PYTHON

Course Notebook
v1.0

Mihai Cătălin Teodosiu

Mihai Cătălin Teodosiu
Visit my LinkedIn profile at www.linkedin.com

First Release: July 2024

CONTENTS

About the author

I'm a Network Engineer (CCNP), QA Specialist (ISTQB) and Python Developer who decided to share his knowledge and skills with anyone looking to learn Python programming from scratch, in an easy-to-understand, learn-by-doing fashion, without the fancy wording and endless rambling and gibberish that most authors tend to include in their books and training courses.

My beginner-friendly teaching methods turned out to be very efficient for over 100,000 students who enrolled in my Python courses, published on various platforms.

From California to Fiji and from Norway to South Africa, I helped programming rookies become proficient in Python, upgrade their skills and nail job interviews.

Now I'm grateful for having the chance to help you advance your career, as well.

PYTHON 3 - FUNDAMENTALS

MEET PYTHON

Welcome to the first chapter of this notebook!

First of all, let's start by addressing some fundamentals about Python 3.

Generally speaking, Python is an interpreted, high-level programming language where you as a developer are writing the source code and then you use an interpreter that executes the statements you write. Python can be used for a plethora of applications across various fields, from science to gaming to automation of tasks. At a deeper level, the interpreted vs. compiled debate is still hot, since in practice, the Python programming architecture combines elements of both.

As far as syntax goes, in my opinion is one of the most beginner-friendly languages out there, with a much cleaner and straightforward syntax than others, using indentation (in the form of 4 consecutive spaces or the Tab) to structure the code blocks and logic inside any application. But more on that, later.

As a general note about this document, it doesn't follow the exact PCEP syllabus when it comes to the **order** in which the topics are presented. The order I used here makes more sense to me personally, however rest assured that there are no topics in the syllabus that have been ignored in this notebook.

Now, let's start discussing some of Python core concepts and features for the PCEP Exam.

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

THE PYTHON INTERPRETER & IDE

First of all, remember that you can run or execute your Python code in two ways. Either by writing your code inside a Python (.py) file, also called a **script**, and then execute that file in the Windows command line or Linux/macOS terminal, or by using the Python interpreter that comes by default with every installation of Python. Of course, there are other more advanced environments that you can use for development as well, but their use is beyond the scope of the PCEP exam.

At a very basic level, 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 the one in the Windows command line or Linux terminal, using the **python** command. This works perfectly, but, as you'll notice, 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 integrated IDE. IDE stands for Integrated Development Environment - that's a mouthful.

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

USER INPUT

Throughout this chapter, 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()*.

Here's an example of asking for user input in Python.

```
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 "store" or "save" 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.

Following the equal sign, we have the *input()* function. And again, don't worry, we will also discuss functions later in this document. 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 used - just to make everything pretty and easy to read.

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

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

```
print(user_says)
```

Ok, that's it! Let's execute this code, assuming you saved it into file1.py.

You can see that the program nicely asks you to enter whatever you want to be displayed on the screen. So, let's try printing out "Hello Python".

```
C:\Windows\System32>python D:\file1.py
Please enter the string you want to print:

Hello Python
```

Sweet, it works like a charm!

You can also convert the user's input from a string (the default) to an integer or float (more on numerical types, a bit later) if the content of the input allows.

```
>>> user_input = input("Enter the price: ") #string (default)
Enter the price: 50
>>> type(user_input)
<class 'str'>

>>> user_input = int(input("Enter the price: ")) #to integer
Enter the price: 50
>>> type(user_input)
<class 'int'>

>>> user_input = float(input("Enter the price: ")) #to float
Enter the price: 50
>>> type(user_input)
<class 'float'>
```

So, keep in mind that you can capture user input, display this input to the screen or even store it using a variable and later reuse it across your script. All of this with the help of the *input()* function. You can also convert it to integer or float if you need to.

Since I also mentioned the *print()* function, this is used to print something to the terminal, as its name implies. Usually, printing a string to the terminal / inside the interpreter, is as easy as:

```
print("Hello")
```

However, the *print()* function also supports two optional arguments that you can pass after the string itself, and they are: **end** and **sep**. You can also use them both simultaneously.

The **end** argument specifies another string that you want to be appended at the end of the original string when printing out the result.

```
print("Hello", end = "123")
Hello123
```

The **sep** argument allows you to specify a separator that you want to add in between the strings you enter as the first arguments of the function.

```
print("Hello", "Python") #no separator
Hello Python
print("Hello", "Python", sep = ".") #separator is the dot
Hello.Python
```

The last thing I'll add here is about comments. Notice above the text I entered after the # sign? That's called a comment in Python and everything following it until the end of the line doesn't get executed at runtime. Instead, it's just there for you to mark and note various code blocks in your code. This is one of many best practices in every programming language, including Python, and it will help you in your career.

More information:

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

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 having a clean and compliant code and 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 below - you should keep it at hand and read it anytime you're in doubt.

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

Source (python.org):

https://docs.python.org/3/reference/lexical_analysis.html#keywords

From now on this this document, whenever you see >>> before a line of code, keep in mind that it's the IDLE prompt, and NOT part of the actual Python syntax.

Now, let's see how we can assign a value to a variable, in the Python interpreter.

Note! You have to define a variable before being able to use it inside a program.

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.

```
>>> a = 100
```

On the left side of the equal sign, you type in the name of the variable and on the right side you type in the value that you want to associate that variable with.

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

You can also perform 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
```

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 or 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 find out the address or identity of **10** in the computer's memory.

```
>>> id(10)
1835063616
```

Let's apply the same function on variable **a**.

```
>>> id(a)
```

```
1835063616
```

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)
1835063616
```

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)
1835063776
```

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**.

```
>>> id(a)
1835063616
```

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 the most in Python 3 and how to handle and use them in your programs.

DATA TYPES OVERVIEW

Ok, following up on the previous discussion, let's see what types of data can be stored in memory and used inside Python 3 applications.

The Python 3 programming language defines many types of data for many types of use cases. We will have a look at the most widely used data types throughout the following chapters.

For now, as an overview, let's enumerate the most famous and useful Python 3 data types - so, we have:

- Strings
- Numbers
- Booleans
- Lists
- Sets
- Tuples
- Ranges
- Dictionaries

...and the **None** type.

All these data types are built 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, for instance.
- **immutable** data types - here we can mention 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 that you want. Now, it's time to have a closer look at a data type we already talked about earlier - strings.

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

PYTHON 3 - STRINGS

INTRODUCTION

What is a string, actually? A string is a sequence of characters - meaning letters, numbers and other characters such as the dollar sign, the space and punctuation marks - enclosed by single quotes, double quotes or even triple quotes when spanning multiple lines.

So, let's open the Python interpreter, create 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
'this is my first string'
```

Then, using single quotes and checking the result again:

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

So, it is the same string, right? It's your choice to use either of them.

However, what if you need to use quotes inside a string? Well, you just need to make sure that you use single quotes inside a string enclosed by double quotes, or vice-versa, like in the examples below:

```
>>> john = "I like John's shirt"
>>> hi = "She came to say 'Hi' to him"
>>> john
"I like John's shirt"
>>> hi
"She came to say 'Hi' to him"
```

In order to use the same type of quotes both for enclosing a string and inside the string itself, you need to place a backslash for the ones inside the string (escaping).

```
>>> hi = "She came to say \"Hi\" to him"
>>> hi
'She came to say "Hi" to him'
```

Now, what do we need triple quotes or triple double-quotes for? We need them whenever we want to enter a string on multiple lines, such as 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, you should 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** characters between the words in the string? Well, **\n** is called a new line character and it signals a new line being created at that 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 can we 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'''

>>> my_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 this book.

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, remember, the last character in a string will always have the index -1, when counting 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 at index 0 in the string, we should type in the following: the name of the variable, **string1,** and then, without inserting any spaces, the index in between square brackets:

```
>>> string1[0]
'C'
```

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? Well, that would be index 2, right?

```
>>> string1[2]
's'
```

This returns "s". Correct, again!

Let's see the 6th element! We need **string1[5]**, right?

```
>>> string1[5]
' '
```

And that returns 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?

```
>>> string1[-1]
'r'
```

This returns the letter "r". Good!

What about **string1[-4]**? Who would that be? Well, let's focus on the `"Cisco Router"` string and count from right to left: index **-1** is "r", then **-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.

```
>>> len(string1)
12
```

This piece of code returns the number of characters in the string, 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, as shown by the *len()* function. So, starting at index 0 and counting from left to right, the last character should have the index 11, right?

So, **string1[11]** would be "r". This is basically the same thing as **string1[-1]**.

```
>>> string1 = "Cisco Router"
>>> string1[11]
'r'
>>> string1[-1]
'r'
```

Now, let's see what happens if we enter **string1[20]**.

```
>>> string1[20]
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
string1[20]
IndexError: string index out of range
```

Well, we got an error. Let's celebrate! It is one of the first errors that you encountered thus far in Python. 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 this book. 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.

STRING 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 important operations on strings.

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 inside the string. So, let's consider variable **a** and assign it the string "Cisco Switch".

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

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

```
>>> a.index("i")
1
```

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

Obviously, this returns **1** as being the index of "i", which is correct since it is the second character of the string.

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 you are referring to, a dot, then the word **count** followed by parentheses and, finally, the letter you want to count, surrounded by quotes.

```
>>> a.count("i")
2
```

This returns **2**, so we have "i" twice inside our string, which is correct.

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 in practice.

```
>>> a.find("sco")
2
```

So, the result indicates a match starting at index 2, which is correct, since the letter "s" is positioned at index 2 inside 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, as well. Let's search for the substring "xyz" inside the initial "Cisco Switch" string.

```
>>> a.find("xyz")
-1
```

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

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, as needed. This can be accomplished by using the *lower()* and *upper()* methods.

So, returning to our initial string, let's perform both of these operations.

```
>>> a.lower()
'cisco switch'
```

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

```
>>> a.upper()
'CISCO SWITCH'
```

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 has not been modified. No changes have been applied to the original string.

```
>>> a
'Cisco Switch'
```

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

You can also verify that a string starts with 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")
True
```

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")
True
```

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

On the other hand, let's try the following:

```
>>> a.endswith("q")
```

```
False
```

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

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.

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()
'Cisco Switch'
```

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

Now, consider that instead of the three spaces on each side of the string, we had three **$** (dollar) 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.

```
>>> c.strip("$")
'Cisco Switch'
```

The above line of code 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 I did below, to get the string clean.

```
>>> b.replace(" ", "")
'CiscoSwitch'
```

Actually, this way you are replacing every Space character in the string with a so-called empty string (the string following the comma in between parentheses). It works, the result is the one we expected, so great job again!

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**.

```
>>> 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, such as a list? Well, in this case, the *split()* method saves the day.

```
>>> d.split(",")
['Cisco', 'Juniper', 'HP', 'Avaya', 'Nortel']
```

Python returns a list where each provider in the string is now 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.

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

For this, we will have to use the *join()* method - first, you type in the character you want to use as a separator, enclosed in single or double quotes. So, this character would be the underscore in our case.

```
>>> "_".join(a)
'C_i_s_c_o_ _S_w_i_t_c_h'
```

This could be read like this: *"use the _ separator to join the elements of string **a**"*. And that's it, you got the desired result using the magic behind the *join()* method.

More information:

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

STRING 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 do 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" #space included before Switch
```

...and, finally, add them together:

```
>>> x + y
'Cisco Switch'
```

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

```
>>> x * 3
'CiscoCiscoCisco'
```

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 whether the character "o" is part of this string. This should return True, obviously.

```
>>> x
'Cisco'
>>> "o" in x
True
```

Now, let's try another one.

```
>>> "b" in x
False
```

...returns False, because there is no letter "b" in **x**. On the other hand:

```
>>> "b" not in x
True
```

This will result in True, because we are negating 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 a 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 (currently **2600XM**), number of slots (currently **2**) and IOS version (currently **12.4**) a couple of times, while running our Python program. So, this would need to be a dynamic way of changing the string each time.

