PYTHON BIBLE

7 IN 1

FLORIAN DEDOV

THE
PYTHON BIBLE

7 IN 1

BY

FLORIAN DEDOV

This book is a 7-in-1 version of all the seven volumes of The Python Bible series.

I recommend reading them in the right order, since the volumes build on top of each other.

If you think that this book has brought value to you and helped you on your programming journey, I would appreciate a quick review on Amazon.

Thank you!

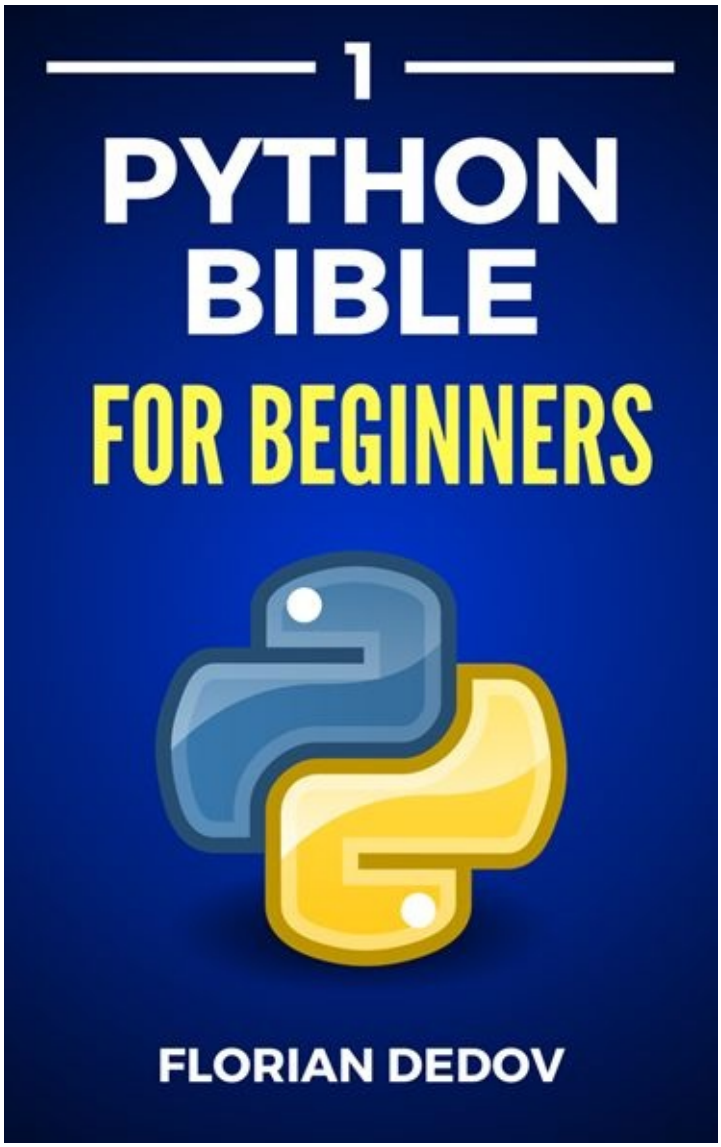

1
PYTHON
BIBLE
FOR BEGINNERS
FLORIAN DEDOV

TABLE OF CONTENT

INTRODUCTION

This book is the first part of a series that is called the Python Bible. In this series, we are going to focus on learning the Python programming language as effective as possible. The goal is to learn it smart and fast without needing to read thousands of pages. We will keep it simple and precise.

In this first book, we will introduce you to the language and learn the basics. It is for complete beginners and no programming, IT or math skills are needed to understand it. At the end, you will be able to write some simple and interesting programs and you will have the necessary basis to continue with volume two.

WHY PYTHON?

One question you might be asking yourself right now is: Why learn Python? Why not Java, C++ or Go?

First of all, a good programmer is fluent in multiple languages, so learning Python doesn't mean that you can't learn C++ or Java additionally. But second of all, Python is probably the best language to start with.

It is extremely simple in its syntax (the way the code is written) and very easy to learn. A lot of things that you need to do manually in other languages are automated in Python.

Besides that, Python's popularity is skyrocketing. According to the TIOBE-Index, Python is the third most popular language with an upward trend. But also in other rankings, you will see Python near the top.

TIOBE: https://www.tiobe.com/tiobe-index/

Also, Python is the lingua franca of machine learning. This means that it is the one language that is used the most in the areas of artificial intelligence, data science, finance etc. All these topics will be part of this Python Bible series. But since Python is a general-purpose language, the fields of application are numerous.

Last but not least, Python is a very good choice because of its community. Whenever you have some problem or some error in your code, you can go online and find a solution. The community is huge and everything was probably already solved by someone else.

HOW TO READ THIS BOOK

In order to get as much value as possible from this book, it is very important that you code the individual examples yourself. When you read and understand a new concept, play around with it. Write some scripts and experiment around. That’s how you learn and grow.

So, stay motivated. Get excited and enjoy your programing journey. If you follow the steps in this book, you will have a solid basis and a decent understanding of the Python language. I wish you a lot of fun and success with your code!

Just one little thing before we start. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

1 – INSTALLING PYTHON

Now, before we get into the code, we need to install Python and our development environment. Python is a so-called *interpreted* language. This means that the code needs a certain software (the interpreter) to be executed.

Other languages like C++ or Java are *compiled* languages. You need a compiler but then the program is converted into machine code and can be executed without extra software. Python scripts can’t be executed without a Python interpreter.

PYTHON INSTALLATION

First of all, you need to visit the official Python website in order to get the latest version from there.

Python: https://www.python.org/downloads/

Download the installer and follow the instructions. Once you are done, you should have the Python interpreter as well as the IDLE on your computer.

The IDLE is an *Integrated Development and Learning Environment*. It is the basic editor, where we can write and execute our code. Just open your start menu and look for it.

DEVELOPMENT ENVIRONMENT

PYTHON IDLE

When it comes to our development environment, we have many options to choose from. The simplest choice is to just use the default IDLE. It is a great tool for writing the code and it has an integrated interpreter. So, you can execute the code directly in the IDLE. For beginners this is definitely enough. If you choose this option, you can just stick with the basic installation. In this book, we are going to assume that you are using the IDLE.

EDITOR AND CLI

If you prefer to use a specific editor, like Atom, Sublime or VS Code, you can run your code directly from the command line. So you basically write your code in your editor and save the file. Then you run CMD (on Windows) or Terminal (on Linux & Mac). You need to use the following syntax in order to run the code:

python <scriptname>.py

This option is a bit less convenient but if you prefer using a specific editor, you may need to do it. Another way would be to look for some Python interpreter plugins for your editor.

Atom Editor: https://atom.io/

Sublime Text: https://www.sublimetext.com/

VS Code: https://code.visualstudio.com/

PYCHARM

Last but not least, you can also decide to use a very professional IDE with a lot of features. For Python this is PyCharm. This development environment is a product of JetBrains, a very well-known and professional company. It has a ton of features, professional syntax highlighting and a great user interface. I would definitely recommend it to every Python developer, but I think it might be a bit too much and not necessary for beginners. But that is your decision. If you are interested, you can get the community edition for free.

PyCharm: https://www.jetbrains.com/pycharm/

Now, let’s get into the code!

2 – OUR FIRST PROGRAM

In order to understand the syntax of Python, we are going to start with a very simple first program. It is a tradition in programming to start with a *Hello World* application, when you are learning a new language. So, we are going to do that in this chapter.

HELLO WORLD

A *Hello World* application is just a script that outputs the text *“Hello World!”* onto the screen. In Python this is especially simple.

```
print("Hello World!")
```

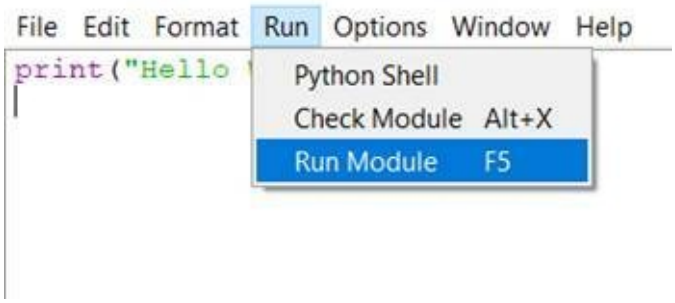
As you can see, this is a one-liner in Python. In other languages, we would have to define a basic structure with functions, classes and more, just to print one text.

But let’s see what’s happening here. The first thing that we can notice is the so-called *function* with the name *print*. When we use that function, it outputs a certain text onto the screen. The text that we want to print needs to be put between the parentheses.

Another thing that is very important here, are the quotation marks. They indicate that the text is a *string* and not a name of something else. A string is a data-type that represents text. When we don’t use quotation marks, the interpreter will think that *Hello World!* is a variable name and not a text. Therefore, we will get an error message. But we will talk about variables and data types in the next chapter.

RUNNING THE SCRIPT

Now, we just need to run the script we just wrote. For that, you need to save the script into a Python file. Then you can use the integrated interpreter of the IDLE. Just click on Run -> Run Module (or F5).



Running Code in Python IDLE

You will then see the results on the screen. That’s how you run your first program.

3 – VARIABLES AND DATA TYPES

Probably, you have already encountered variables in your math classes. Basically, they are just placeholders for values. In programming, that’s the same. The difference is that we have a lot of different data types, and variables cannot only store values of numbers but even of whole objects.

In this chapter we are going to take a look at variables in Python and the differences of the individual data types. Also, we will talk about type conversions.

NUMERICAL DATA TYPES

The types you probably already know from mathematics are numerical data types. There are different kinds of numbers that can be used for mathematical operations.

NUMERICAL DATA TYPES		
DATA TYPE	**KEYWORD**	**DESCRIPTION**
Integer	int	A whole number
Float	float	A floating point number
Complex	complex	A complex number

As you can see, it’s quite simple. An integer is just a regular whole number, which we can do basic calculations with. A float extends the integer and allows decimal places because it is a floating point number. And a complex number is what just a number that has a *real* and an *imaginary* component. If you don’t understand complex numbers mathematically, forget about them. You don’t need them for your programming right now.

STRINGS

A string is just a basic sequence of characters or basically a text. Our text that we printed in the last chapter was a string. Strings always need to be surrounded by quotation marks. Otherwise the interpreter will not realize that they are meant to be treated like text. The keyword for String in Python is *str*.

BOOLEANS

Booleans are probably the most simple data type in Python. They can only have one of two values, namely *True* or *False*. It's a binary data type. We will use it a lot when we get to conditions and loops. The keyword here is *bool*.

SEQUENCES

Sequences are a topic that we will cover in a later chapter. But since sequences are also data types we will at least mention that they exist.

SEQUENCE TYPES		
DATA TYPE	**KEYWORD**	**DESCRIPTION**
List	list	Collection of values
Tuple	tuple	Immutable list
Dictionary	dict	List of key-value pairs

CREATING VARIABLES

Creating variables in Python is very simple. We just choose a name and assign a value.

```
myNumber = 10

myText = "Hello"
```

Here, we defined two variables. The first one is an integer and the second one a string. You can basically choose whatever name you want but there are some limitations. For example you are not allowed to use reserved keywords like *int* or *dict*. Also, the name is not allowed to start with a number or a special character other than the underline.

USING VARIABLES

Now that we have defined our variables, we can start to use them. For example, we could print the values.

```
print(myNumber)

print(myText)
```

Since we are not using quotation marks, the text in the parentheses is treated like a variable name. Therefore, the interpreter prints out the values *10* and *“Hello”*.

TYPECASTING

Sometimes, we will get a value in a data type that we can't work with properly. For example we might get a string as an input but that string contains a number as its value. In this case *"10"* is not the same as *10*. We can't do calculations with a string, even if the text represents a number. For that reason we need to typecast.

```
value = "10"

number = int(value)
```

Typecasting is done by using the specific data type function. In this case we are converting a string to an integer by using the *int* keyword. You can also reverse this by using the *str* keyword. This is a very important thing and we will need it quite often.

4 – OPERATORS

The next thing we are going to learn is operators. We use operators in order to manage variables or values and perform operations on them. There are many different types of operators and in this chapter we are going to talk about the differences and applications.

ARITHMETIC OPERATORS

The simplest operators are arithmetic operators. You probably already know them from mathematics.

ARITHMETIC OPERATORS		
OPERATOR	**NAME**	**DESCRIPTION**
+	Addition	Adds two values
-	Subtraction	Subtracts one value from another
*	Multiplication	Multiplies two values
/	Division	Divides one value by another
%	Modulus	Returns the remainder of a division
**	Exponent	Takes a value to the power of another value
//	Floor Division	Returns the result of a division without decimal places

Let’s take a look at some examples.

```
20 + 10 = 30        20 - 10 = 10
2 * 10 = 20         5 / 2 = 2.5
5 % 2 = 1           5 ** 2 = 25
5 // 2 = 2
```

If you don’t get it right away, don’t worry. Just play around with the operators and print the results. Of course you can also use variables and not only pure

values.

ASSIGNMENT OPERATORS

Another type of operators we already know is assignment operators. As the name already tells us, we use them to assign values to variables.

ASSIGNMENT OPERATORS	
OPERATOR	**DESCRIPTION**
=	Assigns a value to a variable
+=	Adds a value to a variable
-=	Subtracts a value from a variable
*=	Multiplies a value with a variable
/=	Divides the variable by a value
%=	Assigns the remainder of a division
**=	Assigns the result of a exponentiation
//=	Assigns the result of a floor division

Basically we use these operators to directly assign a value. The two statements down below have the same effect. It’s just a simpler way to write it.

```
a = a + 10
```

```
a += 10
```

COMPARISON OPERATORS

When we use comparison operators in order to compare two objects, we always get a Boolean. So our result is binary, either True or False.

COMPARISON OPERATORS		
OPERATOR	**NAME**	**DESCRIPTION**
==	Equal	Two values are the same
!=	Not Equal	Two values are not the same
>	Greater Than	One value is greater than the other
<	Less Than	One value is less than the other
>=	Greater or Equal	One value is greater than or equal to another
<=	Less or Equal	One value is less than or equal to another

We use comparisons, when we are dealing with conditions and loops. These are two topics that we will cover in later chapters.

When a comparison is right, it returns True, otherwise it returns False. Let's look at some examples.

10 == 10 → **True**

20 > 10 → **True**

20 >= 20 → **True**

10 <= 5 → **False**

10 != 10 → **False**

20 > 20 → **False**

20 < 10 → **False**

LOGICAL OPERATORS

Logical operators are used to combine or connect Booleans or comparisons.

LOGICAL OPERATORS	
OPERATOR	**DESCRIPTION**
or	At least one has to be *True*
and	Both have to be *True*
not	Negates the input

I think this is best explained by examples, so let's look at some.

True or **True** → **True**

True or **False** → **True**

False or **False** → **False**

True and **False** → **False**

True and **True** → **True**

False and **False** → **False**

not **True** → **False**

not **False** → **True**

OTHER OPERATORS

There are also other operators like bitwise or membership operators. But some of them we just don’t need and others need a bit more programming knowledge to be understood. So for this chapter we will stick with those.

5 – USER INPUT

Up until now, the only thing we did is to print out text onto the screen. But what we can also do is to input our own data into the script. In this chapter, we are going to take a look at user input and how to handle it.

INPUT FUNCTION

In Python we have the function *input*, which allows us to get the user input from the console application.

```
name = input("Please enter your name:")

print(name)
```

Here, the user can input his name and it gets saved into the variable *name*. We can then call this variable and print it.

```
number1 = input("Enter first number: ")

number2 = input("Enter second number: ")

sum = number1 + number2

print("Result: ", sum)
```

This example is a bit misleading. It seems like we are taking two numbers as an input and printing the sum. The problem is that the function *input* always returns a string. So when you enter 10, the value of the variable is *"10"*, it's a string.

So, what happens when we add two strings? We just append one to the other. This means that the sum of "15" and "26" would be "1526". If we want a mathematical addition, we need to typecast our variables first.

```
number1 = input("Enter first number: ")

number2 = input("Enter second number: ")

number1 = int(number1)

number2 = int(number2)

sum = number1 + number2

print("Result: ", sum)
```

Now our script works well! Always remember that the input function returns a string and you need to typecast it, if you want to do calculations with it.

6 – CONDITIONS

This chapter is about a concept that will make our scripts more interesting. So far, the interpreter always executed one command after the other. With conditions, this changes.

IF, ELIF, ELSE

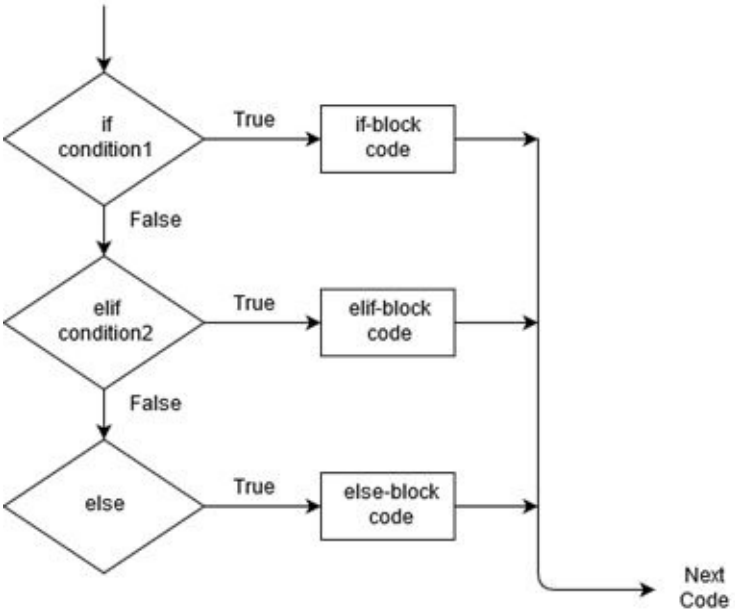
Basically, a condition needs to return *True*, so that our script continues with the code in its block.

```
number = input("Enter a number:")

number = int(number)


if number < 10:

    print("Your number is less than 10")

elif number > 10:

    print("Your number is greater than 10")

else:

    print("Your number is 10")
```

The three important keywords here are *if*, *elif* and *else*. In this script, the user inputs a number that gets converted into an integer. Then our first *if-statement* checks if this number is less than ten. Remember that comparisons always return *True* or *False*. If the return is *True*, the code that is indented here gets executed. We use colons and indentations to mark code blocks in Python.

If this condition returns *False*, it continues to the *elif-block* and checks if this condition is met. The same procedure happens here. You can have as many *elif-blocks* as you want. If no condition is met, we get into the *else-block*.

FLOWCHART



In this flowchart you can see how these basic if, elif and else trees work. Of course, you don’t need an elif or else block. You can just write an if-statement and if the condition is not met, it skips the code and continues with the rest of the script.

NESTED IF-STATEMENTS

You can also put if-blocks into if-blocks. These are called *nested* if-statements.

```
if number % 2 == 0:
    if number == 0:
        print("Your number is even but zero")
    else:
        print("Your number is even")
else:
    print("Your number is odd")
```

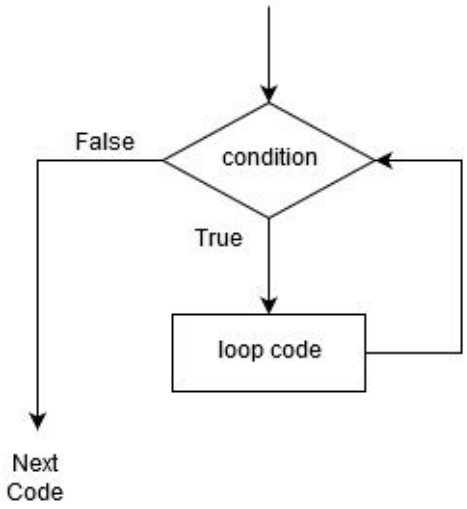
So, here we have the first condition, which checks if the number is even. When it’s even it then checks if it’s a zero or not. That’s a trivial example but you get the concept.

7 – LOOPS

If we want to automate a repetitive process, we can use loops to do that. A loop is a programming structure that executes the same code over and over again, as long as a certain condition is met. This is at least true for the classic *while loop.*

WHILE LOOP

There are two types of loops in Python: *while loops* and *for loops*. A while loop executes the code in its block *while* a condition is met.



As you can see, it goes in circles until the condition returns *False*. Let's have a look at the code.

```
number = 0

while number < 10:

    number += 1

    print(number)
```

We use the *while* keyword to define the loop. Then we state a condition and again the code block is indented after the colon. In this example, we are counting from one to ten. We are initializing the variable *number* with the value zero. In every iteration, we increase it by one and print its value. This is done as long as the number is less than ten.

ENDLESS LOOP

With this knowledge, we can create an endless loop. This might seem useless but in fact it has some applications.

```
while True:
    print("This will print forever")
```

It is done by defining a loop which has the condition *True*. Since it is always *True*, the loop will never end, unless we terminate the script.

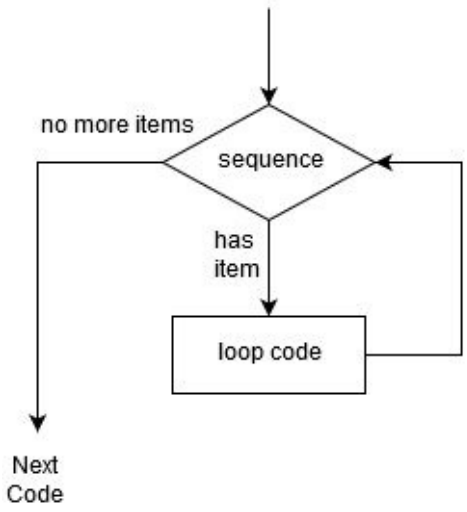
Warning: This might overload your computer, especially if it is a slow one.

FOR LOOP

The *for loop* works a bit differently. Here we don't have a condition. This loop type is used to iterate over sequences. Since these are the topic of the next chapter, we won't get into too much detail here.

```
numbers = [10, 20, 30, 40]

for number in numbers:

    print(number)
```

For now, we won't care about the syntax of sequences. Just notice that we have a list of four numbers. We then use the *for* keyword to iterate over it. The control variable *number* always gets assigned the value of the next element. In this case, we print out all the numbers.



As you can see, the loop continues as long as there is a next element in the list.

RANGE FUNCTION

With the *range* function, we can create lists that contain all numbers in between two numbers.

```
for x in range(100):
```

```
    print(x)
```

This right here is a simple way to print all the numbers up to 100. But what you can also do is start counting from another number.

```
for x in range (20, 80):

    print(x)
```

Our range list here contains the numbers between 20 and 80.

LOOP CONTROL STATEMENTS

In order to manage loops, we have so-called *loop control statements*. They allow us to manipulate the process flow of the loop at a specific point.

BREAK STATEMENT

With the *break* statement, we can end a loop immediately, without caring about the condition.

```
number = 0

while number < 10:

    number += 1

    if number == 5:

        break

    print(number)
```

Here, we have a simple counting loop. As soon as the number reaches the value five, we execute a break statement and the script continues with the rest of the code.

CONTINUE STATEMENT

If we don’t want to break the full loop, but to skip only one iteration, we can use the *continue* statement.

```
number = 0

while number < 10:

    number += 1

    if number == 5:

        continue
```

```
    print(number)
```

In this example, we always increase the number by one and then print it. But if the number is a five, we skip the iteration, after the increment. So, this number doesn't get printed.

PASS STATEMENT

The *pass* statement is a very special statement, since it does absolutely nothing. Actually, it is not really a loop control statement, but a placeholder for code.

```
if number == 10:
    pass
else:
    pass


while number < 10:
    pass
```

Sometimes you want to write your basic code structure, without implementing the logic yet. In this case, we can use the pass statement, in order to fill the code blocks. Otherwise, we can't run the script.

8 – SEQUENCES

The sequence is the most basic data structure in Python. It contains multiple elements and they are indexed with a specific number. In this chapter, we are going to talk about the different types of sequences and their functions.

LISTS

The first sequence type we are looking at is the list. It is what the name says – just a list.

```
numbers = [10, 22, 6, 1, 29]
```

In Python, we define lists by using square brackets. We put the elements in between of those and separate them by commas. The elements of a list can have any data type and we can also mix them.

```
numbers = [10, 22, 6, 1, 29]

names = ["John", "Alex", "Bob"]

mixed = ["Anna", 20, 28.12, True]
```

ACCESSING VALUES

In order to access values of a sequence, we need to first talk about indices. The index is more or less the position of the element. What's important here is that we start counting from zero. So the first element has the index zero, the second has the index one and so on. We can then access the element by using the index.

```
print(numbers[2])

print(mixed[1])

print(names[0])
```

We print the third element of *numbers* (6), the second element of *names* (Alex) and the first element of *mixed* (Anna).

