the file?

Well, no, I didn't!

This is where the concept of "cursor" enters the game. The cursor is the position at which you are currently located inside the file. So, after we previously read the entire file using the **read()** method, we are now

positioned at the end of the file. That's why nothing is returned when we want to read the file content further. At this point, we actually have nothing else to read; we already went through all the text in the file.

To go back to the beginning of the file you have to position the cursor before the first character.

You can do this using the **seek()** method, where in between parentheses you enter the position or, better said, the number of bytes from the beginning of the file at which you want the cursor to be positioned. If you want to start from the very beginning, you will use **seek(0)**. So:

```
myfile.seek(0)
```

0

This also returns the new position at which the cursor is currently positioned.

Furthermore, to see the current position of the cursor, you can also use the **tell()** method and it should return the same result.

```
myfile.tell()
```

0

This shows that we are currently positioned at 0 bytes from the beginning of the file. Just what we were looking to achieve!

Now, we can read the file again. Remember, we wanted to read only the first 5 bytes or characters in the file, right?

```
myfile.read(5)
```

'Cisco'

So, this time, we got 'Cisco' returned, which is indeed correct.

Another useful method is **readline()**. This returns the file content, line by line, one line at a time, each time you call the method.

Let's see this in practice, but first, let's go back to the beginning of the file again.

myfile.seek(0)

myfile.readline()

'Cisco\n'

myfile.readline()

'Juniper\n'

myfile.readline()

'HP\n'

myfile.readline()

'Avaya\n'

myfile.readline()

'Nortel\n'

myfile.readline()

'Arista\n'

myfile.readline()

'Linksys\n'

myfile.readline()

''

So, we got each line of the file printed out and then, in the end, we got an empty string; this signals that we have reached the end of the file.

The third way of reading files is using the **readlines()** method; this method returns a list, where each element of the list is a line in the file; this method is very useful for iterating over a file and it is frequently used when working with files.

myfile.seek(0)

myfile.readlines()

['Cisco\n', 'Juniper\n', 'HP\n', 'Avaya\n', 'Nortel\n', 'Arista\n', 'Linksys\n']

Now, you can use a **for** loop to work with the elements of this list.

First, of course, don't forget about the **seek()** method! If you forget to position the cursor at the very beginning of the file, then you will end up with an empty list when calling the **readlines()** method again. Let me prove this to you.

myfile.readlines()

[ ]

And that's the proof. Now, as I said, let's use the **seek()** method once again.

myfile.seek(0)

Now, let's print out the vendors whose names start with a capital A. For this, we are going to iterate over the list provided by the **readlines()** method and use a condition stating that if the list element, which is a string, starts with a capital A, then go ahead and print that element to the screen.

```
for line in myfile.readlines():
    if line.startswith("A"):
        print(line)
```

Avaya

Arista

Notice here that we also got a blank line after each string. That's because we had a **\n**, a new line character, at the end of each element in the list and that gets printed out, as well.

Let's also test the **"x"** mode on our already existing file and see if it really generates an exception, since the *routers.txt* file already exists.

```
myfile = open("D:\\routers.txt", "x")
```

```
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    myfile = open("D:\\routers.txt", "x")
FileExistsError: [Errno 17] File exists: 'D:\\routers.txt'
```

Ok, so here's our confirmation about the "x" access mode and its associated exception, called FileExistsError. But, what if the file does not exist?

That means we should have a new file created, right? Let's test this, as well.

```
myfile = open("D:\\routers2.txt", "x")
```

And the new file has been created on the D: drive.

One thing to keep in mind here - depending on your current system setup and the type of privileges that you have as a user, you might end up with a PermissionError if the the user you're currently logged in with has no administrative privileges and thus cannot save to the D: drive.

```
myfile = open("D:\\routers2.txt", "x")
```

```
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    myfile = open("D:\\routers2.txt", "x")
PermissionError: [Errno 13] Permission denied: 'D:\\routers2.txt'
```

The solution for this issue is either to change the location where you want to create the new file to another drive or to the Desktop, **or** to open up another Python interpreter and run it as an administrator, this time. To do this, just right-click on the interpreter's icon and select **Run as administrator**, then click Yes. Now, you will be able to complete your file creation on the D: drive.

Python 3 Files - Writing & Appending

We've just seen how to open and read a file. Now, it's time to add some data to the file using the **"w"** access method. This method is used to open a file for writing **only**.

First of all, I should say that the **"w"** method also creates the file for writing, if the file doesn't exist and overrides the file, if the file already exists.

Let me prove this to you. We will create a file named "newfile.txt", using the open() function and the "w" method.

First, in case you don't trust me, let's check the D: drive again to see that newline.txt does not exist. Again, if you choose to store your files on the D: drive, too, then make sure you have the necessary admin privileges. Otherwise, you can always open a Python interpreter and run is as an administrator.

Now, let's create a new file object and a new file for writing.

```
newfile = open("D:\\newfile.txt", "w")
```

Good, we have the file where we want it to be and it is already open for writing, so, let's start writing to the file.

For this, we have the write() method in Python.

```
newfile.write("I like Python!\nDo you?")
```

23

So, what we did is we wrote this string to the file and we got back the number of characters that were written to the file, including spaces and new line characters.

Now, let's check that our text was indeed written to the file.

You may say "well, let's use the read() method for this". Actually, I'm sorry, but you cannot do this, because now we have the file open only for writing, not for writing and reading.

But, for the sake of the discussion, let's try to read it anyway.

newfile.read()

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

newfile.read()

io.UnsupportedOperation: not readable

So, we got an error saying that the file is not readable. Don't worry, I will show you how to open a file for both reading and writing, but, for now, let's check if our string was indeed added to the file, by simply opening the file in Notepad++.

Uhm, disappointment again! There's no text in the file. Why is that? The answer is because we didn't close the file after writing to it and thus our changes were not saved.

Let's go ahead and close the file and then check its content again. We will do this using the **close()** method. We will talk more about closing files in the next lecture, don't worry.

newfile.close()

Now let's open our file again, using Notepad++.

I like Python!

Do you?

Nice, this looks good.

So far, we used the **"w"** access-mode inside the open() function to create a new file for writing.

Now, let's add more data to this file. First, we should open the file for writing again, using the open() function, since we closed the file earlier. Also, don't forget to close the file again after you make all the desired changes.

```
newfile = open("D:\\newfile.txt", "w")

newfile.write("This is a great day for science!")

33

newfile.close()
```

Let's check the file again, using Notepad++.

Notice that the old data is not there anymore. We just have the string we entered a few seconds ago and that's it. This is because once you open an existing file for writing, the content of the file is deleted and Python overwrites the old data with the new data you input using the write() method.

Be very careful when using the open() function and the **"w"** access method!

Another method you can use to write data to a file is **writelines()**.

This method takes a sequence of strings as an argument and writes those strings to the file. Usually, this sequence is a list.

Let's use the same file as before, newfile.txt. Remember that Python will overwrite the data in the file, because we will open the file again for writing.

```
newfile = open("D:\\newfile.txt", "w")

newfile.writelines(["Cisco", "Juniper", "HP"])

newfile.close()
```

Now, let's check the contents of newfile.txt again, using Notepad++.

CiscoJuniperHP

Notice that the elements of the list we passed as an argument are written to the file, without any spaces or newlines in between them. This is how the **writelines()** method works.

You can also pass a tuple to the **writelines()** method.

```
newfile = open("D:\\newfile.txt", "w")

newfile.writelines(("Avaya", "Nortel", "Arista", "\n"))

newfile.close()
```

Let's check the file again, now.

AvayaNortelArista

The new data has been extracted from the tuple and then inserted into the file.

Now, what if you want to keep the data already written to the file and also write some new data at the end of the file? In that case, you should use the **a** access method. This stands for appending. This access mode adds new data to the end of the file, if that file exists. If it doesn't, then it creates the file.

So, we currently have the "AvayaNortelArista" string inside newfile.txt.

Let's open the file again, this time for appending and add a new string.

```
newfile = open("D:\\newfile.txt", a)

newfile.write("This string was appended!")
```

25

```
newfile.close()
```

Let's check the file again.

AvayaNortelArista

This string was appended!

And, indeed, the old string was kept into the file and the new one has been added to the end of the file.