For this, we should use the percent (**%**) operator, followed by **s** for strings, **d** for digits or **f** for floating-point numbers. 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** represents a placeholder for a string that we will specify in between parentheses, at the end of this line. The **%d** operator follows the same logic, but for an integer instead of a string, and, finally, the **%f** refers to a floating-point number, a number with a decimal point.

```
>>> "Cisco model: %s, %d WAN slots, IOS %f" % ("2600XM",2,12.4)
'Cisco model: 2600XM, 2 WAN slots, IOS 12.400000'
```

Now, looking at the result above, notice that 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 other examples just to make sure that the template stays the same and only the values get updated each time.

```
>>> "Cisco model: %s, %d WAN slots, IOS %f" % ("2691XM",4,12.3)
'Cisco model: 2691XM, 4 WAN slots, IOS 12.300000'

>>> "Cisco model: %s, %d WAN slots, IOS %f" % ("7200XR",8,15.4)
'Cisco model: 7200XR, 8 WAN slots, IOS 15.400000'

>>> "Cisco model: %s, %d WAN slots, IOS %f" % ("1841M",1,12.2)
'Cisco model: 1841M, 1 WAN slots, IOS 12.200000'
```

Something you might have noticed is the addition of several decimal places when dealing with floating-point numbers - notice, for instance, the 12.300000 value instead of 12.3. In order to control this behavior, you can easily choose the number of decimal places that you want to print out to the screen. Let's get back to our string and simply insert a dot and a value in between the **%** operator and the letter **f**.

For instance, if we want just a single digit after the decimal point, we should use **.1**, like this:

```
>>> "Cisco model: %s, %d WAN slots, IOS %.1f" % ("2691XM",2,12.3)
'Cisco model: 2691XM, 2 WAN slots, IOS 12.3'
```

For two digits, let's use **%.2f** instead of **%f**.

```
>>> "Cisco model: %s, %d WAN slots, IOS %.2f" % ("2691XM",2,12.3)
'Cisco model: 2691XM, 2 WAN slots, IOS 12.30'
```

Finally, if we want no decimal places at all and no decimal point, we can just enter a dot and we will get the value 12 in return.

```
>>> "Cisco model: %s, %d WAN slots, IOS %.f" % ("2691XM",2,12.3)
'Cisco model: 2691XM, 2 WAN slots, IOS 12'
```

That's awesome, isn't it?

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)
'Cisco model:2600XM,2 WAN slots,IOS 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 as with the previous string formatting approach.

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)
'Cisco model:2600XM,2 WAN slots,IOS 12.4'
```

If we run this line of code, the result is going to be the same, as shown above. 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 in between parentheses are mapped to the operators in the string. Let's see what will happen if we change our code like this:

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

Well, we can notice 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 original 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)
'Cisco model:12.4,2 WAN slots,IOS 12.4'
```

Last, but not least, let's see yet another type of string formatting - using **f-strings**, also called **formatted string literals**. F-strings have been introduced in Python version 3.6 through the PEP 458 Python enhancement proposal, as a way to embed variables and expressions inside strings, using a minimal and elegant syntax.

As shown below, f-strings are prefixed with the letter 'f' and the variables or expressions embedded inside curly braces get replaced with their value in the resulting string.

Considering the same string template that we used earlier, let's define some variables first and then let's use an f-string to include the values referenced by these variables in the string.

```
>>> model = '2600XM'
>>> slots = 4
>>> ios = 12.3
```

Now, using the previously mentioned *format()* method, we can embed these values into the string template by simply specifying the name of each variable inside the parentheses of *format()*.

```
>>> "Cisco model:{},{} WAN slots,IOS {}".format(model,slots,ios)
'Cisco model:2600XM,4 WAN slots,IOS 12.3'
```

But what if we can achieve the same result using a much easier and cleaner syntax? Yes, I am referring to f-strings. Let's see them in action.

```
>>> f"Cisco model: {model}, {slots} WAN slots, IOS {ios}"
'Cisco model: 2600XM, 4 WAN slots, IOS 12.3'
```

What we did here is we prefixed the string with the letter "f" to define it as an f-string and then, in between curly braces, we entered the name of each previously defined variable to obtain the desired result. Pretty straightforward, right?

What if, instead of just the variable name, we want to enter an expression? Well, in that case, all we have to do is to simply write the expression itself in between the curly braces. For instance, let's assume that we want the value referenced by the **slots** variable to be multiplied by 2 inside the resulting string. In this case, we have to just embed this expression in the string, using the name of the variable and the multiplication operator.

```
>>> f"Cisco model: {model}, {slots * 2} WAN slots, IOS {ios}"
'Cisco model: 2600XM, 8 WAN slots, IOS 12.3'
```

Isn't that just awesome?

Now, what if we want to use a string-specific method to convert the letters in the string referenced by the **model** variable to lowercase? So, that would be '2600XM' to '2600xm'. Well, we would have to call the *lower()* method, right? F-strings allow us to do that right in between the curly braces.

```
>>> f"Cisco model: {model.lower()}, {slots} WAN slots, IOS {ios}"
'Cisco model: 2600xm, 4 WAN slots, IOS 12.3'
```

Target achieved. What about embedding only a character (by its index) or a slice *(don't worry, you'll start learning about slicing on the next page)* of the string that the **model** variable points to? Again, just use the necessary syntax inside the curly braces.

```
>>> f"Cisco model: {model[0]}, {slots} WAN slots, IOS {ios}"
'Cisco model: 2, 4 WAN slots, IOS 12.3'

>>> f"Cisco model: {model[0:4]}, {slots} WAN slots, IOS {ios}"
'Cisco model: 2600, 4 WAN slots, IOS 12.3'
```

This way, you can embed only a portion of a predefined string - a single character or a subsequence of characters - into the final string, using a very elegant and straightforward syntax. All of this thanks to the PEP 498 enhancement.

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

STRING SLICING

Generally speaking, slicing allows us to extract certain segments from a sequence of elements, leaving the initial sequence unchanged.

A string itself is a sequence of characters. For instance, using the *list()* function on a string will return a list of the characters in that string. More on lists, later.

```
>>> e = "hello"
>>> list(e)
['h', 'e', 'l', 'l', 'o']
```

The general syntax of a slice is the following:

```
my_sequence[10:15]
```

We have the name of the variable pointing to the sequence (a string, let's say), 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 make this as clear as daylight.

Let's create a string and reference it, using the **string1** variable. Again, the actual 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 a substring - the first IP address in this string, which is 10.110.8.9. The first character in our slice should be "1". This character is positioned at index 5 in the string, right? You can double-check that by simply starting at index 0 and counting from left to right. By the way, remember that Spaces count as characters, too!

Now, the last character in the slice should be "9", which is located at index 14, isn't it? Don't take my word for it, start counting!

Ok, so we know that we should start the slicing at index 5 and go up to index 15, as the upper bound 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]
'10.110.8.9'
```

As expected, the slice above 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. Let me prove this, so we can be on the same page.

```
>>> string1[5:14]
'10.110.8.'
```

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

```
>>> string1[5:]
'10.110.8.9 [160/5] via 10.119.254.6, 0:01:00, Ethernet2'
```

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 go to the Python interpreter and try this.

```
>>> string1[:10]
'O E2 10.11'
```

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[:]
'O E2 10.110.8.9 [160/5] via 10.119.254.6, 0:01:00, Ethernet2'
```

Sweet, it works like a charm! Keep on reading and don't skip anything.

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

```
>>> string1[-1]
'2'
```

Remember that index -1 will always return the last character in the string, which is "2" in our case.

string1[-2] should return the next character, when reading from right to left, right? That would be the second to last character in the string. Let's test this.

```
>>> string1[-2]
't'
```

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" - those would be "E" and "t". Character "t" has the index -2 and "E" has the index -9. Now, we can write the slice.

```
>>> string1[-9:-1]
'Ethernet'
```

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 the correct one, since we got the "Ethernet" substring printed out to the screen.

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 would have to use index -5 and go all the way up 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.

```
>>> string1[-5:]
'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. Therefore, the character at index -6 will be the last character we will include in our slice. Let's test this!

```
>>> string1[:-5]
'O E2 10.110.8.9 [160/5] via 10.119.254.6, 0:01:00, Ethe'
```

One more thing about slices. You can specify a third element within the square brackets, after the start and stop 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]
'OE 01089[6/]va1.1.5.,00:0 tent'
```

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.

Next 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]
'2tenrehtE ,00:10:0 ,6.452.911.01 aiv ]5/061[ 9.8.011.01 2E O'
```

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

Also, we can slice a string using both negative indexes (start & stop) and a negative step. That would mean we're reading the string right to left and extracting a slice like the one below. Let's say we want to get 'hsab' from string2, which are the last 4 characters in reverse order. And we plan to do this using negative indexes. Cool, eh?

```
>>> string2 = "pythonjavarubybash"
>>> string2[-1:-5:-1]
'hsab'
```

So, in the slice above, we're starting at index -1 (which is the last character in the string), up to – but not including – index -5, with a step of -1.

Now, regarding string slices and memory assignments, check out the way Python handles the variables below and notice the difference when it comes to **str4**.

```
>>> str1 = "Hello"
>>> id(str1)

1444540894976
>>> str2 = str1
>>> id(str2)
1444540894976

>>> str3 = str1[:]
>>> str3
'Hello'
>>> id(str3)
1444540894976

>>> str4 = str1[1:3]
>>> str4
'el'
>>> id(str4) #notice that str4 points to a new memory location
1444546121040
```

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.

PYTHON 3 - NUMBERS AND BOOLEANS

NUMBERS - MATH OPERATORS

The first thing I should mention in this chapter 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. 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 **basic math operations** are available using integers and floating-point numbers.

Addition:

```
>>> 1 + 2
3
```

Subtraction:

```
>>> 2 - 1
1
```

Division:

```
>>> 5 / 2
2.5
```

Integer division:

```
>>> 5 // 2
2
```

Multiplication:

```
>>> 4 * 2
8
```

Raising to a power (exponentiation):

```
>>> 4 ** 2
16
```

Modulo - finding out the remainder after dividing one number by another:

```
>>> 5 % 2
1
```

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

Less than:

```
>>> 4 < 5
True
```

Greater than:

```
>>> 5 > 4
True
```

Less than or equal to:

```
>>> 4 <= 5
True
```

Greater than or equal to:

```
>>> 5 >= 4
True
```

Equals:

```
>>> 5 == 5
True
```

Not equals:

```
>>> 4 != 5
True
```

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, considering the expression below.

```
>>> 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 priority, 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, leads to 100 - 10 = 90. Let's check this!

```
>>> 100 - 5 ** 2 / 5 * 2
90.0
```

Notice that the result is 90.0 - a floating-point number, instead of 90 - an integer. That's because of the division operation inside our expression which, by default, produces a float. If we want to make sure we get an integer as a result, we should use the integer division operator instead:

```
>>> 100 - 5 ** 2 // 5 * 2
90
```

Keep in mind that these rules for order of evaluation inside a mathematical expression always apply, unless you use parentheses to change the order of evaluation.

```
>>> 100 - 5 ** 2 / 5 * 2
90.0
>>> (100 - 5) ** 2 / 5 * 2
3610.0
>>> 100 - 5 ** (2 / 5) * 2
96.19269212256825
```

Another thing to remember about operations with the same priority, when having two **** (raising to a power) consecutive operations, they are evaluated **from right to left**.

```
>>> 2 ** 2 ** 4
65536
```

Here, **2**4** is evaluated first, which is **16**. And then **2**16** is 65536.