But instead of only accessing one single element, we can also define a range that we want to access.

```
print(numbers[1:3]) # 22 and 6
```

```
print(numbers[:3])  # 10, 22 and 6

print(numbers[1:])  # 22, 6, 1 and 29
```

By using the colon, we can slice our lists and access multiple elements at once.

MODIFYING ELEMENTS

In a list, we can also modify the values. For this, we index the elements in the same way.

```
numbers[1] = 10

names[2] = "Jack"
```

The second element of the numbers list is now *10* instead of *22* and the third element of the names list is *Jack* instead of *Bob*.

LIST OPERATIONS

Some of the operators we already know can be used when working with lists – addition and multiplication.

LIST OPERATIONS	
OPERATION	**RESULT**
[10, 20, 30] + [40, 50, 60]	[10, 20, 30, 40, 50, 60]
[10, “Bob”] * 3	[10, “Bob”, 10, “Bob”, 10, “Bob”]

LIST FUNCTIONS

When it comes to lists, there are a lot of different functions and methods that we can use. We are not going to talk about all of them, since it’s just not necessary. Our focus will lie on the most important ones.

LIST FUNCTIONS	
FUNCTION	**DESCRIPTION**
len(list)	Returns the length of a list
max(list)	Returns the item with maximum value
min(list)	Returns the item with

	minimum value
list(element)	Typecasts element into list

LIST METHODS	
METHOD	**DESCRIPTION**
list.append(x)	Appends element to the list
list.count(x)	Counts how many times an element appears in the list
list.index(x)	Returns the first index at which the given element occurs
list.pop()	Removes and returns last element
list.reverse()	Reverses the order of the elements
list.sort()	Sorts the elements of a list

TUPLES

The next sequence type we are going to look at is very similar to the list. It's the *tuple*. The only difference between a list and a tuple is that a tuple is immutable. We can't manipulate it.

```
tpl = (10, 20, 30)
```

Notice that a tuple is defined by using parentheses rather than square brackets.

TUPLE FUNCTIONS

Basically, all the reading and accessing functions like *len*, *min* and *max* stay the same and can be used with tuples. But of course it is not possible to use any modifying or appending functions.

DICTIONARIES

The last sequence type in this chapter will be the *dictionary*. A dictionary works a bit like a lexicon. One element in this data structure points to another. We are talking about key-value pairs. Every entry in this sequence has a key and a respective value. In other programming languages this structure is called *hash map*.

```
dct = {"Name": "John",
       "Age": 25,
       "Height": 6.1}
```

We define dictionaries by using curly brackets and the key-value pairs are separated by commas. The key and the value themselves are separated by colons. On the left side there is the key and on the right side the according value.

Since the key now replaces the index, it has to be unique. This is not the case for the values. We can have many keys with the same value but when we address a certain key, it has to be the only one with that particular name. Also keys can’t be changed.

ACCESSING VALUES

In order to access values of a dictionary, we need to address the keys.

```
print(dct["Name"])
print(dct["Age"])
print(dct["Height"])
```

Notice that if there were multiple keys with the same name, we couldn’t get a result because we wouldn’t know which value we are talking about.

DICTIONARY FUNCTIONS

Similar to lists, dictionaries also have a lot of functions and methods. But since

they work a bit differently and they don’t have indices, their functions are not the same.

DICTIONARY FUNCTIONS	
FUNCTION	**DESCRIPTION**
len(dict)	Returns the length of a dictionary
str(dict)	Returns the dictionary displayed as a string

DICTIONARY METHODS	
METHOD	**DESCRIPTION**
dict.clear()	Removes all elements from a dictionary
dict.copy()	Returns a copy of the dictionary
dict.fromkeys()	Returns a new dictionary with the same keys but empty values
dict.get(key)	Returns the value of the given key
dict.has_key(key)	Returns if the dictionary has a certain key or not
dict.items()	Returns all the items in a list of tuples
dict.keys()	Returns a list of all the keys
dict.update(dict2)	Add the content of another dictionary to an existing one
dict.values()	Returns a list of all the values

MEMBERSHIP OPERATORS

One type of operators we haven't talked about yet is membership operators. These are very important when it comes to sequences. We use them to check if an element is a member of a sequence, but also to iterate over sequences.

```
list1 = [10, 20, 30, 40, 50]

print(20 in list1)     # True

print(60 in list1)     # False

print(60 not in list1) # True
```

With the *in* or *not in* operators, we check if a sequence contains a certain element. If the element is in the list, it returns *True*. Otherwise it returns *False*.

But we also use membership operators, when we iterate over sequences with for loops.

```
for x in list1:

    print(x)
```

For every element *in* the sequence *x* becomes the value of the next element and gets printed. We already talked about that in the loops chapter.

9 – FUNCTIONS

Oftentimes in programming, we implement code that we want to use over and over again at different places. That code might become quite large. Instead of re-writing it everywhere we need it, we can use *functions*.

Functions can be seen as blocks of organized code that we reuse at different places in our scripts. They make our code more modular and increase the reusability.

DEFINING FUNCTIONS

In order to define a function in Python, we use the *def* keyword, followed by a function name and parentheses. The code needs to be indented after the colon.

```
def hello():
  print("Hello")
```

Here we have a function *hello* that prints the text *"Hello"*. It's quite simple. Now we can call the function by using its name.

```
hello()
```

PARAMETERS

If we want to make our functions more dynamic, we can define parameters. These parameters can then be processed in the function code.

```
def print_sum(number1, number2):
  print(number1 + number2)
```

As you can see, we have two parameters in between the parentheses – *number1* and *number2*. The function *print_sum* now prints the sum of these two values.

```
print_sum(20, 30)
```

This function call prints the value *50* out onto the screen.

RETURN VALUES

The two functions we wrote were just executing statements. What we can also do is return a certain value. This value can then be saved in a variable or it can be processed. For this, use the keyword *return*.

```
def add(number1, number2):
  return number1 + number2
```

Here we return the sum of the two parameters instead of printing it. But we can then use this result in our code.

```
number3 = add(10, 20)

print(add(10, 20))
```

DEFAULT PARAMETERS

Sometimes we want our parameters to have default values in case we don’t specify anything else. We can do that by assigning values in the function definition.

```
def say(text="Default Text"):

    print(text)
```

In this case, our function *say* prints the text that we pass as a parameter. But if we don’t pass anything, it prints the default text.

VARIABLE PARAMETERS

Sometimes we want our functions to have a variable amount of parameters. For that, we use the *asterisk symbol* (*) in our parameters. We then treat the parameter as a sequence.

```
def print_sum(*numbers):

    result = 0

    for x in numbers:

        result += x

    print(result)
```

Here we pass the parameter *numbers*. That may be five, ten or a hundred numbers. We then iterate over this parameter, add every value to our sum and print it.

```
print_sum(10, 20, 30, 40)
```

SCOPES

The last thing we are going to talk about in this chapter is *scopes*. Scopes are not only important for functions but also for loops, conditions and other structures. Basically, we need to realize the difference between local and global variables.

```
def function():

    number = 10

    print(number)


print(number) # Doesn't work
```

In this example, you see why it's important. When you define a variable inside of a function, a loop, a condition or anything similar, this variable can't be accessed outside of that structure. It doesn't exist.

```
number = 10


def function():

    print(number)
```

This on the other hand works. The variable *number* was defined outside of the function, so it can be *seen* inside the function. But you will notice that you can't manipulate it.

In order to manipulate an object that was defined outside of the function, we need to define it as *global*.

```
number = 10
```

```
def function():
    global number
    number += 10
    print(number)
```

By using the keyword *global* we can fully access and manipulate the variable.

10 – EXCEPTION HANDLING

Programming is full of errors and exceptions. If you coded along while reading and experimented around a little bit, you may have encountered one or two error messages. These errors can also be called *exceptions*. They terminate our script and crash the program if they are not handled properly.

```
result = 10 / 0
```

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    10 / 0
ZeroDivisionError: division by zero
```

Just try to divide a number by zero and you will get a *ZeroDivisionError*. That's because a division by zero is not defined and our script doesn't know how to handle it. So it crashes.

```
text = "Hello"

number = int(text)
```

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    number = int(text)
ValueError: invalid literal for int() with base 10: 'Hello'
```

Alternatively, try to typecast an ordinary text into a number. You will get a *ValueError* and the script crashes again.

TRY EXCEPT

We can handle these errors or exceptions by defining *try* and *except* blocks.

```
try:
    print(10 / 0)
    text = "Hello"
    number = int(text)
except ValueError:
    print("Code for ValueError...")
except ZeroDivisionError:
    print("Code vor ZDE...")
except:
    print("Code for other exceptions...")
```

In the *try* block we put the code that we want to execute and where errors might occur. Then we define *except* blocks that tell our script what to do in case of the respective errors. Instead of crashing, we provide code that handles the situation. This might be a simple error message or a complex algorithm.

Here we defined two specific *except* blocks for the *ValueError* and the *ZeroDivisionError*. But we also defined a general *except* block in case we get an error that doesn't fit these two types.

ELSE STATEMENTS

We can also use else statements for code that gets executed if nothing went wrong.

```
try:
```

```
    print(10 / 0)
except:
    print("Error!")
else:
    print("Everything OK!")
```

FINALLY STATEMENTS

If we have some code that shall be executed at the end no matter what happened, we can write it into a *finally* block. This code will always be executed, even if an exception remains unhandled.

```
try:
    print(10 / 0)
except:
    print("Error!")
finally:
    print("Always executed!")
```

11 – FILE OPERATIONS

Oftentimes, we will need to read data in from external files or to save data into files. In this chapter we will take a look at *file streams* and the various operations.

OPENING AND CLOSING FILES

Before we can read from or write into a file, we first need to open a *file stream*. This returns the respective file as an object and allows us to deal with it.

```
file = open("myfile.txt", "r")
```

We use the function *open* in order to open a new file stream. As a parameter we need to define the file name and the *access mode* (we will talk about that in a second). The function returns the stream and we can save it into our variable *file*.

ACCESS MODES

Whenever we open a file in Python, we use a certain access mode. An access mode is the way in which we access a file. For example *reading* or *writing*. The following table gives you a quick overview over the various access modes.

ACCESS MODE	
LETTER	**ACCESS MODE**
r	Reading
r+	Reading and Writing (No Truncating File)
rb	Reading Binary File
rb+	Reading and Writing Binary File (No Truncating File)
w	Writing
w+	Reading and Writing (Truncating File)
wb	Writing Binary File
wb+	Reading and Writing Binary File (Truncating File)
a	Appending
a+	Reading and Appending
ab	Appending Binary File
ab+	Reading and Appending Binary File

The difference between r+ or rb+ and w+ or wb+ is that w+ and wb+ overwrite existing files and create new ones if they don’t exist. This is not the case for r+ and rb+.

CLOSING FILES

When we are no longer in need of our opened file stream, we should always close it. Python does this automatically in some cases but it is considered good practice to close streams manually. We close a stream by using the method *close*.

```
file = open("myfile.txt", "r+")

# CODE

file.close()
```

WITH STATEMENT

Alternatively, we can open and close streams more effectively by using *with* statements. A *with* statement opens a stream, executes the indented code and closes the stream afterwards.

```
with open("myfile.txt", "r") as file:

    # Some Code
```

It shortens the code and makes it easier to not forget to close your streams.

READING FROM FILES

Once we have opened a file in a reading mode, we can start reading its content. For this, we use the *read* method.

```
file = open('myfile.txt', 'r')

print(file.read())

file.close()
```

Here we open the file in reading mode and then print out its content. We can do the same thing by using the *with* statement.

```
with open('myfile.txt', 'r') as file:

    print(file.read())
```

But we don’t have to read the whole file, if we don’t want to. We can also read the first 20 or 50 characters by passing a parameter to the method.

```
with open('myfile.txt', 'r') as file:

    print(file.read(50))
```

WRITING INTO FILES

When we write into a file, we need to ask ourselves if we just want to add our text or if we want to completely overwrite a file. So we need to choose between writing and appending mode. For writing in general we use the method *write*.

```
file = open('myfile.txt', 'w')

print(file.write("Hello File!"))

file.flush()

file.close()
```

We open our file in writing mode and write our little text into the file. Notice that the text doesn't get written until we *flush* the stream. In this case this is not necessary because when we close a stream it flushes automatically. Let's look at the *with* statement alternative again.

```
with open('myfile.txt', 'w') as file:

    print(file.write("Hello File!"))
```

If we want to append our text, we just have to change the access mode. Everything else stays the same.

```
with open('myfile.txt', 'a') as file:

    print(file.write("Hello File!"))
```

OTHER OPERATIONS

Now if we want to perform other operations than writing, reading and appending, we will need to import and extra module. The basic Python functions and classes are available by default. But many things like mathematics, networking, threading and also additional file operations, require the import of modules. In this case we need to import the *os* module, which stands for operating system.

```
import os
```

This would be one way to import this module. But if we do it like that, we would always need to specify the module when we use a function. To make it easier for us, we will do it like that.

```
from os import *
```

Basically, what we are saying here is: Import all the function and classes from the module *os*. Notice that the import statements of a script should always be the first thing at the top.

DELETING AND RENAMING

For deleting and renaming files we have two very simple functions from the *os* module – *remove* and *rename*.

```
remove("myfile.txt")
```

```
rename("myfile.txt", "newfile.txt")
```

We can also use the *rename* function, to move files into different directories. But the directory has to already be there. This function can't create new directories.

```
rename("myfile.txt", "newdir/myfile.txt")
```

DIRECTORY OPERATIONS

With the *os* module we can also operate with directories. We can create, delete

and navigate through them.

```
mkdir("newdir")

chdir("newdir")

chdir("..")

rmdir("newdir")
```

Here we create a new directory by using the *mkdir* (make directory) function. We then go into that directory with the *chdir (change directory)* function and then back to the previous directory with the same function. Last but not least we remove the directory with the *rmdir (remove directory)* function.

By using *(“..”)* we navigate back one directory. Additionally, if we would want to specify a whole path like “C:\Users\Python\Desktop\file.txt”, we would have to use double backslashes since Python uses single backslashes for different purposes. But we will talk about this in the next chapter in more detail.

12 – STRING FUNCTIONS

Even though strings are just texts or sequences of characters, we can apply a lot of functions and operations on them. Since this is a book for beginners, we won't get too much into the details here, but it is important for you to know how to deal with strings properly.

STRINGS AS SEQUENCES

As I already said, strings are sequences of characters and they can also be treated like that. We can basically index and slice the individual characters.

```
text = "Hello World!"

print(text[:5])

print(text[6:11])
```

The first slice we print is *“Hello”* and the second one is *“World”*. Another thing we can do is to iterate over strings with for loops.

```
text = "Hello World!"

for x in text:

    print(x)
```

In this example, we print the individual characters one after the other.

ESCAPE CHARACTERS

In strings we can use a lot of different *escape characters*. These are non-printable characters like *tab* or *new line*. They are all initiated by a backslash, which is the reason why we need to use double backslashes for file paths (see last chapter).

The following table summarizes the most important of these escape characters. If you are interested in all the other ones just use google but you won't need them for now.

ESCAPE CHARATCERS	
NOTATION	**DESCRIPTION**
\b	Backspace
\n	New Line
\s	Space
\t	Tab

STRING FORMATTING

When we have a text which shall include the values of variables, we can use the % operator and placeholders, in order to insert our values.

```
name, age = "John", 25

print("%s is my name!" % name)

print("I am %d years old!" % age)
```

Notice that we used different placeholders for different data types. We use *%s* for strings and *%d* for integers. The following table shows you which placeholders are needed for which data types.

PLACEHOLDERS	
PLACEHOLDER	**DATA TYPE**
%c	Character
%s	String
%d or %i	Integer
%f	Float
%e	Exponential Notation

If you want to do it more general without specifying data types, you can use the *format* function.

```
name, age = "John", 25

print("My name is {} and I am {} years old"

      .format(name, age))
```

Here we use curly brackets as placeholders and insert the values afterwards using the *format* function.

STRING FUNCTIONS

There are a ton of string functions in Python and it would be unnecessary and time wasting to talk about all of them in this book. If you want an overview just go online and look for them. One website, where you can find them is W3Schools.

W3Schools Python String Functions:
https://www.w3schools.com/python/python_ref_string.asp

In this chapter however, we will focus on the most essential, most interesting and most important of these functions. The ones that you might need in the near future.

CASE MANIPULATING FUNCTIONS

We have five different case manipulating string functions in Python. Let's have a look at them.

CASE MANIPULATING FUNCTIONS	
FUNCTION	**DESCRIPTION**
string.lower()	Converts all letters to lowercase
string.upper()	Converts all letters to uppercase
string.title()	Converts all letters to titlecase
string.capitalize()	Converts first letter to uppercase
string.swapcase()	Swaps the case of all letters

COUNT FUNCTION

If you want to count how many times a specific string occurs in another string, you can use the *count* function.

```
text = "I like you and you like me!"

print(text.count("you"))
```

In this case, the number two will get printed, since the string *"you"* occurs two

times.

FIND FUNCTION

In order to find the first occurrence of a certain string in another string, we use the *find* function.

```
text = "I like you and you like me!"

print(text.find("you"))
```

Here the result is *7* because the first occurrence of *"you"* is at the index *7*.

JOIN FUNCTION

With the *join* function we can join a sequence to a string and separate each element by this particular string.

```
names = ["Mike", "John", "Anna"]

sep = "-"

print(sep.join(names))
```

The result looks like this: Mike-John-Anna

REPLACE FUNCTION

The *replace* function replaces one string within a text by another one. In the following example, we replace the name *John* by the name *Anna*.

```
text = "I like John and John is my friend!"

text = text.replace("John", "Anna")
```

SPLIT FUNCTION

If we want to split specific parts of a string and put them into a list, we use the *split* function.

```
names = "John,Max,Bob,Anna"

name_list = names.split(",")
```

Here we have a string of names separated by commas. We then use the *split* function and define the comma as the separator in order to save the individual names into a list.

TRIPLE QUOTES

The last topic of this chapter is *triple quotes*. They are just a way to write multi-line strings without the need of escape characters.

```
print('''Hello World!

This is a multi-line comment!


And we don't need to use escape characters

in order to write new empty lines!''')
```

WHAT'S NEXT?

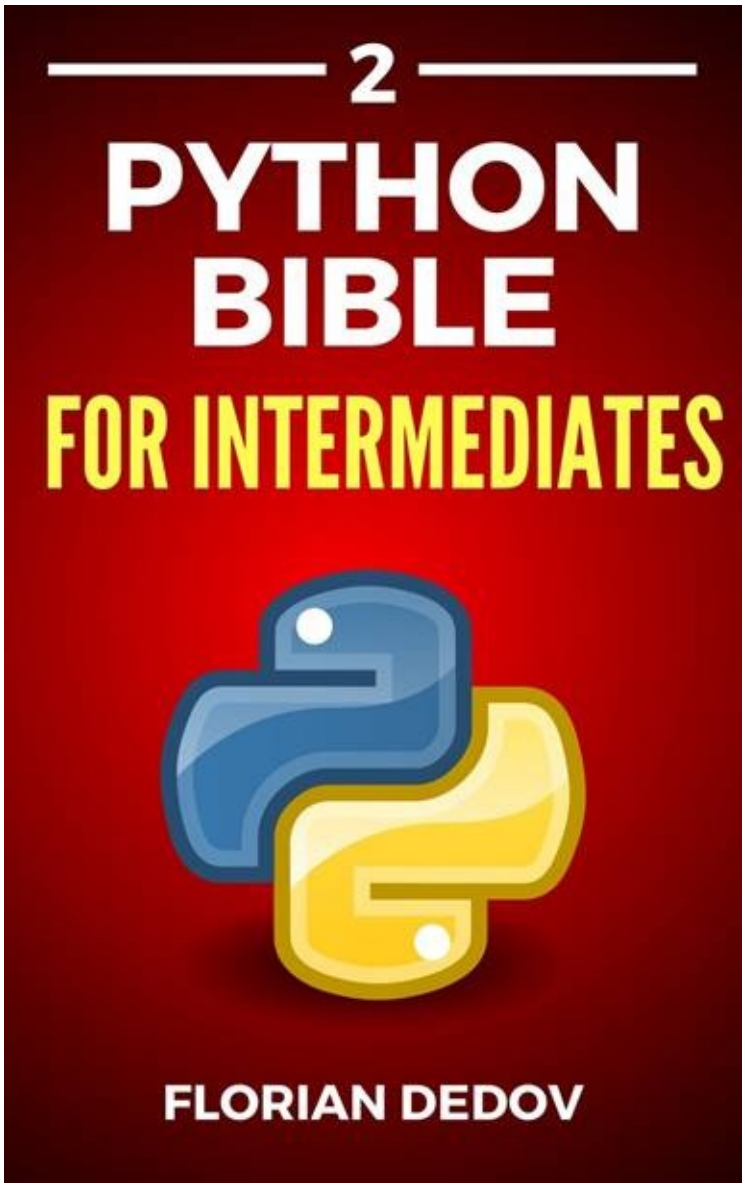
You made it! We covered all of the core basics of Pythons. You now understand how this language is structured but also general programming principles like conditions, loops and functions. Definitely, you are now able to develop a basic calculator or other simple applications. But the journey has just begun.

This is only the first part of the Python Bible Series. We've covered the topics for beginners but the real fun starts when we get into more advanced topics like network programming, threading, machine learning, data science, finance, neural networks and more. With this book you have an excellent basis for the next volumes of the Python Bible and I encourage you to continue your journey.

The next part will be for intermediates and advanced programmers, which you know belong to. So stay tuned and keep coding!

Last but not least, a little reminder. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!


2
PYTHON
BIBLE
FOR INTERMEDIATES
FLORIAN DEDOV

THE
PYTHON BIBLE

VOLUME TWO

INTERMEDIATE AND ADVANCED

BY

FLORIAN DEDOV

TABLE OF CONTENT

INTRODUCTION

I think I don’t have to convince you that Python is one of the most important languages of our time and worth learning. If you are reading this book, I assume that you have already programmed in Python and know the basic concepts of this language. For this book, you will definitely need the foreknowledge from the first volume, since we will build on the skills taught there.

INTERMEDIATE CONCEPTS

So what can you expect from this second volume? Basically, we will dive deep into more advanced topics of Python but also of programming in general. We'll start with object-oriented programming, classes and objects. Then we will talk about multithreading, network programming and database access. Also, we are going to build an efficient port scanner along the way. After that, we will cover recursion, XML processing and other interesting topics like logging and regular expressions.

There is a lot to learn here and the concepts get more and more complex as we go on. So stay tuned and code along while reading. This will help you to understand the material better and to practice implementing it. I wish you a lot of fun and success with your journey and this book!

Just one little thing before we start. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you have learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

1 – CLASSES AND OBJECTS

Python is an object-oriented language which means that the code can be divided into individual units, namely *objects*. Each of these objects is an instance of a so-called *class*. You can think of the class as some sort of blueprint. For example, the blueprint of a car could be the class and an object would be the actual physical car. So a class has specific attributes and functions but the values vary from object to object.

CREATING CLASSES

In Python, we use the keyword *class* in order to define a new class. Everything that is indented after the colon belongs to the class.

```
class Car:

  def __init__(self, manufacturer, model, hp):
    self.manufacturer = manufacturer
    self.model = model
    self.hp = hp
```

After the *class* keyword, we put the class name. In this example, this is *Car*.

CONSTRUCTOR

What we notice first here, is a special function called *__init__*. This is the so-called *constructor.* Every time we create an instance or an object of our class, we use this constructor. As you can see, it accepts a couple of parameters. The first one is the parameter *self* and it is mandatory. Every function of the class needs to have at least this parameter.

The other parameters are just our custom attributes. In this case, we have chosen the manufacturer, the model and the horse power (hp).

When we write *self.attribute*, we refer to the actual attribute of the respective object. We then assign the value of the parameters to it.

ADDING FUNCTIONS

We can simply add functions to our class that perform certain actions. These functions can also access the attributes of the class.

```
class Car:


    def __init__(self, manufacturer, model, hp):
        self.manufacturer = manufacturer
        self.model = model
        self.hp = hp
```

```
    def print_info(self):
        print("Manufacturer: {}, Model: {}, HP; {}"
              .format(self.manufacturer,
                      self.model,
                      self.hp))
```

Here we have the function *print_info* that prints out information about the attributes of the respective object. Notice that we also need the parameter *self* here.

CLASS VARIABLES

In the following code, you can see that we can use one and the same variable across all the objects of the class, when it is defined without referring to *self*.

```
class Car:


    amount_cars = 0


    def __init__(self, manufacturer, model, hp):
        self.manufacturer = manufacturer
        self.model = model
        self.hp = hp
        Car.amount_cars += 1


    def print_car_amount(self):
        print("Amount: {}"
              .format(Car.amount_cars))
```

The variable *amount_cars* doesn’t belong to the individual object since it’s not addressed with *self*. It is a class variable and its value is the same for all objects or instances.