I promised that I'll show you how to open a file for both writing and reading at the same time, to be able to read its contents immediately after writing.

The access methods to use in this case are: **r+**, **w+** and **a+**.

For instance, using **"w+"** will open a file for writing and reading at the same time and, if the file doesn't exist, it will create it.

Now, as we've already seen, if we open newfile.txt for writing using the **"w"** access mode and try to read it, we will get the *not readable* error again, right?

So, let's open the file using the **"w+"** access mode then.

```
newfile = open("D:\\newfile.txt", "w+")
```

```
newfile.write("something else")
```

14

Let's also jump to the beginning of the file to read properly.

```
newfile.seek(0)
```

0

```
newfile.read()
```

'something else'

Cool! So, you can now write to and read the file, at the same time.

Finally, let's close the file to save it and check if the string was indeed written to the file.

something else

Great! Now it's time to talk about file closing.

https://docs.python.org/3/tutorial/inputoutput.html

https://www.tutorialspoint.com/python3/python_files_io.htm

Python 3 Files - Closing. The "with" Statement

After opening a file using the open() function, it is always recommended to close that file, in order to avoid any issues related to operating system processes.

As we've seen so far, Python provides the **close()** method for closing files. After closing a file, you cannot use it anymore for reading or writing. Instead, you are required to open the file again to be able to do any of these operations.

You can also use a file object attribute to check if a file is closed.

Let's see this.

We still have newfile.txt from the previous lecture which is already closed.

```
newfile.closed
```

True

So, Python returns True if the file is indeed closed; otherwise, it returns False.

Another way to close a file is using the **with … as** statement. This was introduced in Python v2.5.

Using this statement, you don't have to worry about closing the file manually using the close() method. Instead, the file is closed automatically.

Also, the **with … as** statement creates a new file object and a new file, if you use the "w" access method.

Let's see how it works. I will write the code and then explain it.

```
with open("D:\\newfile.txt", "w") as f:
    f.write("Hello Python!")
```

13

So, first, I used the "with" keyword, followed by the open() function, which is used in the same way as before. Then, I added the "as" keyword and then the file object name, **f**.

The "as" keyword acts like an assignment operator for the file object.

After typing in the file object name, I also used a colon, like we did previously for **if** statements or functions and then, indented one level to the right from the "with" statement, you type in the code to be executed. In this case, I wanted to write a string to the file and that's it.

Now, let's check two things here. First, if the string was written to the file and second - that the file was automatically closed. So, let's open the file.

Hello Python!

Good, the string is there. Now, back to the Python interpreter.

f.closed

True

So, the file was indeed closed without us having to do it manually. Keep this method of closing files in mind and use it whenever necessary.

https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

Python 3 Regex - "re.match" & "re.search"

First, let's try to define a regular expression or regex, as it is usually abbreviated.

A regular expression is represented by a special syntax that helps you find and match a pattern inside a string.

Let me give you a very basic example. Let's say you have the following string, in the Python interpreter:

mystr = "You can learn any programming language, whether it is Python2, Python3, Perl, Java, javascript or PHP."

Now, what if you want to find the programming language in this string that starts with a capital "P" and has 4 letters. Or the programming languages starting with the substring "java", regardless of "J" being uppercase or lowercase; or maybe you want the last four characters preceding the digit "3". For all of those cases and many more you will use regular expressions.

During this lecture and the next one, we will talk about four methods that Python provides for dealing with regular expressions and text pattern matching.

Another thing that you should know is that all operations related to regular expressions are provided by the **re** built-in module. So, whenever you want to use regular expressions, you should first import this module using the **import re** statement and then prepend the name of the method with "re" and a dot.

Now, let's start discussing the first method, which is **match()**. The match() method searches the pattern you provide as its first argument, in the string you enter as its second argument, but **only at the beginning** of that string.

The general syntax for this method is the following.

a = re.match(pattern, string, optional flags)

…where **a** is called a match object, if the pattern is found inside the string, otherwise **a** will be **None**, if the pattern cannot be matched inside the string.

Now, let's return to our string **mystr** and use the match() method to see if the characters "You" are indeed positioned at the beginning of the string.

```
import re

a = re.match("You", mystr)

a
```

<re.Match object; span=(0, 3), match='You'>

So, you see that **a** is a match object, because Python has found the string given as a pattern, at the beginning of the **mystr** string. Now, if you would try to match a random string at the beginning of **mystr** you should get **None** returned. Don't trust me, let's verify this.

```
a = re.match("abc", mystr)

a

type(a)
```

<class 'NoneType'>

So, indeed, **a** is of type NoneType.

Ok, now, let's redefine **a** as it was before, so that it would return a match object again.

You can have the entire match returned by using the **group()** method on the match object. So, let's see this in action.

```
a.group()
```

'You'

As I said, Python returns the match it found in the string, according to the pattern we provided. In this case, the pattern was the substring itself, "You".

Now, let me give you an example that uses an **optional flag**.

Let's redefine **a** and add a flag that tells Python to ignore the case of the matched letters. This way, Python won't care whether the characters it is trying to match are uppercase or lowercase. It just matches them.

a = re.match("you", mystr, re.I)

a

<re.Match object; span=(0, 3), match='You'>

a.group()

'You'

So, this time, we provided the string "you" in all lowercase, although at the beginning of **mystr** we have "You", written with a capital Y. The **match()** method still returned a valid object, because we specified the **re.I** flag for ignoring the case, as the third argument of the method. There are other optional flags as well; I just wanted to give you an example, for now, because I find this particular flag very useful when working with regular expressions.

Now, for example, let's say we don't have the substring "you" at the beginning of the string. Instead, we have it somewhere in the middle.

mystr = "can learn any programming language, you whether it is Python2, Python3, Perl, Java, javascript or PHP."

a = re.match("you", mystr, re.I)

a

type(a)

<class 'NoneType'>

This time, the **match()** method returns a NoneType value, because the pattern we are searching for is not located at the beginning of the string.

To search for a pattern across the entire string, we will use the **search()** method. That's why this method is used more frequently than the **match()** method.

We will also start using regular expressions specific syntax, rather than just searching for an explicit substring inside a larger string.

The syntax for the **search()** method is similar to the syntax we've seen for the **match()** method. Let's see it.

```
a = re.search(pattern, string, optional flags)
```

Let's define another string now. We will use a more complex string which actually represents ARP entry from the ARP table of a network device. Don't worry if you don't know what an ARP entry really is, its meaning is not relevant to us right now. However, the diversity of characters inside such a string is very useful for our purpose, for the rest of this lecture.

```
arp = "22.22.22.1   0     b4:a9:5a:ff:c8:45 VLAN#222       L"
```

Just like the **match()** method, **search()** will return a match object if the pattern is found in the string, and a NoneType value if nothing is found. It's time to see it in action. Let me write this first and then we will study the syntax.

```
a = re.search(r"(.+?) +(\d) +(.+?)\s{2,}(\w)*", arp)
```

Ok, now, "what language is this?", you might ask. Well, this is the syntax of regular expressions and pattern matching. Don't worry, you'll get used to it.

Now, let's take this one bit at a time. Firstly, you can see that the arguments of the **search()** method are: first, the pattern to be matched and secondly, after the comma, the variable which holds the string in which to search for a pattern match. No optional flags, this time.

Another thing here is the lowercase **"r"** before the pattern. This means that the pattern should be treated like a raw string. A raw string is useful to avoid conflicts between the way Python recognizes an escape sequence and the way the **re** module does it. We will talk more about escape sequences soon.

Now, let's dissect the syntax for the pattern.

First, notice the parentheses surrounding some of the symbols. Any pair of parentheses matches the regular expression in between them and, as defined by Python's official documentation, indicates the start and the end of a group. If a match is found for the pattern inside the parentheses, then the contents of that group can be extracted using the **group()** method, applied to the match object, which in our case is called **a**.

Now, inside the first pair of parentheses we have a dot, a plus sign and a question mark. Each of these symbols has a meaning and matches something in particular.

In regex syntax, a dot represents any character, except a new line character. The plus sign means that the previous expression, which in our case is just a dot, may repeat one or more times. So, this group will look for at least one character, **any** character, doesn't matter if it is a letter, a number, a space or punctuation mark or whatever.

