

Python Beginner Course

David Rieger

Lecture 01: Introduction, Setup and Basics

1 Introduction

In this course you will learn how you set up your Python Environment and get started on Python anywhere. You will learn how python files are structured, what modules are and how to use them, some basic knowledge about logic and code structure and the most common commands and functions. In the scope of this course, you also get the knowledge to make simple scripts that run in your command line. The goal is to have a solid foundation and understanding of python, so you can use your knowledge to tackle more complex problems. Also the knowledge provided in this course should suffice to learn more on your own, if you have problems in solving any given programming task.

2 Setup

To use python, you need the python environment, which you can download here, and a suitable editor. I personally recommend VSCode (Download Link: [VSCode](#)). But you can use other editors, like Atom or Notepad++.

In this course we will set up our python environment with the VSCode editor. You should install python with the *Add python.exe to PATH* box checked. Just download the installer and execute it. Follow the instructions of the installer. After the installation you need to open the computer settings. Click on Apps, then on the left hand side on Apps and Features and click on Alaises for App-Execution. Then turn App-Installer (python.exe) and App-Installer (python3.exe) off. To check if everything worked, you can press your windows key and search for *Python*. There should be an entry called *Python 3.xx* where x is the version number of python. Now execute it and you have a window where you can write python instructions. Unfortunately we can't save our code in there, so we need to have an editor where we can write and save our python code. This saved code can then be run with the python environment you just installed. If you downloaded the VSCode installer, then just execute it and accept the license agreement. You get a window where you can choose some options for your VSCode installation. I recommend the options shown in figure 1.

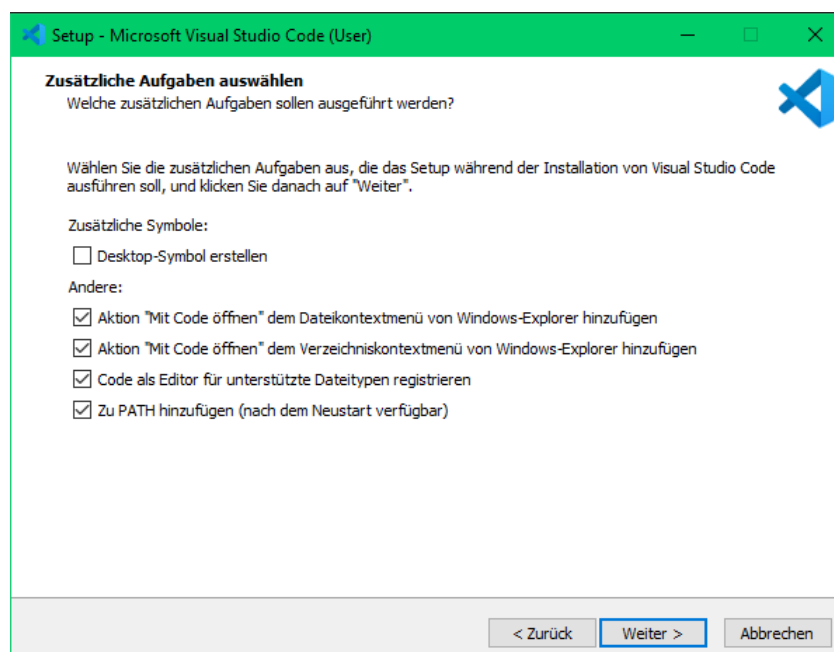


Figure 1: VSCode recommended installation preferences

After you have chosen the settings for the installation, follow through with the installation manager. If the installation finished, you can open VSCode. For autocompletion and syntax highlighting, we then download the Python extension. Click on the extension tab in VSCode and type *Python* in the searchbar. The first

extension that you see should be the Python extension by Microsoft (refer to fig. 2). Simply install that and reopen VSCode.