Whenever we create a new car object, it increases by one. Then, every object can access and print the amount of existing cars.

DESTRUCTORS

In Python, we can also specify a method that gets called when our object gets *destroyed* or *deleted* and is no longer needed. This function is called *destructor* and it is the opposite of the *constructor*.

```
class Car:


    amount_cars = 0


    def __init__(self, manufacturer, model, hp):
        self.manufacturer = manufacturer
        self.model = model
        self.hp = hp
        Car.amount_cars += 1


    def __del__(self):
        print("Object gets deleted!")
        Car.amount_cars -=1
```

The destructor function is called *__del__*. In this example, we print an informational message and decrease the amount of existing cars by one, when an object gets deleted.

CREATING OBJECTS

Now that we have implemented our class, we can start to create some objects of it.

```
myCar1 = Car("Tesla", "Model X", 525)
```

First, we specify the name of our object, like we do with ordinary variables. In this case, our object's name is *myCar1*. We then create an object of the *Car* class by writing the class name as a function. This calls the constructor, so we can pass our parameters. We can then use the functions of our car object.

```
myCar1.print_info()

myCar1.print_car_amount()
```

The results look like this:

```
Manufacturer: Tesla, Model: Model X, HP; 525
Amount: 1
```

What you can also do is directly access the attributes of an object.

```
print(myCar1.manufacturer)

print(myCar1.model)

print(myCar1.hp)
```

Now let's create some more cars and see how the amount changes.

```
myCar1 = Car("Tesla", "Model X", 525)

myCar2 = Car("BMW", "X3", 200)

myCar3 = Car("VW", "Golf", 100)

myCar4 = Car("Porsche", "911", 520)
```

```
del myCar3


myCar1.print_car_amount()
```

Here we first create four different car objects. We then delete one of them and finally we print out the car amount. The result is the following:

```
Object gets deleted!
Amount: 3
```

Notice that all the objects get deleted automatically when our program ends. But we can manually delete them before that happens by using the *del* keyword.

HIDDEN ATTRIBUTES

If we want to create *hidden* attributes that can only be accessed within the class, we can do this with *underlines*.

```
class MyClass:


    def __init__(self):
        self.__hidden = "Hello"
        print(self.__hidden) # Works


m1 = MyClass()
print(m1.__hidden) # Doesn't Work
```

By putting two underlines before the attribute name, we make it invisible from outside the class. The first print function works because it is inside of the class. But when we try to access this attribute from the object, we can’t.

INHERITANCE

One very important and powerful concept of object-oriented programming is *inheritance*. It allows us to use existing classes and to extend them with new attributes and functions.

For example, we could have the *parent class* which represents a *Person* and then we could have many *child classes* like *Dancer, Policeman, Artist* etc. All of these would be considered a person and they would have the same basic attributes. But they are special kinds of persons with more attributes and functions.

```
class Person:


    def __init__(self, name, age):
        self.name = name
        self.age = age


    def get_older(self, years):
        self.age += years


class Programmer(Person):


    def __init__(self, name, age, language):
        super(Programmer, self).__init__(name, age)
        self.language = language


    def print_language(self):
        print("Favorite Programming Language: {}"
              .format(self.language))
```

You can see that we created two classes here. The first one is the *Person* class, which has the attributes *name* and *age*. Additionally, it has a function *get_older* that increases the age.

The second class is the *Programmer* class and it inherits from the *Person* class. This is stated in the parentheses after the class name. In the constructor we have one additional attribute *language*. First we need to pass our class to the *super* function. This function allows us to call the constructor of the parent class *Person*. There we pass our first two parameters. We also have an additional function *print_language*.

```
p1 = Programmer("Mike", 30, "Python")


print(p1.age)

print(p1.name)

print(p1.language)


p1.get_older(5)


print(p1.age)
```

Our *Programmer* object can now access all the attributes and functions of its parent class, additionally to its new values. These are the results of the statements:

```
30
Mike
Python
35
```

OVERWRITING METHODS

When one class inherits from another class, it can overwrite its methods. This is automatically done by defining a method with the same name and the same amount of parameters.

```
class Animal:
```

```
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        print("Some sound!")

class Dog(Animal):

    def __init__(self, name):
        super(Dog, self).__init__(name)

    def make_sound(self):
        print("Bark!")
```

Here the function *make_sound* was overwritten in the child class *Dog*. It now has a different functionality than the function of the parent class *Animal*.

OPERATOR OVERLOADING

When we create a class with various attributes, it is not clear what should happen when we perform certain operations on them. For example, what should happen when we add two humans or when we multiply them? Since there is no default solution for this question, we can *overload* and define the operators ourselves. That allows us to choose what happens when we apply the operators on our objects.

```
class Vector():


    def __init__(self, x, y):
        self.x = x
        self.y = y


    def __str__(self):
        return "X: %d, Y: %d" % (self.x,
                                 self.y)


    def __add__(self, other):
        return Vector(self.x + other.x,
                      self.y + other.y)


    def __sub__(self, other):
        return Vector(self.x - other.x,
```

```
                self.y - other.y)
```

Here you see a class that represents the function of a *Vector*. When you add a vector to another, you need to add the individual values. This is the same for subtracting. If you don't know what vectors are mathematically, forget about them. This is just one example.

We use the functions *__add__* and *__sub__* to define what happens when we apply the + and the – operator. The *__str__* function determines what happens when we print the object.

```
v1 = Vector(3, 5)

v2 = Vector(6, 2)

v3 = v1 + v2

v4 = v1 - v2


print(v1)

print(v2)

print(v3)

print(v4)
```

The results are the following:

```
X: 3, Y: 5
X: 6, Y: 2
X: 9, Y: 7
X: -3, Y: 3
```

2 – MULTITHREADING

Threads are lightweight processes that perform certain actions in a program and they are part of a process themselves. These threads can work in parallel with each other in the same way as two individual applications can.

Since threads in the same process share the memory space for the variables and the data, they can exchange information and communicate efficiently. Also, threads need fewer resources than processes. That’s why they’re often called lightweight processes.

HOW A THREAD WORKS

A thread has a beginning or a start, a working sequence and an end. But it can also be stopped or put on hold at any time. The latter is also called *sleep*.

There are two types of threads: *Kernel Threads* and *User Threads*. Kernel threads are part of the operating system, whereas user threads are managed by the programmer. That’s why we will focus on user threads in this book.

In Python, a thread is a class that we can create instances of. Each of these instances then represents an individual thread which we can start, pause or stop. They are all independent from each other and they can perform different operations at the same time.

For example, in a video game, one thread could be rendering all the graphics, while another thread processes the keyboard and mouse inputs. It would be unthinkable to serially perform these tasks one after the other.

STARTING THREADS

In order to work with threads in Python, we will need to import the respective library *threading*.

```
import threading
```

Then, we need to define our target function. This will be the function that contains the code that our thread shall be executing. Let's just keep it simple for the beginning and write a *hello world* function.

```
import threading


def hello():

    print("Hello World!")


t1 = threading.Thread(target=hello)

t1.start()
```

After we have defined the function, we create our first thread. For this, we use the class *Thread* of the imported *threading* module. As a parameter, we specify the *target* to be the *hello* function. Notice that we don't put parentheses after our function name here, since we are not calling it but just referring to it. By using the *start* method we put our thread to work and it executes our function.

START VS RUN

In this example, we used the function *start* to put our thread to work. Another alternative would be the function *run*. The difference between these two functions gets important, when we are dealing with more than just one thread.

When we use the *run* function to execute our threads, they run serially one after the other. They wait for each other to finish. The *start* function puts all of them

to work simultaneously.

The following example demonstrates this difference quite well.

```
import threading


def function1():
    for x in range(1000):
        print("ONE")


def function2():
    for x in range(1000):
        print("TWO")


t1 = threading.Thread(target=function1)
t2 = threading.Thread(target=function2)
t1.start()
t2.start()
```

When you run this script, you will notice that the output alternates between *ONEs* and *TWOs*. Now if you use the *run* function instead of the *start* function, you will see 1000 times *ONE* followed by 1000 times *TWO*. This shows you that the threads are run serially and not in parallel.

One more thing that you should know is that the application itself is also the main thread, which continues to run in the background. So while your threads are running, the code of the script will be executed unless you wait for the threads to finish.

WAITING FOR THREADS

If we want to wait for our threads to finish before we move on with the code, we can use the *join* function.

```
import threading


def function():
    for x in range(500000):
        print("HELLO WORLD!")


t1 = threading.Thread(target=function)
t1.start()


print("THIS IS THE END!")
```

If you execute this code, you will start printing the text *"HELLO WORLD!"* 500,000 times. But what you will notice is that the last print statement gets executed immediately after our thread starts and not after it ends.

```
t1 = threading.Thread(target=function)
t1.start()


t1.join()


print("THIS IS THE END!")
```

By using the *join* function here, we wait for the thread to finish before we move on with the last print statement. If we want to set a maximum time that we want to wait, we just pass the number of seconds as a parameter.

```
t1 = threading.Thread(target=function)

t1.start()



t1.join(5)



print("THIS IS THE END!")
```

In this case, we will wait for the thread to finish but only a maximum of five seconds. After this time has passed we will proceed with the code.

Notice that we are only waiting for this particular thread. If we would have other threads running at the same time, we would have to call the *join* function on each of them in order to wait for all of them.

THREAD CLASSES

Another way to build our threads is to create a class that inherits the *Thread* class. We can then modify the *run* function and implement our functionality. The *start* function is also using the code from the *run* function so we don't have to worry about that.

```
import threading



class MyThread(threading.Thread):



    def __init__(self, message):
```

```
        threading.Thread.__init__(self)
        self.message = message

    def run(self):
        for x in range(100):
            print(self.message)

mt1 = MyThread("This is my thread message!")
mt1.start()
```

It is basically the same but it offers more modularity and structure, if you want to use attributes and additional functions.

SYNCHRONIZING THREADS

Sometimes you are going to have multiple threads running that all try to access the same resource. This may lead to inconsistencies and problems. In order to prevent such things there is a concept called *locking*. Basically, one thread is locking all of the other threads and they can only continue to work when the lock is removed.

I came up with the following quite trivial example. It seems a bit abstract but you can still get the concept here.

```
import threading

import time


x = 8192


def halve():
    global x
    while(x > 1):
        x /= 2
        print(x)
        time.sleep(1)
    print("END!")


def double():
    global x
```

```
    while(x < 16384):
        x *= 2
        print(x)
        time.sleep(1)
    print("END!")


t1 = threading.Thread(target=halve)
t2 = threading.Thread(target=double)


t1.start()
t2.start()
```

Here we have two functions and the variable *x* that starts at the value *8192*. The first function halves the number as long as it is greater than one, whereas the second function doubles the number as long as it is less than *16384*.

Also, I've imported the module *time* in order to use the function *sleep*. This function puts the thread to sleep for a couple of seconds (in this case one second). So it pauses. We just do that, so that we can better track what's happening.

When we now start two threads with these target functions, we will see that the script won't come to an end. The *halve* function will constantly decrease the number and the *double* function will constantly increase it.

With locking we can now let one function finish before the next function starts. Of course, in this example this is not very useful but we can do the same thing in much more complex situations.

```
import threading
```

```
import time

x = 8192

lock = threading.Lock()

def halve():
    global x, lock
    lock.acquire()
    while(x > 1):
        x /= 2
        print(x)
        time.sleep(1)
    print("END!")
    lock.release()

def double():
    global x, lock
    lock.acquire()
    while(x < 16384):
        x *= 2
```

```
        print(x)
        time.sleep(1)
    print("END!")
    lock.release()


t1 = threading.Thread(target=halve)
t2 = threading.Thread(target=double)


t1.start()
t2.start()
```

So here we added a couple of elements. First of all we defined a *Lock* object. It is part of the *threading* module and we need this object in order to manage the locking.

Now, when we want to try to lock the resource, we use the function *acquire*. If the lock was already locked by someone else, we wait until it is released again before we continue with the code. However, if the lock is free, we lock it ourselves and release it at the end using the *release* function.

Here, we start both functions with a locking attempt. The first function that gets executed will lock the other function and finish its loop. After that it will release the lock and the other function can do the same.

So the number will be halved until it reaches the number one and then it will be doubled until it reaches the number *16384*.

SEMAPHORES

Sometimes we don’t want to completely lock a resource but just limit it to a certain amount of threads or accesses. In this case, we can use so-called *semaphores*.

To demonstrate this concept, we will look at another very abstract example.

```
import threading

import time


semaphore = threading.BoundedSemaphore(value=5)


def access(thread_number):
    print("{}: Trying access..."
          .format(thread_number))
    semaphore.acquire()
    print("{}: Access granted!"
          .format(thread_number))
    print("{}: Waiting 5 seconds..."
          .format(thread_number))
    time.sleep(5)
    semaphore.release()
    print("{}: Releasing!"
```

```
            .format(thread_number))


for thread_number in range(10):
    t = threading.Thread(target=access,
                         args=(thread_number,))
    t.start()
```

We first use the *BoundedSemaphore* class to create our *semaphore* object. The parameter *value* determines how many parallel accesses we allow. In this case, we choose five.

With our *access* function, we try to access the semaphore. Here, this is also done with the *acquire* function. If there are less than five threads utilizing the semaphore, we can acquire it and continue with the code. But when it's full, we need to wait until some other thread frees up one space.

When we run this code, you will see that the first five threads will immediately run the code, whereas the remaining five threads will need to wait five seconds until the first threads *release* the semaphore.

This process makes a lot of sense when we have limited resources or limited computational power in a system and we want to limit the access to it.

EVENTS

With *events* we can manage our threads even better. We can pause a thread and wait for a certain *event* to happen, in order to continue it.

```
import threading


event = threading.Event()


def function():
    print("Waiting for event...")
    event.wait()
    print("Continuing!")


thread = threading.Thread(target=function)
thread.start()


x = input("Trigger event?")
if(x == "yes"):
    event.set()
```

To define an *event* we use the *Event* class of the *threading* module. Now we define our *function* which waits for our event. This is done with the *wait* function. So we start the thread and it waits.

Then we ask the user, if he wants to trigger the event. If the answer is yes, we

trigger it by using the *set* function. Once the event is triggered, our function no longer waits and continues with the code.

DAEMON THREADS

So-called *daemon threads* are a special kind of thread that runs in the background. This means that the program can be terminated even if this thread is still running. Daemon threads are typically used for background tasks like synchronizing, loading or cleaning up files that are not needed anymore. We define a thread as a *daemon* by setting the respective parameter in the constructor for *Thread* to *True*.

```
import threading

import time


path = "text.txt"

text = ""


def readFile():
    global path, text
    while True:
        with open(path) as file:
            text = file.read()
        time.sleep(3)


def printloop():
    global text
    for x in range(30):
```

```
        print(text)

        time.sleep(1)


t1 = threading.Thread(target=readFile, daemon=True)

t2 = threading.Thread(target=printloop)


t1.start()

t2.start()
```

So, here we have two functions. The first one constantly reads in the text from a file and saves it into the *text* variable. This is done in an interval of three seconds. The second one prints out the content of *text* every second but only 30 times.

As you can see, we start the *readFile* function in a *daemon thread* and the *printloop* function in an ordinary thread. So when we run this script and change the content of the *text.txt* file while it is running, we will see that it prints the actual content all the time. Of course, we first need to create that file manually.

After it printed the content 30 times however, the whole script will stop, even though the daemon thread is still reading in the files. Since the ordinary threads are all finished, the program ends and the daemon thread just gets terminated.

3 – QUEUES

In Python, *queues* are structures that take in data in a certain order to then output it in a certain order. The default queue type is the so-called *FIFO queue*. This stands for *first in first out* and the name describes exactly what it does. The elements that enter the queue first are also the elements that will leave the queue first.

```
import queue


q = queue.Queue()


for x in range(5):

    q.put(x)


for x in range(5):

    print(q.get(x))
```

In order to work with queues in Python, we need to import the module *queue*. We can then create an instance of the class *Queue* by using the constructor.

As you can see, we are using two functions here – *put* and *get*. The *put* function adds an element to the queue that can then be extracted by the *get* function.

Here, we put in the numbers one to five into our queue. Then, we just get the elements and print them. The order stays the same, since the default queue is *FIFO*.

QUEUING RESOURCES

Let’s say we have a list of numbers that need to be processed. We decide to use multiple threads, in order to speed up the process. But there might be a problem. The threads don’t know which number has already been processed and they might do the same work twice, which would be unnecessary. Also, solving the problem with a counter variable won’t always work, because too many threads access the same variable and numbers might get skipped.

In this case we can just use queues to solve our problems. We fill up our queue with the numbers and every thread just uses the *get* function, to get the next number and process it.

Let’s say we have the following *worker* function:

```
import threading

import queue

import math


q = queue.Queue()

threads = []


def worker():

    while True:

        item = q.get()

        if item is None:

            break

        print(math.factorial(item))
```

```
        q.task_done()
```

We start out with an empty queue and an empty list for threads. Our function has an endless loop that gets numbers from the list and calculates the factorial of them. For this *factorial* function, we need to import the module *math*. But you can ignore this part, since it is only used because the computation requires a lot of resources and takes time. At the end, we use the function *task_done* of the queue, in order to signal that the element was processed.

```
for x in range(5):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)


zahlen = [134000, 14, 5, 300, 98, 88, 11, 23]


for item in zahlen:
    q.put(item)


q.join()


for i in range(5):
    q.put(None)
```

We then use a for loop to create and start five threads that we also add to our list. After that, we create a list of numbers, which we then all put into the queue.

The method *join* of the *queue* waits for all elements to be extracted and processed. Basically, it waits for all the *task_done* functions. After that, we put *None* elements into the queue, so that our loops break.

Notice that our threads can't process the same element twice or even skip one because they can only get them by using the *get* function.

If we would use a counter for this task, two threads might increase it at the same time and then skip an element. Or they could just access the same element simultaneously. Queues are irreplaceable for tasks like this. We will see a quite powerful application of queues in the chapter about *networking*.

LIFO QUEUES

An alternative to the *FIFO queues* would be the *LIFO queues*. That stands for *last in first out*. You can imagine this queue like some sort of stack. The element you put last on top of the stack is the first that you can get from it.

```
import queue


q = queue.LifoQueue()


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


for x in numbers:
    q.put(x)


while not q.empty():
    print(q.get())
```

By using the *LifoQueue* class from the *queue* module, we can create an instance

of this type. When we now put in the numbers one to five in ascending order, we will get them back in descending order.

The result would be:

```
5 4 3 2 1
```

PRIORITIZING QUEUES

What we can also do in Python, is creating *prioritized queues*. In these, every element gets assigned a level of priority that determines when they will leave the queue.

```
import queue


q = queue.PriorityQueue()


q.put((8, "Some string"))

q.put((1, 2023))

q.put((90, True))

q.put((2, 10.23))


while not q.empty():

    print(q.get())
```

Here, we create a new instance of the class *PriorityQueue*. When we put a new element into this queue, we need to pass a *tuple* as a parameter. The first element of the tuple is the level of importance (the lower the number, the higher the priority) and the second element is the actual object or value that we want to put into the queue.

When we execute the print statement of the loop, we get the following results:

```
(1, 2023)
(2, 10.23)
(8, 'Some string')
(90, True)
```

As you can see, the elements got sorted by their priority number. If you only want to access the actual value, you need to address the index one because it is the second value of the tuple.

```
while not q.empty():

    print(q.get()[1])
```

4 – NETWORK PROGRAMMING

Now we get into one of the most interesting intermediate topics – *network programming*. It is about communicating with other applications and devices via some network. That can be the internet or just the local area network.

SOCKETS

WHAT ARE SOCKETS?

Whenever we talk about networking in programming, we also have to talk about *sockets*. They are the endpoints of the communication channels or basically, the endpoints that talk to each other. The communication may happen in the same process or even across different continents over the internet.

What's important is that in Python we have different access levels for the network services. At the lower layers, we can access the simple sockets that allow us to use the connection-oriented and connectionless protocols like TCP or UDP, whereas other Python modules like *FTP* or *HTTP* are working on a higher layer – the *application layer*.

CREATING SOCKETS

In order to work with sockets in Python, we need to import the module *socket*.

```
import socket
```

Now, before we start defining and initializing our socket, we need to know a couple of things in advance:

- Are we using an internet socket or a UNIX socket?
- Which protocol are we going to use?
- Which IP-address are we using?
- Which port number are we using?

The first question can be answered quite simply. Since we want to communicate over a network instead of the operating system, we will stick with the *internet socket*.

The next question is a bit trickier. We choose between the protocols *TCP (*Transmission Control Protocol) and *UDP (*User Datagram Protocol). TCP is connection-oriented and more trustworthy than UDP. The chances of losing data are minimal in comparison to UDP. On the other hand, UDP is much faster than TCP. So the choice depends on the task we want to fulfil. For our examples, we will stick with TCP since we don't care too much about speed for now.

The IP-address should be the address of the host our application will run on. For

now, we will use *127.0.0.1* which is the *localhost* address. This applies to every machine. But notice that this only works when you are running your scripts locally.

For our port we can basically choose any number we want. But be careful with low numbers, since all numbers up to 1024 are *standardized* and all numbers from 1024 to 49151 are *reserved*. If you choose one of these numbers, you might have some conflicts with other applications or your operating system.

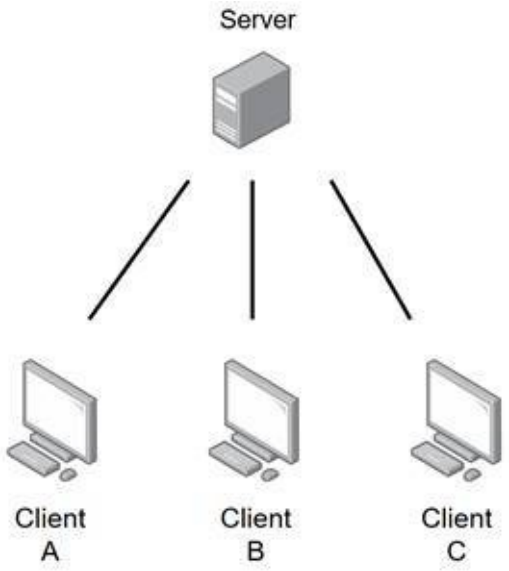
```
import socket



s = socket.socket(socket.AF_INET,

                  socket.SOCK_STREAM)
```

Here we created our first socket, by initializing an instance of the class *socket*. Notice that we passed two parameters here. The first one *AF_INET* states that we want an *internet socket* rather than a *UNIX socket*. The second one *SOCK_STREAM* is for the protocol that we choose. In this case it stands for *TCP*. If we wanted *UDP*, we would have to choose SOCK_DGRAM.

So we have a socket that uses the IP protocol (internet) and the TCP protocol. Now, before we get into the actual setup of the socket, we need to talk a little bit about clients and servers.

CLIENT-SERVER ARCHITECTURE

In a nutshell, the server is basically the one who provides information and *serves* data, whereas the clients are the ones who request and receive the data from the server.



A server opens up a session with every client that connects to it. This way, servers are able to serve multiple clients at once and individually.

SERVER SOCKET METHODS

There are three methods of the *socket* class that are of high importance for the servers.

SERVER SOCKET METHODS	
METHOD	**DESCRIPTION**
bind()	Binds the address that consists of hostname and port to the socket
listen()	Waits for a message or a signal
accept()	Accepts the connection with a client

CLIENT SOCKET METHODS

For the client, there is only one specific and very important method, namely *connect*. With this method the client attempts to connect to a server which then has to *accept* this with the respective method.

OTHER SOCKET METHODS

Also, there are some other socket methods that are quite important in general.

OTHER SOCKET METHODS	
METHOD	**DESCRIPTION**
recv()	Receives a TCP message
send()	Sends a TCP message
recvfrom()	Receives a UDP message
sendto()	Sends a UDP message
close()	Closes a socket
gethostname()	Returns hostname of a socket

CREATING A SERVER

Now that we understand the client-server architecture, we are going to implement our server. We decided that we want to use TCP and an internet socket. For the address we will use the *localhost* address *127.0.0.1* and as a port, we will choose *9999*.

```
s = socket.socket(socket.AF_INET,

                  socket.SOCK_STREAM)

s.bind(("127.0.0.1", 9999))

s.listen()

print("Listening...")
```

Here we initialize our socket like we did in the beginning of this chapter. We then use the method *bind*, in order to assign the IP-address and the port we chose. Notice that we are passing a tuple as a parameter here. Last but not least, we put our socket to listening mode by using the method *listen*.
After that, we just have to create a loop that accepts the client requests that will eventually come in.

server.py

```
import socket


s = socket.socket(socket.AF_INET,

                  socket.SOCK_STREAM)

s.bind(("127.0.0.1", 9999))

s.listen()

print("Listening...")
```

```
while True:
    client, address = s.accept()
    print("Connected to {}".format(address))
    message = "Hello Client!"
    client.send(message.encode('ascii'))
    client.close()
```

The method *accept* waits for a connection attempt to come and accepts it. It then returns a *client* for responses and the *address* of the client that is connected. We can then use this client object in order to send the message. But it’s important that we encode the message first, because otherwise we can’t send it properly. At the end, we *close* the client because we don’t need it anymore.