Now, remember that the plus sign - which, as I said, means **one** or more repetitions of the expression before it, and the asterisk - which means **zero** or more repetitions of the expression before it, are both greedy, meaning they try to match as much text as possible.

Let me show you what I mean. What match will Python find for the first group of parentheses, having the question mark as it is right now? To answer this, we will use the **group()** method with **1** as an argument, meaning the first group in the pattern.

```
a = re.search(r"(.+?) +(\d) +(.+?)\s{2,}(\w)*", arp)
```

```
a.group(1)
```

'22.22.22.1'

Now, let's remove the question mark from the first group and see what does the **search()** method return.

```
a = re.search(r"(.+) +(\d) +(.+?)\s{2,}(\w)*", arp)
```

```
a.group(1)
```

'22.22.22.1 '

Notice the difference? In the second case, the plus sign was so greedy that it also included two of the three whitespaces following it in the matching group; but, when we had the question mark, in the previous case, the match was made in a minimal fashion, matching as few characters as possible. Now, we've seen the role of the question mark in regular expressions. This means that the choice of matching in a greedy or minimal fashion is totally yours.

One more thing about this greedy behavior of the plus and asterisk signs. Up to which point is the plus sign acting greedy, in our example? Well, by looking at the initial string, we see that after the IP address, there are three whitespaces and then the 0 digit and the rest of the string.

That's why inside the regex pattern, I typed in a space after the closing parentheses. My intention was to tell Python that I want to match a group until a Space is encountered.

When the question mark was inside the parentheses, Python understood that it should match any character, one or more times, until the first Space character occurs, so in a non-greedy way.

That's why the matching stopped right before the first Space and all we got was just the IP address.

When I didn't use the question mark, Python understood that it should match any character, one or more times, as greedy as possible, until the last Space character is encountered.

That's why the matching stopped right before the last Space character. Since the matching has been done in a greedy way this time, we got the IP address followed by two Space characters; that's because Python stopped matching right before the third Space. So, notice that for a correct match result, you should be careful at both the greedy behavior and at the characters following the group, because they act as limits or borders of the pattern you want to match.

Now, moving beyond the first group, we have the Space we talked about, followed by a plus sign. This means that, looking at the initial string, we are expecting one or more Space characters after the IP address. We know there are actually three Spaces there, but it is safer to use one Space and a plus sign, for future compatibility. Maybe you use this regular expression inside a test automation suite and the output of some command will slightly change at some point into the future, and, instead of three Spaces, the new output will only have two Spaces, in that spot. You don't want to have failures into your scripts and tests, caused by a fixed number of spaces defined in your regex pattern. I've been through this and, trust me, being somewhat flexible with your pattern matching in Python can save you from a lot of headaches.

Now, let's look at the second group in the pattern. We have a **\d** there inside parentheses. This is called a special sequence in Python and it means any decimal digit, so, any digit from 0 to 9. Python will expect a single digit here, so let's verify this with the **group()** method, using group(2).

a.group(2)

'0'

And, indeed, we have the single digit we were expecting to match, according to the second group; that it this 0 right here.

Next, we have another sequence of Spaces, up to the first non-Space character, which is "b". After this sequence, we have a group similar to the first group we previously discussed at large. Let's say we want this group, which will be group number 3, to match a substring consisting of both the MAC address and the VLAN number. And the Space between the two, implicitly.

For this, we use the same *dot - plus - question mark* syntax, but, we should also specify the limit up to which we should consider this group; the border; so, that would be the first character or sequence of characters at which Python should stop matching this group.

Of course, we could use the same syntax as before, with the single Space character and the plus sign, but that wouldn't be fun, right?

Instead, we can use something even more interesting. First, another special sequence, **\s**, which matches any whitespace character, whether it is a Space, a Tab or a new line character (\n). Secondly, I used a number followed by a comma, in between curly braces. This means Python should expect two or more occurrences of the pattern preceding the curly braces - a whitespace, in our case. Remember that not typing the comma inside the curly braces would tell Python that it should expect **exactly** two repetitions of the previous pattern, meaning the Space.

If we would match just a single repetition of a whitespace here, the matching would stop at that one Space character in between the MAC address and the word "VLAN". Instead, by specifying two or more whitespaces, Python will know that it needs to stop the matching right after "222", cause that's where a lot of spaces reside. The final result for this group would be the substring starting at "b" and up to the last "2", because that's where two or more whitespaces are located; that's actually the border or limit we defined in the pattern.

Finally, the **\w** special sequence followed by an asterisk matches zero or more occurrences of any word character, meaning letters - lowercase **a-z** and uppercase **A-Z**, numbers **0-9** or the underscore character. This means that after the two or more whitespaces, we should match a word character and that is, of course, the letter "L".

Now, it's time to check out all the groups we have defined inside the pattern:

a.group(1)

'22.22.22.1'

a.group(2)

'0'

a.group(3)

'b4:a9:5a:ff:c8:45 VLAN#222'

a.group(4)

'L'

Great! Also, remember that **a.group()** without any arguments and **a.group(0)** both return the same thing - the entire string. Let's verify this.

a.group()

'22.22.22.1 0 b4:a9:5a:ff:c8:45 VLAN#222 L'

a.group(0)

'22.22.22.1 0 b4:a9:5a:ff:c8:45 VLAN#222 L'

Another method we can use when matching patterns is the **groups()** method, which returns all matches found in a given string, in the form of a tuple, where each match is an element of that tuple.

a.groups()

('22.22.22.1', '0', 'b4:a9:5a:ff:c8:45 VLAN#222', 'L')

I think this video is long enough, now. In the next one, we will discuss two other very useful regex methods, **findall()** and **sub()**.

Python 3 Regex - "re.findall" & "re.sub"

The third method used with regular expressions is **findall()**. I think this is, by far, the most used method, because instead of returning a match object, like **match()** and **search()** did, it returns a list, where each element is a pattern that was matched inside the target string.

Remember that the **findall()** method, as its name implies, returns a list of **all** the patterns that were matched.

To see this, let's consider the same target string and let's say we want to match the IP address using regex syntax and the **findall()** method.

```
arp = "22.22.22.1   0     b4:a9:5a:ff:c8:45 VLAN#222       L"
```

The IP address we want to match consists of two digits in the first octet, then a dot, then two digits again, then another dot, then two digits again, another dot and finally a single digit.

Let me write a solution to match this pattern and then I'll explain it in detail.

```
a = re.findall(r"\d\d\.\d{2}\.[0-9][0-9]\.[0-9]{1,3}", arp)
```

Looks awfully complicated, but I think it's not.

First, I typed in two **\d** special sequences, meaning Python should expect two consecutive digits.

Next, I used a backslash and a dot; this is called character escaping.

Because a dot, in the world of regular expressions, means any character except the newline character, as we've seen in the previous lecture, in order to match an actual dot in the string, the dot character itself, you are required to escape it using a backslash. This also is valid for question marks, plus signs, parentheses and any other characters that have a special meaning in regular expressions.

Then, instead of using two **\d**'s, I used a single **\d** and the number 2 inside curly braces, which means the previous character, any digit, 0 to 9, should

occur two times only. Then, another escaped dot, to represent the actual dot in the IP address.

For the third octet of the IP address I used a set of characters, as it is called in the regex world, which defines a range of characters or a character class that should be expected at that particular location in the pattern. In our case, we have this character class doubled, because we are expecting two digits again.

Then, another escaped dot character and, finally, I used the same range of characters, [0-9], which this time is expected to occur between 1 and 3 times; this means either 1, 2 or 3 times.

Let's get one thing clear. I could've used just one of these equivalent notations for representing a group of digits, for instance **\d\d**, but, first of all, it would've been kind of boring and, secondly, I wanted you to learn several ways in which you can represent a number or a set of consecutive digits, inside a pattern. It never hurts knowing how to solve the same problem in multiple ways.

The final result of the **findall()** method is going to be a list having a single element, the IP address matched by the regular expression. Let's check this.

a

['22.22.22.1']

type(a)

<class 'list'>

So, as I previously said, we don't have a match object returned this time; instead, we have a list.

We can also use grouping with the findall() method, by enclosing the groups we want to extract in between parentheses.