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:

```
>>> type(1.7)
<class 'float'>
>>> int(1.7)
1
```

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. On the other hand, we have:

```
>>> type(2)
<class 'int'>
>>> float(2)
2.0
```

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

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

The *abs()* function returns the absolute value for the number provided in between its parentheses. The absolute value represents the distance between the number we pass to the *abs()* function and 0.

```
>>> abs(5)
5
```

...returns 5, since this is the distance between 5 and 0.

```
>>> abs(-5)
5
```

...also returns 5, since the distance to 0 is the same as in the previous case.

Next, let's start comparing two numbers.

```
>>> max(1, 2)
2
```

...returns 2, the largest number in between parentheses.

```
>>> min(1, 2)
1
```

...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 called *pow()*.

```
>>> pow(3, 2)
9
```

...where 3 is the base and 2 is the exponent or power.

The result is, of course, 9, which is correct.

Now, it’s time to have a look at compounding operators, which are sometimes preferred for having a cleaner code. What a compounding operator is, is basically an equal sign preceded by a math operator. See the examples below and the comments.

```
>>> x = 10
>>> x += 5    #same as x = x + 5
>>> x
15

>>> x = 10
>>> x -= 5    #same as x = x - 5
>>> x
5

>>> x = 10
>>> x *= 5    #same as x = x * 5
>>>
50

>>> x = 10
>>> x /= 5    #same as x = x / 5
>>> x
2.0

>>> x = 10
>>> x //= 5    #same as x = x // 5
>>> x
2

>>> x = 10
>>> x **= 5    #same as x = x ** 5
>>> x
100000
```

As a side note, addition and multiplication via compound operators works for strings as well.

```
>>> mystr = "abc"
>>> mystr += "D"
>>> mystr
'abcD'

>>> mystr = "abc"
>>> mystr *= 3
>>> mystr
'abcabcabc'
```

I think this should be more than enough math for now, so let's move on to Booleans.

More information:

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

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**, respectively.

In Python, **True** is always 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 chapters, so they are not completely new to you at this point in the book.

Basically, these two values are used to evaluate whether an expression is true or false and can be further used in conditional or loop structures, as you will see later.

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 <= 2
False
```

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

As a side note, be careful when using **less (greater) than or equal to** operators.

```
>>> 2 =< 3 #invalid operator
SyntaxError: cannot assign to literal
>>> 3 => 2 #invalid operator
SyntaxError: cannot assign to literal
```

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)
True
```

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

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

Next, when performing an AND operation between two False expressions the result will always 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 is going to be False, as well. So, when using OR, it is enough if only one expression is True, in order to have True as a 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**; the empty string **""**; the empty list **[ ]**; the empty set **set()**; the empty tuple **( )**; the empty dictionary **{ }**

Python provides the *bool()* function to help us evaluate expressions and values 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.

```
>>> bool(1)
True
>>> bool(2.2)
True
>>> bool("Hello")
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.

Now, let's also take a look at Bitwise logical operators, which are analogous to the AND, OR and NOT operators we've seen earlier. Bitwise logical operators are used on a **bit-by-bit level** for specific operations and use cases. In our day-to-day coding, it's best to stick to the classical AND, OR and NOT operators 90% of the time.

```
(1 == 1) & (2 == 2) #corresponds to AND, but on a bit level

(1 == 1) | (2 == 3) #corresponds to OR, but on a bit level

~(1 == 2) #corresponds to NOT, but on a bit level
```

Now, one other bitwise operator we haven't talked about is XOR, exclusive OR.

It doesn't have an associated logical operator like the ones above, however it can prove to be useful in bitwise operations that require the evaluation of two **mutually exclusive conditions**. At the end, XOR tells you whether exactly one of the conditions is met (true). We can see this using a custom function that emulates the logic of XOR.

The two expressions – x and y – cannot be both True, since "a" cannot be a string and an integer at the same time. The XOR bitwise operator is normally ^

```
def xor():
    i = "a"
    x = type(i) is str
    y = type(i) is int
    return (x and not y) or (not x and y)

>>> xor()
True
```

More information: 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? In short, a list is 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, booleans or even other lists; and a list may have any number of elements.

As with 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 perform slicing 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 you will get to test that very soon.

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

```
>>> list1 = [ ]
```

So, in order to have an empty list, you just have to type in 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)
<class 'list'>
```

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)
6
```

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.

```
>>> list1[0]
'Cisco'
>>> list1[1]
'Juniper'
>>> list1[-1]
-11
>>> list1[-2]
10.5
```

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

```
>>> list1[100]
Traceback (most recent call last):
File "<pyshell#5>", line 1, in <module>
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?

```
>>> list1[2]
'Avaya'
```

Now, let's replace "Avaya" with another vendor, let's say "HP". We just have to use the equal sign to modify this.

```
>>> 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. So we can play around with indexes, for instance by updating an element at a negative index, or replacing one element of the list with another element from the same list, using the power of indexes.

```
>>> list1 = ["Cisco", "Juniper", "Avaya", 10, 10.5, -11]
>>> list1[-1] = 222 #updating element at negative index
>>> list1
['Cisco', 'Juniper', 'Avaya', 10, 10.5, 222]

>>> list1[-1] = list1[1] #assigning the value of another element
>>> list1
['Cisco', 'Juniper', 'Avaya', 10, 10.5, 'Juniper']
```

Finally, let's also check out how memory assignment works when slicing lists.

```
list1 = [1, 2, 3]
id(list1)
1444546143808

list2 = list1
id(list2)
1444546143808 #same memory location, same list

list3 = list1[:]
id(list3)
1444546043264 #different memory location than list1

list4 = list1[2:]
id(list4)
1444545959680 #different memory location than list1
```

Following up on the same logic:

```
>>> listA = [10, 20, 30]
>>> id(listA)
1444545959104

>>> listB = listA #here listB points to listA - it's the same list
>>> id(listB)
1444545959104 #same memory location

>>> listB[0] = 5 #modifying listB also modifies listA
>>> id(listA)
1444545959104

>>> id(listB)
1444545959104
>>> listA
[5, 20, 30]
>>> listB
[5, 20, 30]
```

On the other hand:

```
>>> listX = [4, 5, 6]
>>> id(listX)
1444502426368

>>> listY = listX[1:] #listY is not = listX (like B was to A)
>>> id(listY)
1444546041536

>>> listX
[4, 5, 6]
>>> listY
[5, 6]

>>> listY[0] = 99 #modifying listY is not modifying listX
>>> listX
[4, 5, 6]
>>> listY
[99, 6]
```

Finally, always remember that lists are mutable, unlike strings or numbers.

I hope this is clear and now we can move on to list methods.

LIST 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, let's find out the maximum value in our list, as well.

```
>>> max(list2)
12
```

...which returns 12. Again, that's correct.

Now, what about a list of strings?

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

In this case, what would be the minimum and maximum values in the list?

```
>>> min(list3)
'a'
```

...will return "a", pretty easy to guess, considering the alphabetical order.

```
>>> max(list3)
'c'
```

...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. 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)
Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>
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, considering 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 the list.

```
>>> list1
['Cisco', 'Juniper', 'HP', 10, -11, 100]
```

Another way to remove an element by its index is using the *pop()* method.

```
>>> list1.pop(0)
'Cisco'
```

...will remove the first element in the list, which is "Cisco", right? Now, if we check **list1** again, we can spot the absence of "Cisco".

```
>>> list1
['Juniper', 'HP', 10, -11, 100]
```

As a side note, remember that using *pop()* with no arguments (no index specified) will remove the element at index -1, meaning the last element of the list!

```
>>> list1 = ['Cisco', 'Juniper', 'HP', 10, 10.5, -11]
>>> list1.pop()
-11
>>> list1
['Cisco', 'Juniper', 'HP', 10, 10.5]
```

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")
>>> list1
['Juniper', 10, 'Nortel', -11, 100]
```

...where 2 is the index at which where we want this new element to be inserted, and "Nortel" is the new element.

How about inserting an element at a negative index in the list? Is that possible?

```
>>> list1 = ['Juniper', 10, 'Nortel', -11, 100]
>>> list1.insert(-2, "XYZ")
>>> list1
['Juniper', 10, 'Nortel', 'XYZ', -11, 100]
```

So, in the example above, inserting the element "XYZ" at index -2 basically means you're inserting the new element before element that is currently sitting at index -2. In list1 above, we see -11 having index -2, right? When inserting "XYZ" we're basically telling Python to insert it before the element at index -2. Thus, -11 keeps its existing position (index), whilst "XYZ" gets inserted at index -3. Keep that in mind.

Another interesting list operation is appending a list to another list. So, let's consider **list2** as being:

```
>>> list2 = [9, 99, 999]
```

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

```
>>> list1
['Juniper', 10, 'Nortel', -11, 100]
>>> list2
[9, 99, 999]
```

And now, we can just extend **list1** with the content of **list2**.

```
>>> list1.extend(list2)
>>> 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 inside the list.

```
>>> list1
['Juniper', 10, 'Nortel', -11, 100, 9, 99, 999]
>>> list1.index(-11)
3
```

...returns the index of the element -11 in the list.

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
[9, 99, 999]
>>> list2.append(1)
>>> list2.append(25)
>>> list2.append(500)
>>> list2
[9, 99, 999, 1, 25, 500]
```

Ok. Now, let's say that we want to have them sorted in ascending order. We will simply apply the *sort()* method on **list2**.

```
>>> 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 available 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 in memory; 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]
```

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 the variable pointing to the list.

```
>>> sorted(list2, reverse = True)
[999, 500, 99, 25, 9, 1]
```

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 in the past, but, if you want to check out the available arguments for a function, you should use *help()* inside the Python interpreter, passing the name of the function you want to know more about, as an argument.

```
>>> help(sorted)
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
Return a new list containing all items from the iterable in
ascending order.
A custom key function can be supplied to customize the sort order,
and the reverse flag can be set to request the result in
descending order.
```

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

```
>>> list1 + list2
['Juniper', 10, 'Nortel', -11, 100, 9, 99, 999, 10, 999, 500, 99,
25, 9, 1]
```

...adds the elements of both lists together, creating a single list.

```
>>> list2 * 3
[999, 500, 99, 25, 9, 1, 999, 500, 99, 25, 9, 1, 999, 500, 99, 25,
9, 1]
```

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

Notice here that duplicate elements are allowed, so don't worry about that.

As with strings, you can verify the (non-)existence of a certain element in a list.

```
>>> list1 = ['Cisco', 'Juniper', 'HP', 10, 10.5, -11]
>>> 10 in list1
True
>>> 10 not in list1
False
```

Next, we are going to discuss list slicing.

LIST SLICING

Remember - list slices allow us to return various parts or segments 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:

```
>>> list1[5:10]
```

So, we have 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.

```
>>> 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.

```
>>> 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.

```
>>> 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 exactly what we aimed for. 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.

```
>>> 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 - for instance, 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 that would be 2.

```
>>> list3[::2]
[1, 3, 'b']
```

Secondly, if you want to have the list returned 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?

```
>>> list3[::-1]
['c', 'b', 'a', 3, 2, 1]
```

Also, we can slice a list using both negative indexes (start & stop) and a negative step. That would mean we're reading the list right to left and extracting a slice like the one below.

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> list1[-2:-6:-2]
[7, 5]
```

So, in the slice above, we're starting at index -2 in the list (which is 7), up to (but not including) index -6 (which is 3), with a step of -2, so the result is [7, 5].

Given our discussion and the examples thus far, as with strings and other "slice-able" data types, the following operations are invalid and return either an IndexError or an empty list.

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7, 8]
list1[8]
Traceback (most recent call last):
  File "<pyshell#153>", line 1, in <module>
    list1[8]
IndexError: list index out of range

>>> list1[4:2]
[]
>>> list1[-4:-2:-1]
[]
```

On the other hand, the following slice is valid, although the second index is outside the range of the list (aka is grater than the index of the last element in the list).