CREATING A CLIENT

Now our server is done and we just need some clients that connect to it. Our clients shall request a resource from the server. In this case, this is the message *“Hello Client!”*.

For our client we also need a socket but this time it will not use the function *bind* but the function *connect*. So let’s start writing our code into a new file.

```
import socket



s = socket.socket(socket.AF_INET,

                  socket.SOCK_STREAM)

s.connect(("127.0.0.1", 9999))
```

We just create an ordinary internet socket that uses TCP and then connect it to the localhost IP-address at the port 9999.

To now get the message from the server and decode it, we will use the *recv* function.

client.py

```
import socket



s = socket.socket(socket.AF_INET,

                  socket.SOCK_STREAM)

s.connect(("127.0.0.1", 9999))

message = s.recv(1024)

s.close()
```

```
print(message.decode('ascii'))
```

After we connect to the server, we try to receive up to 1024 bytes from it. We then save the message into our variable and then we decode and print it.

CONNECTING SERVER AND CLIENT

Now in order to connect these two entities, we first need to run our server. If there is no server listening on the respective port, our client can’t connect to anything. So we run our *server.py* script and start listening.

After that, we can run our *client.py* script many times and they will all connect to the server. The results will look like this:

Server

```
Listening...
Connected to ('127.0.0.1', 4935)
Connected to ('127.0.0.1', 4942)
Connected to ('127.0.0.1', 4943)
Connected to ('127.0.0.1', 4944)
Connected to ('127.0.0.1', 4945)
```

Client

```
Hello Client!
```

One thing you might optimize on that script if you want is the exception handling. If there is no server listening and our client tries to connect, we get a *ConnectionRefusedError* and our script crashes. Now you can fix this with the knowledge from the first book.

Hint: Use try and except!

PORT SCANNER

Now we have learned a lot about multithreading, locking, queues and sockets. With all that knowledge, we can create a highly efficient and well working *port scanner*.

What a port scanner basically does is: It tries to connect to certain ports at a host or a whole network, in order to find loopholes for future attacks. Open ports mean a security breach. And with our skills, we can already code our own penetration testing tool.

WARNING: *Port scanning is not allowed on any hosts or networks which you don't have explicit permission for. Only scan your own networks or networks for which you were given permission. I don't take any liability for what you do with this knowledge, since I warned you!*

```
import socket


target = "10.0.0.5"


def portscan(port):
    try:
        s = socket.socket(socket.AF_INET,
                          socket.SOCK_STREAM)
        conn = s.connect((target, port))
        return True
    except:
```

```
        return False


for x in range(1, 501):
    if(portscan(x)):
        print("Port {} is open!".format(x))
    else:
        print("Port {} is closed!".format(x))
```

So this scanner is quite simple. We define a target address. In this case, this is *10.0.0.5*. Our function *portscan* simply tries to connect to a certain port at that host. If it succeeds, the function returns *True*. If we get an error or an exception, it returns *False*.

This is as simple as a port scan can get. We then use a for loop to scan the first 500 ports and we always print if the port is open or closed.

Just choose a target address and run this script. You will see that it works.

```
Port 21 is closed!
Port 22 is open!
Port 23 is closed!
Port 24 is closed!
Port 25 is open!
```

But you will also notice that it is extremely slow. That's because we serially scan one port after the other. And I think we have already learned how to handle that.

THREADED PORT SCANNER

In order to speed up the scanning process, we are going to use *multithreading*. And to make sure that every port gets scanned and also that no port is scanned twice, we will use *queues.*

```
import socket

from queue import Queue

import threading


target = "10.0.0.5"


q = Queue()

for x in range(1,501):

    q.put(x)


def portscan(port):

    try:

        s = socket.socket(socket.AF_INET,

                          socket.SOCK_STREAM)

        conn = s.connect((target, port))

        return True

    except:
```

```
        return False


def worker():
    while True:
        port = q.get()
        if portscan(port):
            print("Port {} is open!"
                  .format(port))
```

So we start by creating a queue and filling it up with all numbers from 1 to 500. We then have two functions. The *portscan* function does the scanning itself and the *worker* function gets all the ports from the queue in order to pass them to the *portscan* function and prints the result. In order to not get confused with the output, we only print when a port is open because we don't care when a port is closed.

Now we just have to decide how many threads we want to start and then we can go for it.

```
for x in range(30):
    t = threading.Thread(target=worker)
    t.start()
```

In this example, we start 30 threads at the same time. If you run this, you will see that it increases the scanning speed a lot. Within a few seconds, all the 500 ports are scanned. So if you want, you can increase the number to 5000.

The results for my virtual server are the following:

```
Port 25 is open!
Port 22 is open!
Port 80 is open!
Port 110 is open!
Port 119 is open!
```

Port 143 is open!
Port 443 is open!
Port 465 is open!

As you can see, there are a lot of vulnerabilities here. You now just have to google which ports are interesting and depending on your side you may either prepare for an attack or fix the security breaches. For example port 22 is SSH and quite dangerous.

5 – DATABASE PROGRAMMING

Databases are one of the most popular ways to store and manage data in computer science. Because of that, in this chapter we are going to take a look at database programming with Python.

Notice that for most databases we use the query language *SQL*, which stands for *Structured Query Language*. We use this language in order to manage the database, the tables and the rows and columns. This chapter is not about database structure itself, nor is it about SQL. Maybe I will write a specific SQL book in the future but here we are only going to focus on the implementation in Python. We are not going to explain the SQL syntax in too much detail.

CONNECTING TO SQLITE

The database that comes pre-installed with Python is called *SQLite*. It is also the one which we are going to use. Of course, there are also other libraries for *MySQL*, *MongoDB* etc.

In order to use *SQLite* in Python, we need to import the respective module – *sqlite3*.

```
import sqlite3
```

Now, to create a new database file on our disk, we need to use the *connect* method.

```
conn = sqlite3.connect('mydata.db')
```

This right here creates the new file *mydata.db* and connects to this database. It returns a connection object which we save in the variable *conn*.

EXECUTING STATEMENTS

So, we have established a connection to the database. But in order to execute *SQL* statements, we will need to create a so-called *cursor*.

```
c = conn.cursor()
```

We get this cursor by using the method *cursor* of our connection object that returns it. Now we can go ahead and execute all kinds of statements.

CREATING TABLES

For example, we can create our first table like this:

```
c.execute("""CREATE TABLE persons (
            first_name TEXT,
            last_name TEXT,
            age INTEGER
            )""")
```

Here we use the *execute* function and write our query. What we are passing here is SQL code. As I already said, understanding SQL is not the main objective here. We are focusing on the Python part. Nevertheless, it's quite obvious what's happening here. We are creating a new *table* with the name *persons* and each person will have the three attributes *first_name*, *last_name* and *age*.

Now our statement is written but in order to really execute it, we ne need to commit to our connection.

```
conn.commit()
```

When we do this, our statement gets executed and our table created. Notice that this works only once, since after that the table already exists and can't be created again.

At the end, don’t forget to close the connection, when you are done with everything.

```
conn.close()
```

INSERTING VALUES

Now let’s fill up our table with some values. For this, we just use an ordinary *INSERT* statement.

```
c.execute("""INSERT INTO persons VALUES
            ('John', 'Smith', 25),
            ('Anna', 'Smith', 30),
            ('Mike', 'Johnson', 40)""")


conn.commit()
conn.close()
```

So basically, we are just adding three entries to our table. When you run this code, you will see that everything went fine. But to be on the safe side, we will try to now extract the values from the database into our program.

SELECTING VALUES

In order to get values from the database, we need to first execute a *SELECT* statement. After that, we also need to *fetch* the results.

```
c.execute("""SELECT * FROM persons
             WHERE last_name = 'Smith'""")


print(c.fetchall())
```

```
conn.commit()

conn.close()
```

As you can see, our *SELECT* statement that gets all the entries where the *last_name* has the value *Smith*. We then need to use the method *fetchall* of the cursor, in order to get our results. It returns a list of tuples, where every tuple is one entry. Alternatively, we could use the method *fetchone* to only get the first entry or *fetchmany* to get a specific amount of entries. In our case however, the result looks like this:

[('John', 'Smith', 25), ('Anna', 'Smith', 30)]

CLASSES AND TABLES

Now in order to make the communication more efficient and easier, we are going to create a *Person* class that has the columns as attributes.

```
class Person():


    def __init__(self, first=None,
                 last=None, age=None):
        self.first = first
        self.last = last
        self.age = age


    def clone_person(self, result):
        self.first = result[0]
        self.last = result[1]
        self.age = result[2]
```

Here we have a constructor with default parameters. In case we don't specify any values, they get assigned the value *None*. Also, we have a function *clone_person* that gets passed a sequence and assigns the values of it to the object. In our case, this sequence will be the tuple from the *fetching* results.

FROM TABLE TO OBJECT

So let's create a new *Person* object by getting its data from our database.

```
c.execute("""SELECT * FROM persons
```

```
                WHERE last_name = 'Smith'""")


person1 = Person()

person1.clone_person(c.fetchone())



print(person1.first)

print(person1.last)

print(person1.age)
```

Here we fetch the first entry of our query results, by using the *fetchone* function. The result is the following:

John
Smith
25

FROM OBJECT TO TABLE

We can also do that the other way around. Let's create a person objects, assign values to the attributes and then insert this object into our database.

```
person2 = Person("Bob", "Davis", 23)



c.execute("""INSERT INTO persons VALUES

             ('{}', '{}', '{}')"""

          .format(person2.first,

                  person2.last,

                  person2.age))
```

```
conn.commit()

conn.close()
```

Here we used the basic *format* function in order to put our values into the statement. When we execute it, our object gets inserted into the database. We can check this by printing all objects of the table *persons*.

```
c.execute("SELECT * FROM persons")

print(c.fetchall())
```

In the results, we find our new object:

[('John', 'Smith', 25), ('Anna', 'Smith', 30), ('Mike', 'Johnson', 40), **('Bob', 'Davis', 23)**]

PREPARED STATEMENTS

There is a much more secure and elegant way to put the values of our attributes into the SQL statements. We can use *prepared statements*.

```
person = Person("Julia", "Johnson", 28)


c.execute("INSERT INTO persons VALUES (?, ?, ?)",

          (person.first, person.last, person.age))


conn.commit()

conn.close()
```

We replace the values with question marks and pass the values as a tuple in the function. This makes our statements cleaner and also less prone to SQL injections.

MORE ABOUT SQL

For this book, we are done with database programming. But there's a lot more to learn about SQL and databases. As I said, I might publish a detailed SQL book in the future so keep checking my author page on Amazon.

However, if you are interested in learning SQL right now, you can check out the W3Schools tutorial.

W3Schools: https://www.w3schools.com/sql/

6 – RECURSION

In this short chapter, we are going to talk about a programming concept that I would say should be taught in a book for intermediates. This concept is *recursion* and basically it refers to a function calling itself.

```
def function():

    function()


function()
```

So what do you think happens, when you call a function like that? It is a function that calls itself. And this called function calls itself again and so on. Basically, you get into an endless recursion. This is not very useful and in Python we get an *RecursionError* when the maximum recursion depth is exceeded.

Every program has a stack memory and this memory is limited. When we run a function we allocate stack memory space and if there is no space left, this is called *Stack Overflow*. This is also where the name of the famous forum comes from.

FACTORIAL CALCULATION

But recursion can also be useful, if it's managed right. For example, we can write a recursive function that calculates the *factorial* of a number. A factorial is just the value you get, when you multiply a number by every lower whole number down to one.

So 10 factorial would be 10 times 9 times 8 and so on until you get to times 1.

```
def factorial(n):
    if n < 1:
        return 1
    else:
        number = n * factorial(n-1)
        return number
```

Look at this function. When we first call it, the parameter *n* is our base number that we want to calculate the factorial of. If *n* is not smaller than one, we multiply it by the factorial of *n-1*. At the end, we return the number.

Notice that our first function call doesn't return anything until we get down to one. This is because it always calls itself in itself over and over again. At the end all the results are multiplied by the last *return* which of course is *one*. Finally, we can print the end result.

This might be quite confusing, if you have never heard of recursion before. Just take your time and analyze what's happening step-by-step here. Try to play around with this concept of *recursion*.

7 – XML PROCESSING

Up until now, we either saved our data into regular text files or into professional databases. Sometimes however, our script is quite small and doesn’t need a big database but we still want to structure our data in files. For this, we can use *XML*.

XML stands for *Extensible Markup Language* and is a language that allows us to hierarchically structure our data in files. It is platform-independent and also application-independent. XML files that you create with a Python script, can be read and processed by a C++ or Java application.

XML PARSER

In Python, we can choose between two modules for *parsing* XML files – *SAX* and *DOM*.

SIMPLE API FOR XML (SAX)

SAX stands for *Simple API for XML* and is better suited for large XML files or in situations where we have very limited RAM memory space. This is because in this mode we never load the full file into our RAM. We read the file from our hard drive and only load the little parts that we need right at the moment into the RAM. An additional effect of this is that we can only read from the file and not manipulate it and change values.

DOCUMENT OBJECT MODEL (DOM)

DOM stands for *Document Object Model* and is the generally recommended option. It is a language-independent API for working with XML. Here we always load the full XML file into our RAM and then save it there in a hierarchical structure. Because of that, we can use all of the features and also manipulate the file.

Obviously, DOM is a lot faster than SAX because it is using the RAM instead of the hard disk. The main memory is way more efficient than the hard drive. We only use SAX when our RAM is so limited that we can’t even load the full XML file into it without problems.

There is no reason to not use both options in the same projects. We can choose depending on the use case.

XML STRUCTURE

For this chapter, we are going to use the following XML file:

```
<?xml version="1.0"?>
<group>
    <person id="1">
        <name>John Smith</name>
        <age>20</age>
        <weight>80</weight>
        <height>188</height>
    </person>
    <person id="2">
        <name>Mike Davis</name>
        <age>45</age>
        <weight>82</weight>
        <height>185</height>
    </person>
    <person id="3">
        <name>Anna Johnson</name>
        <age>33</age>
        <weight>67</weight>
```

```
        <height>167</height>
    </person>
    <person id="4">
        <name>Bob Smith</name>
        <age>60</age>
        <weight>70</weight>
        <height>174</height>
    </person>
    <person id="5">
        <name>Sarah Pitt</name>
        <age>12</age>
        <weight>50</weight>
        <height>152</height>
    </person>
</group>
```

As you can see, the structure is quite simple. The first row is just a notation and indicates that we are using XML version one. After that we have various tags. Every tag that gets opened also gets closed at the end.

Basically, we have one *group* tag. Within that, we have multiple *person* tags that all have the attribute *id*. And then again, every *person* has four tags with their values. These tags are the attributes of the respective person. We save this file as *group.xml*.

XML WITH SAX

In order to work with *SAX*, we first need to import the module:

```
import xml.sax
```

Now, what we need in order to process the XML data is a *content handler*. It handles and processes the attributes and tags of the file.

```
import xml.sax



handler = xml.sax.ContentHandler()



parser = xml.sax.make_parser()

parser.setContentHandler(handler)

parser.parse("group.xml")
```

First we create an instance of the *ContentHandler* class. Then we use the method *make_parser,* in order to create a *parser* object. After that, we set our *handler* to the content handler of our parser. We can then parse the file by using the method *parse*.

Now, when we execute our script, we don’t see anything. This is because we need to define what happens when an element gets parsed.

CONTENT HANDLER CLASS

For this, we will define our own *content handler* class. Let’s start with a very simple example.

```
import xml.sax
```

```
class GroupHandler(xml.sax.ContentHandler):

    def startElement(self, name, attrs):

        print(name)


handler = GroupHandler()

parser = xml.sax.make_parser()

parser.setContentHandler(handler)

parser.parse("group.xml")
```

We created a class *GroupHandler* that inherits from *ContentHandler*. Then we overwrite the function *startElement*. Every time an element gets processed, this function gets called. So by manipulating it, we can define what shall happen during the parsing process.

Notice that the function has two parameters – *name* and *attr*. These represent the tag name and the attributes. In our simple example, we just print the tag names. So, let's get to a more interesting example.

PROCESSING XML DATA

The following example is a bit more complex and includes two more functions.

```
import xml.sax


class GroupHandler(xml.sax.ContentHandler):


    def startElement(self, name, attrs):

        self.current = name

        if self.current == "person":

            print("--- Person ---")

            id = attrs["id"]
```

```
            print("ID: %s" % id)

    def endElement(self, name):
        if self.current == "name":
            print("Name: %s" % self.name)
        elif self.current == "age":
            print("Age: %s" % self.age)
        elif self.current == "weight":
            print("Weight: %s" % self.weight)
        elif self.current == "height":
            print("Height: %s" % self.height)
        self.current = ""

    def characters(self, content):
        if self.current == "name":
            self.name = content
        elif self.current == "age":
            self.age = content
        elif self.current == "weight":
            self.weight = content
        elif self.current == "height":
            self.height = content

handler = GroupHandler()
parser = xml.sax.make_parser()
parser.setContentHandler(handler)
parser.parse("group.xml")
```

The first thing you will notice here is that we have three functions instead of one. When we start processing an element, the function *startElement* gets called. Then we go on to process the individual *characters* which are *name*, *age*, *weight* and *height*. At the end of the element parsing, we call the *endElement* function.

In this example, we first check if the element is a *person* or not. If this is the case

we print the *id* just for information. We then go on with the *characters* method. It checks which tag belongs to which attribute and saves the values accordingly. At the end, we print out all the values. This is what the results look like:

```
--- Person ---
ID: 1
Name: John Smith
Age: 20
Weight: 80
Height: 188
--- Person ---
ID: 2
Name: Mike Davis
Age: 45
Weight: 82
Height: 185
--- Person ---
...
```

XML WITH DOM

Now, let's look at the DOM option. Here we can not only read from XML files but also change values and attributes. In order to work with DOM, we again need to import the respective module.

```
import xml.dom.minidom
```

When working with DOM, we need to create a so-called *DOM-Tree* and view all elements as collections or sequences.

```
domtree = xml.dom.minidom.parse("group.xml")
group = domtree.documentElement
```

We parse the XML file by using the method *parse*. This returns a DOM-tree, which we save into a variable. Then we get the *documentElement* of our tree and in our case this is *group*. We also save this one into an object.

```
persons = group.getElementsByTagName("person")


for person in persons:
    print("--- Person ---")
    if person.hasAttribute("id"):
        print("ID: %s" % person.getAttribute("id"))


    name = person.getElementsByTagName("name")[0]
    age = person.getElementsByTagName("age")[0]
    weight = person.getElementsByTagName("weight")[0]
    height = person.getElementsByTagName("height")[0]
```

Now, we can get all the individual elements by using the *getElementsByTagName* function. For example, we save all our *person* tags into a variable by using this method and specifying the name of our desired tags. Our *persons* variable is now a sequence that we can iterate over.

By using the functions *hasAttribute* and *getAttribute*, we can also access the attributes of our tags. In this case, this is only the *id*. In order to get the tag values of the individual person, we again use the method *getElementsByTagName*.

When we do all that and execute our script, we get the exact same result as with *SAX*.

--- Person ---
ID: 1
Name: John Smith
Age: 20
Weight: 80
Height: 188
--- Person ---
ID: 2
Name: Mike Davis
Age: 45
Weight: 82
Height: 185
--- Person ---
...

MANIPULATING XML FILES

Since we are now working with *DOM*, let's manipulate our XML file and change some values.

```
persons = group.getElementsByTagName("person")

persons[0].getElementsByTagName("name")[0].childNodes[0].nodeValue = "New Name"
```

As you can see, we are using the same function, to access our elements. Here we adress the *name* tag of the first *person* object. Then we need to access the *childNodes* and change their *nodeValue*. Notice that we only have one element *name* and also only one child node but we still need to address the index zero, for the first element.

In this example, we change the name of the first person to *New Name*. Now in order to apply these changes to the real file, we need to write into it.

```
domtree.writexml(open("group.xml", "w"))
```

We use the *writexml* method of our initial *domtree* object. As a parameter, we pass a file stream that writes into our XML file. After doing that, we can look at the changes.

```
<person id="1">
    <name>New Name</name>
    <age>20</age>
    <weight>80</weight>
    <height>188</height>
</person>
```

We can also change the attributes by using the function *setAttribute*.

```
persons[0].setAttribute("id", "10")
```

Here we change the attribute *id* of the first person to *10*.

```
<person id="10">
    <name>New Name</name>
    <age>20</age>
    <weight>80</weight>
    <height>188</height>
</person>
```

CREATING NEW ELEMENTS

The last thing that we are going to look at in this chapter is creating new XML

elements by using DOM. In order to do that, we first need to define a new *person* element.

```
newperson = domtree.createElement("person")

newperson.setAttribute("id", "6")
```

So we use the *domtree* object and the respective method, to create a new XML element. Then we set the *id* attribute to the next number.

After that, we create all the elements that we need for the person and assign values to them.

```
name = domtree.createElement("name")

name.appendChild(domtree.createTextNode("Paul Smith"))


age = domtree.createElement("age")

age.appendChild(domtree.createTextNode("45"))


weight = domtree.createElement("weight")

weight.appendChild(domtree.createTextNode("78"))


height = domtree.createElement("height")

height.appendChild(domtree.createTextNode("178"))
```

First, we create a new element for each attribute of the person. Then we use the method *appendChild* to put something in between the tags of our element. In this case we create a new *TextNode*, which is basically just text.

Last but not least, we again need to use the method *appendChild* in order to define the hierarchical structure. The attribute elements are the childs of the *person* element and this itself is the child of the *group* element.

```
newperson.appendChild(name)

newperson.appendChild(age)
```

```
newperson.appendChild(weight)

newperson.appendChild(height)


group.appendChild(newperson)


domtree.writexml(open("group.xml", "w"))
```

When we write these changes into our file, we can see the following results:

```
<person id="6">
    <name>Paul Smith</name>
    <age>45</age>
    <weight>78</weight>
    <height>178</height>
</person>
```

8 – LOGGING

No matter what we do in computer science, sooner or later we will need logs. Every system that has a certain size produces errors or conditions in which specific people should be warned or informed. Nowadays, everything gets logged or recorded. Bank transactions, flights, networking activities, operating systems and much more. Log files help us to find problems and to get information about the state of our systems. They are an essential tool for avoiding and understanding errors.

Up until now, we have always printed some message onto the console screen when we encountered an error. But when our applications grow, this becomes confusing and we need to categorize and outsource our logs. In addition, not every message is equally relevant. Some messages are urgent because a critical component fails and some just provide nice information.

SECURITY LEVELS

In Python, we have got five security levels. A higher level means higher importance or urgency.

1. DEBUG
2. INFO
3. WARNING
4. ERROR
5. CRITICAL

Notice that when we choose a certain security level, we also get all the messages of the levels above. So for example, *INFO* also prints the messages of *WARNING*, *ERROR* and *CRITICAL* but not of *DEBUG*.

DEBUG is mainly used for tests, experiments or in order to check something. We typically use this mode, when we are looking for errors (troubleshooting).

We use *INFO* when we want to log all the important events that inform us about what is happening. This might be something like *"User A logged in successfully!"* or *"Now we have 17 users online!"*

WARNING messages are messages that inform us about irregularities and things that might go wrong and become a problem. For example messages like *"Only 247 MB of RAM left!"*

An *ERROR* message gets logged or printed when something didn't go according to the plan. When we get an exception this is a classical error.

CRITICAL messages tell us that critical for the whole system or application happened. This might be the case when a crucial component fails and we have to immediately stop all operations.

CREATING LOGGERS

In order to create a logger in Python, we need to import the *logging* module.

```
import logging
```

Now we can just log messages by directly using the respective functions of the *logging* module.

```
logging.info("First informational message!")

logging.critical("This is serious!")
```

This works because we are using the *root* logger. We haven't created our own loggers yet. The output looks like this:

CRITICAL:root:This is serious!
INFO:root:Logger successfully created!

So let's create our own logger now. This is done by either using the constructor of the *Logger* class or by using the method *getLogger*.

```
logger = logging.getLogger()

logger = logging.Logger("MYLOGGER")
```

Notice that we need to specify a name for our logger, if we use the constructor. Now we can log our messages.

```
logger.info("Logger successfully created!")

logger.log(logging.INFO, "Successful!")

logger.critical("Critical Message!")

logger.log(logging.CRITICAL, "Critical!")
```

Here we also have two different options for logging messages. We can either

directly call the function of the respective security level or we can use the method *log* and specify the security level in the parameters.