Let's consider the same initial string and almost the same pattern as an argument to the findall() method, this time grouping each IP address octet inside parentheses.

a = re.findall(r"(\d\d)\.(\d{2})\.([0-9][0-9])\.([0-9]{1,3})", arp)

Be careful to leave the escaped dots **\.** between the octets outside parentheses.

Let's see the result, now.

a

[('22', '22', '22', '1')]

This time, Python returns a list of tuples, where each tuple element is a group from within the pattern. If we would've had two IP addresses inside the initial string, then we would've ended up with a list containing two tuples.

Actually, let's check this by randomly inserting another IP address in the initial string. So, I'll add 10.10.10.10 at the end of the string, just for fun.

arp = '22.22.22.1 0 b4:a9:5a:ff:c8:45 VLAN#222 L 10.10.10.10'

Now, let's use the **findall()** method again to find all the matching patterns inside the string.

a = re.findall(r"(\d\d)\.(\d{2})\.([0-9][0-9])\.([0-9]{1,3})", arp)

a

[('22', '22', '22', '1'), ('10', '10', '10', '10')]

And, indeed, we got a list of two tuples, as expected. Cool!

Lastly, let's talk about the fourth method you should know about, when using regular expressions - **sub()**. This method simply replaces all occurrences of the specified pattern in the target string with another string that you provide as an argument. Let's see this in practice.

Let's consider the same initial string and replace all the digits inside the string with 7. For this, we should apply the **sub()** method, by specifying the pattern to be replaced, as the first argument, then the character replacing each occurrence of the pattern, as the second argument and, finally, the variable pointing to the original string, as the third argument. So, let's check this out.

b = re.sub(r"\d", "7", arp)

b

'77.77.77.7 7 b7:a7:7a:ff:c7:77 VLAN#777 L 77.77.77.77'

You can see that each digit that was matched by the **\d** special sequence, was replaced with 7. Great!

I know that the regular expressions topic is a hard topic to digest, at first. To better understand it and also learn additional ways of matching patterns, my only advice is to get into the Python interpreter and start playing around with patterns and specific methods. Sadly, there is no better way of grasping and mastering this topic, than to practice as much as you can.

https://docs.python.org/3/howto/regex.html

https://docs.python.org/3/library/re.html

Python 3 Classes - Objects

Apart from what we've seen until now, Python also has an object-oriented approach. Up to this point, we have seen one way of programming in Python, using functions.

Object-oriented programming is based on classes, methods and objects. We will analyze what each of these terms means in this lecture and the next one.

In short, a **class** is a data type containing its own variables, attributes and functions (which, by the way, in object-oriented programming are called **methods**). A standard definition of a class would tell you that a class is like a blueprint for creating objects.

An **object** may be regarded as an instance of a defined class and the attribute values for a particular object define the state of that object.

Another term that is very much used when discussing classes is **inheritance**. This means that a new class may inherit all the names and functionalities from an existing class. We will talk more about inheritance in the next lecture.

Now, let's take this one step at a time. First, we should see how to define a class. I will create a new file on my D: drive and write the code inside this file. I will name the file myclass.py.

In order to properly define a class, you will always use the **class** keyword, then type in the class name.

Now, be careful here because the convention is to use camelcase for class names. This means each word in the name of the class will be capitalized and, also, no spaces are allowed. Actually, except the camelcase rule, all the rules regarding variable and function names apply to class names, as well.

So, let's name our class simply **MyRouter**.

After the name of the class, in between parentheses, you should type in the word "object", all lowercase. This is the new style of defining classes, starting with Python version 2.2.

The thing you should keep in mind here is that if a class doesn't inherit from another class, then you should always type in the word **object** inside parentheses, when defining a class. This is a default setting and it means

that this class inherits from a default class named **object**. I know this may seem confusing, so, we won't get into any more details on this topic. Just don't forget to add that word, **object**.

So, we have **class MyRouter(object)** then, as for any other block of code we've seen so far, we'll type in a colon.

class MyRouter(object):

On the next row, using one level of indentation, of course, we shall input the content of the class. As with functions, on the first row after the class definition you can type in a documentation string or docstring, in between quotes, to provide a hint about that class' functionality. So, let's enter some text in between double quotes.

"This is a class that describes the characteristics of a router."

Following the optional docstring, the first thing you define inside a class is the special __**init**__ method, also called a class constructor. The word **init** will be preceded and followed by double underscore. This is the way Python identifies a special method.

def __init__(self, routername, model, serialno, ios):

The role of __**init**__ is to initialize some variables and the method is called whenever you create a new instance of the class in which it resides. Actually, it is the first code that is executed whenever you create a new instance of the class.

Any special method or regular method within a class is defined using the **def** keyword, as you do with regular functions. The difference here is that each time you define a method inside a class, the first parameter inside the parentheses is **self**. You have to remember to always input this word as the first parameter of every class method.

self is no more than just a reference to the current instance of the class.

Now, after typing in **self**, you define any other parameters that you want to be defined and initialized whenever you create a new instance of the class in which it resides.

In our case, we want to define some parameters that characterize our router, so I have added **routername**, **model**, **serialno** and **ios**.

Now, let's define the object or instance attributes we need to describe the router, according to the parameters of the __**init**__ method. Remember that **self** is used to point out that we are referring to the current instance of the class. The next lines of code will be again indented under the definition of the __**init**__ method.

```
self.routername = routername

self.model = model

self.serialno = serialno

self.ios = ios
```

That's how you define object attributes. Now, let's also define a new method inside this class, that will do nothing more than just print out the attributes and concatenate the model of the router with the manufacturing date. The definition of this new method will sit at the same level of indentation as the definition of the __**init**__ method.

```
def print_router(self, manuf_date):
```

Notice that **self** is again inserted as the first parameter. You should do this for every method you define inside a class. Next, inside the method, again indenting our code one level to the right, we enter the **print()** functions we need.

```
print("The router name is: ", self.routername)

print("The router model is: ", self.model)

print("The serial number of: ", self.serialno)

print("The IOS version is: ", self.ios)

print("The model and date combined: ", self.model + manuf_date)
```

Now, let's see what's the deal with the objects we talked so much about.

An object is actually an instance of the class. You can create as many objects as you want or need, by simply calling the class name and entering the arguments required by the __**init**__ method, in between parentheses; all of them, except **self**, which is automatically passed by Python.

First of all, let's copy and paste the code inside the file into the Python interpreter.

Now, let's create our first object.

router1 = MyRouter('R1', '2600', '123456', '12.4')

router1

<__main__.MyRouter object at 0x05713AF0>

Indeed, we see that we have created an object. Python confirms it.

Now, what can you do with this object?

First, you can access each of its attributes. Let's see how.

router1.model

'2600'

router1.ios

'12.4'

router1.serialno

'123456'

router1.routername

'R1'

Cool! What else?

We can also access the method we defined inside the class, **print_router()**. The method borrows the attributes defined in the **__init__** method and uses them within its own block of code.

Also, notice that we should specify a value for the **manuf_date** parameter, when calling this method. Again, **self** is invoked by Python in the background, so you shouldn't worry about it.

To use the object for calling the method we'll do the following.

router1.print_router("20150101")

The router name is: R1

The router model is: 2600

The serial number of: 123456

The IOS version is: 12.4

The model and date combined: 260020150101

Now, let's create another object and do the same operations, just to prove we can create multiple objects.

router2 = MyRouter('R2', '7200', '101010', '12.2')

router2.model '7200'

router2.ios '12.2'

router2.serialno '101010'

router2.routername 'R2'

router2.print_router("20150202")

The router name is: R2

The router model is: 7200

The serial number of: 101010

The IOS version is: 12.2

The model and date combined: 720020150202

Great! You can also change an attribute of an object, like this:

router2.ios

'12.2'

router2.ios = "12.3"

router2.ios

'12.3'

Python also provides some functions to play around with object attributes.

Let's see them.