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> list1[3:25]
[4, 5, 6, 7, 8]
```

Ok, I think we covered most of what you need to know about lists as a beginner. Of course, there are many more methods and operations that you can apply on this data type and you can find them all in Python's official documentation on data structures.

Also, check out a bonus section on List Comprehensions at the end of this document. It will surely prove to be useful both at the exam and in real life coding.

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

PYTHON 3 - SETS

INTRODUCTION

Let's start this chapter by asking ourselves - what is a set? Well, in short, 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'>
```

To prove that sets indeed are unordered, as I mentioned at the beginning of this chapter, let's see an example. Considering a string - which is a sequence of characters

- let's apply the *set()* function on this string and see if the initial order of elements is kept. This will clearly demonstrate the unordered nature of sets.

```
>>> j = "0123456789"
>>> set(j)
{'0', '8', '7', '1', '5', '3', '4', '2', '9', '6'}
```

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 - according to **python.org** - so Python 3.x is included, as well.

```
>>> 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 *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
```

Please keep in mind that sets are mutable, therefore we can easily add or remove elements from a set, using the *add()* and *remove()* methods.

```
>>> 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 you try to add an element which already exists in the set, Python will not agree, 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}
```

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

SET 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 that **set2** doesn't, by using the *difference()* method. By 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)
{1, 2, 4}
```

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

Another thing you can do is remove an element in the set by using the *pop()* method. Using *pop()* you remove the element with the smallest value in the set. Since sets are unordered, you cannot remove elements by index since there are no indexes.

```
>>> set1
{1, 2, 3, 4}
>>> set1.pop()
1
>>> set1
{2, 3, 4}
```

So, element 1 has been removed and also printed out to the screen 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 document.

Just in case, please check out the following link to Python's official documentation regarding sets.

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

PYTHON 3 - TUPLES

INTRODUCTION

Tuples are yet 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 whenever 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 practicing and create our first tuple. Tuples are enclosed by parentheses and their elements are separated by commas.

```
>>> my_tuple = ()
>>> type(my_tuple)
<class 'tuple'>
```

If you want to create a tuple with just a single element you have to use a trick, meaning that, although you have only one element inside the tuple, you have to insert 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'>
```

On the other hand:

```
>>> my_tuple = (9,)
>>> type(my_tuple)
<class 'tuple'>
```

Or, creating a tuple in which the only element is another tuple:

```
>>> my_tup = ((1,2),)
>>> type(my_tup)
```

```
<class 'tuple'>
```

Now you have a tuple set up. You should remember this when creating tuples having a single element. Next, let's populate our tuple with more elements.

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

Additionally, you can create a tuple like below, without adding parentheses.

```
>>> my_tuple = 1, 2, 3, 4, 5
>>> my_tuple
(1, 2, 3, 4, 5)
>>> type(my_tuple)
<class 'tuple'>
```

Just like strings and lists, tuples support indexing too, 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
Traceback (most recent call last):
File "<pyshell#121>", line 1, in <module>
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]
Traceback (most recent call last):
File "<pyshell#122>", line 1, in <module>
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 can 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 regard 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 a different number of elements, a ValueError will be returned.

```
>>> tuple2 = (1, 2, 3, 4)
>>> (x, y, z) = tuple2
Traceback (most recent call last):
File "<pyshell#131>", line 1, in <module>
(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 automatically generate the ValueError exception.

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.

Other interesting operations with tuple indexing and packing/unpacking:

```
my_tuple = ("a", "b", "c", "d", "e", "f")
my_tuple = my_tuple[2:-2] #assigning the variable to a slice
my_tuple
('c', 'd')
my_tuple = my_tuple[1] #assigning the variable to an element
my_tuple
'd'
my_other_tuple = [1, 2, 3, 4, 5, 6]
my_other_tuple[my_other_tuple[1]] #passing an element as index
3
my_other_tuple[my_other_tuple[-3]] #passing an element as index
5
```

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

TUPLE 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)
>>> 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 addition and multiplication operators.

```
>>> tuple2
(1, 2, 3, 4)
>>> tuple2 + (5, 6, 7)
(1, 2, 3, 4, 5, 6, 7)
>>> tuple2 * 3
(1, 2, 3, 4, 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 too much 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[::2]
(1, 3)
>>> tuple2[::-1]
(4, 3, 2, 1)
```

Another thing you can do with tuples is you can check if an element is a member of a tuple or not, using the **in** and **not in** operators. Let's test 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** keyword to delete the entire tuple.

```
>>> del tuple2
>>> tuple2
Traceback (most recent call last):
File "<pyshell#169>", line 1, in <module>
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 keep them in mind.

More information:

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

PYTHON 3 - RANGES

INTRODUCTION

In this chapter 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, open up http://www.codeskulptor.org/, which is an online Python 2 interpreter and, after clearing any code in the left pane, 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.

So, in the left pane type in:

```
print range(10)
```

Hit the Play button and check the result in the right pane.

```
[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 consecutive integers returned, inside a list.

But don't worry - the start, stop and step values can be customized. 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)
```

And the result is, I think, pretty easy to guess.

```
[-2, -4, -6, -8]
```

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 returned the same values, 0 to 9. However, the difference laid in what data type does each of these functions return. In Python 2, the *range()* function returned a list, as we've just seen. On the other hand, the *xrange()* function returned an iterator. Iterators are outside the scope of this document. 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(10000000000000)*. That would generate a gigantic list, starting at 0 and up to this huge value, am I right? Such an enormous number of 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 pages.

RANGE METHODS

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

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 useful.

Finally, keep in mind that in order to slice a range you can perform one of the two operations below, both of them requiring the use of the *list()* function.

```
>>> list(r)[2:5]
[2, 3, 4]
```

or:

```
>>> r[2:5]
range(2, 5)
>>> list(r[2:5])
[2, 3, 4]
```

Now, let's see some examples of valid and invalid ranges.

```
>>> list(range(7)) #valid
[0, 1, 2, 3, 4, 5, 6]
>>> list(range(2, 7)) #valid
[2, 3, 4, 5, 6]
>>> list(range(-7, -2)) #valid
[-7, -6, -5, -4, -3]
>>> list(range(7, 2)) #invalid
[]
>>> list(range(-2, -7)) #invalid
[]
```

As a fun experiment, what would be the result of passing the length of a range to the *range()* function?

```
>>> list(range(2,7))
[2, 3, 4, 5, 6]
>>> len(range(2,7))
5
>>> range(len(range(2,7))) #that would be range(5)
range(0, 5)
>>> list(range(len(range(2,7))))
[0, 1, 2, 3, 4]
```

As a simple rule, remember that when having only one argument passed to the *range()* function, the first element is 0 and **len(range(x)) = x** like in the code below:

```
len(range(10))
10
len(range(55))
55
len(range(2024))
2024
```

More information: 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 a 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 network device, 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 methods. 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. You will see this in practice below, 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
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.4', 'Ports': '4'}
```

Each dictionary element consists of a key, a colon and a value, followed by a comma. Now, let's notice a few things here.

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

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

Keep in mind that up to Python version 3.5 inclusively, the **dict** data type was considered unordered, basically meaning that the order in which its elements were stored and displayed wasn't necessary the same as the order they've been inserted in at creation time. However, the approach was initially changed in Python 3.6 and the ultimate decision has been made in Python 3.7 and now the insertion-order preservation nature of **dict** objects is official.

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. The immutability and mutability of the data types used as keys and values, respectively, is a crucial concept to keep in mind when using dictionaries.

DICTIONARY METHODS

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

```
>>> dict1
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.4', 'Ports': '4'}
```

First, let's have the corresponding value returned 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.

```
>>> dict1["IOS"]
'12.4'
>>> dict1["Vendor"]
'Cisco'
>>> dict1["Model"]
'2600'
>>> dict1["Ports"]
'4'
```

However, you cannot access multiple keys at once using this syntax.

```
>>> dict1["Vendor","IOS"] #invalid, returns KeyError
```

Earlier in this chapter I said that dictionaries are mutable. So, let's try to add a new key-value pair to our dictionary. This is accomplished by simply assigning a new value to the new key.

```
>>> dict1["RAM"] = "128"
>>> dict1
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.4', 'Ports': '4',
'RAM': '128'}
```

Now, maybe we want to modify and update the value for the IOS data of this device. Don't worry, it's simple - just use the same syntax like for adding a new pair.

```
>>> dict1["IOS"] = "12.3"
>>> dict1
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.3', 'Ports': '4',
'RAM': '128'}
```

You can also easily modify a value via concatenation (for strings) or a math operation (for numbers).

```
>>> dict1["IOS"] += "6"
>>> dict1
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.36', 'Ports': '4'}
```

```
>>> dict2 = {0: 1, 1: 2, 2: 3}
>>> dict2[1] *= 5
>>> dict2
{0: 1, 1: 10, 2: 3}
```

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

```
>>> del dict1["Ports"]
>>> dict1
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.3', 'RAM': '128'}
```

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
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.3', 'RAM': '128'}
>>> 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 used for dealing with the keys and values of 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.

```
>>> dict1.keys()
dict_keys(['Vendor', 'Model', 'IOS', 'RAM'])
>>> type(dict1.keys())
<class 'dict_keys'>
>>> list(dict1.keys())
['Vendor', 'Model', 'IOS', 'RAM']
```

A similar method is *values()* to get a list having the values in a dictionary as elements.

```
>>> dict1.values()
dict_values(['Cisco', '2600', '12.3', '128'])
>>> type(dict1.values())
<class 'dict_values'>
>>> list(dict1.values())
['Cisco', '2600', '12.3', '128']
```

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()
```

```
dict_items([('Vendor', 'Cisco'), ('Model', '2600'), ('IOS',
'12.3'), ('RAM', '128')])
>>> type(dict1.items())
<class 'dict_items'>
>>> list(dict1.items())
[('Vendor', 'Cisco'), ('Model', '2600'), ('IOS', '12.3'), ('RAM',
'128')]
```

When using the *keys()*, *values()* and *items()* methods, since dictionaries are ordered data types starting with version 3.7, 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
{'Vendor': 'Cisco', 'Model': '2600', 'IOS': '12.3', 'RAM': '128'}
```

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

```
>>> list(dict1.keys())
['Vendor', 'Model', 'IOS', 'RAM']
>>> list(dict1.values())
['Cisco', '2600', '12.3', '128']
>>> list(dict1.items())
[('Vendor', 'Cisco'), ('Model', '2600'), ('IOS', '12.3'), ('RAM',
'128')]
```

So, you can notice that Python helps us by keeping an internal indexing system when using dictionary methods.

Next, we can also make a copy of the dictionary. This operation does not modify the original dictionary in any way.

```
>>> dict2 = {0: 1, 1: 2, 2: 3}
>>> id(dict2)
1444546051776
>>> dict3 = dict2.copy()
>>> id(dict3)
1444546049728
>>> dict3
{0: 1, 1: 2, 2: 3}
```

Finally, we can delete all the elements of a dictionary using the *clear()* method.

```
>>> dict3.clear()
>>> dict3
{}
```

More information:

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

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 an integer to a string.

```
>>> num2 = str(num)
>>> num2
'2'
>>> type(num2)
<class 'str'>
```

Converting a floating-point number to a string.

```
>>> f2 = str(f)
>>> f2
'2.5'
>>> type(f2)
<class 'str'>
```

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

```
>>> str1 = "5"
>>> type(str1)
<class 'str'>
>>> int1 = int(str1)
>>> int1
5
>>> 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 you've just seen.

```
>>> num
2
>>> type(num)
```

```
<class 'int'>
>>> f = float(num)
>>> f
2.0
>>> type(f)
<class 'float'>
```

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

```
>>> f
2.0
>>> int1 = int(f)
>>> int1
2
>>> type(int1)
<class 'int'>
```

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

Now, just for fun, let's convert an integer to a string and then to a float, within a single line of code.

```
>>> i = 20
>>> type(i)
<class 'int'>
>>> float(str(i))
20.0
```

Of course, generally speaking, you have to be careful about what string you choose to convert to a float, otherwise you might end up with an exception.

```
>>> s = "hello"
>>> float(s)
Traceback (most recent call last):
File "<pyshell#331>", line 1, in <module>
float(s)
ValueError: could not convert string to float: 'hello'
```

Obviously, "hello" cannot be converted to a numerical data type.