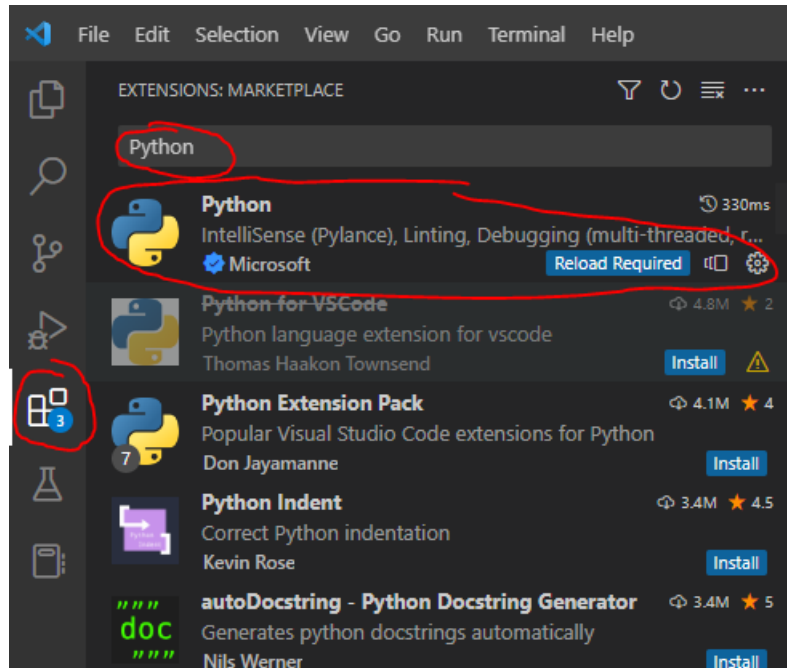


Figure 2: Python extension for VSCode

Now we navigate to the Explorer tab, which is the one in the upper left corner of the screen, and create a workspace. For that, you just click on File and then on open folder. Create a new folder for this course in a location of your choice. There you can create other folders or files. If you close and reopen VSCode, then VSCode will open the last workspace, that you worked on. To run python files you need to tell VSCode which interpreter it should use. To specify the interpreter, press `Ctrl+Shift+P`. At the top of the window a selection will appear. There only should be one entry called *Python 3.xx* where *x* is the version. Simply click on this entry and VSCode is successfully set up for python.

3 Python Basics

Python is an interpreted language, so unlike C++ where you have a compiler, that translates your code directly to machine code which then gets executed on your machine, python code is first interpreted by the python interpreter and then compiled to machine code. That means that the interpreter guesses, what your code should do and declares variable types and statements according to the interpretation of your written code. That means, that we don't need to declare specific datatypes for variables. Also the interpreter can convert variables from one datatype to another. In C++ for example, you need to declare every variable with the datatype it should have, and you can't change that datatype in the ongoing execution of your code. In figure 3 we can see an example of these datatype assignments to variables.

```
5 a = 5          # variable a is now an integer
6 print(type(a))
7 a = 1.5        # variable a is now a float
8 print(type(a))
9 a = 'String'   # variable a is now a string
10 print(type(a))
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

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

```
1 int main() {
2     int a = 1;    // variable a is declared as an integer
3     float a = 1.0; // illegal
4     char a = 'x'; // illegal
5 }
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

* Executing task: C/C++: gcc.exe Aktive Datei kompilieren

Kompilierung wird gestartet...

```
C:\msys64\ucrt64\bin\gcc.exe -fdiagnostics-color=always -g "C:\Users\david\OneDrive\Desktop\Python Course\datatype.cpp: In function 'int main()':
C:\Users\david\OneDrive\Desktop\Python Course\datatype.cpp:3:11: error: redeclaration of 'float a' with different type
3 |     float a = 1.0;
  |           ^
C:\Users\david\OneDrive\Desktop\Python Course\datatype.cpp:2:9: note: previous declaration of 'a' with type 'int'
2 |     int a = 1;
  |     ^
C:\Users\david\OneDrive\Desktop\Python Course\datatype.cpp:4:10: error: redeclaration of 'char a' with different type
4 |     char a = 'x';
  |          ^
C:\Users\david\OneDrive\Desktop\Python Course\datatype.cpp:2:9: note: previous declaration of 'a' with type 'int'
2 |     int a = 1;
  |     ^
```

Figure 3: (top) python, (bottom) c++

You can see that the python code runs fine, but the C++ code throws errors, because you can't change the specified datatype of a variable in C++. In python the interpreter handles these datatype conversions. Also function returns are interpreted, so there is no need to specify the return datatype.

