TECHNION
Azrieli Continuing Education and External Studies Division

Python Basics

Notebook

Contents

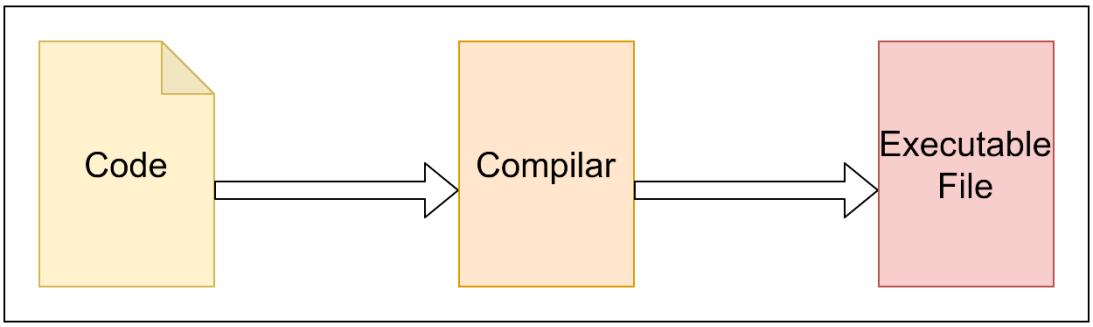
Part I: Introduction

Python is the most popular programming language according to various statistics. It is widely used in the world of Cyber Security and being able to write and read code is an essential skill that can significantly simplify various tasks.

Python considered to be easy to learn as it simplifies a lot of "complex" programing concepts.
The following book describes the essential Python skills that are required to write script and automations as well as basic programming concepts.

Compilation and Interpretation

To execute code, it must first be translated into machine code. There are two ways to translate the code. The first way is to translate all the code before execution and create an executable file. That method is called Compilation. Languages like C, C++, C#, Rust and Go are examples of compiled languages.
The second way to run the code is by executing each line of code separately. That method is known as



Interpretation. During the process of the interpretation the Interpreter translates each line of code into machine code and executes it. Languages like Python, Php and Ruby are examples of such languages. For example, in order to execute Python code, you have to use the Python's interpreter.

The interpreter reads the code line by line and executes it. Both methods have their advantages and disadvantages. For example, Python considered to be extremely easy but at the same time slow language, especially compared to languages like C or Go.

Installing Python

Python comes preinstalled on most Linux distributions and it does not require any additional manipulation to install. On Windows it is not installed by default but can easily downloaded and installed from the official website. To verify that Python is installed on your system, run the following command in the terminal and check the output.

```
user@host:~$ python3 --version
Python 3.7.3
```

Note: For this book, it doesn't really matter which exact version of Python you have as long as it is ***Python 3****.*

Executing Python Code

The files that contain the Python code are simple text files with the .py extension. Aside from the extension there is nothing special about those files. The code can be written using any text editor. Usually people prefer to use IDEs (Integrated Development Environment) to write the code, although it is not mandatory. The code that will be presented in this book will work in any type of environment.
Python's code is referred to as scripts a lot. It is due to the fact that many Python programs are actually scripts. A script is a simple program that is used to perform some linear operations.
The first Python script that you are going to write is a simple piece of code that prints a message to the screen. To do that, create a new file using the text editor/IDE. Do not forget the .py extension. (If you are using an IDE, it will probably do that automatically)
In the file that you create write the following lines of code.

```
#!/usr/bin/env python3

print('This is a python script')
```

The first line in the code is called Shebang. The Shebang is used to tell the system which interpreter to use when executing the code. The Shebang always placed in the first line of the file.
Note: The Shebang is not mandatory, but it is a good practice to have it in the script. Also in some cases, if there is no Shebang, the code will not execute.
After the Shebang, comes the print function. It is used to print text to the screen. The print function might be used for various cases, such as, interaction with the user, printing the status of the running code, displaying errors, etc. Notice that after the name of the function there is a pair of brackets. Each time you call a function it must be followed by brackets. Inside the brackets comes the string (text) that will be printed. After the script is written, save the file.
There are several methods to execute the script. If you are using an IDE, just click the Run/Start button. If you are using the command line, then there are two options. The first one is to call Python and specify the name of the file.

```
user@host:~$ python script_1.py
This is a python script
```