First, we have **getattr()** for getting the value of an attribute. Let's use it on the **router2** object.

getattr(router2, "ios")

'12.3'

We can also set a new value for an attribute, using **setattr()**.

setattr(router2, "ios", "12.1")

getattr(router2, "ios")

'12.1'

Another function you can use is hasattr(), to check whether an object attribute exists or not.

hasattr(router2, "ios")

True

hasattr(router2, "ios2")

False

Finally, you can delete an attribute using the **delattr()** function.

delattr(router2, "ios")

Now, let's check if the object still has this attribute.

hasattr(router2, "ios")

False …so, indeed, it doesn't have that attribute anymore.

Let's also see what error does Python return when trying to get an non-existent attribute.

getattr(router2, "ios")

AttributeError: 'MyRouter' object has no attribute 'ios'

So, as expected, we got an AttributeError saying that this MyRouter instance has no attribute named "ios".

We can also verify if an object is an instance of a particular class. This is useful especially when you have multiple classes and objects created and you want to keep track of them. Let's see this in action for router2.

isinstance(router2, MyRouter)

True

And now, we have the confirmation that the object named **router2** is indeed an instance of the MyRouter class.

In short, this is how object-oriented programming or, simply, OOP, works. We won't use classes and OOP to build the applications in this course, although we could do it. It's really a matter of choice and, to be honest, I like the classical way of programming more.

Next, we'll have a look at class inheritance.

Python 3 Classes - Inheritance

We've seen that when creating a new class, we should type in **object** in between parentheses, when defining the class. We can say that the new class inherits from the default class, called **object**.

But, for the sake of modularity and the DRY best practice in Python - this stands for Don't Repeat Yourself - we can also create a class that inherits the attributes and methods from an already existing class. This is called class inheritance.

In this case, the class inheriting the attributes and methods is called a child and the class the child inherits from is called - you guessed it - a parent.

To tell Python that you want to inherit from a certain class, you should type in the parent's name in between parentheses, instead of **object**, when you define the child. Let's see this.

We still have the old class, MyRouter, in memory, from the previous lecture. This will act as the parent. Now, let's create the child and name it MyNewRouter.

```
class MyNewRouter(MyRouter):
```

Now, let's talk about the __**init**__ method. You are allowed to skip the definition of an __**init**__ method for the child class, because the new class MyNewRouter actually extends the MyRouter class and thus the __**init**__ method is inherited from the parent.

But, what if you want to inherit all the attributes from the parent class and also define new ones inside the child class? Then, you are required to redefine the __**init**__ method, tell Python that you want all the attributes in the parent to be available inside the child, as well and then enter the new attributes.

I will write the __**init**__ method for this class and then I'll explain every bit of code.

```
class MyNewRouter(MyRouter):
    def __init__(self, routername, model, serialno, ios, portsno):
        MyRouter.__init__(self, routername, model, serialno, ios)
```

```
        self.portsno = portsno
```

So, first, I defined the **__init__** method using the **def** keyword, as usual, then, in between parentheses, I have inserted the parameters as they were defined in the parent class, but, since I also want to define a new attribute for this class, namely the number of ports on the device, I have added the **portsno** parameter, at the end.

On the next row, I called the **__init__** method from the parent class, by typing in the parent class name, then a dot and then the **__init__** method, having the same parameters in between parentheses. I have actually imported, so to say, the **__init__** method's attributes from the parent class.

Finally, I defined the new attribute, specific to this class only, called **portsno** and that's it.

Then, I defined an additional method inside the child class, that simply concatenates a string given as argument when calling the method and the value of the **model** attribute, inherited from the parent.

```
def print_new_router(self, string):
    print(string + self.model)
```

Now, let's verify the inheritance process, by pasting the child class into the Python interpreter, where we still have the parent defined, and then creating an instance of the child class.

```
new_router1 = MyNewRouter("newr1", "1800", "111111", "12.2", "10")
```

Notice that this time, when invoking the child class, you should also add a value for the number of ports, as the last argument inside parentheses. I added this "10" right here, in my case.

Let's see the attribute for the number of ports.

```
new_router1.portsno
```

'10'

Now, let's see if we can access the attributes from the parent class using this instance of the child class.

```
new_router1.model
```

'1800'

new_router1.ios

'12.2'

Great! Now, let's access the method from the parent class using the same instance of the child class.

new_router1.print_router("aaabbbccc")

The router name is: newr1

The router model is: 1800

The serial number of: 111111

The IOS version is: 12.2

The model and date combined: 1800aaabbbccc

Let's do the same thing, this time using the method defined in the child class, called **print_new_router()**.

new_router1.print_new_router("aaabbbccc")

aaabbbccc1800

Two more things on class inheritance. First, a child class can inherit from two or multiple parents at the same time; the difference is that when you define the child class, you should enter all the parents in between parentheses, like this:

class ChildClass(Parent1, Parent2, Parent3):

The second thing I'll show you is a function that checks if a certain class is the child of another class or not.

Let's use it with the classes we defined so far.

issubclass(MyNewRouter, MyRouter)

True

This will return either True or False; in our case it is True, obviously.

Ok, I think you now have a pretty good understanding about classes, objects and inheritance. Of course, the object-oriented programming topic is much more complex and cannot be exhausted in just a few lectures or even an entire course, to be honest. Next, let's have a look at more advanced Python programming topics. See you in the next lecture!

https://docs.python.org/3/tutorial/classes.html

Python 3 - List / Set / Dictionary Comprehensions

The concept of comprehensions allows us to build sequences from other sequences, in an easy, quick and elegant way. Let's start working with list comprehensions, because they are, perhaps, the most widely used. They are actually a quick alternative to something we've already seen and used thus far.

Let's say we have a list and want to iterate over it and perform some sort of action on each of its elements.

We would use a **for** loop, right? Let's build a list of all the elements in the range generated by range(10), having each element raised to the power of two. First, let's define an empty list. This is where the results are going to be appended.

```
list1 = [ ]
```

Now, it's time to iterate over range(10), do the math and append the results to list1.

```
for i in range(10):
    result = i ** 2
    list1.append(result)
list1
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Nice! We got the result we were looking for.

But, what if we could write this whole block of code on a single line? Python allows you to do this, using a list comprehension. Let's see it in action.

```
list2 = [x ** 2 for x in range(10)]
```

So, we've created list2, then, in between square brackets - because we are trying to build a list - we first write the action to be performed on each element and then, on the same line, without any colons, commas or indentation, we write the loop structure, as we would in the old fashion way. Pretty cool, I think! Now, let's see the result. It should be the same as list1.

list2

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81] …and indeed it is.

But, what if you want to also add a conditional statement inside this looping construct? Let's say you want only the numbers greater than 5 to be raised to the power of 2. The solution is to enter the **if** statement right after the **for** statement, at the end of the list comprehension. Let's do this.

list3 = [x ** 2 for x in range(10) if x > 5]

So, as you can see, we have the **for** statement in the middle; on the left side, we have the action to be performed and on the right side we have the conditional statement.

Again, no commas or colons are needed. Let's see the new list, now.

list3

[36, 49, 64, 81] …Great! Just what we wanted.

Now, let's see set and dictionary comprehensions.

The way to build and write them is the same as for list comprehensions, the only difference being the enclosing character for the comprehension.

For both set and dictionary comprehensions, we use the curly braces as enclosing characters, instead of square brackets.

Let's better see an example for each of them, starting with set comprehensions.

set1 = {x ** 2 for x in range(10)}

set1

{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

type(set1)

<class 'set'>

So, indeed, we built a set using a set comprehension. Cool! Now, let's do the same, for dictionaries.

Let's remember that dictionary elements are actually key-value pairs, where the key is separated from the value by a colon. So, this is the format we should use for dictionary comprehensions, as well.

dict1 = {x: x * 2 for x in range(10)}

dict1

{0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18}

type(dict1)

<class 'dict'>

Now, we have a dictionary built from scratch, using a dictionary comprehension.

Of course, you can also have nested **for** loops or **if** statements and you can do many other types of operations or iterations inside a comprehension structure. I just wanted you to learn a basic set of operations, so that you get familiar with the comprehension concept itself.

I think that list, set and dictionary comprehensions can be a useful and elegant way to write some code, especially when you want to perform simple **for** loops and save some space within your program. So, keep them in mind! I can almost guarantee that you will need them at some point in your programming adventure.

https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

https://docs.python.org/3/tutorial/datastructures.html#nested-list-comprehensions

Python 3 - Lambda Functions

Python allows us to create small, anonymous functions at runtime, called lambda functions.

Lambda functions are a great Python tool, enabling us to quickly insert some functionality into our code, especially in places that would not allow the definition of a regular function; you know, using the **def** keyword, the **return** statement and all that.

Unlike regular functions, you can embed a lambda function wherever you want inside your code and you don't need a **return** statement to output something from that function.

Although not mandatory, you can assign a lambda function to a variable, so you can use that variable further into the program. If you don't, then the lambda function will run right there where it has been created, then the result is thrown away.

Without further ado, let's see the general syntax for lambda functions and then some examples.

lambda arg1, arg2, … , arg n: an expression using the arguments

Let's do a couple of examples in the Python interpreter and let's assign the lambda function to a variable in each example.

a = lambda x, y: x * y …and that's the lambda function definition

Now, let's check that this is indeed a function.

a

<function <lambda> at 0xb742cf44>

type(a)

<class 'function'>

Great! Now, let's call our function and provide it with some arguments to perform the multiplication.

a(2, 10)

20

a(100, 100)

10000

So, it works! See how easy it is? Now, let's do another example.

Hold on tight, this will be a nice one!

Let's combine a lambda function and a list comprehension inside which we'll have a nested **for** loop. But, first, let's do this old fashion, using a regular function and the classic **for** loop syntax.

Let's say that we want to obtain a list containing all the results of multiplying each element in the range generated by range(10) - so, 0 through 9 - with each element in the range generated by range(5) - meaning 0 through 4 - and concatenate this list with a list passed as an argument to the function.

How would we normally do this?

We would create a function, a **for** loop nested inside another **for** loop and return the list of results concatenated with the list we pass as an argument. So, let me write this first.