3.1 Create first python file

To create a python file in VSCode, simply right-click in your explorer window and create a file with the ending `.py`. A good start would be a *Hello World!* program. simply type `print('Hello World!')` or `print("Hello World!")`

```
5 print('Hello World!')
6 print("Hello World!")
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

Hello World!
Hello World!

Figure 4: Python Hello World! program

Notice, that it doesn't matter in what quotes we encapsulate our string. Single quotes and double quotes do the exact same thing. This is useful, if you want to emphasize a string.

```
5 print("Hello World!")
6 print('Hello World!')
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

"Hello World!"
'Hello World!'

Figure 5: Use of single quotes and double quotes in strings

Inputs and print instructions

To give an output to an user, the use of *print()* instructions is common. Print instructions can output a string, the value of a variable or function returns. Note that every print instruction in python outputs in a new line. You can also get an user input with the *input()* instruction. We can define a prompt string for the input instruction, to give the user an idea what to input. A few examples for print and input instructions are in figure 6.

```
5 print('Line 1')
6 user_input = input("Write something: ")
7 print(user_input)
8 print(len(user_input)) # prints the length of the input string
9 print("characters are in your input!")
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

Line 1
Write something: Line 2
Line 2
6
characters are in your input!

Figure 6: print() and input() instructions

3.2 Formatted Strings

Sometimes it is useful to have the values of variables embedded in strings. Therefore formatted strings can be used. Python has some built-in functions, that help with embedding variable values in output or input strings. There are a few different ways to do this. In figure 7 we can see some examples.

```

5 value1 = 7
6 value2 = 3
7 print(f'This is the number {value1} and this {value2}')
8 print('This is the number {value1} and this {value2}'.format(value1=value1, value2=value2))
9 print('This is the number %x and this %x' % (value1, value2))

```

PROBLEMS 2 OUTPUT DEBUG CONSOLE **TERMINAL** JUPYTER