The second method is to add execute permissions to the script and run it directly.

```
user@host:~$ chmod u+x script_1.py
user@host:~$ ./script_1.py
This is a python script
```

The result of both methods will be the same. There is no performance difference between the methods.
*Note: For the second method, the **Shebang must** be in the script.*
As you can see, after the execution of the script, it simply prints the text to the screen and that's it. You can play around with it a little and make it print different texts to the screen. You can also add few additional print functions to make it print more lines of text to the screen.
It is relatively simple to execute Python code and it is due to the fact that Python is an interpreted language, so you do not have to compile the code before the execution.

Comments

Comments are lines in the code that are ignored by the compiler/interpreter. In Python, comments can be written using one of the two methods. The first method is to use a number sign (#). Every line that starts with # will be ignored by the interpreter.

```
#!/usr/bin/env python3

# This is a comment
```

The # can be used not only in the beginning of the line but it can also be added at the end of the line.

```
#!/usr/bin/env python3

print('Hello') # This is a comment
```

The second method is to surround the lines of code with three either single or double quotes. Any lines between the quotes will be considered a comment. Make sure that you use the same type of quotes when you open and close the comment.

```
#!/usr/bin/env python3

'''
This line is a comment
This is also a comment
'''
```

Part II: Variables and Data Types

As you saw previously, it is very simple to execute python scripts. The first script that you wrote is just a simple piece of code that prints one sentence to the screen, so no matter how many times the script will be executed, the result will always be the same. In reality, you want your code to be more flexible and ready for changes during run time (the execution). For situations like that, variables are used. A variable is just a simple placeholder for data. The data can be numbers, characters, text and more. For example, say that you want to write a script that asks the user for his name, that means that you must store the data that the user gave you. To store it a variable will be used.

The following piece of code demonstrates usage of a variable.

```
#!/usr/bin/env python3

msg = 'This is my text'

print(msg)
```

In line 4, a variable *msg* is declared, and the text that will be printed is placed into the variable. And then, in line 6 the text is printed to the screen. So, if you need to print the same text more than once, you can use the variable instead of writing the text again and again. Variables can be used in many situations which will be demonstrated in the book.

Data Types

Before proceeding to the use cases of variables, first we have to cover the types of data that can be placed into the variables. As stated before, variables can store any types of data, but it is very important to know which types exist. The following data types are the basic types that exist in python.
Note: Lists, tuples and dictionaries will be covered later.

Integer

An integer is a simple number without the decimal point. It can be either positive, negative or zero. To declare a variable that stores an integer, write the name of the variable and the value that you want to assign to it.

```
number_1 = 3
number_2 = -91
number_3 = 0
number_4 = 1984
```

Note: Unlike some other languages, in Python the integers can store any number, there is no upper or lower limit. The only limit that exists is the amount of available memory.

Float

A float is a decimal point number, it can store any value either positive or negative.

```
number_1 = 3.333
number_2 = -91.11
number_3 = 0.0
number_4 = 1984.987
```

Note: 0.0 is also a float as it contains the decimal point.

String

A string is a sequence of characters. For example, a text that is printed to the screen. The computer does not understand human languages, thus for it any text is just a sequence of characters.

```
str_1 = 'Hello'
str_2 = 'This is my Python program'
```

Unlike integers and floats, strings must be encapsulated with either single (‘) or double (“) quotes. Python threats both types of quotes the same. There is no strict rule on which types of quotes must be used, but usually the single quotes are more popular as they are quicker to type.
Note: Remember that spaces are also characters as all the others.

Boolean

Booleans are the simplest type of data. A Boolean has only two possible options, True or False. They are mostly used to performs various checks which will be discussed later.

```
is_human = True
is_dog = False
```

Variable Declaration

In the previous examples you seen some declaration of variables. There is a simple set of rules that defines how variables should be declared.

- The variable names can contain letters, numbers, and underscore (_).
- Variable name cannot start with a number.
- Spaces are not allowed for variable names.
- Variable names are case sensitive.
- In Python, the accepted way of naming multi word variables is by using snake case.
- Most of the variables use lowercase characters and numbers only.
- Uppercase variables are used as global variables.