```
def myfunc(mylist):
    list_xy = [ ]
    for x in range(10):
        for y in range(5):
            result = x * y
            list_xy.append(result)
    return list_xy + mylist
```

So, this function takes a single parameter, a list. We then initialize an empty list to store the future results of multiplying the elements of the two ranges. Each result is going to be saved using a variable, called "result", and then appended to the empty list we just created. Finally, the function returns the final result of concatenating the two lists.

Now, let's call the function and use a list as an argument.

myfunc([100,101,102])

[0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 2, 4, 6, 8, 0, 3, 6, 9, 12, 0, 4, 8, 12, 16, 0, 5, 10, 15, 20, 0, 6, 12, 18, 24, 0, 7, 14, 21, 28, 0, 8, 16, 24, 32, 0, 9, 18, 27, 36, 100, 101, 102]

We got the expected result - a list containing the results of multiplying the elements of the two ranges, at the end of which we added three elements: 100, 101 and 102.

Now, can we achieve the same result with just a single line of code, using the lambda function and list comprehensions? Sure, we can!

Let me type in the line of code first.

b = lambda mylist: [x * y for x in range(10) for y in range(5)] + mylist

So, after the **lambda** keyword, we have a single parameter, named "mylist", as we did with the regular function. Then, instead of initializing a list, writing the **for** loops and appending the results to that list, we used a list comprehension with a **for** loop nested inside another **for** loop, without needing colons or indentation. Finally, we added our own list to the result of the list comprehension and that's it. Now, let's check the result.

b([100,101,102])

[0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 0, 2, 4, 6, 8, 0, 3, 6, 9, 12, 0, 4, 8, 12, 16, 0, 5, 10, 15, 20, 0, 6, 12, 18, 24, 0, 7, 14, 21, 28, 0, 8, 16, 24, 32, 0, 9, 18, 27, 36, 100, 101, 102]

As you can see, we achieved the same result, using a lambda function and a list comprehension, using only one line of code instead of seven lines.

Lambda functions are widely used in conjunction with three Python functions: map() and filter(). We will discuss each of them in the next lecture.

https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions

Python 3 - Map() and Filter()

Let's talk about two other useful functions that Python provides and that are very often used along with lambda functions.

First, we have the **map()** function, which takes a function and a sequence as arguments and applies the function to all the elements of the sequence, returning an iterable object, as the result. The function taken as the first argument may be a previously defined function or a lambda function.

So, let's try this, first using a predefined function that we will create.

Let's define a function that takes a parameter and multiplies it by 10.

```
def product10(a):
    return a * 10
```

Now, let's create a range object.

```
r1 = range(10)
```

Next, let's use the **map()** function to apply the **product10()** function to each element in range(10).

```
map(product10, r1)
```

<map object at 0x05963490>

The result, just I said, is an iterable object. In order to get a list, instead of this object, we can simply apply the **list()** function on it.

```
list(map(product10, r1))
```

[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

And, indeed, this returns a list, where each element is the result of multiplying each of the elements in range(10) with 10.

Let's do the same thing, this time using a lambda function.

```
map((lambda a: a * 10), r1)
```

<map object at 0x059634D0>

```
list(map((lambda a: a * 10), r1))
```

[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

Notice that the same result is achieved, this time using a single line of code.

By the way, note that the lambda function body may or may not be enclosed by parentheses. You should better use parentheses, though, for better readability.

The **filter()** function also takes a function and a sequence as arguments and its role is to extract all the elements in the sequence, for which the function returns True. Let's go ahead and use a lambda function and the same range, **r1**.

filter(lambda a: a > 5, r1)

<filter object at 0x05963490>

list(filter(lambda a: a > 5, r1))

[6, 7, 8, 9]

So, this is pretty intuitive, I think. The lambda function returns True if, applied on **r1**, it finds elements greater than 5. Then, the **filter()** function returns an object referencing those elements only.

Therefore, remember these two functions that Python provides to help you deal more easily with different tasks. They may come in handy, some day.

https://docs.python.org/3/library/functions.html#map

https://docs.python.org/3/library/functions.html#filter

Python 3 - Iterators and Generators

It's now time to talk about a slightly advanced topic in Python - iterators and generators.

Let's start with iterators. We've talked about iterators and iterable objects a couple of times, thus far in the course. Now, we're going to dive a bit deeper into the topic.

According to the official definition, an iterator is an object which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation.

In Python, we have the **iter()** function available, that takes the iterable object as an argument and returns an iterator. Now, because I'm not a big fan of definitions and long speeches on theoretical topics, let me get into the Python interpreter and show you how we can use an iterator.

First, let's consider a list. Let's say:

my_list = [1, 2, 3, 4, 5, 6, 7]

Now, using the knowledge we have about iterations, how would we traverse this sequence, in order to get each element printed out? Well, using a **for** loop, right? Let's say:

```
for element in my_list:
    print(element)
```

1

2

3

4

5

6

7

Ok, simple enough. Now, let's use the **iter()** function I was talking about, to create an iterator. We can do this pretty simple, just by typing in the name of the object and then use the **iter()** function.

```
my_iter = iter(my_list)
```

Now, let's confirm that this is indeed an iterator, of course, using the same old **type()** function.

```
type(my_iter)
```

<class 'list_iterator'>

Ok, there you have it! You just built your first iterator.

Now, how can we use this to our advantage?

Well, here's where we get to learn a new function, which is called **next()**. To use this function, we just have to apply it on our iterator, so:

```
next(my_iter)
```

1

```
next(my_iter)
```

2

```
next(my_iter)
```

3

```
next(my_iter)
```

4

```
next(my_iter)
```

5

```
next(my_iter)
```

6

```
next(my_iter)
```

7

```
next(my_iter)
```

Traceback (most recent call last):

File "<pyshell#63>", line 1, in <module>

next(my_iter)

StopIteration

Notice that each time we apply the **next()** function to our iterator, we get the next element in the list. Pretty cool! Now, after exhausting all the elements in the sequence, we got StopIteration, meaning we have reached the end of the sequence.

Now, why do we use iterators? Well, first, because they are simple and elegant to write and implement, as you've just seen. Secondly, they save computer resources, because you can access only the next element in the list or string or whatever sequence you have, without having to keep all the elements in memory and then have them returned.

Other things can be said and tested in relation to iterators. I just wanted you to know about them and see how to use them in your Python code.

Now, let's have a look at generators.

The formal definition says that a generator is a special routine, that can be used to control the iteration behavior of a loop. Similar to a function, a generator is defined using the **def** keyword, can have parameters and it can be called. As opposed to a function, which returns an entire array with all the values, a generator function yields the values, one at a time. That's why it can be also called a resumable function. You can traverse a sequence up to a certain point, get the result and then actually suspend the execution. Later, you can return to that point and resume the execution.

This is a great advantage, because this way less memory is required. Let's see how a generator function works. Let me write such a function and then we will discuss it.