But when you now execute the script, you will notice that it will only print the critical messages. This has two reasons. First of all, we need to adjust the level of the logger and second of all, we need to remove all of the *handlers* from the default *root* logger.

```
for handler in logging.root.handlers:

    logging.root.removeHandler(handler)

logging.basicConfig(level=logging.INFO)
```

Here we just use a for loop in order to remove all the handlers from the root logger. Then we use the *basicConfig* method, in order to set our logging level to *INFO*. When we now run our code again, the output is the following:

```
INFO:MYLOGGER:Logger successfully created!
INFO:MYLOGGER:Successful!
CRITICAL:MYLOGGER:Critical Message!
CRITICAL:MYLOGGER:Critical!
```

LOGGING INTO FILES

What we are mainly interested in is logging into files. For this, we need a so-called *FileHandler*. It is an object that we add to our logger, in order to make it log everything into a specific file.

```
import logging


logger = logging.getLogger("MYLOGGER")

logger.setLevel(logging.INFO)


handler = logging.FileHandler("logfile.log")

handler.setLevel(logging.INFO)


logger.addHandler(handler)

logger.info("Log this into the file!")

logger.critical("This is critical!")
```

We start again by defining a logger. Then we set the security level to *INFO* by using the function *setLevel*. After that, we create a *FileHandler* that logs into the file *logfile.log*. Here we also need to set the security level. Finally, we add the handler to our logger using the *addHandler* function and start logging messages.

FORMATTING LOGS

One thing that you will notice is that we don’t have any format in our logs. We don’t know which logger was used or which security level our message has. For this, we can use a so-called *formatter*.

```
import logging


logger = logging.getLogger()

logger.setLevel(logging.INFO)


handler = logging.FileHandler("logfile.log")

handler.setLevel(logging.INFO)


formatter = logging.Formatter('%(asctime)s: %(levelname)s - %(message)s')

handler.setFormatter(formatter)


logger.addHandler(handler)

logger.info("This will get into the file!")
```

We create a formatter by using the constructor of the respective class. Then we use the keywords for the timestamp, the security level name and the message. Last but not least, we assign the formatter to our handler and start logging again. When we now look into our file, we will find a more detailed message.

```
2019-06-25 15:41:43,523: INFO - This will get into the file!
```

These log messages can be very helpful, if they are used wisely. Place them wherever something important or alarming happens in your code.

9 – REGULAR EXPRESSIONS

In programming, you will oftentimes have to deal with long texts from which we want to extract specific information. Also, when we want to process certain inputs, we need to check for a specific pattern. For example, think about emails. They need to have some text, followed by an @ character, then again some text and finally a *dot* and again some little text.

In order to make the validations easier, more efficient and more compact, we use so-called *regular expressions*.

The topic of regular expressions is very huge and you could write a whole book only about it. This is why we are not going to focus too much on the various placeholders and patterns of the expressions themselves but on the implementation of *RegEx* in Python.

So in order to confuse you right in the beginning, let’s look at a regular expression that checks if the format of an email-address is valid.

^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$

Now you can see why this is a huge field to learn. In this chapter, we are not going to build regular expressions like this. We are going to focus on quite simple examples and how to properly implement them in Python.

IDENTIFIER

Let’s get started with some basic knowledge first. So-called *identifiers* define what kind of character should be at a certain place. Here you have some examples:

|REGEX IDENTIFIERS	
IDENTIFIER	**DESCRIPTION**
\d	Some digit
\D	Everything BUT a digit
\s	White space
\S	Everything BUT a white space
\w	Some letter
\W	Everything BUT a letter
.	Every character except for new lines
\b	White spaces around a word
\.	A dot

MODIFIER

The *modifiers* extend the regular expressions and the identifiers. They might be seen as some kind of operator for regular expressions.

REGEX MODIFIERS	
MODIFIER	**DESCRIPTION**
{x,y}	A number that has a length between x and y
+	At least one
?	None or one
*	Everything
$	At the end of a string
^	At the beginning of a string
|	Either Or Example: x | y = either x or y
[]	Value range
{x}	x times
{x,y}	x to y times

ESCAPE CHARACTERS

Last but not least, we have the classic *escape characters.*

REGEX ESCAPE CHARATCERS	
CHARACTER	**DESCRIPTION**
\n	New Line
\t	Tab
\s	White Space

APPLYING REGULAR EXPRESSIONS

FINDING STRINGS

In order to apply these regular expressions in Python, we need to import the module *re*.

```
import re
```

Now we can start by trying to find some patterns in our strings.

```
text = '''

Mike is 20 years old and George is 29!

My grandma is even 104 years old!

'''



ages = re.findall(r'\d{1,3}', text)

print(ages)
```

In this example, we have a text with three ages in it. What we want to do is to filter these out and print them separately.

As you can see, we use the function *findall* in order to apply the regular expression onto our string. In this case, we are looking for numbers that are one to three digits long. Notice that we are using an *r* character before we write our expression. This indicates that the given string is a regular expression.

At the end, we print our result and get the following output:

['20', '29', '104']

MATCHING STRINGS

What we can also do is to check if a string matches a certain regular expression.

For example, we can apply our regular expression for mails here.

```
import re


text = "test@mail.com"


result = re.fullmatch(r"^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-
]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-
Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$", text)


if result != None:

    print("VALID!")

else:

    print("INVALID!")
```

We are not going to talk about the regular expression itself here. It is very long and complicated. But what we see here is a new function called *fullmatch*. This function returns the checked string if it matches the regular expression. In this case, this happens when the string has a valid mail format.

If the expression doesn't match the string, the function returns *None*. In our example above, we get the message *"VALID!"* since the expression is met. If we enter something like *"Hello World!"*, we will get the other message.

MANIPULATING STRINGS

Finally, we are going to take a look at manipulating strings with regular expressions. By using the function *sub* we can replace all the parts of a string that match the expression by something else.

```
import re
```

```
text = """

Mike is 20 years old and George is 29!

My grandma is even 104 years old!

"""


text = re.sub(r'\d{1,3}', "100", text)

print(text)
```

In this example, we replace all ages by *100*. This is what gets printed:

Mike is 100 years old and George is 100!
My grandma is even 100 years old!

These are the basic functions that we can operate with in Python when dealing with regular expressions. If you want to learn more about regular expressions just google and you will find a lot of guides. Play around with the identifiers and modifiers a little bit until you feel like you understand how they work.

WHAT’S NEXT?

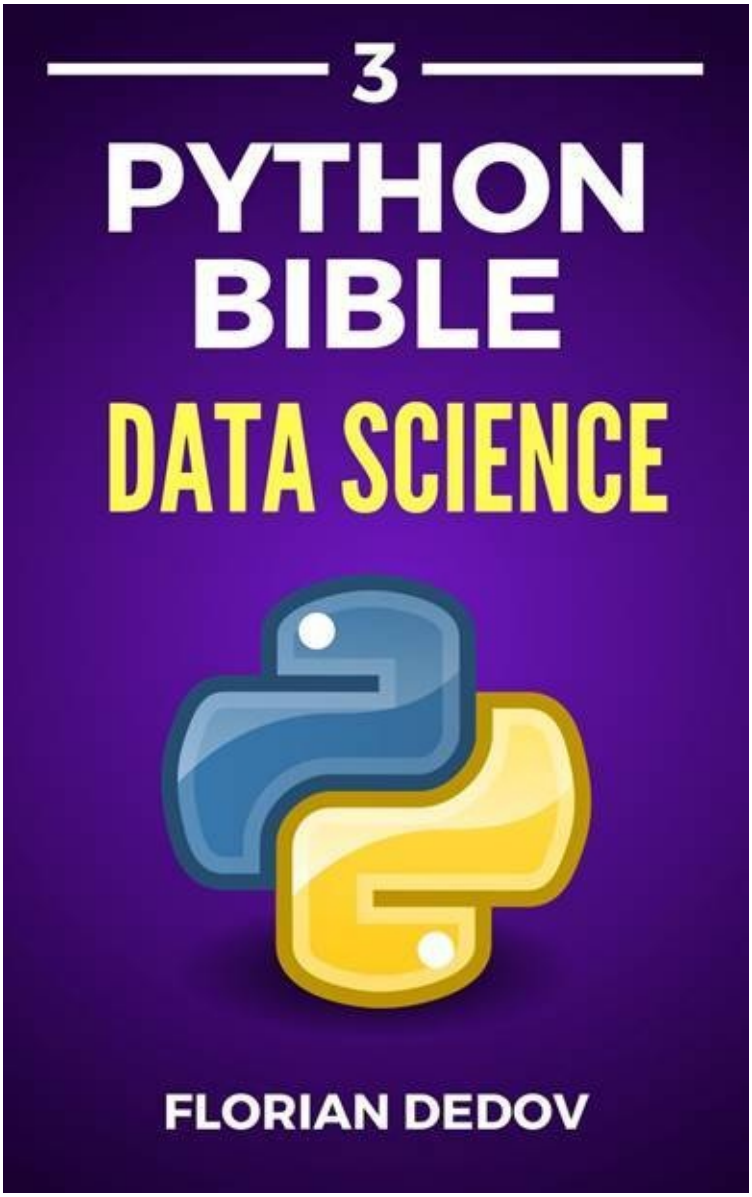
Now you have finished reading the second volume of this Python Bible series. This one was way more complex than the first one and it had a lot more content. Make sure that you practice what you’ve learned. If necessary, reread this book a couple of times and play around with the code samples. That will dramatically increase the value that you can get out of this book.

However, you are now definitely able to develop some advanced and professional Python applications. You can develop a chat, a port scanner, a string formatter and many more ideas. But this is still just the beginning. Even though you can now consider yourself to be an advanced Python programmer, there is much more to learn.

With the next volumes we are going to dive deep into the fields of machine learning, data science and finance. By having read the first two volumes you already have an excellent basis and I encourage you to continue your journey. I hope you could get some value out of this book and that it helped you to become a better programmer. So stay tuned and prepare for the next volume!

Last but not least, a little reminder. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!


3
PYTHON
BIBLE
DATA SCIENCE
FLORIAN DEDOV

THE
PYTHON BIBLE

VOLUME THREE

DATA SCIENCE

BY

FLORIAN DEDOV

TABLE OF CONTENT

INTRODUCTION

In our modern time, the amount of data grows exponentially. Over time, we learn to extract important information out of this data by analyzing it. The field which is primarily focusing on exactly that is called *data science*. We use data science to analyze share prices, the weather, demographics or to create powerful artificial intelligences. Every modern and big system has to deal with tremendous amounts of data that need to be managed and analyzed intelligently.

Therefore, it is more than reasonable to educate yourself in this area as much as possible. Otherwise you might get overrun by this fast-growing trend instead of being part of it.

THIS BOOK

If you have read the first two volumes of this series, you are already a decent Python programmer. You are able to develop complex scripts using advanced techniques like multithreading or network programming. A lot of these skills will be needed for this volume, since it’s going to be quite complex and detailed.

Now in this volume, we are going to start by talking about the major libraries or modules for data science in Python. We are taking a look at advanced arrays and lists, professional data visualization, statistical analysis and advanced data science with data frames. At the end, you will be able to prepare, analyze and visualize your own big data sets. This will lay the foundations for future volumes about machine learning and finance.

This book is again full of new and more complex information. There is a lot to learn here so stay tuned and code along while reading. This will help you to understand the material better and to practice implementing it. I wish you a lot of fun and success with your journey and this book!

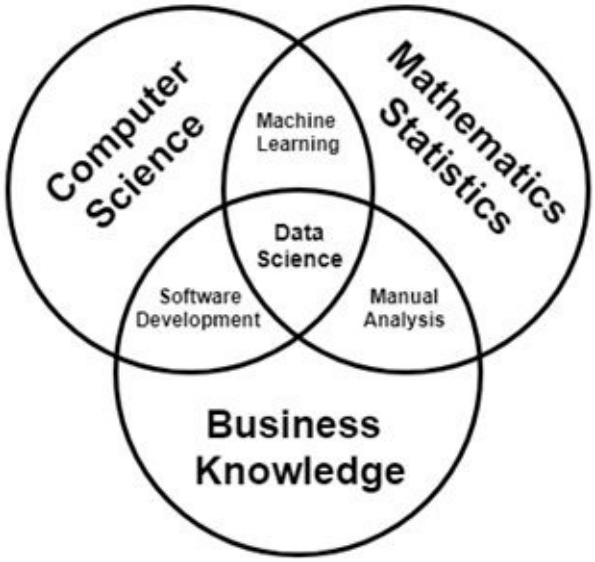
Just one little thing before we start. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you have learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

1 – WHAT IS DATA SCIENCE?

Now before we do anything at all, we need to first define what we are even talking about when using the term *data science*. What is data science?

When we are dealing with data science or data analysis, we are always trying to generate or extract knowledge from our data. For this, we use models, techniques and theories from the areas of mathematics, statistics, computer science, machine learning and many more.



The figure above illustrates pretty accurately what data science actually is. When you combine computer science and business knowledge, you get software development and create business applications. When you combine your business knowledge with mathematics and statistics, you can also analyze the data but you have to do it manually, since you are missing the computational component. When you only combine computer science and statistics, you get machine learning, which is very powerful, but without the necessary business knowledge, you won't get any significant information or conclusions. We need to combine all of these three areas, in order to end up with data science.

However, in this volume we are not going to focus too much on the mathematics and the statistics or the machine learning algorithms. This will be the topic of future volumes. In this book we are focusing on the structuring, visualization and analyzing of the data.

WHY PYTHON?

Now, you should already know why Python is a good choice and a good programming language to learn. But why should we use it for data science? Aren't there better alternatives?

And although I hate polarizing answers and generalization, I have to bluntly say **NO!** You have some alternatives like the programming language *R* or *MATLAB* but they are not as big, as powerful and as simple as Python.

One of the main reasons for Python's popularity in this area is the large amount of libraries and modules for data science but also machine learning and scientific computing. We already have professional open-source libraries for managing lists, linear algebra, data visualization, machine learning, neural networks and much more.

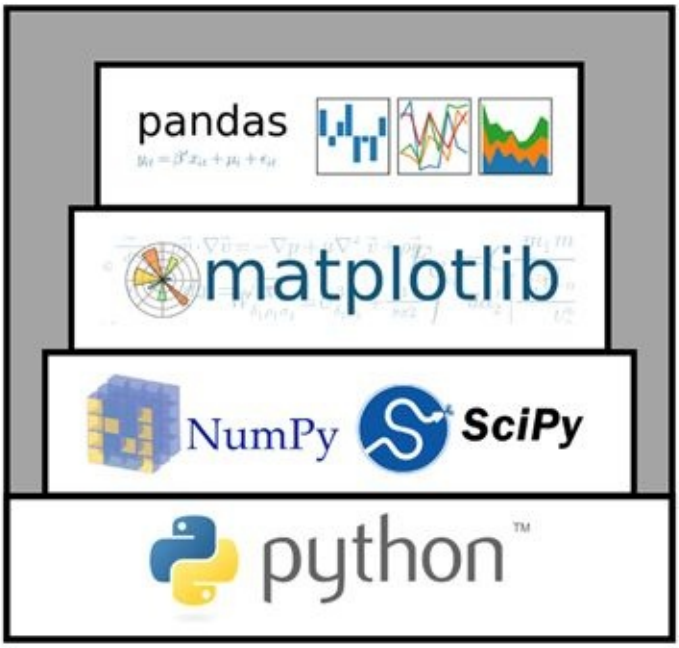
Also, alternatives like R or MATLAB are very specialized in one single area like statistics or mathematical programming. Python on the other hand is a general-purpose language. We use it to code network scripts, video games, professional web applications, artificial intelligences and much more. Self-driving cars use Python, professional modelling software uses Python and also Pinterest was developed with Django, which is a Python framework.

For these reasons, Python has become one of the most popular programming languages out there, especially for machine learning and data science.

2 – INSTALLING MODULES

So the last thing we need to talk about before we get into the coding itself is the modules or libraries that we are going to use.

The following figure illustrates the structure of the modules that are used for data science and scientific computing.



As you can see, we have four major modules here and they all build on core Python. Basically, this is the hierarchy of these modules. *NumPy* builds on Python, *Matplotlib* uses or builds on *NumPy* and *Pandas* builds on top of that. Of course there are other libraries that then build on top of *Pandas* as well. But for now, these are the modules that interest us.

Now in order to clear up the confusion, let’s look at the purpose and functionalities of the individual libraries.

NUMPY

The *NumPy* module allows us to efficiently work with vectors, matrices and multi-dimensional arrays. It is crucial for linear algebra and numerical analysis. Also, it offers some advanced things like Fourier transforms and random number generation. It basically replaces the primitive and inefficient Python *list* with very powerful *NumPy arrays*.

Another thing worth mentioning is that NumPy was built in the *C* programming language. This means that it is a lot faster and more efficient than other Python libraries.

SCIPY

SciPy is a module which we are actually not going to use in this book. Nevertheless, it is worth mentioning because it is a very powerful library for scientific computing (maybe there will be a future volume about this).

However, SciPy can be seen as the application of NumPy to real problems. NumPy is basically just managing the arrays and lists. It is responsible for the operations like indexing, sorting, slicing, reshaping and so on. Now, SciPy actually uses NumPy to offer more abstract classes and functions that solve scientific problems. It gets deeper into the mathematics and adds substantial capabilities to NumPy.

MATPLOTLIB

On top of that, we have *Matplotlib*. This library is responsible for plotting graphs and visualizing our data. It offers numerous types of plotting, styles and graphs.

Visualization is a key step in data science. When we see our data in form of a graph, we can extract information and spot relations much easier. With Matplotlib we can do this professionally and very easy.

PANDAS

Last but not least, we have *Pandas*. This is the most high-level of our libraries and it builds on top of them. It offers us a powerful data structure named *data frame*. You can imagine it to be a bit like a mix of an Excel table and an SQL database table.

This library allows us to efficiently work with our huge amounts of interrelated data. We can merge, reshape, filter and query our data. We can iterate over it and we can read and write into files like CSV, XLSX and more. Also, it is very powerful when we work with databases, due to the similar structure of the tables.

Pandas is highly compatible with NumPy and Matplotlib, since it builds on them. We can easily convert data from one format to the other.

INSTALLING MODULES WITH PIP

Since all these modules don’t belong to core Python, we will need to install them externally. For this, we are going to use *pip*. This is a recursive name and stands for *pip installs packages*.

In order to use pip, we just need to open up our terminal or command line. On windows this is CMD and on Mac and Linux it is the terminal. We then just use the following syntax, in order to install the individual packages.

pip install <package-name>

So what we need to do is to execute the following commands:

pip install numpy

pip install scipy (optional)

pip install matplotlib

pip install pandas

3 – NUMPY ARRAYS

We can’t do a lot of data science with NumPy alone. But it provides the basis for all the high-level libraries or modules for data science. It is essential for the efficient management of arrays and linear algebra.

In order to use NumPy, we of course have to import the respective module first.

```
import numpy as np
```

As you can see, we are also defining an *alias* here, so that we can address NumPy by just writing *np*.

CREATING ARRAYS

To create a NumPy array, we just use the respective function *array* and pass a list to it.

```
a = np.array([10, 20, 30])

b = np.array([1, 77, 2, 3])
```

Now we can access the values in the same way as we would do it with a list.

```
print(a[0])

print(b[2])
```

MULTI-DIMENSIONAL ARRAYS

The arrays we created are one-dimensional arrays. With NumPy, we can create large multi-dimensional arrays that have the same structure as a matrix.

```
a = np.array([

    [10, 20, 30],

    [40, 50, 60]

])



print(a)
```

Here, we pass two lists within a list as a parameter. This creates a 2x3 matrix. When we print the array, we get the following result:

```
[[10 20 30]
 [40 50 60]]
```

Since we now have two dimensions, we also need to address two indices, in order to access a specific element.

```
print(a[1][2])
```

In this case, we are addressing the second row (index one) and the third element or column (index two). Therefore, our result is *60*.

We can extend this principle as much as we want. For example, let's create a much bigger array.

```
a = np.array([
    [
        [10,20,30,40], [8,8,2,1], [1,1,1,2]
    ],
    [
        [9, 9, 2, 39], [1,2,3,3], [0,0,3,2]
    ],
    [
        [12,33,22,1], [22,1,22,2], [0,2,3,1]
    ]
], dtype=float)
```

Here we have a 3x3x4 matrix and slowly but surely it becomes a bit irritating and we can't really grasp the structure of the array. This is especially the case when we get into four or more dimensions, since we only perceive three dimensions in everyday life.

You can imagine this three-dimensional array as a cube. We have three rows, four columns and three pages or layers. Such visualizations fail in higher dimensions.

Another thing that is worth mentioning is the parameter *dtype*. It stands for data

type and allows us to specify which data type our values have. In this case we specified *float* and therefore our values will be stored as floating point numbers with the respective notation.

FILLING ARRAYS

Instead of manually filling our arrays with values, we can also use pre-defined functions in certain cases. The only thing we need to specify is the desired function and the shape of the array.

FULL FUNCTION

By using the *full* function for example, we fill an array of a certain shape with the same number. In this case we create a 3x5x4 matrix, which is filled with sevens.

```
a = np.full((3,5,4), 7)
```

```
print(a)
```

When we print it, we get the following output:

```
[[[7 7 7 7]
  [7 7 7 7]
  [7 7 7 7]]

 [[7 7 7 7]
  [7 7 7 7]
  [7 7 7 7]]]
```

ZEROS AND ONES

For the cases that we want arrays full of zeros or ones, we even have specific functions.

```
a = np.zeros((3,3))
```

```
b = np.ones((2,3,4,2))
```

Here we create a 3x3 array full of zeros and a four-dimensional array full of ones.

EMPTY AND RANDOM

Other options would be to create an empty array or one that is filled with random numbers. For this, we use the respective functions once again.

```
a = np.empty((4,4))

b = np.random.random((2,3))
```

The function *empty* creates an array without initializing the values at all. This makes it a little bit faster but also more dangerous to use, since the user needs to manually initialize all the values.

When using the *random* function, make sure that you are referring to the module *np.random*. You need to write it two times because otherwise you are calling the library.

RANGES

Instead of just filling arrays with the same values, we can fill create sequences of values by specifying the boundaries. For this, we can use two different functions, namely *arange and linspace*.

```
a = np.arange(10, 50, 5)
```

The function *arange* creates a list with values that range from the minimum to the maximum. The step-size has to be specified in the parameters.

```
[10 15 20 25 30 35 40 45]
```

In this example, we create have count from 10 to 45 by always adding 5. The result can be seen above.

By using *linspace* we also create a list from a minimum value to a maximum value. But instead of specifying the step-size, we specify the amount of values that we want to have in our list. They will all be spread evenly and have the same distance to their neighbors.

```
b = np.linspace(0, 100, 11)
```

Here, we want to create a list that ranges from 0 to 100 and contains 11 elements. This fits smoothly with a difference of 10 between all numbers. So the result

looks like this:

```
[ 0.  10.  20.  30.  40.  50.  60.  70.  80.  90. 100.]
```

Of course, if we choose different parameters, the numbers don’t be that “beautiful”.

NOT A NUMBER (NAN)

There is a special value in NumPy that represents values that are not numbers. It is called *NaN* and stands for *Not a Number*. We basically just use it as a placeholder for empty spaces. It can be seen as a value that indicates that something is missing at that place.

When importing big data packets into our application, there will sometimes be missing data. Instead of just setting these values to zero or something else, we can set them to NaN and then filter these data sets out.

ATTRIBUTES OF ARRAYS

NumPy arrays have certain attributes that we can access and that provide information about the structure of it.

NUMPY ARRAY ATTRIBUTES	
ATTRIBUTE	**DESCRIPTION**
a.shape	Returns the shape of the array e.g. (3,3) or (3,4,7)
a.ndim	Returns how many dimensions our array has
a.size	Returns the amount of elements an array has
a.dtype	Returns the data type of the values in the array

MATHEMATICAL OPERATIONS

Now that we know how to create an array and what attributes it has, let's take a look at how to work with arrays. For this, we will start out with basic mathematical operations.

ARITHMETIC OPERATIONS

```
a = np.array([

    [1,4,2],

    [8,8,2]

])



print(a + 2)

print(a - 2)

print(a * 2)

print(a / 2)
```

When we perform basic arithmetic operations like addition, subtraction, multiplication and division to an array and a scalar, we apply the operation on every single element in the array. Let's take a look at the results:

```
[[ 3  6  4]
 [10 10  4]]

[[-1  2  0]
 [ 6  6  0]]

[[ 2  8  4]
 [16 16  4]]

[[0.5 2.  1. ]
 [4.  4.  1. ]]
```

As you can see, when we multiply the array by two, we multiply every single value in it by two. This is also the case for addition, subtraction and division. But

what happens when we apply these operations on two arrays?

```
a = np.array([
    [1,4,2],
    [8,8,2]
])

b = np.array([
    [1,2,3]
])

c = np.array([
    [1],
    [2]
])

d = np.array([
    [1,2,3],
    [3,2,1]
])
```

In order to apply these operations on two arrays, we need to take care of the shapes. They don't have to be the same, but there has to be a reasonable way of performing the operations. We then again apply the operations on each element of the array.