```

This is the number 7 and this 3
This is the number 7 and this 3
This is the number 7 and this 3

```

Figure 7: Three different ways to implement formatted strings

Obviously the first implementation is the shortest and therefore the preferred one. This function is called *f-Strings* the second implementation is called *str.format()* and the third one is called *%-formatting*. Either one can be used and functions the exact same.

3.3 Datatypes and Datastructures

Even though python doesn't care which datatype gets written in a variable, python uses several datatypes and datastructures to store values and do arithmetic with. In table 1 is a specification of those types.

Datatype	Category	Description
str	Text Type	String. A single character or a combination of characters in every given length. Has to be surrounded by quotes
int	Numeric type	An integer
float	Numeric type	Decimal number. We use a decimal point . not a comma ,
complex	Numeric type	A complex number
list	Sequence type	A list of any datatype, datatypes can be mixed. Lists have to be surrounded by [] and are unordered and changeable
tuple	Sequence type	A tuple of any datatype, datatypes can be mixed. tuples have to be surrounded by () and are unordered and unchangeable
range	Sequence type	A list of only integers, that are ordered and have a fixed step-size
dict	Mapping type	A dictionary is composed of a key and corresponding values. Dictionaries have to be surrounded by {} and are ordered and changeable
set	Set type	A set is a collection, that has to be surrounded by {} and is unordered, unchangeable and unindexed, but you can add or delete items
frozenset	Set type	A frozenset is a set, that is immutable, which means you cannot add or delete items
bool	Boolean type	A datatype which can be True or False
bytes	Binary type	Bytes are a hexadecimal representation of numbers (0..255) which are immutable
bytearray	Binary type	A bytearray is similar to bytes, but is NOT immutable
NoneType	None type	None indicates no value

Table 1: Python datatypes

3.4 Statements and Loops

If-Else-Statements

The If-Statement is used to check several conditions (refer to table 2) and execute the code of the statement, if the condition is true. If the condition is false, the statement is skipped and the program continues after the statement. The elif-Statement is a addition to the if-statement and only gets checked if the if-condition was false. The else-Statement gets executed if non of the if- or elif-conditions turned out true. Check figure 8.

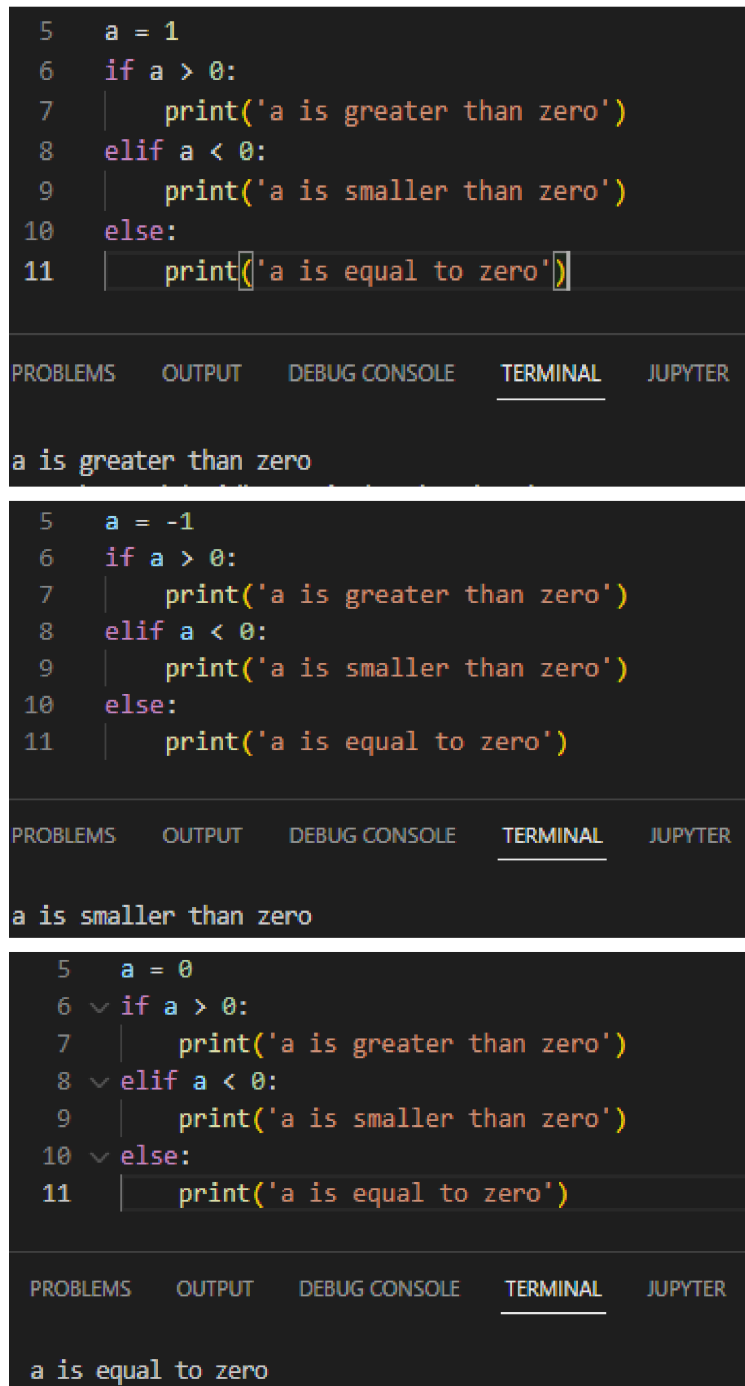


Figure 8: Python if-else-statements

Condition	Meaning	Description
<code>a == b</code>	Equals	a has the same value as b
<code>a != b</code>	Unequals	a has not the same value as b
<code>a < b</code>	Less than	a is smaller than b
<code>a > b</code>	Greater than	a is greater than b
<code>a <= b</code>	Less or equal to	a is smaller or equal to b
<code>a >= b</code>	Greater or equal to	a is greater or equal to b
<code>a in b</code>	Is in	a is part of b
<code>a not in b</code>	is not in	a is not part of b

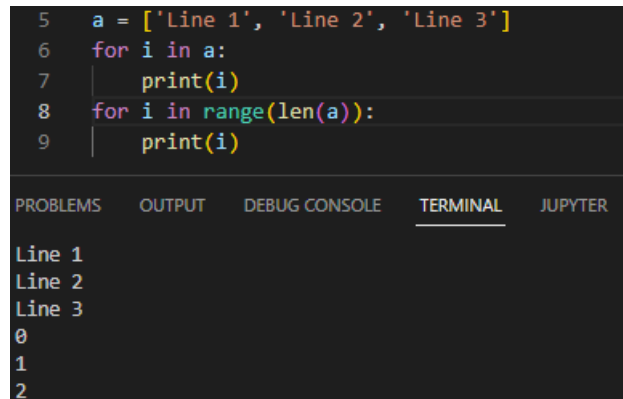
Table 2: Logical conditions for statements

You can have more than one condition in an if-statement. You can take two conditions and combine them

with a logical AND or a logical OR the keywords are *and* and respectively *or*.

3.5 For- and While-Loops

Loops can be useful to write continuous code or to iterate through a list. There are two types of loops in python. The for-loop is a conditional loop with a fixed loop count, which means that the loop continues until a specific condition is met, which is the end of the iterable values. Different to other languages, python can iterate through values. That is helpful if you want to get values in a sequence type. Check figure 9 for examples. Unlike other programming languages that encapsulate code blocks in curly brackets, python uses the indentation system, which means that code blocks belonging to a statement or function have to be indented by a tab.



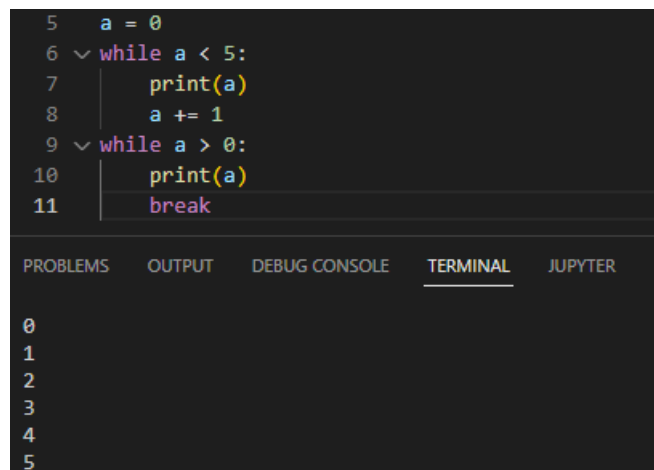
```
5 a = ['Line 1', 'Line 2', 'Line 3']
6 for i in a:
7     print(i)
8 for i in range(len(a)):
9     print(i)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