Once the name of a variable is written it will be followed by an equal sign '='. After the equal sign comes the value that will be stored in that variable. The following is an example of a correct variable declaration.

```
first_name = 'john'
age = 26
refund = 499.99
is_secure = True
number_1 = 13
number_2 = 42
```

When a variable is declared, a value must be put into it. If a variable is declared without putting a value into it an error will occur. The value that is assigned into the variable can be changed at any time. Once a new value is put into a variable it overwrites the previous data that was stored in that variable. Additionally, Python allows to change the type of a value that is store in a variable. For example, if there is a variable that stores an integer, it is possible to replace it with a float, string, Boolean, etc.

```
number = 13
print(number)
number = 99
print(number)

OUTPUT:

13
99
```

```
var_1 = 91
print(var_1)
var_1 = 'This is a string'
print(var_1)

OUTPUT:

91
This is a string
```

As you can see, variables in python are very flexible, especially compared to other languages.
Once important thing to remember when dealing with variables it that the names of the variables must be named clearly. So, when the code is vied later or by someone else, it will be much easier to understand it.

Arithmetic Operations

Python allows to perform arithmetic operations on numbers (integers/floats). All the operations are performed during the runtime of the script/program. The result of the operation can be either stored in a variable or used immediately. The available operations are:

+	Plus
-	Minus
*	Multiplication
/	Division
**	Power
%	Modulo

The operations can be done with hardcoded numbers, variables, or both. Additionally, Python deals with all the mathematical rules automatically. The following code shows examples of arithmetic operations.

```
#!/usr/bin/env python3

num_1 = 3 + 9
num_2 = 5 - 2
num_3 = 4 + 1 * 2

sum_1 = num_1 + num_2
sum_2 = num_1 + 98

num_1 += 3
num_2 *= 6
```

As you can see in line 5, operations can be combined, and Python will handle the order automatically (multiplication before addition). In lines 10 and 11 there is a shortcut, the same two lines can be written as:

```
num_1 = num_1 + 3
num_2 = num_2 * 6
```

Also, in those lines you can see that you can store the result of the operation in the same variable that was part of the calculation.
When performing operations with numbers it is important to remember the data types. An operation that is done on two integers will also return an integer in all the cases except for division. If the operation is done with float and an integer, the result will be an integer.

String Operations

One of the interesting features of Python is that it allows to multiply strings. Say for example, you need to print a line on the screen. It can be done by printing the '-' symbol multiple times in a row.

```
line = '-' * 30
print(line)

OUTPUT:

------------------------------
```

Or you can do the multiplication directly in the *print* function.

```
print('-' * 30)

OUTPUT:

------------------------------
```

Multiplication can also be done on strings longer than one character. For example, if need to print the same string repeatedly you can use the following pattern.

```
print('-|-' * 10)

OUTPUT:

-|--|--|--|--|--|--|--|--|-
```

```
print('hello ' * 3)

OUTPUT:

hello hello hello
```

In addition to the multiplication, it is also possible to add (concatenate) one string to another using the *+* operator.

```
first_name = 'Jonh'
last_name = 'Doe'

full_name = first_name + ' ' + last_name

print(full_name)

OUTPUT:

John Doe
```

Working with Input and Output

A lot of programs require some sort of input from the user to perform operations with the provided data. The input can come in various methods but one of the simplest ways to get data from the user is by using the *input* function. The input function gives the user the ability to enter data to the program.
The following code demonstrates an example of *input* usage.

```
#!/usr/bin/env python3

print('Enter your name:')

name = input()

print(name)
```

On line three, the script tells the user what he needs to enter, and on line 5 the input function receives the input from the user and stores it in the *name* variable. The input function accepts input until the *Enter* key is pressed. Once the *Enter* is pressed, the execution continues.