```
def my_gen(x, y):
    for i in range(x):
        print("i is %d" % i)
```

```
        print("y is %d" % y)
        yield i * y
```

So, as I said, you use the **def** keyword to define the generator function, which takes two parameters, x and y. Then, using a **for** loop, you iterate over range(x), print the values of **i** and **y** and then yield the result of multiplying each element in range(x) with y. The **yield** keyword suspends the function execution and sends a value back to the caller. It also saves the state of the execution, so when the execution is then resumed, the control picks up after the **yield** statement. Very interesting. Let's see this in action.

Let's create a generator object, first, and then call it.

```
my_object = my_gen(10, 5)
```

Now we should check that we have indeed created a generator.

```
my_object
```

<generator object my_gen at 0x05944CB0>

```
type(my_object)
```

<class 'generator'> …ok, so, we have a created a generator. Good!

Now, let's use the next() function to manually get the next element returned by the my_gen() function, so, each result of **i * y**, one at a time.

```
next(my_object)
```

i is 0

y is 5

0

Ok, so, for the first run of the **next()** function we got "i is 0", because this is the first element in range(10); remember we made **x** equal to 10 when we created the generator object, so range(x) becomes range(10); then, we have "y is 5" and, finally, the result of multiplying them is, of course, 0.

Now, let's run next() again.

```
next(my_object)
```

i is 1

y is 5

5

This time, we got the results of the next iteration, where **i** is 1, **y** is still 5 and the final result is now 5.

Let's see all the iterations, one at a time, until range(10) is exhausted.

next(my_object)

i is 2

y is 5

10

next(my_object)

i is 3

y is 5

15

next(my_object)

i is 4

y is 5

20

next(my_object)

i is 5

y is 5

25

next(my_object)

i is 6

y is 5

30

next(my_object)

i is 7

y is 5

35

next(my_object)

i is 8

y is 5

40

next(my_object)

i is 9

y is 5

45

next(my_object)

Traceback (most recent call last):

File "<pyshell#80>", line 1, in <module>

next(my_object)

StopIteration

Of course, in the end, we get StopIteration, because all the elements of range(10), meaning 0 through 9, were exhausted. Pretty nice.

Finally, let's look at another type of generator construct, namely generator expressions.

They are very similar to the comprehensions you saw earlier in the course, only this time we use parentheses instead of square brackets.

Let's create a very basic generator expression and let's then see what we get as a result.

gen_exp = (x for x in range(5))

gen_exp

<generator object <genexpr> at 0x05BD6D70>

type(gen_exp)

<class 'generator'>

So, we get a generator object, which we can now use to extract each value in the range generated by range(5), one value at a time, using the **next()** function. Let's try it.

next(gen_exp)

0

next(gen_exp)

1

next(gen_exp)

2

next(gen_exp)

3

next(gen_exp)

4

next(gen_exp)

Traceback (most recent call last):

File "<pyshell#90>", line 1, in <module>

next(gen_exp)

StopIteration

…and since 4 is the last element in range(5), we got StopIteration. Cool!

I hope you enjoyed this lecture and learning new Python concepts. I'll see you in the next one, to discuss other awesome Python tools.

Python 3 - Itertools

Everyone, it's time to learn about a new and exciting Python module, called **itertools**.

This module includes a lot of functions for working with iterable data sets and we will see some of them in this lecture. I think this would be a great follow-up on the previous lecture about iterators and generators.

Let's get into the Python interpreter and see what's the deal with itertools. First, as always, we have to import the module, so we can have all of its functions available in our namespace. So, let's use:

```
from itertools import *
```

For this lecture, I've chosen five functions from the itertools module, to discuss and test in the Python interpreter.

You will find all other functions inside this module using a link I have attached to this lecture, so check out that link, for more information about itertools.

Now, let's get to work and let's consider two lists to be the iterable data sets on which to work on.

```
list1 = [1, 2, 3, 'a', 'b', 'c']
```

```
list2 = [101, 102, 103, 'X', 'Y']
```

The first function we're going to discuss is **chain()**. This function simply takes several sequences as arguments and chains them together. Let's see this.

```
chain(list1, list2)
```

<itertools.chain object at 0xb74bc4cc>

Ok, what we got back is an iterator object.

Now, to see its contents, we can either loop over it using a **for** loop and print each element or just feed it to the list() function. Let's test both methods.

```
for i in chain(list1, list2):
        print(i)
```

1

2

3

a

b

c

101

102

103

X

Y

Ok, so, this way, we have each element of the chained sequences printed out on the screen.

Secondly, we can just use the **list()** function applied on the **chain()** function.

```
list(chain(list1, list2))
```

[1, 2, 3, 'a', 'b', 'c', 101, 102, 103, 'X', 'Y']

…and, as expected, we got the chained sequence.

Next, the **count()** function returns an iterator that generates consecutive integers, until you stop it; otherwise, it will go on forever. That's why you should use it wisely, so you won't end up with an infinite loop and a huge computer resource consumption.

So, let me put this inside a **for** loop.

```
for i in count(10, 2.5):
    if i <= 50:
        print(i)
    else:
        break
```

10

12.5

15.0

17.5

20.0

22.5

25.0

27.5

30.0

32.5

35.0

37.5

40.0

42.5

45.0

47.5

50.0

So, let's translate this code in plain English: for each element in the sequence returned by the **count()** function, if that element is less than or equal to 50, then print it out to the screen. Otherwise, just break out of the loop. Notice that I passed two values as arguments to the **count()** function. The first value is the starting point of the sequence and the second value is the step to increment by. This way, we get the numbers between 10 and 50 inclusively, with a step of 2.5.

The next function that we're going to discuss is **cycle()**. This function returns an iterator that simply repeats the value given as an argument, infinitely. Again, if you use it inside a program, you have to find a way to break out of the infinite loop. Otherwise, your program will crash. Let's see it in action.

```
a = range(11, 16)
for i in cycle(a):
    print(i)
```

11

12

13

14

15

11

12

13

14

15

…

So, as you can see, we got an infinite loop which repeats the elements of the range, forever, theoretically.

Always remember to use **Ctrl+C** in the Python interpreter to break out of this never-ending loop.

Next, let's study the **filterfalse()** function. Remember the **filter()** function we've seen in the lecture about map() and filter()? Well, it works similarly to that function, with a small twist; filterfalse() will return the elements for which the function you provide as an argument returns False. Let's check this out; let me write this, first.

```
filterfalse(lambda x: x < 5, [1, 2, 3, 4, 5, 6, 7])
```

<itertools.filterfalse object at 0x05BD2370>

So, first you write the function name. Then in between parentheses, you enter two arguments; the first argument is the function you want to consider - I hope you remember lambda functions from earlier in the course.

The second argument is the iterable sequence you want to apply the function to.

So, the final result should be the elements of that list, for which the lambda function returns False, meaning all the elements which are **not** less than 5. Now, let's see if I'm right and let's apply the **list()** function to get the results.

list(filterfalse(lambda x: x < 5, [1, 2, 3, 4, 5, 6, 7]))

[5, 6, 7]

Ok, great, we got 5, 6 and 7, which is, indeed, the correct result.

Next, let's see another function, **islice()**. Remember list slices? Or string slices? This function basically does the same thing. You can even specify a starting point in the sequence, an end point and a step, just like we did with slices thus far, in the course.

First, let's consider the range generated by range(10), then convert it to a list and slice it in the old fashion way.

range(10)

range(0, 10)

list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

list(range(10))[2:9:2]

[2, 4, 6, 8]

So, we start the slice at the element positioned at index 2, we slice the list up to, but **not** including the element at index 9, with a step of 2. The result is this list, right here. Now let's do the same thing using islice().

islice(range(10), 2, 9, 2)

<itertools.islice object at 0xb74da874>

So, we got an iterator object.

Now, let's use the **list()** function to see the final result.

list(islice(range(10), 2, 9, 2))

[2, 4, 6, 8]

Ok, the result is identical, which is what we wanted, in the first place.

I hope you had fun with the itertools module. Don't forget to check out the link in the attachment, for further reading on this topic and additional functions.

https://docs.python.org/3/library/itertools.html

Python 3 - Decorators

Decorators are a powerful feature of Python, but what exactly is a decorator?

A decorator is actually a function that takes another function as a parameter and extends its functionality and behavior without modifying it. Think of a decorator like a candy wrapper. You take a candy and just wrap it using a nice wrapper. The candy itself is not modified in any way, but its functionality was extended, because now you can sell it and keep its cleanness and taste. Maybe it's not the perfect example, but I hope you understand it.

Now, to **make sure** you understand it, let's see a simple example of a decorator. I will write it inside a new text file first, for better readability, and then I will explain it and show you the results.