Next, moving on 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]
>>> type(list1)
<class 'list'>
>>> 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]
>>> type(list1)
<class 'list'>
>>> set1 = set(list1)
>>> type(set1)
<class 'set'>
```

Let's try the same procedure for converting a tuple to a set.

```
>>> tup1 = (1,2,3)
>>> type(tup1)
<class 'tuple'>
>>> set1 = set(tup1)
>>> 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 the binary and hex formats back to the decimal notation, we will use the *int()* function.

```
>>> bin_num = int(num_bin, 2)
>>> bin_num
10
>>>
>>> hex_num = int(num_hex, 16)
>>> hex_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 90% of 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 even thousands, but you will rarely use them in your day-to-day programs or even not use them at all.

I hope you enjoyed these chapters and I'll see you in the next one!

PYTHON 3 - CONDITIONALS AND LOOPS

CONDITIONALS - IF / ELIF / ELSE

In Python, we have the **if, elif** and **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 4 consecutive Space characters or the **Tab** key.

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.

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 text file on the local computer and call it **test1**, add pick the appropriate extension - **.py** - and that's it.

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 a certain way and run

a piece of code consequently. Let's use the **if** statement to execute a certain block of code, if the expression we provide will be evaluated as **True**. Therefore, let's write the following code inside our new Python script.

```
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.

After I finished writing my **if** statement and adding the colon at the end, 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 that 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.

```
x = 10
if x > 5:
   print("x is greater than 5")
   print(x * 2)
```

Now, let's save the script using **Ctrl+S** in Notepad++ and let's try to guess the result. We said **x** was equal to 10 and then, if **x** is greater than 5, we should have *"x is greater than 5"* and the result of **x * 2** printed out to the screen. Since 10 is indeed greater than 5, then the **x > 5** expression is evaluated as **True** and the *print()* functions indented under the **if** clause get executed. That's why we should obtain the string and the value of 20 as a result.

Let's test this by simply opening the Windows command line (cmd) and running our script, using the **python** command and the path to the Python script.

```
C:\Windows\System32>python D:\test1.py
```

The result of running the script is the one we anticipated.

```
x is greater than 5
2
```

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

```
x = 4
if x > 5:
   print("x is greater than 5")
   print(x * 2)
```

Now, in the Windows cmd:

```
C:\Windows\System32>python D:\test1.py

C:\Windows\System32>
```

In this case, our **if** block does not return anything, since 4 is not greater than 5 and the expression in the **if** clause 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 have the **elif** and **else** clauses available.

First, let's see how to use the **else** clause by returning 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 is going to be:

```
x is NOT greater than 5
```

So, to use the **else** statement, you must hit Enter after *print("x is greater than 5")* line, then return to the beginning of the new line using Backspace 4 times, and then type in the **else** keyword, followed by a colon and, finally, hit Enter again. Now, on this new line use the Tab key once again to get to the next level of indentation and write the code to be executed in case x is not greater than 5: *print("x is NOT greater than 5")*. For every block of code positioned under an **if/elif/else** clause we must use indentation to assign that code to the associated clause above it.

The **else** clause is used to cover all the other cases not covered by the **if** clause above it. Therefore, if the expression following the **if** keyword is evaluated as True, then the indented code below this clause gets 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 one or more **elif** clauses.

Let's say that 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 during the program execution and *"x is less than 5"* for all the other cases. We should then add the **elif** clause in between **if** and **else**. The **else** clause 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 equal to 5")
```

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

Save the file and run it once again in the Windows command line. Of course, the result is going to be:

```
x is equal to 5
```

So, the syntax for **elif** is similar to the syntax of the **if** clause, 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 code.

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

Also, keep in mind that you can have as many **elif** clauses as you want after an **if** clause, but remember that there can be **only one else clause at the end**, covering all the other possible cases not being covered by the **if** and **elif** clauses. Actually, what Python does is it evaluates these clauses in order, from top to bottom. 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 to keep in mind here is that there may be more than one expression being evaluated in an **if** or **elif** clause. Let's check this.

We still 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 using the same **if** clause, 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 it 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, you can see that I used parentheses to surround both conditions specified in each of the **if** and **elif** clauses. This is completely optional, and the final result would've been the same if I would've skipped the parentheses, however they definitely increase the readability of the code when dealing with multiple conditions on the same line of code.

Next, it's obvious that Python printed out the string in the **if** block and the value of **x** because both expressions mentioned in the **if** clause were evaluated as **True**,

which leads to a final value of True when using the **AND** logical operator. Python executes the indented code below the **if** clause and exits, without checking any other clauses or expressions below. If the expression in the **if** clause would have been evaluated as False, then Python would've jumped to the subsequent **elif** clause and then maybe to the next one and then maybe to a final **else** clause, 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 it is not equal to 5")
```

This time, the final result is:

```
x is an integer, but it is not equal to 5
```

In this case, the expression in the **if** clause 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 clause - 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** clause, 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 test this, too, by making **x = 100**.

```
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 it is not equal to 5")
```

Now save your code and run this script in the Windows command line.

```
C:\Windows\System32>python D:\test1.py

C:\Windows\System32>
```

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. In this case, we don't have any other lines of code in our script besides the **if/elif** block, so there's nothing to return as a result.

In conclusion, remember that Python handles **if/elif/else** clauses 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.

More information:

https://docs.python.org/3/tutorial/controlflow.html#if-statements

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 in the Python interpreter.

```
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. Let's write this.

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

So, we start by typing in the **for** keyword, then we enter **the iterating variable**, which is 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. Now, let's run this **for** loop in the Python interpreter:

```
vendors = ["Cisco", "HP", "Nortel", "Avaya", "Juniper"]
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 test it in the Python interpreter.

```
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 book. A **range** 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, multiplied by 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 original 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.

One thing I'll add here is that you can also use an **else** clause 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 run 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
```

Going back to the iterating variable which is temporary and user-defined, you can also use an underscore if you don't plan on using that variable inside the indented code block underneath the **for** statement. Basically, you don't care about that variable in particular, you only want to iterate over a sequence and perform other operations.

```
for _ in range(1, 3):
    print("Whatever, I don't need the iterating variable.")
```

```
Whatever, I don't need the iterating variable.
Whatever, I don't need the iterating variable.
```

Another use case might be to multiply a pre-defined variable with each element of a **for** loop, thus performing multiple consecutive math operations, such as this:

```
>>> my_list = [0, 2, 4, 6]
>>> my_num = 3
for i in my_list:
    my_num *= i

>>> my_num
0
```

Here, on the first iteration, **my_num = 3** and it gets multiplied by the first element in the list, 0. The result is 0. At the second iteration, **my_num** which is now 0 is now multiplied by the second element in the list and the result is again 0. This keeps happening until the list is exhausted. For this reason, the final **my_num** is equal to 0.

That's most of what you need to know 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.

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

LOOPS - WHILE / WHILE-ELSE

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

Well, unlike a **for** loop, which executes a code block several 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 and run 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 should 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**.

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

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

As a quick reminder, you can use `x += 1` instead of `x = x + 1`, the two expressions being equivalent and returning the same result.

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

```
1
2
3
4
```

```
5
6
7
8
9
10
```

Now, let me explain what just happened. Python takes the first value of **x**, which is initially 1, evaluates it against the **x <= 10** condition as being **True**, prints **x** to the screen and then increments **x** by 1. Now, **x** being 2, it is again compared to 10 and the result is, again, **True** - therefore 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 to 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** anymore.

But what would happen if we wouldn't specify the `x = x + 1` 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 to 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:

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

```
1
1
1
1
1
[output omitted]
```

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
```

Another thing on **while** loops is that a while loop can have its own **else** clause, where 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 gets executed. Pretty cool, I think. Very useful stuff.

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**.

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

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** clause inside another **if** clause, a **for** loop inside another **for** loop or a **while** loop inside another **while** loop. Let's use some basic examples to see what I mean.

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

When nesting an **if** clause inside another **if** clause, we are actually telling Python that the indented code below the nested **if** clause should be executed only if both the inner clause and the outer clause are evaluated as **True**. Let's see what I mean by that.

```
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** clause - the outer clause - checks whether the letter "i" is part of this string or not. If this statement is evaluated as **True** - and it is - then Python moves further and evaluates the condition in the second **if** clause, which is also **True** since the string has more than 3 characters. Finally, since both statements have been evaluated as being **True**, the code below the inner **if** clause executed.

The result of this nested **if** structure is the one below.

```
C:\Windows\System32>python D:\nesting.py
Cisco 5
```

The code inside **nesting.py** is actually performing the same tasks as using the **AND** logical operator between the two expressions, within a single **if** clause.

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

The result is, indeed, the same.

```
C:\Windows\System32>python D:\nesting.py
Cisco 5
```

The parentheses surrounding each of the conditions inside the **if** clause are completely optional and their role is to provide better readability of your code. I strongly recommend using them, in order to have your code as clean 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** clause is indented one level to the right from the first **if** clause, using the Tab key.

Then, the code under the second **if** clause is indented two levels to the right from the first **if** clause and one level to the right from the **if** clause right above it. 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 very careful when adding a new level of indentation and also when returning to the previous indentation level.

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

Let's assume that 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, taking each element into account, then we need to do the multiplications and return the results.

```
list1 = [4, 5, 6]
list2 = [10, 20, 30]
for i in list1:
   for j in list2:
          print(i * j)
```

Now let's save the file and run it once again in the Windows command line.

```
C:\Windows\System32>python D:\nesting.py
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!". All clear?

Keep in mind that you can also go from the second indentation level back 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 in practice. Let's say that after doing the multiplications we did for each element of **list1**, we want to also print that element of **list1** to the screen, for each iteration. For this, we can go back one level of indentation and just type in **print(i)** at the same level with the second **for** loop.

Please see the code on the next page.

```
list1 = [4, 5, 6]
list2 = [10, 20, 30]
for i in list1:
   for j in list2:
         print(i * j)
   print(i)
```

Now, after saving the new version of our Python script let's run it in the cmd.

```
C:\Windows\System32>python D:\nesting.py
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 each of the elements of **list2**. So, nesting a **for** loop inside another **for** loop can be pretty useful sometimes.

Now, let's also have a look at **while** loops, too and we shall consider the same example we discussed in the chapter about **while** loops.

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

Ok, let's save this file and run it using the Windows command line.

```
C:\Windows\System32>python D:\nesting.py
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's write a nested **while** structure and then we will analyze it, as always.

Please see the code on the next page.

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

Now, let's see the results of running this new piece of code.

```
C:\Windows\System32>python D:\nesting.py
5
6
7
8
9
[output omitted]
5
6
7
8
9
10
```

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 equal to 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 you'll find it useful.

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.

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

Let's save and run this Python script now once again and see the results.

```
C:\Windows\System32>python D:\nesting.py
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 out 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** clause. 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 command line.

I think you now have a very good understanding of nesting and indentation, so it's time to move on to a new set of concepts and skills.

BREAK, CONTINUE, PASS

The **break** and **continue** statements are used to handle the flow of a **while** or a **for** loop inside 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.

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

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

```
C:\Windows\System32>python D:\break.py
0
1
2
3
4
5
6
```

Ok, what this **for** loop does is it takes the numbers in the range generated by the *range()* function - so 0 up to 9, inclusively - and then compares them to 7, using the nested **if** clause.