The *input* function interprets the provided input as strings. Even if the user typed numbers they will still be accepted as a string. Thus, if you want to perform some mathematical calculations with the data provided from the user you must convert the string into an integer. To do the conversion, the *int* function is used. The *int* function converts the provided data into an integer. Let's assume that you want to write a script that calculates the area of a square. To do that, you need to ask the user to enter two numbers, the height, and the width of the square. The following code provides an example of such script.

```
#!/usr/bin/env python3


print('Enter the height of the square:')

height_str = input()
height_int = int(height_str)

print('Enter the width of the square:')

width_str = input()
width_int = int(width_str)

area = height_int * width_int

print(area)
```

As you can see on lines 6 and 11, the int function is used. The int function can only convert numbers into integers, thus if the provided data from user contains any character other than 0-9, it will result in an error. The above code takes 15 lines (including the shebang and the empty lines). Some of the operations in the code above can be simplified and shortened. The first this that can be shortened is lines 5-6 and 10-11. The input and int function can be used directly one on another.

```
#!/usr/bin/env python3


print('Enter the height of the square:')

height = int(input())

print('Enter the width of the square:')

width = int(input())

area = height * width

print(area)
```

Function can be placed one into another, in that case the input function is placed *into* the *int* function. The input function also can print text to the user, it can be done by putting the text into the brackets of the *input* function.

```
#!/usr/bin/env python3


height = int(input('Enter the height of the square: '))

width = int(input('Enter the width of the square: '))

area = height * width

print(area)
```

If a text is put into the *input* function it will be printed before the user's input. The only difference from the *print* function (aside from the main purpose) is that the *input* function does not uses new line (\n) at the end of the string. The code can be shortened even more, but that is not the goal now. Also remember that if the code is shorted it does **NOT** mean that it will be executed faster, in some cases it might even make it slower.

Type Conversion

In addition to the *int* function there are other functions that are used to change the type of the data. The need to convert one type into another happens a lot, thus it is important to know which functions can be used for that.

Type Function

The first function related to data types is the *type* function. The *type* function tells which data type is stored in a variable.

```
var_1 = 777
var_2 = 'Python'
```

```
var_3 = 4 / 3
var_4 = True

print(type(var_1))
print(type(var_2))
print(type(var_3))
print(type(var_4))

OUTPUT:

<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
```

In the output you can see the types of each of the variables.

Int Function

The *int* function takes data as an input and returns an integer. The function can convert strings, floats and Booleans into an integer.

If the given data is a string, then int function will return the same number as an integer. But the string must not contain any characters aside from numbers. Decimal point is also not allowed.

```
num_str = '954'
num_int = int(num_str)
```

The int function can also convert floats to integers. When the function receives a float, it returns the same number without the decimal point. The function does not round the numbers, thus if the number was 2.9, the function will return 2.

```
num_1 = 9.5
num_2 = 4.8

print(int(num_1))
print(int(num_2))

OUTPUT:

9
4
```

Another conversion that can be done with the *int* function is conversion from Boolean to integer. If the value of the Boolean is True, then the result will be 1. With False it will be 0.

```
bool_1 = True
bool_2 = False
```

```
print(int(bool_1))
print(int(bool_2))

OUTPUT:

1
0
```

Float Function

The *float* function converts given data into a float. The function can take strings, integers and Booleans and convert them.

The most common use case of the function is to convert a string into a float. Unlike the *int* function, float can accept a decimal point in the string.

```
var_1 = '5.56'

print(float(var_1))
print(type(float(var_1)))

OUTPUT:

5.56
<class 'float'>
```

Part III: Conditional Statements

Conditional statements are used to control the flow of the script and make it adjustable for various situation. Conditions are checked using the *if* statement. After the *if* statement comes the condition itself. A typical *if* statement looks like the following.

```
num = 13

if num > 10:
  print('The number is greater than 10')
```

As you can see, after the *if* statement comes the condition. If the condition is true, the code in the body of the *if* statement will be executed. Note that the line under the if statement is moved to the right. That movement is called indentation. By using indentation, we tell the interpreter if the line belongs to a body of other statement, in this case the *if* statement. In the code above, the content of the variable *num* will be checked, if the value that is stored in *num* is greater than 10, the print function will be executed. Otherwise, the interpreted will ignore that line an continue the execution.