For example, look at *a* and *b*. They have different shapes but when we add these two, they share at least the amount of columns.

```
print(a+b)
```
```
[[ 2  6  5]
 [ 9 10  5]]
```

Since they match the columns, we can just say that we add the individual columns, even if the amount of rows differs.

The same can also be done with *a* and *c* where the rows match and the columns differ.

```
print(a+c)
```
```
[[ 2  5  3]
 [10 10  4]]
```

And of course it also works, when the shapes match exactly. The only problem is when the shapes differ too much and there is no reasonable way of performing the operations. In these cases, we get *ValueErrors*.

MATHEMATICAL FUNCTIONS

Another thing that the NumPy module offers us is mathematical functions that we can apply to each value in an array.

NUMPY MATHEMATICAL FUNCTIONS	
FUNCTION	**DESCRIPTION**
np.exp(a)	Takes *e* to the power of each value
np.sin(a)	Returns the sine of each value
np.cos(a)	Returns the cosine of each value
np.tan(a)	Returns the tangent of each value
np.log(a)	Returns the logarithm of each value
np.sqrt(a)	Returns the square root of each value

AGGREGATE FUNCTIONS

Now we are getting into the statistics. NumPy offers us some so-called *aggregate functions* that we can use in order to get a key statistic from all of our values.

NUMPY AGGREGATE FUNCTIONS	
FUNCTION	**DESCRIPTION**
a.sum()	Returns the sum of all values in the array
a.min()	Returns the lowest value of the array
a.max()	Returns the highest value of the array
a.mean()	Returns the arithmetic mean of all values in the array
np.median(a)	Returns the median value of the array
np.std(a)	Returns the standard deviation of the values in the array

MANIPULATING ARRAYS

NumPy offers us numerous ways in which we can manipulate the data of our arrays. Here, we are going to take a quick look at the most important functions and categories of functions.

If you just want to change a single value however, you can just use the basic indexing of lists.

```
a = np.array([
    [4, 2, 9],
    [8, 3, 2]
])
```

```
a[1][2] = 7
```

SHAPE MANIPULATION FUNCTIONS

One of the most important and helpful types of functions are the *shape manipulating functions*. These allow us to restructure our arrays without changing their values.

SHAPE MANIPULATION FUNCTIONS	
FUNCTION	**DESCRIPTION**
a.reshape(x,y)	Returns an array with the same values structured in a different shape
a.flatten()	Returns a flattened one-dimensional copy of the array
a.ravel()	Does the same as *flatten* but works with the actual array instead of a copy
a.transpose()	Returns an array with the same values but swapped dimensions
	Returns an array with the same

a.swapaxes()	values but two swapped axes
a.flat	Not a function but an iterator for the flattened version of the array

There is one more element that is related to shape but it's not a function. It is called *flat* and it is an iterator for the flattened one-dimensional version of the array. *Flat* is not callable but we can iterate over it with *for* loops or index it.

```
for x in a.flat:
    print(x)


print(a.flat[5])
```

JOINING FUNCTIONS

We use *joining functions* when we combine multiple arrays into one new array.

JOINING FUNCTIONS	
FUNCTION	**DESCRIPTION**
np.concatenate(a,b)	Joins multiple arrays along an existing axis
np.stack(a,b)	Joins multiple arrays along a new axis
np.hstack(a,b)	Stacks the arrays horizontally (column-wise)
np.vstack(a,b)	Stacks the arrays vertically (row-wise)

In the following, you can see the difference between *concatenate* and *stack*:

```
a = np.array([10, 20, 30])

b = np.array([20, 20, 10])
```

```
print(np.concatenate((a,b)))

print(np.stack((a,b)))
```
[10 20 30 20 20 10]
[[10 20 30]
 [20 20 10]]

What *concatenate* does is, it joins the arrays together by just appending one onto the other. *Stack* on the other hand, creates an additional axis that separates the two initial arrays.

SPLITTING FUNCTIONS

We can not only join and combine arrays but also split them again. This is done by using *splitting functions* that split arrays into multiple sub-arrays.

SPLITTING FUNCTIONS	
FUNCTION	**DESCRIPTION**
np.split(a, x)	Splits one array into multiple arrays
np.hsplit(a, x)	Splits one array into multiple arrays horizontally (column-wise)
np.vsplit(a, x)	Splits one array into multiple arrays vertically (row-wise)

When splitting a list with the *split* function, we need to specify into how many sections we want to split our array.

```
a = np.array([
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90],
    [100, 110, 120]
])
```

```
print(np.split(a, 2))

print(np.split(a, 4))
```

This array can be split into either two or four equally sized arrays on the default axis. The two possibilities are the following:

1: [[10, 20, 30],[40, 50, 60]]
2: [[70, 80, 90],[100, 110, 120]]

OR

1: [[10, 20, 30]]
2: [[40, 50, 60]]
3: [[70, 80, 90]]
4: [[100, 110, 120]]

ADDING AND REMOVING

The last manipulating functions that we are going to look at are the ones which allow us to *add* and to *remove* items.

ADDING AND REMOVING FUNCTIONS	
FUNCTION	**DESCRIPTION**
np.resize(a, (x,y))	Returns a resized version of the array and fills empty spaces by repeating copies of a
np.append(a, […])	Appends values at the end of the array
np.insert(a, x, …)	Insert a value at the index x of the array
np.delete(a, x, y)	Delete axes of the array

LOADING AND SAVING ARRAYS

Now last but not least, we are going to talk about loading and saving NumPy arrays. For this, we can use the integrated NumPy format or CSV-files.

NUMPY FORMAT

Basically, we are just serializing the object so that we can use it later. This is done by using the *save* function.

```
a = np.array([
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90],
    [100, 110, 120]
])
```

```
np.save('myarray.npy', a)
```

Notice that you don't have to use the file ending *npy*. In this example, we just use it for clarity. You can pick whatever you want.

Now, in order to load the array into our script again, we will need the *load* function.

```
a = np.load('myarray.npy')
print(a)
```

CSV FORMAT

As I already mentioned, we can also save our NumPy arrays into CSV files, which are just comma-separated text files. For this, we use the function *savetxt*.

```
np.savetxt('myarray.csv', a)
```

Our array is now stored in a CSV-file which is very useful, because it can then also be read by other applications and scripts.

In order to read this CSV-file back into our script, we use the function *loadtxt*.

```
a = np.loadtxt('myarray.csv')

print(a)
```

If we want to read in a CSV-file that uses another separator than the default one, we can specify a certain delimiter.

```
a = np.loadtxt('myarray.csv', delimiter=';')

print(a)
```

Now it uses semi-colons as separator when reading the file. The same can also be done with the saving or writing function.

4 – MATPLOTLIB DIAGRAMS

We have already mentioned that visualizing our data is crucial for data science. It gives us an overview and helps us to analyze data and make conclusions. Therefore, we will talk quite a lot about *Matplotlib*, the library which we use for plotting and visualizing.

PLOTTING MATHEMATICAL FUNCTIONS

Now, let's start out by drawing some mathematical functions first. In order to do so, we need to import the *matplotlib.pyplot* module and also NumPy.

```
import numpy as np
import matplotlib.pyplot as plt
```

Notice that we are also using an alias for *pyplot* here. In this case, it is *plt*.

In order to plot a function, we need the x-values or the input and the y-values or the output. So let us generate our x-values first.

```
x_values = np.linspace(0, 20, 100)
```

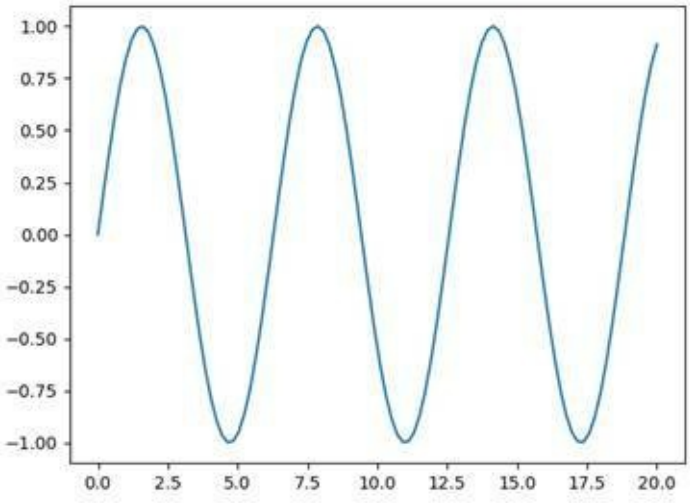
We are doing this by using the already known *linspace* function. Here we create an array with 100 values between 0 and 20. To now get our y-values, we just need to apply the respective function on our x-values. For this example, we are going with the sine function.

```
y_values = np.sin(x_values)
```

Remember that the function gets applied to every single item of the input array. So in this case, we have an array with the sine value of every element of the x-values array. We just need to plot them now.

```
plt.plot(x_values, y_values)
plt.show()
```

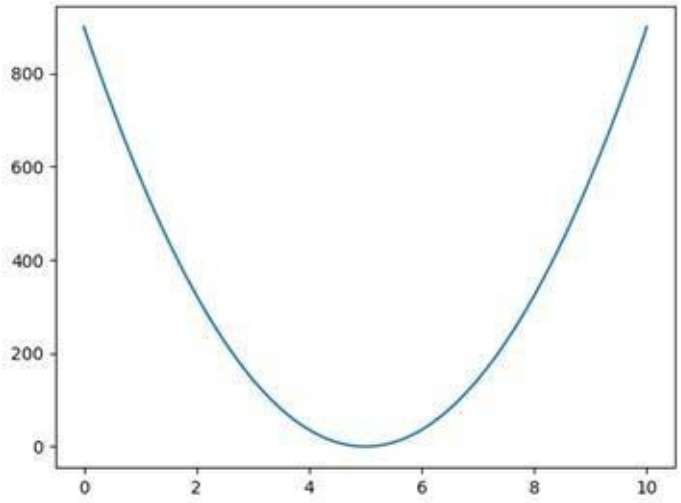
We do this by using the function *plot* and passing our x-values and y-values. At the end we call the *show* function, to display our plot.



That was very simple. Now, we can go ahead and define our own function that we want to plot.

```
x = np.linspace(0, 10, 100)
y = (6 * x - 30) ** 2

plt.plot(x, y)
plt.show()
```

The result looks like this:



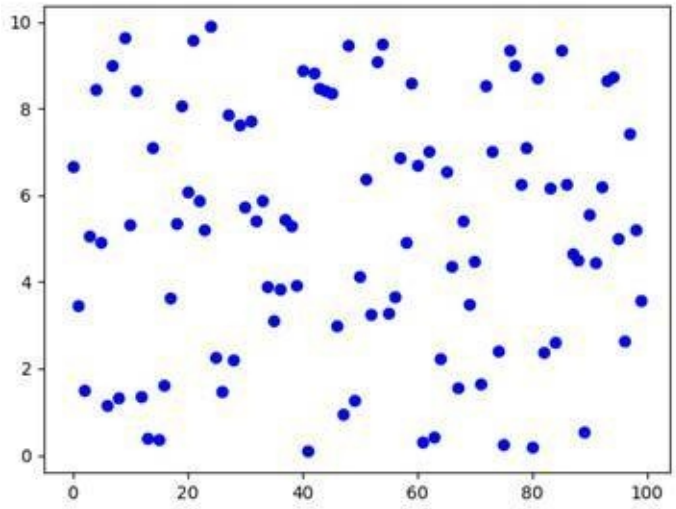
This is just the function $(6x – 30)^2$ plotted with Matplotlib.

VISUALIZING VALUES

What we can also do, instead of plotting functions, is just visualizing values in form of single dots for example.

```
numbers = 10 * np.random.random(100)

plt.plot(numbers, 'bo')
plt.show()
```

Here we are just generating 100 random numbers from 0 to 10. We then plot these numbers as blue dots. This is defined by the second parameter *'bo'*, where the first letter indicates the color (blue) and the second one the shape (dots). Here you can see what this looks like:

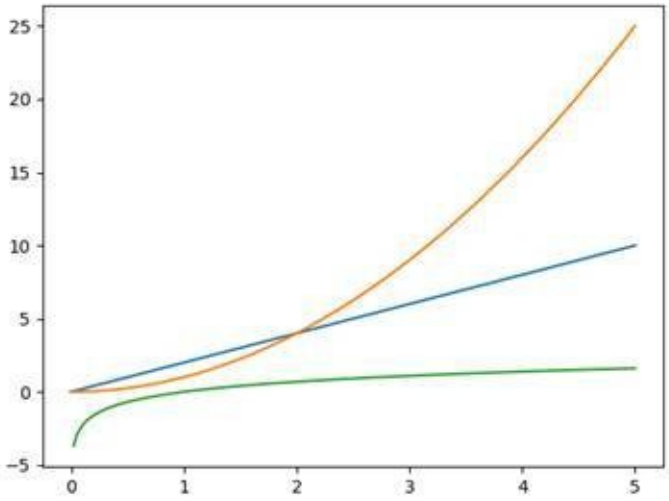


MULTIPLE GRAPHS

Our plots are not limited to only one single graph. We can plot multiple functions in different color and shape.

```
x = np.linspace(0,5,200)
y1 = 2 * x
y2 = x ** 2
y3 = np.log(x)

plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)
plt.show()
```

In this example, we first generate 200 x-values from 0 to 5. Then we define three different functions *y1*, *y2* and *y3*. We plot all these and view the plotting window. This is what it looks like:

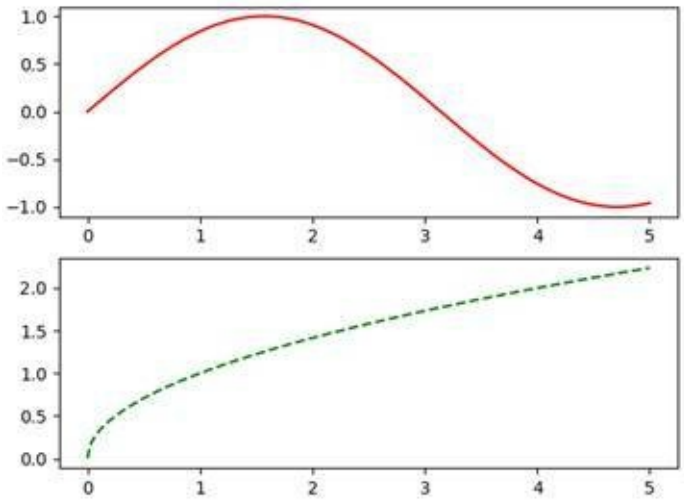


SUBPLOTS

Now, sometimes we want to draw multiple graphs but we don't want them in the same plot necessarily. For this reason, we have so-called *subplots*. These are plots that are shown in the same window but independently from each other.

```
x = np.linspace(0,5,200)
y1 = np.sin(x)
y2 = np.sqrt(x)

plt.subplot(211)
plt.plot(x, y1, 'r-')

plt.subplot(212)
plt.plot(x, y2, 'g--')

plt.show()
```

By using the function *subplot* we state that everything we plot now belongs to this specific subplot. The parameter we pass defines the grid of our window. The first digit indicates the number of rows, the second the number of columns and the last one the index of the subplot. So in this case, we have two rows and one column. Index one means that the respective subplot will be at the top.



As you can see, we have two subplots in one window and both have a different color and shape. Notice that the ratios between the x-axis and the y-axis differ in the two plots.

MULTIPLE PLOTTING WINDOWS

Instead of plotting into subplots, we can also go ahead and plot our graphs into multiple windows. In Matplotlib we call these *figures*.

```
plt.figure(1)
plt.plot(x, y1, 'r-')

plt.figure(2)
plt.plot(x, y2, 'g--')
```

By doing this, we can show two windows with their graphs at the same time. Also, we can use subplots within figures.

PLOTTING STYLES

Matplotlib offers us many different plotting styles to choose from. If you are interested in how they look when they are applied, you can see an overview by going to the following website (I used a URL shortener to make it more readable):

https://bit.ly/2JfhJ4o

In order to use a style, we need to import the *style* module of Matplotlib and then call the function *use*.

```
from matplotlib import style

style.use('ggplot')
```

By using the *from … import …* notation we don't need to specify the parent module *matplotlib*. Here we apply the style of *ggplot*. This adds a grid and some other design changes to our plots. For more information, check out the link above.

LABELING DIAGRAMS

In order to make our graphs understandable, we need to label them properly. We should label the axes, we should give our windows titles and in some cases we should also add a legend.

SETTING TITLES

Let’s start out by setting the titles of our graphs and windows.

```
x = np.linspace(0,50,100)
y = np.sin(x)

plt.title("Sine Function")
plt.suptitle("Data Science")
plt.grid(True)
plt.plot(x,y)

plt.show()
```

In this example, we used the two functions *title* and *suptitle*. The first function adds a simple title to our plot and the second one adds an additional centered title above it. Also, we used the *grid* function, to turn on the grid of our plot.

If you want to change the title of the window, you can use the *figure* function that we already know.

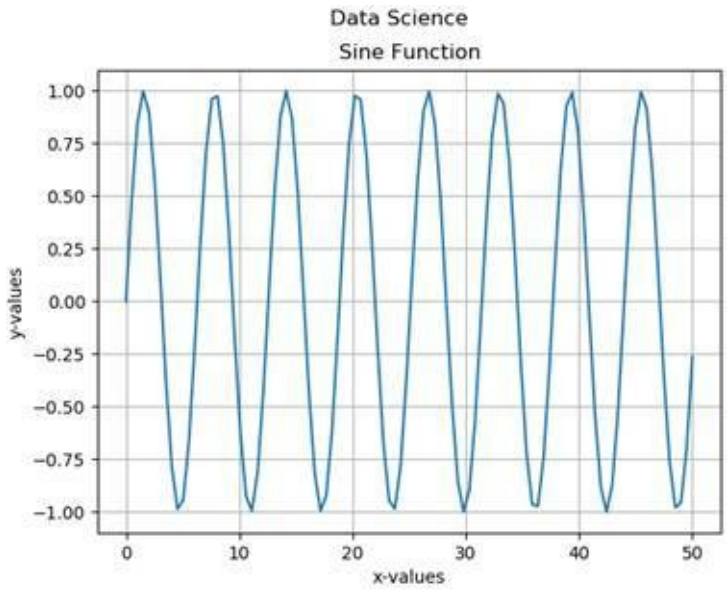
```
plt.figure("MyFigure")
```

LABELING AXES

As a next step, we are going to label our axes. For this, we use the two functions *xlabel* and *ylabel*.

```
plt.xlabel("x-values")
plt.ylabel("y-values")
```

You can choose whatever labels you like. When we combine all these pieces of code, we end up with a graph like this:



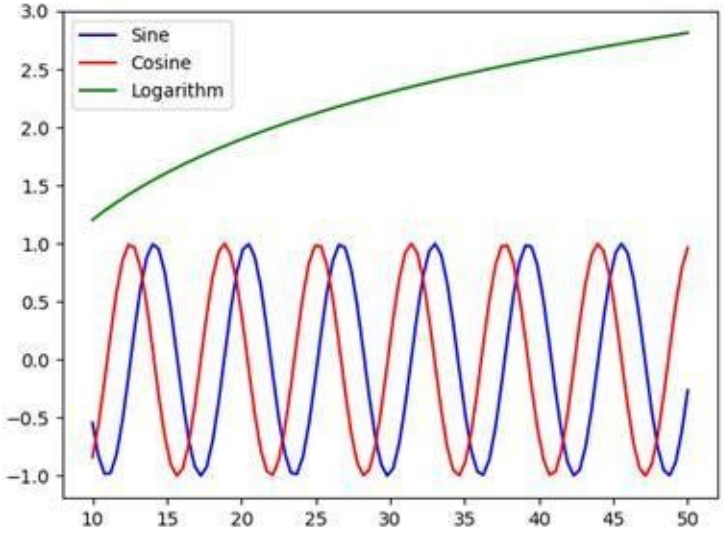
In this case, the labels aren't really necessary because it is obvious what we see here. But sometimes we want to describe what our values actually mean and what the plot is about.

LEGENDS

Sometimes we will have multiple graphs and objects in a plot. We then use legends to label these individual elements, in order to make everything more readable.

```
x = np.linspace(10,50,100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.log(x/3)

plt.plot(x,y1,'b-',label="Sine")
plt.plot(x,y2,'r-',label="Cosine")
plt.plot(x,y3,'g-',label="Logarithm")

plt.legend(loc='upper left')

plt.show()
```

Here we have three functions, *sine*, *cosine* and a *logarithmic* function. We draw all graphs into one plot and add a label to them. In order to make these labels visible, we then use the function *legend* and specify a location for it. Here we chose the *upper left*. Our result looks like this:



As you can see, the legend makes our plot way more readable and it also looks more professional.

SAVING DIAGRAMS

So now that we know quite a lot about plotting and graphing, let’s take a look at how to save our diagrams.

```
plt.savefig("functions.png")
```

Actually, this is quite simple. We just plot whatever we want to plot and then use the function *savefig* to save our figure into an image file.

5 – MATPLOTLIB PLOT TYPES

In the last chapter, we mainly plotted functions and a couple of values. But Matplotlib offers a huge arsenal of different plot types. Here we are going to take a look at these.

HISTOGRAMS

Let's start out with some statistics here. So-called *histograms* represent the distribution of numerical values. For example, we could graph the distribution of heights amongst students in a class.

```
mu, sigma = 172, 4

x = mu + sigma * np.random.randn(10000)
```

We start by defining a mean value *mu* (average height) and a standard deviation *sigma*. To create our x-values, we use our *mu* and *sigma* combined with 10000 randomly generated values. Notice that we are using the *randn* function here. This function generates values for a *standard normal distribution*, which means that we will get a bell curve of values.

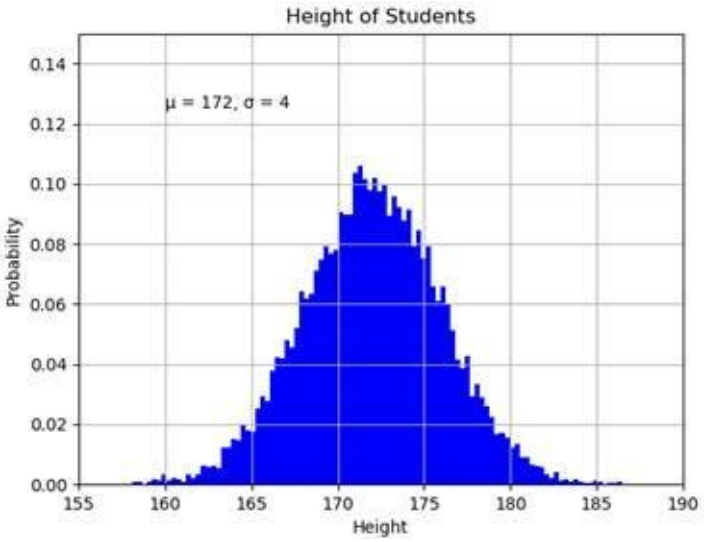
```
plt.hist(x, 100, density=True, facecolor="blue")
```

Then we use the *hist* function, in order to plot our histogram. The second parameter states how many values we want to plot. Also, we want our values to be normed. So we set the parameter *density* to *True*. This means that our y-values will sum up to one and we can view them as percentages. Last but not least, we set the color to blue.

Now, when we show this plot, we will realize that it is a bit confusing. So we are going to add some labeling here.

```
plt.xlabel("Height")

plt.ylabel("Probability")

plt.title("Height of Students")

plt.text(160, 0.125,"μ = 172, σ = 4")

plt.axis([155,190,0,0.15])

plt.grid(True)
```

First we label the two axes. The x-values represent the height of the students, whereas the y-values represent the probability that a randomly picked student has the respective height. Besides the title, we also add some text to our graph. We place it at the x-value 160 and the y-value of 0.125. The text just states the values for μ (mu) and σ (sigma). Last but not least, we set the ranges for the two axes. Our x-values range from 155 to 190 and our y-values from 0 to 0.15. Also, the grid is turned on. This is what our graph looks like at the end:



We can see the Gaussian bell curve which is typical for the standard normal distribution.

BAR CHART

For visualizing certain statistics, *bar charts* are oftentimes very useful, especially when it comes to categories. In our case, we are going to plot the skill levels of three different people in the IT realm.

```
bob = (90, 67, 87, 76)

charles = (80, 80, 47, 66)

daniel = (40, 95, 76, 89)



skills = ("Python", "Java", "Networking", "Machine
Learning")
```

Here we have the three persons *Bob*, *Charles* and *Daniel*. They are represented by tuples with four values that indicate their skill levels in Python programming, Java programming, networking and machine learning.