Since the numbers 0 to 6 do not meet the condition in the **if** clause, the **break** statement is not executed for any of them. Therefore, Python goes further into the program and prints them out to the screen, according to the *print()* function.

When the number 7 is selected from the range, it is then passed to the **if** clause and the condition is now evaluated as **True**, because 7 == 7. This is where Python executes the indented code below the **if** clause, 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** loop, if any.

Now, what if we have a **for** loop nested inside another **for** loop? Let's test this scenario, as well.

```
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 see the results.

```
C:\Windows\System32>python D:\break.py
40
Outside the nested loop
50
Outside the nested loop
60
Outside the nested loop
```

So, as we've seen in the previous chapter, 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. However, in this case we have the **if** clause inside the nested **for** loop and the **break** statement under this **if** clause. Therefore, each time Python tries to multiply the elements of **list1** with 20, the **break** statement tells it to stop the execution and exit the loop.

When dealing with nested loops, 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 **for** loops.

Notice that each time the **break** statement interrupts the multiplication process, we still get the "Outside the nested loop" string printed out to the screen. That's because the *print("Outside the nested loop")* line is not part of the nested **for** 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.

Instead, 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** inside 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 see the results.

```
C:\Windows\System32>python D:\break.py
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 it did 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** clause, 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 the **continue** statement 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 return anything. 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 application, and you want to design the behavior of this **for** loop later.

But what if you just don't write anything below the **for** statement? Let's try it inside a Python script and see what happens.

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

That's it. Save the script as it is and run it in the Windows cmd.

```
C:\Windows\System32>python D:\break.py
File "D:\break.py", line 2
^
SyntaxError: unexpected EOF while parsing
```

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.

More information:

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

PYTHON EXCEPTIONS

In Python, you may encounter two 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)

Traceback (most recent call last):
File "<pyshell#30>", line 1, in <module>
for i in rnge(10):
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 that the variable or function I'm trying to use is not defined in my namespace. Don't worry, we will talk more 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
Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
4 / 0
ZeroDivisionError: division by zero
```

Ok, fair enough, we cannot divide a number by 0, so Python raises a special exception for this particular case, called *ZeroDivisionError*, also specifying the cause that generated this exception - *"division by zero"*.

Moreover, please notice that Python helps us identify the location where the exception was originated by also displaying the line of code that generated the exception and the line number, as well.

Another type of Python exception is the KeyError. We get this exception if, for instance, we try to reference an invalid key in a dictionary, like in the example below.

```
>>> dict1 = {1: "a", 2: "b"}
>>> del dict1[3]
Traceback (most recent call last):
  File "<pyshell#391>", line 1, in <module>
    del dict1[3]
KeyError: 3
```

There are many types of exceptions in Python. We won't analyze all of them in this chapter because you will most likely encounter most of the 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 included below.

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

More information:

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

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

TRY / EXCEPT / ELSE / FINALLY

Ok, we've seen some Python exceptions and what they look like, but, given the fact that an exception interrupts the normal execution of your Python code, it would be more elegant and convenient for us to properly handle an exception when it occurs.

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

Let's consider the *ZeroDivisionError* exception from the previous chapter 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 (to the right) 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 that you want to be executed.

We will use the same **for** loop from the previous examples.

```
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 that we want to check for exceptions. This code will also be indented one level to the right (one Tab) 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 *ZeroDivisionError* 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 execute this code.

```
C:\Windows\System32>python D:\try.py
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 *ZeroDivisionError* 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 *ZeroDivisionError* 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 *ZeroDivisionError* exception may occur in the code under the **try** clause, but instead we wanted to cover any *NameError* exception which could have been generated by the program.

Therefore, 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, we should still get a *ZeroDivisionError*, right?

```
C:\Windows\System32>python D:\try.py
Traceback (most recent call last):
File "D:\try.py", line 3, in <module>
print(i / 0)
ZeroDivisionError: division by zero
```

As you can see, this time Python returned the standard exception message, because we didn't specify any customized message for the *ZeroDivisionError* exception.

Another thing I want to clarify is that if the code under the **try** clause does not generate any exceptions at all, then Python doesn't even look at the **except** clause and moves on to the rest of the lines of code in the program, if any.

To prove this, I'm going to change the denominator for the division operation to 1, instead of 0, so that the operation will be allowed.

```
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 execute the code once again.

```
C:\Windows\System32>python D:\try.py
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 that 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** clause and then executes the rest of the code - the *print()* function - without even caring that **except** is standing right there. Pretty rude, shame on you, Python!

Another thing to mention 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 the **try** clause. For instance, let's add a couple of **except** clauses to our existing code, just so you'll know how to handle multiple potential exceptions in your Python applications.

```
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 any exception type to be expected, just like you see below.

```
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 at runtime and hide the actual root cause of the exception. I strongly recommend creating 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 run the code.

```
C:\Windows\System32>python D:\try.py
2.0
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 operation 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 piece of code that doesn't generate any exceptions under the **try** clause.

```
try:
   print(4 / 2)
except ZeroDivisionError:
   print("Division Error!")
finally:
   print("I don't care, I'm getting printed either way!")
```

Let's execute the code.

```
C:\Windows\System32>python D:\try.py
2.0
I don't care, I'm getting printed either way!
```

The result is 2.0 followed by 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!")
```

Running the code once again.

```
C:\Windows\System32>python D:\try.py
Division Error!
I don't care, I'm getting printed either way!
```

Ok, so both scenarios were covered by the **finally** clause, as expected. That's most of what you need to know about exceptions and exception handling for your future programming tasks. However, in the next chapter you're going to learn more about how to manage and fix the errors and exceptions that occur in your code.

More information:

https://docs.python.org/3/tutorial/errors.html#handling-exceptions

PYTHON 3 - HANDLING ERRORS AND EXCEPTIONS

FIXING SYNTAX ERRORS

In this section of the notebook we're going to take a closer look at errors and exceptions in Python and how to manage and fix them inside your code.

In Python there are two types of errors that your code might generate: syntax errors and exceptions - and we're going to take a look at both of these types of errors and see how to properly read, interpret and fix them.

First of all, please keep in mind that any error that's not a syntax error is an exception in Python. Exceptions are also called runtime errors, because they are generated at some point during the execution of your Python code, whilst syntax errors are, as their name implies, just small errors that are related to the way you wrote your code - in short, this refers to the syntax you misused inside your Python script.

So, for now, let's focus on syntax errors and let's leave exceptions for later.

I have a sample Python script that you can see below, called **syntax.py**, containing a couple of syntax errors. Please keep in mind that Python evaluates each line of code **in order, from top to bottom**, validating the syntax of each line and throwing an error whenever the syntax is incorrect.

```
myvar = [1, 2, 3]

print myvar

for i in myvar
   print(i * 10)

myint = 10

float(myint]
```

Let's run this script and see the results.

```
C:\Windows\System32>python D:\syntax.py
File "D:\syntax.py", line 3
print myvar
^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print(myvar)?
```

Ok, so, as you might have already guessed, Python throws a syntax error. Let's read the error together. On the first line, Python provides us with the path and name of the file that generated the error - in my case that is **D:\syntax.py**. Furthermore, we also get the line in the file where the syntax error has occurred - so, we have line 3. If we check our code, on line 3 we have the **print myvar** piece of code, which is incorrect in Python 3, because *print()* is a function and we skipped the parentheses that should've enclosed **myvar**.

Let's move further and read the error text itself. It says *SyntaxError* - so we know that this is indeed an error related to the syntax we used in the code. Next, the error message provides more details and a potential solution for fixing the syntax, as well.

Missing parentheses in call to 'print'. Did you mean print(myvar)?

Indeed, in order to fix this issue, we should add the parentheses surrounding the name of the variable, **myvar**. So, let's go ahead and do this inside our script.

```
myvar = [1, 2, 3]

print(myvar)

for i in myvar
   print(i * 10)

myint = 10

float(myint]
```

Ok, it's fixed. Let's run our code once again in the Windows command line.

```
C:\Windows\System32>python D:\syntax.py
File "D:\syntax.py", line 5
for i in myvar
^
SyntaxError: invalid syntax
```

It seems we have yet another syntax error, in the same file, but this time on line 5, when building the **for** loop.

This time, the up-arrow is pointing to the exact location of the syntax error, because, as you might have already noticed, we missed the colon after the **for** statement. On the last line of the error, Python tells us that this is indeed a syntax error and that our syntax is invalid. So, let's go to line 5 inside our script and fix this right away.

```
myvar = [1, 2, 3]

print(myvar)

for i in myvar:
   print(i * 10)

myint = 10
float(myint]
```

Now, let's run the script again and see if we have any other errors in the code.

```
C:\Windows\System32>python D:\syntax.py
File "D:\syntax.py", line 10
float(myint]
^
SyntaxError: invalid syntax
```

Yes, we do have another error in the code, on line 10, again being a syntax error. This time, obviously, I used a square bracket instead of a closing parenthesis for the *float()* function. So, let's fix the code once again and run it for the last time.

```
myvar = [1, 2, 3]

print(myvar)

for i in myvar:
   print(i * 10)

myint = 10
float(myint)
```

The result:

```
C:\Windows\System32>python D:\syntax.py
[1, 2, 3]
10
20
30
```

Great, this time we didn't get any more syntax errors, so at this point we can say that our code is correctly written.

FIXING EXCEPTIONS

In this chapter we're going to discuss some of the exceptions that you might encounter on a regular basis while running your Python code. Notice that I said: "while running your Python code". So, keep in mind that exceptions are generated at runtime, as a result of incorrect operations within your code. Let's use another script I wrote - **exceptions.py**, that you can find down below - and let's see some of these exceptions in action. There are many more types of exceptions that you might encounter in your Python programming career, however, we will focus only on 7 of them for now, just to see how to read the error messages that Python throws and how to identify the root cause of the exception.

```
from sys import abcdef

import abcdef

var1 = 10
var2 = 0

r1 = var1 / var2
print(r1)

r2 = var1 + "hello"

text = "Python"
r4 = text[20]

while True:
   print("hi")
```

First of all, we have the **from...import...** statement at the top of our script, which attempts to import a name - *abcdef* - from within the **sys** module, which is a valid built-in Python module. Since *abcdef* is not a valid name inside the **sys** module, when running our code Python evaluates each line, from top to bottom, and finds this name to be invalid, thus throwing an exception. Let's run the script to see the result.

```
C:\Windows\System32>python D:\exceptions.py
Traceback (most recent call last):
File "D:\exceptions.py", line 1, in <module>
from sys import abcdef
ImportError: cannot import name 'abcdef' from 'sys' (unknown
location)
```

Let's read the text here. So, Python traces the cause of this exception in the file we just ran - that would be **exceptions.py**, on my D: drive - on line 1. Next, Python also prints out the line of code that raised the exception - *from sys import abcdef*.

On the next line in this error message we have the type of exception that was raised, called *ImportError*, and the error message saying that Python *"cannot import name abcdef from sys (unknown location)"*, meaning that this name wasn't found inside the module. So, this is one of the many types of exceptions that Python can generate.

Let's move further and comment out the **from...import...** line, so that Python will ignore it and evaluate the next line in the file.

```
#from sys import abcdef

import abcdef

var1 = 10
var2 = 0

r1 = var1 / var2
print(r1)

r2 = var1 + "hello"

text = "Python"
r4 = text[20]

while True:
  print("hi")
```

After saving the file, let's run it once again.

```
C:\Windows\System32>python D:\exceptions.py
Traceback (most recent call last):
File "D:\exceptions.py", line 3, in <module>
import abcdef
ModuleNotFoundError: No module named 'abcdef'
```

This time, Python encountered yet another error and raised another type of exception. So, as you can see on line 3 - which is *import abcdef* - Python throws the the *ModuleNotFoundError* exception, saying that there's no module named *abcdef*. Pretty straightforward, I think, so we can comment out this line as well and move on to discover other exceptions that our code might generate.