There can be more than one line under the *if* statement, the only thing that tells if a line belongs to that statement, is indentation. For example, in the following code you can see multiple lines under the *if* statement.

```
num = 13

if num > 10:
  print('The number is greater than 10')

  num -= 4

  result = num * 2
```

Comparison Operators

There are several operators that can be used with the *if* statement.

Operator	Meaning
==	Equals
!=	Not equals
>	Greater than
<	Less than
>=	Greater than or equals
<=	Less than or equals

The operators can be applied to integers, floats, strings, Booleans, etc. Also, it is possible to perform comparison between integers and floats.

```
price = 499.99

if price > 300:
  discount = True
```

As you can see in the code, it is possible to compare an integer and a float. Although, if you will type to do a comparison between an integer/float and a string you will get an error.

```
price = '499.99' # Note it is a string

if price > 300:
  discount = True

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-23-d2984154b0cb> in <module>()
      4 price = '499.99' # Note it is a string
      5
----> 6 if price > 300:
      7   discount = True

TypeError: '>' not supported between instances of 'str' and 'int'
```

In addition to the operators listed above, there is also an operator that is used to check if something is in something else. For example, a character in a string, or a string in another string.

```
# Example I
ip = '192.168.0.1'

if '192.168' in ip:
  print('Looks like a private IP')

# Example II:
msg = 'This is a Python script'

if 'Python' in msg:
  print('Python is awesome')

# Example III
name = 'John'

if 'o' in name:
  print('You have an "o" in your name')
```

In the above examples, there are various checks for substrings in strings. It is important to remember that the check is case sensitive. So, in the second example (lines 11 – 14), if the word 'Python' was spelled with lowercase p, the check would fail.

Conditions and Booleans

Each time a condition is checked a Boolean value is return. So, to run a code that is written under an *if* statement, the condition should be *True*. If you open Python's interpreter and type a condition, you will see that it returns either *True* or *False*, depending on the condition.

```
>>> 1 > -3
True
>>> 7 == 9
False
>>> 'i' in 'Hello'
False
>>> 4.11 >= 3.12
True
```

Each time a condition is checked, either True or False are returned. It the condition is True, the code in that belongs to the statement will be executed, otherwise it will be skipped.

The fact that the if statement works only if it gets True, also allowed to perform conditional checks on Booleans.

```
price = 500
discount = True

if discount == True:
  price = price * 0.8

print('The final price is:', price)
```

In line 7, we check if the value of *discount* is True, and if it is, we reduce the price. But, as stated above, the if statement gets either True or False and then executes the code. So, the same piece of code can be written a little simpler.

```
price = 500
discount = True

if discount:
  price = price * 0.8

print('The finale price is:', price)
```

The value of *discount* is already True. So, we can skip the unnecessary comparison and directly operate on the value of *discount*.

If, elif, and else

If is not the only conditional statement, there are also elif (else if) and else. As you have seen before, if the result of the condition is False, then the code in the body of the statement will not be executed. In many cases, when a condition is used, we want the code to flow in one of two (or more) given directions. For that we can use the else and elif statements. In the following example, a check is done on the age variable, and one of two things can happen.

```
age = 19

if age >= 21:
  print('You are old enough')
else:
  print('You are too young')
```

Either the first *print* or the second. It is impossible for both to happen, and it is also impossible that none of them will happen. In a situation where you have an and *if else* statement, one of them will be executed. So, if the *if* statement is False, the code under the *else* statement will be executed.

In many cases, there is a need for more than an *if – else* statement. In situations like those, we use the *elif* statement. The *elif* is placed between the *if* and the *else* statements. There is no limit on how many *elif* statements can be used. For example, say you are writing a script that presents a simple menu to the user and acts accordingly. A script like that can look like the following.

```
print('Option 1')
print('Option 2')
print('Option 3')

user_selection = int(input('Enter your selection: '))

if user_selection == 1:
  print('You selected option 1')
elif user_selection == 2:
  print('You selected option 2')
elif user_selection == 3:
  print('You selected option 3')
else:
  print('Wrong option', user_selection)
```

In the code, there are 4 possible option that can happen. Either one of the three options selected by the user, or the last options that prints an error message. It is important to remember that only one of the options can be triggered. In a situation where you have a chain of *if*, *elif*, *else* statements, only **one** of them can be executed. Once one of them is executed the rest will be skipped. Thus, it is a good practice to place the most common condition at the top of the chain if it possible to reduce the execution time of the script/program. Although it is not always possible to predict which condition will be the most common. Also remember that if you have a chin of *if* and *elif* statements, it is not mandatary to add an *else* statement at the end. It all depends on the situation.