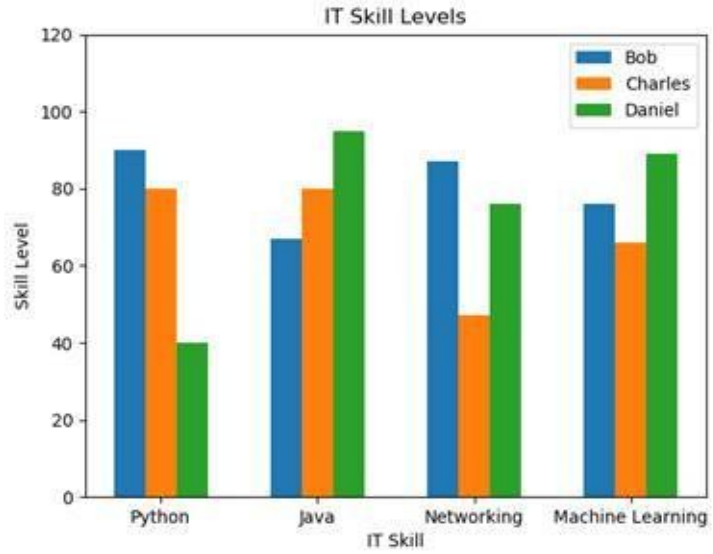
```
width = 0.2

index = np.arange(4)

plt.bar(index, bob,

        width=width, label="Bob")

plt.bar(index + width, charles,

        width=width, label="Charles")

plt.bar(index + width * 2, daniel,

        width=width, label="Daniel")
```

We then use the *bar* function to plot our bar chart. For this, we define an array with the indices one to four and a bar width of 0.2. For each person we plot the

four respective values and label them.

```
plt.xticks(index + width, skills)

plt.ylim(0,120)

plt.title("IT Skill Levels")

plt.ylabel("Skill Level")

plt.xlabel("IT Skill")

plt.legend()
```

Then we label the x-ticks with the method *xticks* and set the limit of the y-axis to 120 to free up some space for our legend. After that we set a title and label the axes. The result looks like this:



We can now see who is the most skilled in each category. Of course we could also change the graph so that we have the persons on the x-axis with the skill-colors in the legend.

PIE CHART

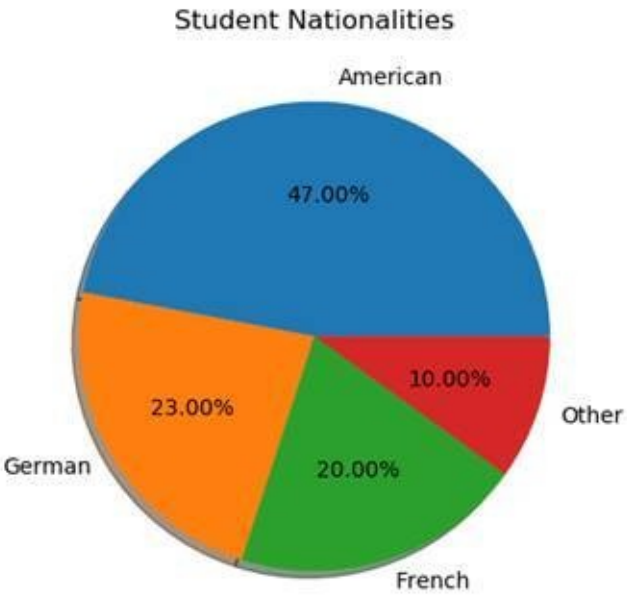
Pie charts are used to display proportions of numbers. For example, we could graph how many percent of the students have which nationality.

```
labels = ('American', 'German', 'French', 'Other')

values = (47, 23, 20, 10)
```

We have one tuple with our four nationalities. They will be our labels. And we also have one tuple with the percentages.

```
plt.pie(values,labels=labels,

        autopct="%.2f%%", shadow=True)

plt.title("Student Nationalities")


plt.show()
```

Now we just need to use the *pie* function, to draw our chart. We pass our values and our labels. Then we set the *autopct* parameter to our desired percentage format. Also, we turn on the *shadow* of the chart and set a title. And this is what we end up with:



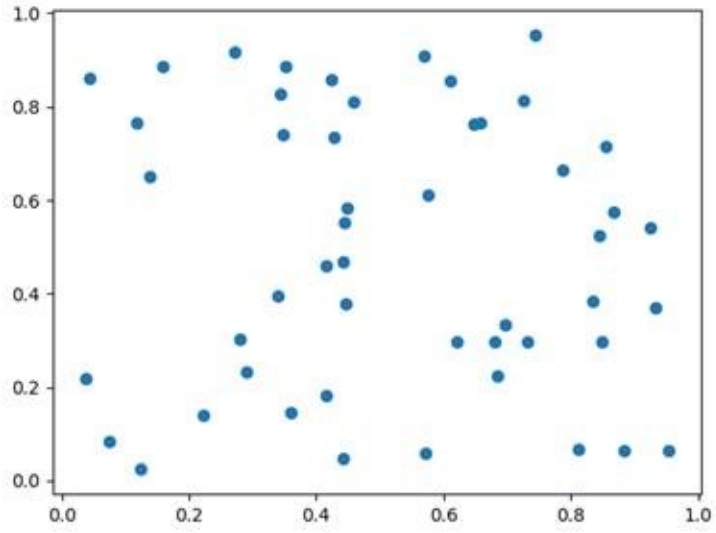
As you can see, this chart is perfect for visualizing percentages.

SCATTER PLOTS

So-called *scatter plots* are used to represent two-dimensional data using dots.

```
x = np.random.rand(50)

y = np.random.rand(50)


plt.scatter(x,y)


plt.show()
```

Here we just generate 50 random x-values and 50 random y-values. By using the *scatter* function, we can then plot them.

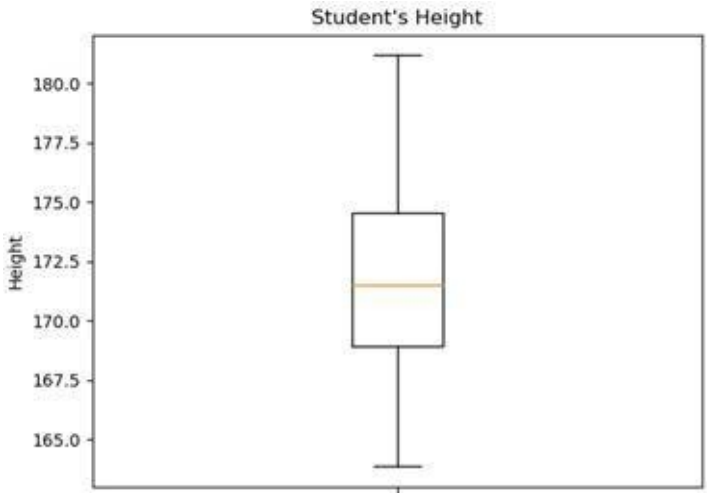


BOXPLOT

Boxplot diagrams are used, in order to split data into *quartiles*. We do that to get information about the distribution of our values. The question we want to answer is: How widely spread is the data in each of the quartiles.

```
mu, sigma = 172, 4

values = np.random.normal(mu,sigma,200)


plt.boxplot(values)

plt.title("Student's Height")

plt.ylabel("Height")

plt.show()
```

In this example, we again create a normal distribution of the heights of our students. Our mean value is 172, our standard deviation 4 and we generate 200 values. Then we plot our boxplot diagram.



Here we see the result. Notice that a boxplot doesn't give information about the frequency of the individual values. It only gives information about the spread of the values in the individual quartiles. Every quartile has 25% of the values but

some have a very small spread whereas others have quite a large one.

3D PLOTS

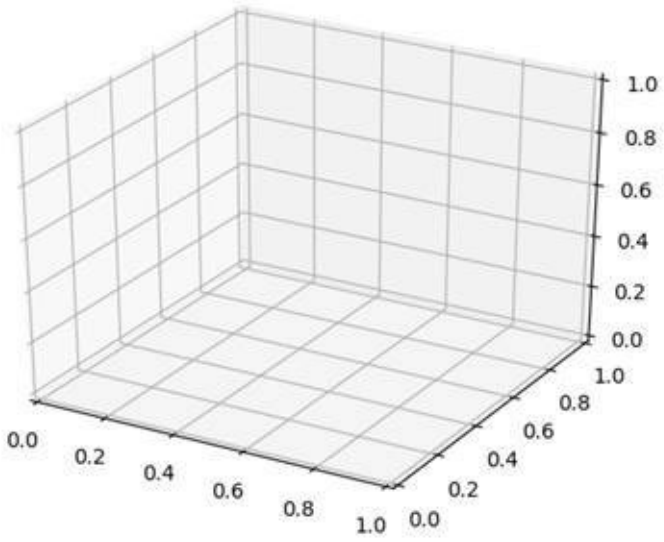
Now last but not least, let's take a look at 3D-plotting. For this, we will need to import another plotting module. It is called *mpl_toolkits* and it is part of the Matplotlib stack.

```
from mpl_toolkits import mplot3d
```

Specifically, we import the module *mplot3d* from this library. Then, we can use *3d* as a parameter when defining our axes.

```
ax = plt.axes(projection='3d')
plt.show()
```

We can only use this parameter, when *mplot3d* is imported. Now, our plot looks like this:

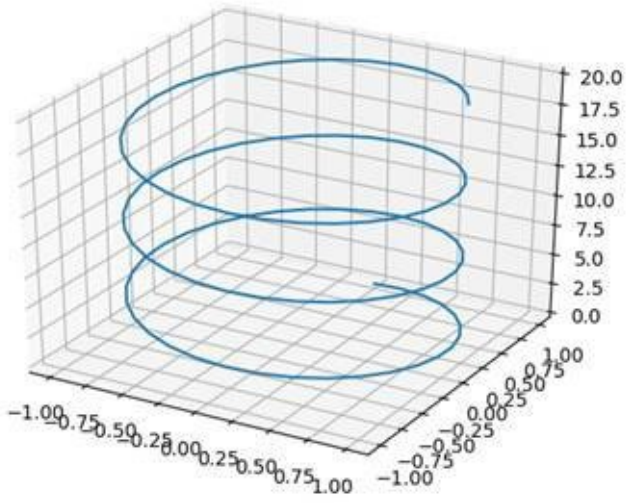


Since we are now plotting in three dimensions, we will also need to define three axes.

```
z = np.linspace(0, 20, 100)

x = np.sin(z)

y = np.cos(z)
```

```
ax = plt.axes(projection='3d')

ax.plot3D(x,y,z)

plt.show()
```

In this case, we are taking the z-axis as the input. The z-axis is the one which goes upwards. We define the x-axis and the y-axis to be a sine and cosine function. Then, we use the function *plot3D* to plot our function. We end up with this:



SURFACE PLOTS

Now in order to plot a function with a surface, we need to calculate every point on it. This is impossible, which is why we are just going to calculate enough to estimate the graph. In this case, x and y will be the input and the z-function will be the 3D-result which is composed of them.

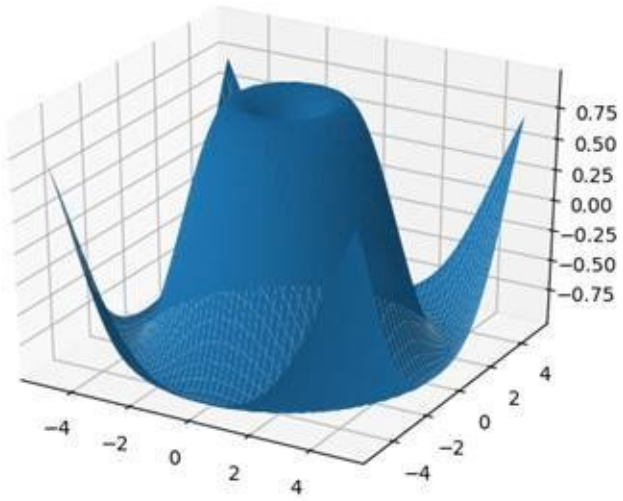
```
ax = plt.axes(projection='3d')


def z_function(x, y):

    return np.sin(np.sqrt(x ** 2 + y ** 2))
```

```
x = np.linspace(-5, 5, 50)

y = np.linspace(-5, 5, 50)
```

We start by defining a *z_function* which is a combination of sine, square root and squaring the input. Our inputs are just 50 numbers from -5 to 5.

```
X, Y = np.meshgrid(x,y)

Z = z_function(X,Y)



ax.plot_surface(X,Y,Z)

plt.show()
```

Then we define new variables for x and y (we are using capitals this time). What we do is converting the x- and y-vectors into matrices using the *meshgrid* function. Finally, we use the *z_function* to calculate our z-values and then we plot our surface by using the method *plot_surface*. This is the result:



Play around with these charts and plots until you really understand them. Visualizing functions and data is very important in data science.

6 – PANDAS DATA ANALYSIS

Pandas is probably the most powerful libraries of this book. It provides high-performance tools for data manipulation and analysis. Furthermore, it is very effective at converting data formats and querying data out of databases. The two main data structures of Pandas are the *series* and the *data frame*. To work with Pandas, we need to import the module.

```
import pandas as pd
```

PANDAS SERIES

A series in Pandas is a one-dimensional array which is labeled. You can imagine it to be the data science equivalent of an ordinary Python dictionary.

```
series = pd.Series([10, 20, 30, 40],
                   ['A', 'B', 'C', 'D'])
```

In order to create a series, we use the constructor of the *Series* class. The first parameter that we pass is a list full of values (in this case numbers). The second parameter is the list of the indices or keys (in this case strings). When we now print our series, we can see what the structure looks like.

```
A    10
B    20
C    30
D    40
dtype: int64
```

The first column represents the indices, whereas the second column represents the actual values.

ACCESSING VALUES

The accessing of values works in the same way that it works with dictionaries. We need to address the respective index or key to get our desired value.

```
print(series['C'])

print(series[1])
```

As you can see, we can choose how we want to access our elements. We can either address the key or the position that the respective element is at.

CONVERTING DICTIONARIES

Since series and dictionaries are quite similar, we can easily convert our Python dictionaries into Pandas series.

```
myDict = {'A':10, 'B':20, 'C':30}

series = pd.Series(myDict)
```

Now the keys are our indices and the values remain values. But what we can also do is, to change the order of the indices.

```
myDict = {'A':10, 'B':20, 'C':30}

series = pd.Series(myDict, index=['C','A','B'])
```

Our series now looks like this:

```
C    30
A    10
B    20
dtype: int64
```

PANDAS DATA FRAME

In contrast to the series, a data frame is not one-dimensional but multi-dimensional and looks like a table. You can imagine it to be like an Excel table or a data base table.

```
data = {'Name': ['Anna', 'Bob', 'Charles'],
        'Age': [24, 32, 35],
        'Height': [176, 187, 175]}


df = pd.DataFrame(data)
```

To create a Pandas data frame, we use the constructor of the class. In this case, we first create a dictionary with some data about three persons. We feed that data into our data frame. It then looks like this:

```
     Name  Age  Height
0    Anna   24     176
1     Bob   32     187
2 Charles   35     175
```

As you can see, without any manual work, we already have a structured data frame and table.

To now access the values is a bit more complicated than with series. We have multiple columns and multiple rows, so we need to address two values.

```
print(df['Name'][1])
```

So first we choose the column *Name* and then we choose the second element (index one) of this column. In this case, this is *Bob*.

When we omit the last index, we can also select only the one column. This is useful when we want to save specific columns of our data frame into a new one. What we can also do in this case is to select multiple columns.

```
print(df[['Name', 'Height']])
```

Here we select two columns by addressing a list of two strings. The result is the following:

```
      Name  Height
0     Anna     176
1      Bob     187
2  Charles     175
```

DATA FRAME FUNCTIONS

Now, let us get a little bit more into the functions of a data frame.

BASIC FUNCTIONS AND ATTRIBUTES

For data frames we have a couple of basic functions and attributes that we already know from lists or NumPy arrays.

BASIC FUNCTIONS AND ATTRIBUTES	
FUNCTION	**DESCRIPTION**
df.T	Transposes the rows and columns of the data frame
df.dtypes	Returns data types of the data frame
df.ndim	Returns the number of dimensions of the data frame
df.shape	Returns the shape of the data frame
df.size	Returns the number of elements in the data frame
df.head(n)	Returns the first *n* rows of the data frame (default is five)
df.tail(n)	Returns the last *n* rows of the data frame (default is five)

STATISTICAL FUNCTIONS

For the statistical functions, we will now extend our data frame a little bit and add some more persons.

```
data = {'Name': ['Anna', 'Bob', 'Charles',
                 'Daniel', 'Evan', 'Fiona',
                 'Gerald', 'Henry', 'India'],
        'Age': [24,32,35,45,22,54,55,43,25],
        'Height': [176,187,175,182,176,
```

```
                189,165,187,167]}

df = pd.DataFrame(data)
```

STATISTICAL FUNCTIONS	
FUNCTION	**DESCRIPTION**
count()	Count the number of non-null elements
sum()	Returns the sum of values of the selected columns
mean()	Returns the arithmetic mean of values of the selected columns
median()	Returns the median of values of the selected columns
mode()	Returns the value that occurs most often in the columns selected
std()	Returns standard deviation of the values
min()	Returns the minimum value
max()	Returns the maximum value
abs()	Returns the absolute values of the elements
prod()	Returns the product of the selected elements
describe()	Returns data frame with all statistical values summarized

Now, we are not going to dig deep into every single function here. But let's take a look at how to apply some of them.

```
print(df['Age'].mean())

print(df['Height'].median())
```

Here we choose a column and then apply the statistical functions on it. What we get is just a single scalar with the desired value.

37.22222222222222
176.0

We can also apply the functions to the whole data frame. In this case, we get returned another data frame with the results for each column.

```
print(df.mean())
```

```
Age       37.222222
Height    178.222222
dtype: float64
```

APPLYING NUMPY FUNCTIONS

Instead of using the built-in Pandas functions, we can also use the methods we already know. For this, we just use the *apply* function of the data frame and then pass our desired method.

```
print(df['Age'].apply(np.sin))
```

In this example, we apply the sine function onto our ages. It doesn't make any sense but it demonstrates how this works.

LAMBDA EXPRESSIONS

A very powerful in Python are *lambda expression*. They can be thought of as nameless functions that we pass as a parameter.

```
print(df['Age'].apply(lambda x: x * 100))
```

By using the keyword *lambda* we create a temporary variable that represents the individual values that we are applying the operation onto. After the colon, we define what we want to do. In this case, we multiply all values of the column *Age* by 100.

```
df = df[['Age', 'Height']]
```

```
print(df.apply(lambda x: x.max() - x.min()))
```

Here we removed the *Name* column, so that we only have numerical values. Since we are applying our expression on the whole data frame now, *x* refers to the whole columns. What we do here is calculating the difference between the maximum value and the minimum value.

```
Age      33
Height   24
dtype: int64
```

The oldest and the youngest are 33 years apart and the tallest and the tiniest are 24 centimeters apart.

ITERATING

Iterating over data frames is quite easy with Pandas. We can either do it in the classic way or use specific functions for it.

```
for x in df['Age']:

    print(x)
```

As you can see, iterating over a column's value is very simple and nothing new. This would print all the ages. When we iterate over the whole data frame, our control variable takes on the column names.

STATISTICAL FUNCTIONS	
FUNCTION	**DESCRIPTION**
iteritems()	Iterator for key-value pairs
iterrows()	Iterator for the rows (index, series)
itertuples()	Iterator for the rows as named tuples

Let's take a look at some practical examples.

```
for key, value in df.iteritems():

    print("{}: {}".format(key, value))
```

Here we use the *iteritems* function to iterate over key-value pairs. What we get is a huge output of all rows for each column.

On the other hand, when we use *iterrows*, we can print out all the column-values for each row or index.

```
for index, value in df.iterrows():

    print(index,value)
```

We get packages like this one for every index:

```
0 Name      Anna
Age         24
Height     176
Name: 0, dtype: object
```

SORTING

One very powerful thing about Pandas data frames is that we can easily sort them.

SORT BY INDEX

```
df = pd.DataFrame(np.random.rand(10,2),
                  index=[1,5,3,6,7,2,8,9,0,4],
                  columns=['A','B'])
```

Here we create a new data frame, which is filled with random numbers. We specify our own indices and as you can see, they are completely unordered.

```
print(df.sort_index())
```

By using the method *sort_index*, we sort the whole data frame by the index column. The result is now sorted:

	A	B
0	0.193432	0.514303
1	0.391481	0.193495
2	0.159516	0.607314
3	0.273120	0.056247
…	…	…

INPLACE PARAMETER

When we use functions that manipulate our data frame, we don’t actually change it but we return a manipulated copy. If we wanted to apply the changes on the actual data frame, we would need to do it like this:

```
df = df.sort_index()
```

But Pandas offers us another alternative as well. This alternative is the parameter *inplace*. When this parameter is set to *True*, the changes get applied to our actual data frame.

```
df.sort_index(inplace=True)
```

SORT BY COLUMNS

Now, we can also sort our data frame by specific columns.

```
data = {'Name': ['Anna', 'Bob', 'Charles',
                 'Daniel', 'Evan', 'Fiona',
                 'Gerald', 'Henry', 'India'],
        'Age': [24,24,35,45,22,54,54,43,25],
        'Height': [176,187,175,182,176,
                   189,165,187,167]}


df = pd.DataFrame(data)


df.sort_values(by=['Age', 'Height'],
               inplace=True)


print(df)
```

Here we have our old data frame slightly modified. We use the function *sort_values* to sort our data frames. The parameter *by* states the columns that we are sorting by. In this case, we are first sorting by age and if two persons have the same age, we sort by height.

JOINING AND MERGING

Another powerful concept in Pandas is *joining* and *merging* data frames.

```
names = pd.DataFrame({
    'id': [1,2,3,4,5],
    'name': ['Anna', 'Bob', 'Charles',
             'Daniel', 'Evan'],
})


ages = pd.DataFrame({
    'id': [1,2,3,4,5],
    'age': [20,30,40,50,60]
})
```

Now when we have two separate data frames which are related to one another, we can combine them into one data frame. It is important that we have a common column that we can merge on. In this case, this is *id*.

```
df = pd.merge(names,ages,on='id')

df.set_index('id', inplace=True)
```

First we use the method *merge* and specify the column to merge on. We then have a new data frame with the combined data but we also want our *id* column to be the index. For this, we use the *set_index* method. The result looks like this:

id	name	age
1	Anna	20
2	Bob	30
3	Charles	40

```
4    Daniel   50
5     Evan    60
```

JOINS

It is not necessarily always obvious *how* we want to merge our data frames. This is where *joins* come into play. We have four types of joins.