```
#from sys import abcdef
#import abcdef

var1 = 10
var2 = 0

r1 = var1 / var2
print(r1)

r2 = var1 + "hello"

text = "Python"
r4 = text[20]

while True:
  print("hi")
```

Next, we can see that I've defined two variables, **var1** and **var2**, pointing to the values of 10 and 0, respectively. On the next line, after defining these variables, we are dividing **var1** by **var2**, then storing the result using the **r1** variable and finally we're trying to print out the value of this variable to the screen. Let's run our code and see what happens.

```
C:\Windows\System32>python D:\exceptions.py
Traceback (most recent call last):
File "D:\exceptions.py", line 8, in <module>
r1 = var1 / var2
ZeroDivisionError: division by zero
```

Ok, so we have yet another exception, this time it's the *ZeroDivisionError* exception because, obviously, we tried dividing a number by 0, which is mathematically incorrect - and Python notices our mistake and acts accordingly. As you can notice, this exception is raised on line 8 of the file, where the division operation is attempted. The error message shows the line number, the line of code itself and, of course, the cause of the exception - *"division by zero"*.

Let's comment out line 8 as well - the one performing the division operation - and run our code once again to see yet another type of exception in Python.

```
#from sys import abcdef
#import abcdef

var1 = 10
var2 = 0

#r1 = var1 / var2
print(r1)

r2 = var1 + "hello"

text = "Python"
r4 = text[20]

while True:
   print("hi")
```

The result of this new version of our code is:

```
C:\Windows\System32>python D:\exceptions.py
Traceback (most recent call last):
File "D:\exceptions.py", line 9, in <module>
print(r1)
NameError: name 'r1' is not defined
```

This time, we got the *NameError* exception on line 9. Why? Because line 9 attempts to print the value of **r1**; however, we just commented out the line of code defining **r1** and, since Python ignores comments, **r1** doesn't exist anymore in our program. That's why Python doesn't know who **r1** is when it reaches line 9 of the code and therefore throws the *NameError* exception saying that *"name 'r1' is not defined"*, which is pretty self-explanatory.

Ok, moving forward, let's also comment out line 9, run the script once again and see what other surprises lay ahead.

```
#from sys import abcdef
#import abcdef

var1 = 10
var2 = 0

#r1 = var1 / var2
#print(r1)

r2 = var1 + "hello"

text = "Python"
r4 = text[20]

while True:
   print("hi")
```

Now, the result of running the code above is:

```
C:\Windows\System32>python D:\exceptions.py
Traceback (most recent call last):
File "D:\exceptions.py", line 11, in <module>
r2 = var1 + "hello"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

This time notice we got the *TypeError* exception, on line 11. Why? That's because on this line we are attempting to add an integer and a string - **var1** and **"hello"**, which is an illegal operation in Python. Notice that the error message is very clear about the cause of this exception - *"unsupported operand type(s) for +: 'int' and 'str'"*. This means the plus operator is not supported between an integer and a string, as I've already mentioned. The *TypeError* exception is one of the most frequent types of exceptions that you may encounter in Python and being able to read and interpret the error message is key to troubleshooting your code.

I want to show you two more types of Python exceptions and then we're going to wrap this up. First, let's comment out line 11 as well and let Python evaluate the rest of the code in our script.

```
#from sys import abcdef
#import abcdef

var1 = 10
var2 = 0

#r1 = var1 / var2
#print(r1)

#r2 = var1 + "hello"

text = "Python"
r4 = text[20]
```

```
while True:
   print("hi")
```

Now, let's run the script again.

```
C:\Windows\System32>python D:\exceptions.py
Traceback (most recent call last):
File "D:\exceptions.py", line 14, in <module>
r4 = text[20]
IndexError: string index out of range
```

Here, you can notice the occurrence of the *IndexError* exception on line 14. Why? Because we previously defined the variable called **text**, pointing to this string - "Python". This string contains 6 characters, so the largest index inside this string is going to be 5, right?

However, on the next line, we're trying to extract the character positioned at index 20 inside our string, which, of course, does not exist. Therefore, Python raises the *IndexError* exception, saying that the string index we asked for is out of range. Basically, this means that our string does not have an element positioned at index 20 because it lacks the necessary length to have such an index. Which is pretty obvious.

Finally, let's comment out line 14 as well and focus on the **while** loop.

```
#from sys import abcdef
#import abcdef

var1 = 10
var2 = 0

#r1 = var1 / var2
#print(r1)
#r2 = var1 + "hello"

text = "Python"
#r4 = text[20]

while True:
   print("hi")
```

First of all, we have **while True** - which basically creates an infinite loop. So, **while True**, print out the string in between the parentheses of the *print()* function to the screen. Of course, this will result in this sting being printed out indefinitely. As you may remember from earlier, in order to break out of an infinite loop, we can use the **Ctrl+C** key combination.

So, let's run our code once again, see the infinite loop running and then interrupt this loop.

```
C:\Windows\System32>python D:\exceptions.py
hi
hi
hi
hi
hi
[output omitted]
```

```
hi
hi
hi
Traceback (most recent call last):
File "D:\exceptions.py", line 17, in <module>
print("hi")
KeyboardInterrupt
```

Immediately after hitting **Ctrl+C** the infinite loop is stopped and Python throws the *KeyboardInterrupt* exception, pointing to the *print()* function at line 17. Great job!

So, these were some examples of raising and troubleshooting exceptions in Python. Although you may encounter many other types of exceptions, the procedure for identifying and fixing the root cause is the same. Again, don't fear these exceptions. They are an incredible source of learning, especially when you're just starting your programming career.

Finally, whenever you're in doubt about why you ended up with a particular exception or how you should fix it, the first step every programmer does is copying the exception message - for instance: `IndexError: string index out of range` - and pasting it into Google's search engine.

Most probably, you're going to end up on the stackoverflow.com website, which is perhaps the best source of troubleshooting information for any programmer. Click on the first link, check out the results and see if any of the replies make any sense to you. Try that with multiple search results and focus on putting everything together to find out more about the exception you're getting and its potential root cause.

PYTHON 3 - FUNCTIONS

BASICS OF FUNCTIONS

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 Python applications. But, first of all, what is a function good for?

Well, you can use a function to organize your code in blocks that can be later reused. This is extremely helpful when you want better readability for your code, modularity, and also time saving when designing and running your code.

Before we start coding, please keep in mind that functions follow the same syntax rules as other structures that we've discussed so far, but also add some features that we're going to analyze shortly.

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 that you want to store inside 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 in which 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. You 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 that we want this function to execute, indented one level to the right, 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, the docstring is entirely optional. Perhaps you can use it when you define complex functions within complex applications, to remember the role of each 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()* built-in function.

```
>>> help(my_first_function)
Help on function my_first_function in module __main__:

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 you've learned up to this point in the notebook. 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 mentioned 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!
```

As you can notice, each time we call the function the code inside that function gets executed. Of course, this is a very basic example of a function, but it should give you an overall understanding of this concept.

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. Therefore, we can say that functions are dynamic structures. Let's redefine our function and change the code inside it. Then, on the next function 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 chapter, when you're going to learn about 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. 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 of this function to perform a certain task. 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, whereas **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.

We will discuss arguments in more detail in the following chapter. 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 to the screen.

You can also insert multiple parameters during a function definition.

```
def my_first_function(x, y):
   print("Hello " + x)
   print("Hello " + y)
```

According to the piece of code above, we expect that when calling the function and passing two strings as arguments inside the parentheses they will both be printed out to the screen, 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.

```
def my_first_function(x, y):
   sum = x + y
```

In the code sample above we have created a variable inside our function, called **sum**, 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)
>>>
```

Nothing is returned. That's because we haven't specified what exactly we are 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 **x** and **y**. However, it doesn't return anything when called. 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, when calling the function.

Now, we can change the function yet again and call it using another set of **arguments**. For instance, let's add another **parameter**, called **z**, change the mathematical 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 and check out the result.

```
>>> my_first_function(1, 2, 3)
81
```

The result is the correct one, since ((1 + 2) * 3) ** 2 is 81.

The last thing I should add here is that if you just use the **return** statement, without specifying what you want to get out of the function, Python will return the *None* value. Therefore, **return** without specifying what to return is, actually, the same thing as **return None**. Let's test this.

Finally, you can also return **multiple values** using the **return** statement, by simply adding a comma in between the values to be returned. The result will be a **tuple** containing the results, on which you can perform specific operations such as slicing.

```
>>> my_first_function(2, 3, 4)
(20, 60, 120)
>>> type(my_first_function(2, 3, 4))
<class 'tuple'>
>>> my_first_function(2, 3, 4)[1]
60
>>> my_first_function(2, 3, 4)[:1]
(20,)
```

Recursion is another topic that comes up a lot when writing Python functions.

With recursion, a function calls (or invokes) itself in order to fulfill the needs of the programmer when designing more advanced applications. Let's take a basic example of recursion.

```
def my_func(x):
    if x == 3:
        return x + 1
    return x * my_func(x - 1)

>>> my_func(5)
80
```

First of all, notice how *my_func()* invokes itself inside the **return** statement.

Now, when calling the function using 5 as an argument the logic is as follows:

- The **if** statement checks whether the argument is == 3, which is **False**.
- Therefore, the idented code below the **if** statement is not executed.
- Then, in the **return** statement, 5 is multiplied by the result of my_func(5-1), so my_func(4). Now who is my_func(4)?
- The function is invoked once again, and this time 4 still doesn't meet the condition in the **if** statement, so the final **return** is executed again.
- Now, we have 4 multiplied by the result of my_func(4-1), so my_func(3). The function is invoked once again.
- This time, x == 3. Therefore x gets incremented by 1, as instructed by the **return** statement indented under **if**.
- So, during the initial function call, x was 5, then it was multiplied by (5-1) and finally by (3+1). Therefore, 5 * 4 * 4 = 80.

Now, let's move on and discuss about function arguments.

FUNCTION ARGUMENTS

Ok, we've seen how to use function **arguments**, we went through some examples, but why do we need a special chapter 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 chapter.

The type of arguments we've seen in the previous chapter 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 raise a *TypeError* exception. Let's test this by considering 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* exception, 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)
Traceback (most recent call last):
File "<pyshell#91>", line 1, in <module>
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 to **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. Let's see a couple of examples.

```
>>> my_first_function(x = 1, y = 2, z = 3)
9
```

Keyword arguments allow us to also change order of arguments during the function call, 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 throw 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**.

```
>>> my_first_function(4, 7, z = 3)
33
>>> my_first_function(14, 35, z = 3)
147
>>> my_first_function(11.5, 29, z = 3)
121.5
```

What else? You can also define a **default value** for a parameter, 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**. Therefore, 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 yet again, but this time using several arguments and see if the tuple gets populated.

```
>>> my_first_function("hello", 100, 200)
hello
(100, 200)
```

We have the confirmation that the **args** variable is indeed a tuple. But how can we print out 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 i in args:
           print(i)
```

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
```

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, the only difference being the type of arguments it applies to. 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.

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 or functions that we create inside our programs.

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 this book. 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 inside a Python script.

The global namespace contains all the variables or functions that you define or import directly into your program, whereas the local namespace refers to the names that you use only inside a particular function. Let's better see this by creating a Python script and playing around with some variables.

I'm going to create a new file called **namespaces.py**. 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 and run the file.

```
C:\Windows\System32>python D:\namespaces.py
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ok, so the script 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**. So, instead of the previous *print()* function just type in:

```
print(my_var)
```

Let's save the file again and try to run it one more time.

```
C:\Windows\System32>python D:\namespaces.py
Traceback (most recent call last):
File "D:\namespaces.py", line 1, in <module>
print(my_var)
NameError: name 'my_var' is not defined
```

As soon as we run the script, 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. Therefore, Python doesn't know where to get it from when trying to print out 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 script again, we notice that it 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.

```
C:\Windows\System32>python D:\namespaces.py
10
```

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 once again 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.

```
C:\WINDOWS\system32>python D:\namespaces.py
10
Traceback (most recent call last):
File "D:\namespaces.py", line 6, in <module>
print(my_var * 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.

Therefore, 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 once 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.