And, Or, Not

When using the *if* statement, there is an option the create more complicated conditions. To do that we use the *and*, *or* and *not* operators. Using those, we can create more complex and granular conditions.
There are various situations in which you need to check two conditions simultaneously. To do that you can either nested if statements or the *and* operator. The following code shows an example of nested *if* statements.

```
price = 899
discount = True

if price > 700:
  if discount:
    price *= 0.7

print('Your price is:', price)
```

As you can see, it is possible to place one if statement into another, but you must remember that indentation is very important. As you can see on line 9, there are two tabs before that line of code. The indentations tell the interpreter that the line of code belongs to the second *if* statement.

The same code can be written using one if statement thus making it shorted and easier to read.

```
price = 899
discount = True

if price > 700 and discount:
  price *= 0.7

print('Your final price is:', price)
```

On line 7 in the *if* statement, there is an *and* operator. When *and* is used it checks that both conditions return True. If one of them returns False, then the final condition will also return False.

In addition to the *and* operator, there is also the *or* operator which returns True if one of the conditions returns True. Thus, if you check two conditions simultaneously, it is enough that one of them will return True. Consider the previous code with a simple change.

```
price = 899
discount = False

if price > 700 or discount:
  price *= 0.7

print('Your price is:', price)
```

In this piece of code, it is enough that one of the conditions is True. In the presented code, the price is above 700 which makes the first condition True. The second one checks if the *discount* variable is True, which is not. But it is still enough for the whole statement to be True thanks to the *or* operator.

The last operator is the *not* operator. The not operator reverses the value of the return statement of the condition. Thus, if the condition returned True, and the *not* operator is used, it will be False. Consider the following code as an example.

```
price = 599
discount = False

if not discount:
  print('You do not deserve a discount')
```

The value of discount is False, thus the condition is False as well. But, because the *not* operator is used the final result is True. Remember that the not operator can be combined with any other condition that you use, as it just opposites the return of the condition.

Part IV: Strings

Strings have been already mentioned before in the book, but there is a large set of operations that can be done with and on strings. Strings are just sequences of characters. For example, you have a string that looks like the following.

```
str_1 = 'Python'
```

Indexing and Slicing

The string *str_1* contains 6 characters, each one of them has its own index (address). Thus, if you need to refer to specific character from a string you can do it by its index. For every string, the first index will always be 0, and the last index is the length of the string minus 1. For str_1 the indexes look like following.

Character	P	y	t	h	o	n
Index	0	1	2	3	4	5

Using the index, we can reference specific character from a string. It is done by using the square brackets.

```
str_1 = 'This is a string'

print(str_1[0])
print(str_1[1])
print(str_1[2])
print(str_1[3])
print(str_1[4])
print(str_1[5])

OUTPUT:

T
h
i
s

i
```

The act of calling a specific character or a set of characters from a string is called string slicing. Using slicing we can get more than one character at a time. It is done by adding a colon in to square brackets. In the code above, if you want to get the first word from the sentence, you can use the following structure.

```
str_1 = 'This is a string'

print(str_1[0:4])

OUTPUT:

This
```

By using the colon, you can specify the start and the end of the string that you want to get. Notice that the second number represents the up to index. If the first number that you want to write is 0 you can omit it and just leave it blank.

```
str_1 = 'This is a string'

print(str_1[:4])

OUTPUT:

This
```

When performing slicing, you do not have to start from the beginning of the string. For example, if you want to get the second word from *str_1* you can do the following.

```
str_1 = 'This is a string'

print(str_1[5:8])

OUTPUT:

is
```

Same as it is with the first number in the square brackets, if the second number is not specified it will grab the whole string until the end.