Line 1
Line 2
Line 3
0
1
2

Figure 9: Python for-loops

Note that the first for-loop iterates through the values and the second iterates through the indexes of the list. You can also iterate through strings in the same way. While-loops are conditional loops with an unfixed loop count. That means, that while-loops continue until the condition isn't met anymore, not caring about the loop count. For- and while-loops can be broken out of with the keyword *break*. If *break* gets executed, python jumps out of the loop, even if the loop condition is true. In figure 10 are examples for while-loops and the *break* statement.



```
5 a = 0
6 while a < 5:
7     print(a)
8     a += 1
9 while a > 0:
10    print(a)
11    break
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

0
1
2
3
4
5
0

Figure 10: While-loops and break statement

The second while-loop should continue infinitely, because the condition of the loop is always met. With the *break* instruction we can break out of the loop, so the output is only one print instruction.

3.6 Functions and declaration of those

If identical code is occurring several times in the program, functions are useful. Functions can be called with or without an argument, which means that you can give the function variables or values that it can work with. These functions then execute code and can return a return-value. Functions don't have to return a value. They can just execute code and don't give a feedback. Functions are defined with the keyword *def*

followed by the function name with normal () brackets appended to the name. In these brackets we can define arguments, that the function can work with. Also these arguments, as well as the return datatype can be defined. The instruction ends like statements with a colon. After that the function code can be written. Functions can be called with the function name and the argument in the brackets. Examples in figure 11

```
5 def thisIsAFunction():
6     print('String from function')
7 def functionWithArgument(a):
8     print(a)
9 def functionDefined(a: int) -> int:
10     return a+10
11 thisIsAFunction()
12 functionWithArgument(1)
13 print(functionDefined(2))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
String from function
1
12
```

Figure 11: Examples for function declaration and function calling

The return of a function is given with the keyword *return*.

3.7 Modules

Modules in python are what libraries are to C++. Modules have pre-written functions and variables, that you can import to your python project. To use modules, you need to have them in your python environment. Python comes with a selection of the most useful modules like math, os, or time. If you need specific modules for your project, you can download them with pip (Package installer for Python). A very useful module is the numpy module which you can use to do matrix arithmetics. To download numpy type *pip install numpy* in your VSCode terminal. Pip takes care of the download and the installation. If everything was successful, numpy can be imported and used in your project. In order to import a module, you can import the whole module or you can import certain functions you need. Also you can rename the module name, to have better readable code. Check figure 12 for examples and usage of module functions.