```
def my_var_func():
   my_var = 10
   print(my_var)

my_var = 20
my_var_func()
print(my_var * 10)
```

```
C:\WINDOWS\system32>python D:\namespaces.py
10
200
```

This time, we got the same value of 10, as before, by calling the *my_var_func()* 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

my_var_func()
```

Let's run the file once again.

```
C:\WINDOWS\system32>python D:\namespaces.py
Traceback (most recent call last):
File "D:\namespaces.py", line 5, in <module>
my_var_func()
File "D:\namespaces.py", line 2, in my_var_func
print(my_var)
UnboundLocalError: local variable 'my_var' referenced before
assignment
```

This time, we got the *UnboundLocalError* exception, saying that the *"local variable 'my_var' referenced before assignment"*. That's because Python reads the lines of code in order, from top to bottom. Therefore, if you first use the **my_var** variable inside a *print()* function and **then** create that variable, Python will be unaware that you've 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 your 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* exception, 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. So, let me show you how to do it. Inside the function, before any other statement that refers to the global variable you want to use, you just type in **global** and the name of the global variable that you want to reference.

```
my_var = 5

def my_var_func():
   global my_var
   print(my_var)
   my_var = 10

my_var_func()
```

Let's save the file as it is and run it.

```
C:\WINDOWS\system32>python D:\namespaces.py
5
```

The result is the value referenced by the global variable, printed out to the screen using the *print()* function inside the *my_var_func()* 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 code once again and return to the code we used at the beginning of this chapter, 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.

```
def my_var_func():
   my_var = 10
   print(my_var)
   return my_var

my_var_func()
print(my_var * 10)
```

The result of running the code above is:

```
C:\WINDOWS\system32>python D:\namespaces.py
10
Traceback (most recent call last):
File "D:\namespaces.py", line 7, in <module>
print(my_var * 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. In order to have this variable available outside the function, you can make use of the **return** statement 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 * 10)
```

Let's see the result now.

```
C:\WINDOWS\system32>python D:\namespaces.py
10
100
```

Now, we got 10 returned by the first *print()* function and the function call itself, and then 100 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 10.

So, this is the way you can propagate the value of **my_var** from the local namespace of the function all the way down to the *print()* function at the end.

Following up on the previous discussion, let's assume that we're passing a list to a function. Keep in mind that changing the list itself inside the function will not broadcast the change outside the function, however changing its contents does. Let's see an example to illustrate this.

```
def my_func(x):
    x = [1, 2, 3]


>>> my_list = [4, 5, 6]
>>> my_func(my_list) #passing my_list as argument to my_func()
>>> print(my_list) notice that after the call, my_list is the same
[4, 5, 6]
```

Now, if we modify out function to perform a modification in the list it receives as an argument, we notice that the change we made is not observable outside *my_func()*.

```
def my_func(x):
    x[-1] = 99


>>> my_list = [4, 5, 6]
>>> my_func(my_list) #passing my_list as argument to my_func()
>>> print(my_list) #after the call, the change is now visible
[4, 5, 99]
```

Ok, so in my view this is probably most of what you need to know about functions for your current stage in your programming path, as well as for the PCEP Exam. Of course, no document or course can cover absolutely every concept or nuance that may be included with the exam, but with the information in this document you should be able to get a passing score on the exam. Of course, if you study and practice.

PYTHON 3 – LISTS (BONUS)

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. List comprehensions are 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 2. 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 add 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 expecting. 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**. 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]
```

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]
```

Also, you might see list comprehensions that don't perform any operations before the **for** statement. Something as simple as printing a string a number of times.

```
>>> list4 = ["Hi" for x in range(10) if x < 5]
>>> list4
['Hi', 'Hi', 'Hi', 'Hi', 'Hi']
```

Now, let's see set and dictionary comprehensions, as well. 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 key-value pairs, where the key is separated from the value by a colon.

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 very 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.

More information:

https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

https://docs.python.org/3/tutorial/datastructures.html#nested-list-comprehensions

PYTHON 3 - FILE OPERATIONS

OPENING & READING

Python provides all the tools that 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 other file on 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 that Python provides to the user.

First of all, let's create a new text file - I'll do that on my D: drive - and insert some data into the file, and then we will start playing around with it. My file is called **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. Let's open up the Python interpreter and start working. First, you should type in a name for the file object that

you want to 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 the *open()* function and, in between its parentheses, you should first specify the name of the file that you want to open and the path to that 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.

Please note that for Windows file paths you should always use double backslash (\\ ) instead of a single backslash, to escape any potential special sequence that might interfere with the path and generate an error. Find more information on escape sequences here: https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals.

This means that a path such as *D:\Data\Stuff\Webinar\webinar.txt* should always be referred to as *D:\\Data\\Stuff\\Webinar\\webinar.txt* when using it in a Python application to avoid any unexpected behavior.

Now, let's open our text file for reading; for this, we will use **"r"** as the access mode. The most common mode in which you can open a file is **"r"** for read-only - 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.

Other frequently used modes are **"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, PDFs, executable files and so on; the binary mode should be used for all files that don't contain text.

Finally, we also 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 it 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. Please find more information about the built-in *open()* function and Python's file access modes here: https://docs.python.org/3/library/functions.html#open.

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 object, like this:

```
>>> myfile = open("D:\\routers.txt", "r")
>>> myfile.mode
'r'
```

Again, as I said mentioned, 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 book.

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. In between its 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)*.

```
>>> 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 contents, 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)
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'
>>> myfile.readline()
''
```

So, we got each line in the file printed out to the screen 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; *readlines()* is very useful for iterating over a file and it is frequently used when working with files.

```
>>> myfile.seek(0)
0
>>> myfile.readlines()
['Cisco\n',    'Juniper\n',    'HP\n',    'Avaya\n',    'Nortel\n',
'Arista\n', 'Linksys']
```

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 you will end up with an empty list when calling the *readlines()* method again. Let me prove this to you.

```
>>> myfile.readlines()
['Cisco\n',    'Juniper\n',    'HP\n',    'Avaya\n',    'Nortel\n',
'Arista\n', 'Linksys']
>>> myfile.readlines()
[]
```

And that's the proof. Now, as I said, let's use the *seek()* method once again.

```
>>> myfile.seek(0)
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 out 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#30>", 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.

```
>>> myfile = open("D:\\routers2.txt", "x")
```

And the new file has been created on my D: drive. You should try this, as well.

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 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 IDLE's icon and select **Run as administrator**, then click Yes. Now, you will be able to complete your file creation on the desired drive.

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"** mode 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"** file access mode. Now, let's create a new file object and a new file for writing.

```
>>> newfile = open("D:\\newfile.txt", "w")
```

Good, we now 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?")
22
```

So, what we did here 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#36>", 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 up the file in Notepad++. Go ahead and do that on your own.

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 soon, 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 with the *open()* function to create a new file for writing. Now, let's add more data to this file. First, we should open up 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!")
32
>>> newfile.close()
```

Go ahead and check the file once 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"** file access mode!

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 inside 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 out the contents of *newfile.txt* again, using Notepad++.

```
CiscoJuniperHP
```

Notice that the elements of the list that 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 mode. 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 once again.

```
AvayaNortelArista
This string was appended!
```

And, indeed, the old string was kept 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 up a file for writing and reading at the same time and, if the file doesn't exist, it will create it. Now, as you've already seen, if you open *newfile.txt* for writing using the **"w"** access mode and try to read it, we will get the `io.UnsupportedOperation: not readable` error again, right?

So, let's open up 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 read and write to the file, at the same time. Finally, don't forget to close the file!

```
>>> newfile.close()
```

More information:

https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

FILE CLOSING

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. I still have *newfile.txt* from earlier, 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've 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 previously did 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 up the file in Notepad++.

```
Hello Python!
```

Good, the string is there. Now, back to the Python interpreter.

```
>>> f.closed
True
```

Therefore, 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.

DELETING FILE CONTENTS

In this small chapter you're going to learn how to delete the contents of a text file, either totally or partially.

I have created a new file on my D: drive to test this out. The file is called *test.txt* and it contains some text.

```
Python is the greatest programming language of all time. Do you agree? :)
```

Now, let's head over to the Python interpreter and see how we can remove text from this file, using a special method that Python provides us with. First of all, let's open up the file for reading, using the *open()* method, as usual.

```
>>> f = open("D:\\test.txt")
```

If we check f, we see that it has been opened for reading - notice the **mode='r'**.

```
>>> f
<_io.TextIOWrapper name='D:\\test.txt' mode='r' encoding='cp1252'>
```

Now, let's use the *read()* method to see the contents of this file.

```
>>> f.read()
'Python is the greatest programming language of all time. Do you
agree? :)'
```

Great! Now, let's also check the number of characters inside this file, using the *len()* function, of course.

```
>>> len(f.read())
0
```

Oups, we got 0 characters. Why is that? Well, as you might have already guessed, we already read the entire file using the *read()* method and now the cursor is positioned at the end of the file. Our job is to reposition this cursor at the beginning. What was the method we used to achieve this? You guessed it, the seek() method.

```
>>> f.seek(0)
0
```

Now, let's try using the *len()* function once again.

```
>>> len(f.read())
73
```

And it seems that we have 73 characters inside our text file. Now, the method to use when you want to delete the contents of a text file is called *truncate()*. So, let's try it on our file and see if the text inside gets deleted or not.

```
>>> f.truncate()
Traceback (most recent call last):
```

```
File "<pyshell#6>", line 1, in <module>
f.truncate()
io.UnsupportedOperation: File not open for writing
```

And, surprise, we get an error. Why? Because in order to delete text from the file, we need to also have the file open for writing, but in our case, we only have the file available for reading, which is not enough. Therefore, let's close the file and open it once again, for reading and writing at the same time, ok?

```
>>> f.close()
>>> f = open("D:\\test.txt", "r+")
```

We're using the **"r+"** file access mode to open the file for reading and writing simultaneously. Great! Now, let's try to use the *truncate()* method once again.

```
>>> f.truncate()
0
```

Ok, this time we didn't receive an error, but we did receive the value of 0. This means that 0 characters are left inside the file after the use of the *truncate()* method. Now, you can check if this is indeed correct using Notepad++ and, indeed, after reloading the file we can now see that our text is gone. Great job!

However, at the beginning of this chapter I said that we can either remove the entire content of the file or just a part of it. Let's see how to partially delete the contents inside the file. First of all, let's close our file and re-insert the text that we previously had.

```
>>> f.close()
```

Next, let's open up the file once again using the **"r+"** access mode.

```
>>> f = open("D:\\test.txt", "r+")
```

And now, let's truncate this file and leave only the first characters in the file and delete the rest of the text. To do that, we just have to enter an integer in between the parentheses of the *truncate()* method. This integer is going to be the number of characters to keep inside the file. So, let's say we want the contents of the file to be deleted, except the first 10 characters.

```
>>> f.truncate(10)
10
```

Now, let's also close the file and then check the results:

```
>>> f.close()
```

As you can clearly see, we only have 10 characters left inside the file.

```
'Python is '
```

Also, we can open the file once again for reading, and check the result using the *len()* function, as we did before.

```
>>> f = open("D:\\test.txt", "r+")
>>> len(f.read())
10
>>> f.close()
```

Indeed, we got the results we were looking for, so great job, again. Cheers! :)

* * *

This e-book is useful both as an overall refresher for your Python 3 skills, as well as a foundation for practicing for your next test, exam or interview. I hope you enjoyed it.

Let's connect on my LinkedIn profile. Cheers!

~ The End ~