In addition to the start and end of the slice, you can also specify the step. If you want to get every second character from a string you can use the following structure.

```
str_1 = 'This is a string'

print(str_1[::2])

OUTPUT:

Ti sasrn
```

So far, you have that it is possible to reference characters from a string by using the indexes of those characters. In addition to the “regular” indexing, you can reference characters by their order from the end. The last character in a string is the -1 character, the one before it is the -2 character and so on.

Character	P	y	t	h	o	n
Index	-6	-5	-4	-3	-2	-1

String Methods

Methods are actions that are applied on objects. A string is also a type of object and using methods we can modify or perform various checks on strings. For example, if you have a string that you want to convert into uppercase you can user the upper() method.

```
#!/usr/bin/env python3

str_1 = 'python'

print(str_1.upper())

OUTPUT:

PYTHON
```

In addition to upper, there are many additional methods that can be used to manipulate the string. A full list of all the available methods can be found here. Some of the methods like upper, lower or capitalize are used to alter a string, others are used to perform various checks. For example, say you have a string that stores an IP address, and you need to check if that IP starts with a specific octet. To do that you can use the startswith() method.

```
#!/usr/bin/env python3

address = '192.168.0.13'

if address.startswith('192.168'):
  print('It\'s a match')
```

Using string methods, you can also replace characters in a string. The replace() method takes to strings as arguments and replaces the first string with the seconds one.

```
#!/usr/bin/env python3

address = '192.168.0.13'

address = address.replace('13', '177')

print(address)

OUTPUT:

192.168.0.177
```

Another very useful method is strip(). This method strips characters from the start and end of a string. It is very handy in certain situations.

```
#!/usr/bin/env python3

name = '  John Doe      '

print(name.strip())

OUTPUT:

John Doe
```

Notice that the string *name* contains spaces at the beginning and the end of the string. Once the strip method is used, it removes the leading and trailing spaces. By default, *strip* removes spaces, tabs and new lines. If there is a need to remove a specific character from the beginning/end, it can be passed as an argument.

```
#!/usr/bin/env python3

msg = 'This is a sentence.'

print(msg.strip('.'))

OUTPUT:

This is a sentence
```

Part V: Lists

Lists are used to store multiple elements in one place. We know that one variable can hold one piece of information at a time. Lists on other hand store an array of elements in one place. For example, if you want to store multiple numbers together you can place them into a list. To declare a list we give it a name (same as for a variable) and put the data that we want to store inside square brackets separated by commas.

```
numbers = [5, 99, 138, 11, 99, 0]
```

To reference a specific element from a list we use the index of the element (same as with strings). Thus, to get the first element from a list, use the index 0.

```
print(numbers[0])

OUTPUT:

5
```

The same rules of slicing that were shown previously are also applied here. So if you want to get all the elements except for the last one from a list, you can do the following.

```
print(numbers[0:-1])

OUTPUT:

[5, 99, 138, 11, 99]
```

Notice the difference in the output between the two prints. The first time, we just got the number The second time we also got the data we wanted, but it was printed as a list.

Lists can contain multiple data type at the same time. They are not limited to just integers. If you need a list that contains multiple types at once, you can just put them together into a list.

```
lst = [1, 99.9, 'Python', True, False]
```

Typically, strings will contain only one type of data in them. Although again it is not mandatory. Additionally, lists are not limited in their size. The only limit is the amount of available memory.

List Methods

Once a string is declared it can be modified during the runtime of the program. You can add and remove elements from a list using various methods. If you need to append (add) a new element into a list, you can use the *append* method. To remove an element from a list, the *remove* method is used. There is number of methods which are applied for strings, they can be found here. Below are few examples of list methods.

```
names = ['John', 'James', 'Bob']

names.append('alice')

print(names)

OUTPUT:

['John', 'James', 'Bob', 'alice']
```

```
names = ['John', 'James', 'Bob']

names.remove('John')

print(names)

OUTPTU:

['James', 'Bob']
```

One of the methods that exists for lists is *copy*. The *copy* method creates a copy of a list. The reason to use it and not just the equals sign is shown in the following example.

```
l1 = [1, 2, 3]
l2 = l1

l1.remove(1)

print(l1)
print(l2)

OUTPTU:

[2, 3]
[2, 3]
```

As you can see, we removed an element from the first list, but the second one also got affected. To prevent it, use the *copy* method as shown below.

```
l1 = [1, 2, 3]
l2 = l1.copy()
```

```
l1.remove(1)

print(l1)
print(l2)

OUTPTU:

[2, 3]
[1, 2, 3]
```

Now, each list is a separate entity which means that if the first list is modified, the second one is not affected by it. This behavior does not occur with regular variables, but it is important to remember it when working with strings.