```
3 import numpy as np
4 from math import factorial
5 import os
6
7 os.system("cls" if os.name == "nt" else "clear")
8 print(factorial(4))
9 a = np.ndarray(shape=(1,1))
10 print(type(a))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
24
<class 'numpy.ndarray'>
```

Figure 12: Examples for different module import options

If we import the whole module, then we need to give the module function the module prefix like in figure 12 line 7. If we import a module as a name, we can alter the prefix to a name we choose. If we import a specific function from a module, we don't need to give the prefix and can call the function directly (refer to fig. 12 line 8).

3.8 Useful instructions and functions

In table 3 are the most useful and common instructions listed.

Statement	Description
print()	Prints a variable value or string in the command line
len()	Finds the length of a string or sequence type
type()	Returns the datatype of a variable
range()	Returns an ordered integer list from start value to end value with a specified step-size
.join()	Appends a string to another string
.split()	Splits a string at the specified characters
round()	Rounds a decimal number to specified digits
input()	Returns a value that is inputted in the command line
.capitalize()	Makes all characters in a string capital
.to_lowercase()	Makes all characters in a string lowercase
find()	Finds the index of the first occurrence of a specified character in a string
copy()	Returns a copy of a list as a new sequence type
insert()	Inserts a new value into a list at the specified index
pop()	Removes an element in a list at the specified position and returns the value of the removed element
reverse()	Reverses a sequence type
sort()	Sorts a list in specified order, default order is ascending
append()	Appends values to a list

Table 3: Useful and common python instructions

Operands are used to do arithmetics, assignments and bit-manipulation. These operands are categorized in arithmetic operators, assignment operators, comparison operators, logical operators, identity operators, membership operators and bitwise operators. Identity operators, membership operators and comparison operators were already discussed in this lecture (refer to tab. 2). In table 4 the remaining operators are covered.

Operator	Category	Description
+	Arithmetic operator	Adds numbers together
−	Arithmetic operator	Subtracts numbers
*	Arithmetic operator	Multiplies numbers together
/	Arithmetic operator	Divides numbers
%	Arithmetic operator	Modulus division, returns the remainder of an integer division
**	Arithmetic operator	Exponentiation
//	Arithmetic operator	Floor division. Returns an integer rounded down
=	Assignment operator	Assigns a value to a variable
+=	Assignment operator	Incrementation
− =	Assignment operator	Decrementation
* =	Assignment operator	Factor multiplication
/ =	Assignment operator	Factor division
% =	Assignment operator	Assigns remainder of division
// =	Assignment operator	Assigns the return of a floor division
** =	Assignment operator	Assigns the return of an exponentiation
& =	Assignment operator	Assigns the return of a bitwise AND operation
=	Assignment operator	Assigns the return of a bitwise OR operation
^ =	Assignment operator	Assigns the return of a bitwise XOR operation
>>=	Assignment operator	Assigns the return of a bit shift operation
<<=	Assignment operator	Assigns the return of a bit shift operation
and	Logical operator	Returns True if both statements are true
or	Logical operator	Returns True if one of the statements is true
not	Logical operator	Reverses the result, returns True if statement is False and the other way around
&	Bitwise operator	AND. Sets bit to 1 if both bits are 1
	Bitwise operator	OR. Sets bit to 1 if one or both of the bits is 1
^	Bitwise operator	XOR. Sets bit to 1 if only one bit is 1
~	Bitwise operator	NOT. Inverts all bits
<<	Bitwise operator	Zero fill left shift. Shifts bits to the left by inserting zeros on the right
>>	Bitwise operator	Signed right shift. Shifts bits to the right by inserting a copy of the left most bit to the left

Table 4: Operators in python