```
def my_decorator(target_function):
    def function_wrapper():
        return "Python is such a " + target_function() + " language!"
    return function_wrapper
```

Ok, so, as with any other function, you define a decorator using the **def** keyword. This decorator takes a function, called **target_function**, as a parameter - remember the candy from earlier. I will define that function in just a moment.

Then, inside the decorator, you define an inner function, a nested function, which I called **function_wrapper()**. This function returns the result of concatenating some strings. The string in the middle will be the candy, which is actually the result returned by the function I'm about to write. Finally, the decorator returns this inner function as a result. Now, let me write the target function, aka "the candy".

```
def target_function():
    return "cool"
```

Ok, simple enough, this function simply returns the string "cool". Nothing special here.

Now, let's "decorate" it, using our decorator. This is done using the @ (at) sign and the name of the decorator, right above the function definition, so:

```
@my_decorator
def target_function():
    return "cool"
```

And that's it! Finally, let's call the function.

```
target_function()
```

Ok, we should now copy these functions into the Python interpreter and see the results.

'Python is such a cool language!'

Nice! So, what we did is actually we fed our **target_function()** to the decorator, as a parameter, and the decorator did its job of concatenating the strings together.

Of course, this was a basic example of a Python decorator, but this concept should definitely be in your programming toolbelt.

I have attached a link to this lecture, if you're interested in further reading on this topic.

https://www.python.org/dev/peps/pep-0318/

Python 3 - Threading Basics

Every process running inside an operating system contains at least one thread and can further initiate multiple threads, which are then executed simultaneously, in the same process space. This is similar to executing multiple applications, at the same time.

In Python, threading can be used to run several function calls or other tasks concurrently, especially if those tasks involve some waiting times. This is the case for networking and web-related tasks, among others.

Threading is a large and advanced topic in Python and that's why we will discuss only the basics of threading in this course.

Python provides a specific module for threading purposes; this module is simply called **threading** and it is a built-in module.

Of course, before using this module, you should first import it using the same old **import** statement.

Now, we will use the **Thread** class from within the threading module, written with a capital T, prepended, of course, by the module name.

From within this class, the most useful and used methods are **start()** and **join()**.

The **start()** method simply starts or initiates the thread.

The **join()** method makes sure the program waits for all threads to terminate.

Ok, let's see a simple example of threading in Python and let's create a file in which to write our code. I will name this file mythreads.py and save it to the D: drive on my computer.

First, we should import the **threading** module. Also, we will need a module we haven't talked about yet, the **time** module, so, we will import this module, as well.

From within the **time** module, we will need the **sleep()** method, which basically interrupts the execution of the program for as many seconds as you enter in between its parentheses. So, for instance, **time.sleep(1)** will hold the execution for a second.

```
import threading

import time
```

I said that threading can be used to run multiple function calls at the same time, so, let's first define a function that prints something, then waits for three seconds and then prints something else.

I will use this simple function just to prove that, using threading, you can run multiple function calls simultaneously.

```
def myfunction():

    print("Start a thread")

    time.sleep(3)

    print("End a thread")
```

Now, it's time to configure threading but, first, let's set a goal for this application. We would like to run myfunction() five times, concurrently and, as a bonus, we would like the program to wait for all threads to terminate.

To achieve our goal, I will first create an empty list called **threads**.

```
threads = [ ]
```

Then, because we want five concurrent instances of the function to be executed, we can make use of a **for** loop and the **range()** function.

```
for i in range(5):
```

Now, let's use the Thread class form the **threading** module and tell it the target function to be executed, using the **target** argument.

```
th = threading.Thread(target = myfunction)
```

Next, it's time to start the thread and then also append each threading object to the **threads** list.

As I said at the beginning of this lecture, you will use the **start()** method to initialize a thread.

```
th.start()

threads.append(th)
```

Finally, using another **for** loop to iterate over the **threads** list, we will use the **join()** method, in order to instruct the program to wait for all threads to finish.

```
for th in threads:

    th.join()
```

Now, let's save the file and run it in the Windows command line, to see threading in action.

```
D:\Windows\System32>python D:\mythreads.py
```

Start a thread

Start a thread

Start a thread

Start a thread

Start a thread

End a thread

End a thread

End a thread

End a thread

End a thread

This clearly demonstrates that the function is called five times, concurrently, because it printed out the first string 5 times, then it stopped for 3 seconds and then printed the second string 5 times.

If we would've called the function 5 times without using threading, then what would've happened? Well, the first call would've printed the first string, then waited 3 seconds, then printed the second string. Then all the subsequent function calls would've done the same thing, so, a total of 15 seconds of interruption. Let's see this, by instructing the program to **not** use threading, but, instead, just call the function 5 times in a row.

```
for i in range(5):
        myfunction()
```

```
D:\Windows\System32>python D:\mythreads.py
```

Start a thread

End a thread

Start a thread

End a thread

Start a thread

End a thread

Start a thread

End a thread

Start a thread

End a thread

So, there it is - the difference between normal function calling and threaded function calling, which can be very useful in some scenarios and can save you a lot of time when running your applications.

I think you now have a pretty good idea about threading. Let's move on.

https://docs.python.org/3/library/threading.html

Python 3 - Coding Best Practices

Ok, let me give you some advice on the way you should write your applications, from what I've learned so far, as a programmer.

I will provide you with some basic best practices, which I have learned while writing Python code, over the past 5 years.

Let's go into the file from the previous lecture, **mythreads.py** and let me add a few things, first.

```
import threading

import time


def myfunction():

    "Function to be executed"

    print("Start a thread")

    time.sleep(3)

    print("End a thread")


#Define an empty list of threads

threads = [ ]


#Runs 5 concurrent sessions of myfunction()

for i in range(5):

    th = threading.Thread(target = myfunction)

    th.start()
```

```
        threads.append(th)


#Waiting for all threads to terminate
for th in threads:
        th.join()
```

As I said earlier in the course, one crucial thing to be careful about is indentation. You should always be consistent about indentation. I would advise you to use the Tab key, pressed once for one indentation level. I always use it in my code and haven't had a problem until now.

Another thing to keep in mind is that you should always write your **import** statements at the beginning of the file, before any other lines of code.

Now, maybe the biggest advantage of Python is the readability of the code; that's why you should keep your code as readable and as clean as possible.

For example, each time you create a variable and assign it a value, it's better to leave a space between the equal sign and the characters you type in on the left and on the right side of it.

Another thing that might make your code more readable is leaving a blank line between functions, classes or any other code blocks.

Docstrings might also help you when you have a large program and you want to have some help available. Feel free to add docstrings to functions, classes or even class methods.

If you feel the need for any comments anywhere inside your code, use the hash sign for single-line comments and triple quotes for multiline comments. They will help you 6 months later, when you return to your code. Trust me on that. You can't imagine how easy it is to forget what your own code does after several months or a year. Of course, this is applicable for larger programs.

Also, try to name your variables, functions or classes as suggestive as you can, thinking about what that particular function or class does. For instance, if you have a function that simply adds two numbers, name it **sumxy()**, instead of **function1()**. Ok, I know this is a very basic example, but you got the point.

I encourage you to read the entire list of best practices for coding in Python, using the link I have attached to this lecture.

Just keep in mind that keeping you code clean, readable and compliant will help you a lot in your coding adventures. Hope that helps!

https://www.python.org/dev/peps/pep-0008/

~ The End ~