Lists will be used further away, especially combined with loops.

Part VI: Loops

Many times, there is a need to perform some operation repeatedly when writing code. To accomplish that we use loops. The idea of a loop is to take a piece of code and execute it X number of times. There are two types of loops, the *while* loop and the *for* loop.

Each time we a loop in our code, it will perform some action, or set of action for a given number of times. Each execution of the loop is called **iteration**. Iteration is a very common word that is used in the field of programing.

While Loop

The while loop checks a condition and if the condition is then the code inside the loop will be executed. It is also important to remember, that indentation is very important when dealing with loops as well. To tell the interpreter which lines belongs to the loop we use indentation.

```
num = 1

while num < 6:
  print(num)

  num += 1

OUTPTU:

1
2
3
4
5
```

At the declaration of the loop there is a condition. As long as the value of num is lower than 10, the code in the loop will be executed. It is very important to remember that when dealing with while loops, you must have something that will stop the loop. If there is mechanism that stops the loop, it will keep running forever.

In some cases, the desired action is to run the loop forever, and then there is no need for stopping condition. To create a loop that never stops, you can create a *while True* loop.

```
while True:
  print('This loop runs forever')
```

There is no limit on how many times the loop can run, or how many lines of code are inside the body of the loop. In addition to that, you can place conditions inside loops and vice versa. Let's take a simple example of a code that prints all the numbers that can be divided by three from 1 to 30. A code that does that can look like the following.

```
number = 1

while number <= 30:
  if number % 3 == 0:
    print(number)

  number += 1

OUTPTU:

3
6
9
12
15
18
21
24
27
30
```

As you can see, you can “nest” statements into each other. But you must pay attention to the indentations. Loops can also be nested into each other, a situation like that is called nested loop.

For Loop

The second type of loops that exist in python are *for* loops. The difference between the two loops it that the *while* loop can run for an unknow number of iterations, and the *for* loop always has a predefined number of iterations. Thus, it is impossible to create a *for* loop that will run forever as the *while True* loop. The reason that for loops cannot run forever is that for loops work in combination with lists, and we already know that list have a finite number of elements.

The following code shows a *for* loop that iterates over the elements in a list.

```
cars = ['Toyota', 'Subaru', 'Suzuki']

for car in cars:
  print(car)

OUTPUT:

Toyota
Subaru
Suzuki
```

The declaration of the *for* loop is done by using the word *for* followed by a new variable name. In that case we have a list named *cars*. The loop takes one element of the list in each iteration and places it in the variable that we named *car*. In the body of the loop, we simply print the name of the variable. The name of

the variable that we declare inside the for loop does not have to be similar to the name of the list. In many cases you will see that the name if the declared variable is just *i* which stands for *iteration*. But you are free to name the variable as you like.

```
cars = ['Toyota', 'Subaru', 'Suzuki']

for i in cars:
  print(i)

OUTPUT:

Toyota
Subaru
Suzuki
```

The for loop does not change the values of the items in a list, it just iterates over each one of them.
The *for* loop can also iterate over a range of numbers, to do that, the *range* function is used.

```
for i in range(5):
  print(i)

OUTPUT:

0
1
2
3
4
```

The *range* function takes numbers as arguments and generates a sequence of numbers. If you specify only one number, the sequence will start from 0 up to the given number. Alternatively, you can specify the start and the end of the range.

```
for i in range(1, 6):
  print(i)

OUTPUT:

1
2
3
4
5
```

You can also specify the step of the range by providing a third number.

```
for i in range(1, 11, 2):
  print(i)

OUTPUT:

1
3
5
7
9
```