JOIN MERGE TYPES	
JOIN	**DESCRIPTION**
left	Uses all keys from left object and merges with right
right	Uses all keys from right object and merges with left
outer	Uses all keys from both objects and merges them
inner	Uses only the keys which both objects have and merges them (default)

Now let's change our two data frames a little bit.

```
names = pd.DataFrame({
    'id': [1,2,3,4,5,6],
    'name': ['Anna', 'Bob', 'Charles',
             'Daniel', 'Evan', 'Fiona'],
})


ages = pd.DataFrame({
    'id': [1,2,3,4,5,7],
    'age': [20,30,40,50,60,70]
})
```

Our *names* frame now has an additional index *6* and an additional name. And our *ages* frame has an additional index *7* with an additional name.

```
df = pd.merge(names,ages,on='id', how='inner')

df.set_index('id', inplace=True)
```

If we now perform the default *inner join*, we will end up with the same data frame as in the beginning. We only take the keys which both objects have. This means one to five.

```
df = pd.merge(names,ages,on='id', how='left')

df.set_index('id', inplace=True)
```

When we use the *left join*, we get all the keys from the *names* data frame but not the additional index *7* from ages. This also means that *Fiona* won't be assigned any age.

id	name	age
1	Anna	20.0
2	Bob	30.0
3	Charles	40.0
4	Daniel	50.0
5	Evan	60.0
6	Fiona	NaN

The same principle goes for the *right join* just the other way around.

```
df = pd.merge(names,ages,on='id', how='right')

df.set_index('id', inplace=True)
```

id	name	age
1	Anna	20
2	Bob	30
3	Charles	40
4	Daniel	50
5	Evan	60
7	NaN	70

Now, we only have the keys from the *ages* frame and the *6* is missing. Finally, if

we use the *outer join*, we combine all keys into one data frame.

```
df = pd.merge(names,ages,on='id', how='outer')
```

```
df.set_index('id', inplace=True)
```

id	name	age
1	Anna	20.0
2	Bob	30.0
3	Charles	40.0
4	Daniel	50.0
5	Evan	60.0
6	Fiona	NaN
7	NaN	70.0

QUERYING DATA

Like in databases with SQL, we can also query data from our data frames in Pandas. For this, we use the function *loc*, in which we put our expression.

```
print(df.loc[df['Age'] == 24])

print(df.loc[(df['Age'] == 24) &

             (df['Height'] > 180)])

print(df.loc[df['Age'] > 30]['Name'])
```

Here we have some good examples to explain how this works. The first expression returns all rows where the value for *Age* is 24.

```
   Name  Age  Height
0  Anna   24     176
1   Bob   24     187
```

The second query is a bit more complicated. Here we combine two conditions. The first one is that the age needs to be 24 but we then combine this with the condition that the height is greater than 180. This leaves us with one row.

```
  Name  Age  Height
1  Bob   24     187
```

In the last expression, we can see that we are only choosing one column to be returned. We want the names of all people that are older than 30.

```
2    Charles
3     Daniel
5      Fiona
6     Gerald
7      Henry
```

READ DATA FROM FILES

Similar to NumPy, we can also easily read data from external files into Pandas. Let's say we have an CSV-File like this (opened in Excel):

	A	B	C	D
1	id	name	age	height
2	1	Anna	20	178
3	2	Bob	30	172
4	3	Charles	40	189
5	4	Daniel	50	192
6	5	Evan	60	183
7	6	Fiona	70	165
8				

The only thing that we need to do now is to use the function *read_csv* to import our data into a data frame.

```
df = pd.read_csv('data.csv')

df.set_index('id', inplace=True)

print(df)
```

We also set the index to the *id* column again. This is what we have imported:

```
      name  age  height
id
1     Anna   20     178
2      Bob   30     172
3  Charles   40     189
4   Daniel   50     192
5     Evan   60     183
6    Fiona   70     165
```

This of course, also works the other way around. By using the method *to_csv*, we can also save our data frame into a CSV-file.

```
data = {'Name': ['Anna', 'Bob', 'Charles',
                 'Daniel', 'Evan', 'Fiona',
                 'Gerald', 'Henry', 'India'],
```

```
        'Age': [24,24,35,45,22,54,54,43,25],
        'Height': [176,187,175,182,176,
                   189,165,187,167]}

df = pd.DataFrame(data)
df.to_csv('mydf.csv')
```

Then we have this CSV-file (opened in Excel):

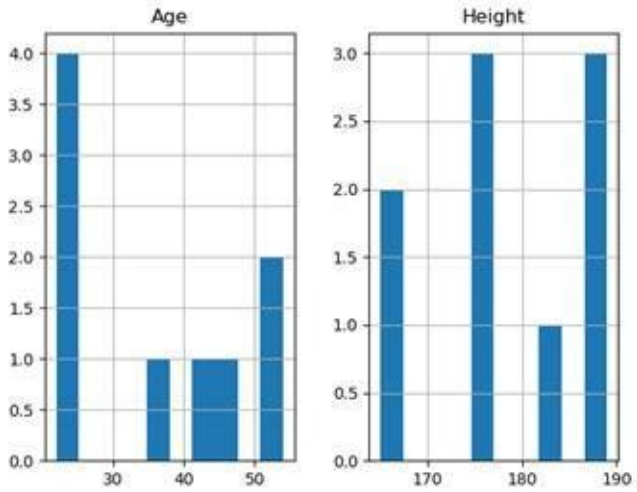
	A	B	C	D
1		Name	Age	Height
2	0	Anna	24	176
3	1	Bob	24	187
4	2	Charles	35	175
5	3	Daniel	45	182
6	4	Evan	22	176
7	5	Fiona	54	189
8	6	Gerald	54	165
9	7	Henry	43	187
10	8	India	25	167
11				

PLOTTING DATA

Since Pandas builds on Matplotlib, we can easily visualize the data from our data frame.

```
data = {'Name': ['Anna', 'Bob', 'Charles',
                 'Daniel', 'Evan', 'Fiona',
                 'Gerald', 'Henry', 'India'],
        'Age': [24,24,35,45,22,54,54,43,25],
        'Height': [176,187,175,182,176,
                   189,165,187,167]}


df = pd.DataFrame(data)

df.sort_values(by=['Age', 'Height'])

df.hist()

plt.show()
```

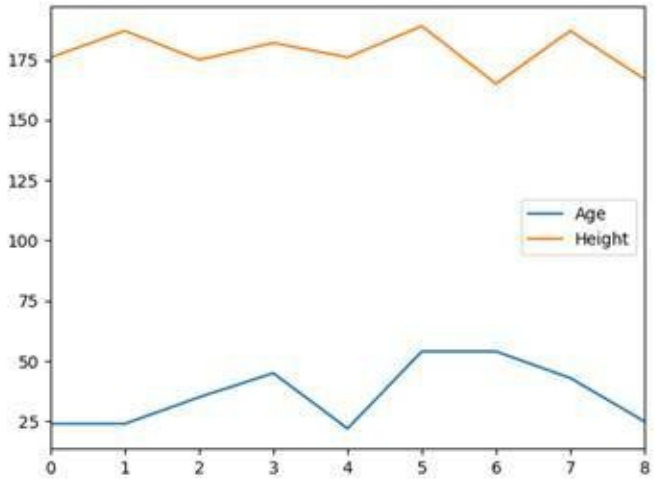
In this example, we use the method *hist* to plot a histogram of our numerical columns. Without specifying anything more, this is what we end up with:



But we can also just use the function *plot* to plot our data frame or individual columns.

```
df.plot()

plt.show()
```

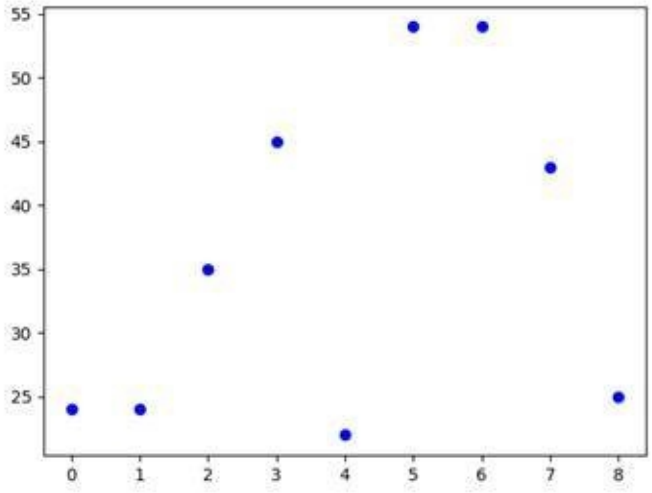
The result is the following:



Of course we can also just use the Matplotlib library itself and pass the columns as parameters.

```
plt.plot(df['Age'], 'bo')
```

```
plt.show()
```



WHAT'S NEXT?

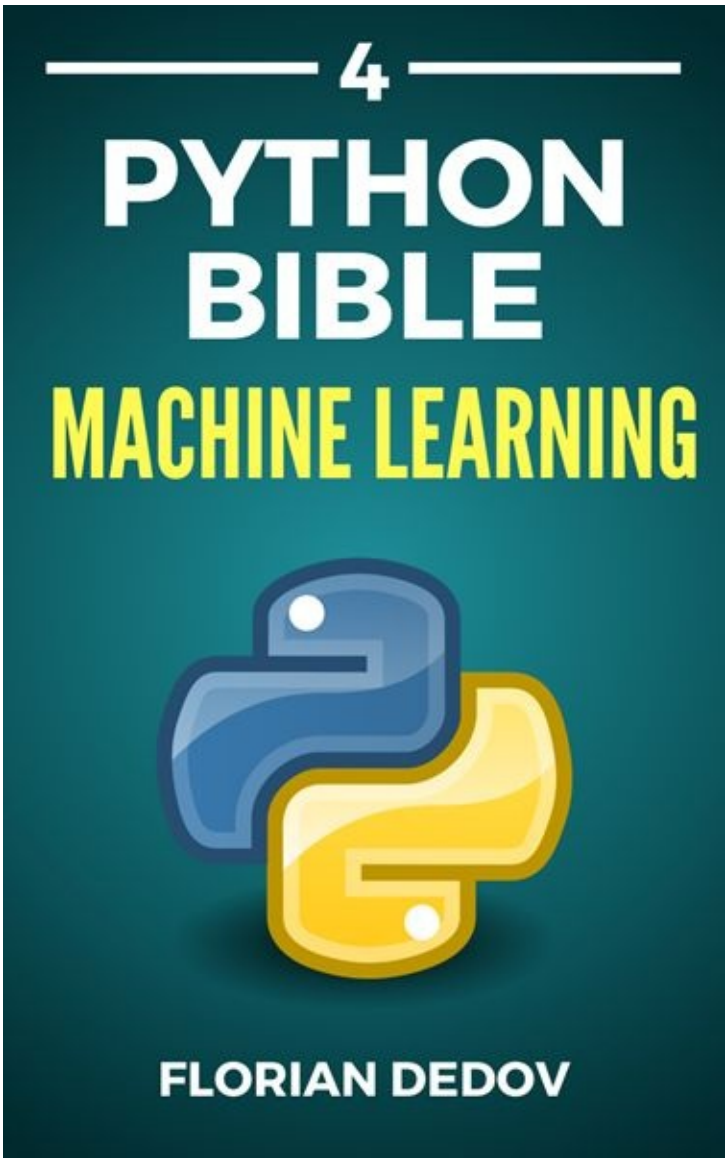
Finally, we are done with the third volume of the Python Bible series. It was very practical and went deep into the topic of data science. This has now laid the foundation for more complex topics like machine learning and finance, which will be the follow-ups to this book. You are on a very good path! Just make sure you practice everything until you really understand the material that we talked about.

You are now definitely able to find some huge data sets online (maybe in CSV-format) and analyze them with Python. And I encourage you to do that. We only learn by doing and practicing. In the next volumes we will also import data from online sources and APIs. And we are not only going to analyze this data but also to make predictions with it.

Now that you've read the first three volumes of this series, I encourage you to continue on this journey because it is NOW that things get really interesting. I hope you could get some value out of this book and that it helped you to become a better programmer. So stay tuned and prepare for the next volume!

Last but not least, a little reminder. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!


4
PYTHON
BIBLE
MACHINE LEARNING
FLORIAN DEDOV

THE
PYTHON BIBLE

VOLUME FOUR

MACHINE LEARNING

By

FLORIAN DEDOV

TABLE OF CONTENT

INTRODUCTION

With this book, we get into some really advanced territory and things get more and more complicated. In the last book, we were learning about data science and data analysis. We now know how to analyze and visualize big data sets. But except for some statistical values we didn't really extract any knowledge out of the data and we were certainly not able to predict future data.

This is where the topic of this book comes into play – *machine learning*. It's a much hyped term and nowadays it can be found almost everywhere. In robots, video games, the stock market, home appliances or even in cars. And it's constantly growing. The development of artificial intelligences can't be stopped and it bears almost unlimited potential (for both – good and evil). The people who don't educate themselves on this matter will be overrun by the development instead of benefiting from it.

Python is definitely the language that dominates the AI market. Of course, artificial intelligences are developed in all sorts of languages but Python has become the *lingua franca* of machine learning in the past few years. Therefore, if you want to be part of this future, you will need to be fluent in Python and get a good understanding of machine learning.

THIS BOOK

In this volume of The Python Bible series, we will dig deep into the machine learning realm and the Python language. We will train and apply complex machine learning models and at the end you will be able to develop and optimize your own AI suited for your specific tasks.

What you will need for this book is the knowledge from the previous three volumes. You will need to be fluent in the basic and intermediate concepts of the Python language. Also, you will need some basic understanding of data science and the libraries *NumPy, Pandas* and *Matplotlib*. If you have already read volume one to three, you are good to go. A decent understanding of mathematics (high school level) is definitely beneficial.

We will start by discussing what machine learning actually is and what types there are. Then, we will install the necessary modules and start with the programming part. First, we will look at linear regression, which is a pretty basic statistical machine learning model. After that, we will cover classification, clustering and support vector machines. Then, we will discuss neural networks and build a model that predicts handwritten digits. At the end, we will take a look at the optimization of models.

This book is again full of new and more complex information. There is a lot to learn here so stay tuned and code along while reading. This will help you to understand the material better and to practice implementing it. I wish you a lot of fun and success with your journey and this book!

Just one little thing before we start. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you have learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

1 – WHAT IS MACHINE LEARNING?

As always, before we start to learn how something works, we first want to precisely define what it is, that we are learning about. So what is this hyped concept of *machine learning*? I will not try to give you some sophisticated and complex scientific definition here. We will try to explain it as simply as possible.

What machine learning is fundamentally is just the *science,* in which we focus on teaching machines or computers to perform certain tasks without being given specific instructions. We want our machines to learn how to do something themselves without explaining it to them.

In order to do this, we oftentimes look at how the human brain works and try to design virtual brains that work in a similar manner.

Notice that machine learning and artificial intelligence are not the same. Artificial intelligence is a broad field and every system that can learn and solve problems might be considered an AI. Machine learning is *one specific approach* to this broad field. In machine learning the AI doesn’t receive any instructions. It isn’t static and it doesn’t follow clear and strict steps to solve problems. It’s dynamic and restructures itself.

SUPERVISED LEARNING

In machine learning we have different approaches or types. The two main approaches are *supervised learning* and *unsupervised learning*. So let's first talk about supervised learning.

Here, we give our model a set of inputs and also the corresponding outputs, which are the desired results. In this way, the model learns to match certain inputs to certain outputs and it adjusts its structure. It learns to make connections between what was put in and what the desired output is. It understands the correlation. When trained well enough, we can use the model to make predictions for inputs that we don't know the results for.

Classic supervised learning algorithms are regressions, classifications and support vector machines.

UNSUPERVISED LEARNING

With unsupervised learning on the other hand, we don’t give our model the desired results while training. Not because we don’t want to but because we don’t know them. This approach is more like a kind of pattern recognition. We give our model a set of input data and it then has to look for patterns in it. Once the model is trained, we can put in new data and our model will need to make decisions.

Since the model doesn’t get any information about classes or results, it has to work with similarities and patterns in the data and categorize or cluster it by itself.

Classic unsupervised learning algorithms are clustering, anomaly detection and some applications of neural networks.

REINFORCEMENT LEARNING

Then there is a third type of machine learning called *reinforcement learning*. Here we create some model with a random structure. Then we just observe what it does and reinforce or encourage it, when we like what it does. Otherwise, we can also give some negative feedback. The more our model does what we want it to do, the more we reinforce it and the more “rewards” it gets. This might happen in form of a number or a grade, which represents the so-called *fitness* of the model.

In this way, our model learns what is right and what is wrong. You can imagine it a little bit like natural selection and survival of the fittest. We can create 100 random models and kill the 50 models that perform worst. Then the remaining 50 reproduce and the same process repeats. These kinds of algorithms are called *genetic algorithms.*

Classic reinforcement learning algorithms are genetic or evolutional algorithms.

DEEP LEARNING

Another term that is always confused with machine learning is *deep learning*. Deep learning however is just one area of machine learning, namely the one, which works with neural networks. Neural networks are a very comprehensive and complex topic. Even though there is a chapter about them in this book, we won't be able to dig too deep into the details here. Maybe I will write a separate volume which just focuses only on neural networks in the future.

FIELDS OF APPLICATION

Actually, it would be easier to list all the areas in which machine learning doesn’t get applied rather than the fields of application. Despite that, we will take a quick look at some of the major areas, in which machine learning gets applied.

- Research
- Autonomous Cars
- Spacecraft
- Economics and Finance
- Medical and Healthcare
- Physics, Biology, Chemistry
- Engineering
- Mathematics
- Robotics
- Education
- Forensics
- Police and Military
- Marketing
- Search Engines
- GPS and Pathfinding Systems
- …

We could go on forever. I think it is very clear why we should educate ourselves in this area. With this book, you are going into the right direction.

WHY PYTHON?

Now before we go on to the next chapters, let’s once again address the question of why we should use Python for machine learning instead of another language. I already mentioned that it has become the *lingua franca* of machine learning. This means that it has become the main language in this field. You might compare it to English in the western world.

There are also other languages like R, MATLAB or LISP which may be considered competition of Python but they are quite specialized for one specific field of application, whereas Python is a general-purpose language.

Python’s community is great and it is massively gaining popularity. The language is simple, easy to learn, easy to use and offers a huge arsenal of powerful open-source libraries for data science, scientific computing and machine learning. Of course other languages have their advantages over Python, but for the field of machine learning there is probably no better choice than Python at the current moment.

2 – INSTALLING MODULES

Again, before we get into the coding, we will need to install the modules and libraries that we are going to use. If you have read volume three of this series, you will already be familiar with the first three. Nevertheless, we are going to discuss them quickly one more time.

NUMPY

The *NumPy* module allows us to efficiently work with vectors, matrices and multi-dimensional arrays. It is crucial for linear algebra and numerical analysis. It basically replaces the primitive and inefficient Python *list* with very powerful *NumPy arrays*.

MATPLOTLIB

On top of NumPy, we have *Matplotlib*. This library is responsible for plotting graphs and visualizing our data. It offers numerous types of plotting, styles and graphs.

PANDAS

Pandas offers us a powerful data structure named *data frame*. You can imagine it to be a bit like a mix of an Excel table and an SQL database table.

This library allows us to efficiently work with our huge amounts of interrelated data. We can merge, reshape, filter and query our data. We can iterate over it and we can read and write into files like CSV, XLSX and more. Also, it is very powerful when we work with databases, due to the similar structure of the tables.

Pandas is highly compatible with NumPy and Matplotlib, since it builds on them. We can easily convert data from one format to the other.

SCIKIT-LEARN

Now, the first new library of this book is *scikit-learn.* This is probably the most important Python library for traditional machine learning. It features classification, regression and clustering algorithms. Also, it allows us to work with support vector machines and more. Scikit-learn is designed to work with NumPy and Matplotlib, which will make everything much easier for us.

TENSORFLOW

Tensorflow is one of the most popular machine learning frameworks out there and it was developed by Google. It is a whole ecosystem for developing modern deep learning models. This means that it is mainly used for the development and training of models that use neural networks. It also has its own data structures and ways of visualizing data.

INSTALLING MODULES WITH PIP

We now need to install our models. In this book, we are using *pip* to do that.

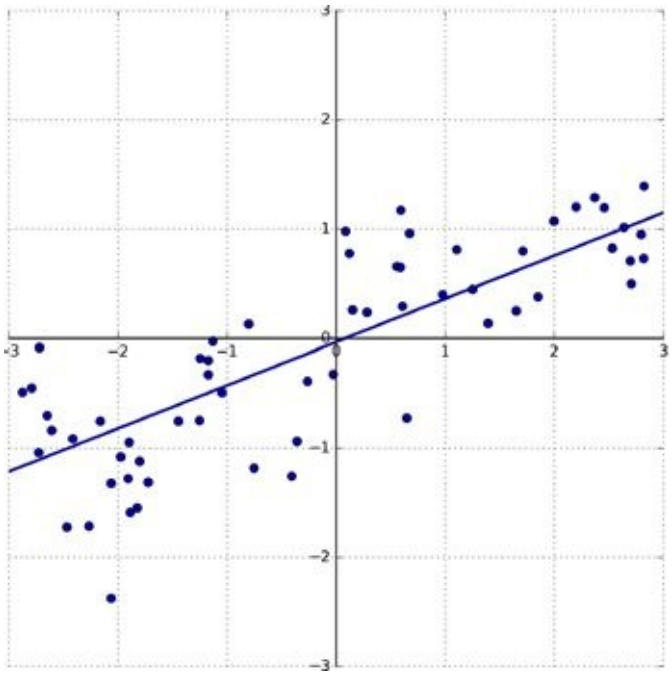
```
pip install numpy
pip install matplotlib
pip install pandas
pip install scikit-learn
pip install tensorflow
```

3 – LINEAR REGRESSION

The easiest and most basic machine learning algorithm is *linear regression*. It will be the first one that we are going to look at and it is a supervised learning algorithm. That means that we need both – inputs and outputs – to train the model.

MATHEMATICAL EXPLANATION

Before we get into the coding, let us talk about the mathematics behind this algorithm.



In the figure above, you see a lot of different points, which all have an x-value and a y-value. The x-value is called the *feature*, whereas the y-value is our *label*. The label is the result for our feature. Our *linear regression model* is represented by the blue line that goes straight through our data. It is placed so that it is as close as possible to all points at the same time. So we “trained” the line to fit the existing points or the existing data.

The idea is now to take a new x-value without knowing the corresponding y-value. We then look at the line and find the resulting y-value there, which the model predicts for us. However, since this line is quite generalized, we will get a relatively inaccurate result.

However, one must also mention that linear models only really develop their effectiveness when we are dealing with numerous features (i.e. higher dimensions).

If we are applying this model to data of schools and we try to find a relation between missing hours, learning time and the resulting grade, we will probably

get a less accurate result than by including 30 parameters. Logically, however, we then no longer have a straight line or flat surface but a hyperplane. This is the equivalent to a straight line, in higher dimensions.

LOADING DATA

To get started with our code, we first need data that we want to work with. Here we use a dataset from UCI.

Link: https://archive.ics.uci.edu/ml/datasets/student+performance

This is a dataset which contains a lot of information about student performance. We will use it as sample data for our models.

We download the ZIP-file from the *Data Folder* and extract the file student-mat.csv from there into the folder in which we code our script.

Now we can start with our code. First of all, we will import the necessary libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

Besides the imports of the first three libraries that we already know, we have two more imports that are new to us. First, we import the *LinearRegression module*. This is the module that we will use for creating, training and using our regression model. Additionally, we import the *train_test_split* module, which we will use to prepare and split our data.

Our first action is to load the data from the CSV file into a Pandas DataFrame. We do this with the function read_csv.

```
data = pd.read_csv('student-mat.csv', sep=';')
```

It is important that we change our separator to semicolon, since the default separator for CSV files is a comma and our file is separated by semicolons.

In the next step, we think about which features (i.e. columns) are relevant for us, and what exactly we want to predict. A description of all features can be found on the previously mentioned website. In this example, we will limit ourselves to

the following columns:

Age, Sex, Studytime, Absences, G1, G2, G3 (label)

```
data = data[['age', 'sex', 'studytime',
             'absences', 'G1', 'G2', 'G3']]
```

The columns *G1, G2* and *G3* are the three grades that the students get. Our goal is to predict the third and final grade by looking at the other values like first grade, age, sex and so on.

Summarized that means that we only select these columns from our DataFrame, out of the 33 possible. *G3* is our label and the rest are our features. Each feature is an axis in the coordinate system and each point is a record, that is, one row in the table.

But we have a little problem here. The *sex* feature is not numeric, but stored as *F (for female)* or *M (for male)*. But for us to work with it and register it in the coordinate system, we have to convert it into numbers.

```
data['sex'] = data['sex'].map({'F': 0, 'M': 1})
```

We do this by using the *map* function. Here, we map a dictionary to our feature. Each *F* becomes a zero and every *M* becomes a one. Now we can work with it.

Finally, we define the column of the desired label as a variable to make it easier to work with.

```
prediction = 'G3'
```

PREPARING DATA

Our data is now fully loaded and selected. However, in order to use it as training and testing data for our model, we have to reformat them. The sklearn models do not accept Pandas data frames, but only NumPy arrays. That's why we turn our features into an x-array and our label into a y-array.

```
X = np.array(data.drop([prediction], 1))

Y = np.array(data[prediction])
```

The method *np.array* converts the selected columns into an array. The *drop* function returns the data frame without the specified column. Our *X* array now contains all of our columns, except for the final grade. The final grade is in the *Y* array.

In order to train and test our model, we have to split our available data. The first part is used to get the hyperplane to fit our data as well as possible. The second part then checks the accuracy of the prediction, with previously unknown data.

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.1)
```

With the function *train_test_split*, we divide our *X* and *Y* arrays into four arrays. The order must be exactly as shown here. The *test_size* parameter specifies what percentage of records to use for testing. In this case, it is 10%. This is also a good and recommended value. We do this to test how accurate it is with data that our model has never seen before.

TRAINING AND TESTING

Now we can start training and testing our model. For that, we first define our model.

```
model = LinearRegression()

model.fit(X_train, Y_train)
```

By using the constructor of the *LinearRegression* class, we create our model. We then use the *fit* function and pass our training data. Now our model is already trained. It has now adjusted its hyperplane so that it fits all of our values.

In order to test how well our model performs, we can use the *score* method and pass our testing data.

```
accuracy = model.score(X_test, Y_test)

print(accuracy)
```

Since the splitting of training and test data is always random, we will have slightly different results on each run. An average result could look like this:

```
0.9130676521162756
```

Actually, 91 percent is a pretty high and good accuracy. Now that we know that our model is somewhat reliable, we can enter new data and predict the final grade.

```
X_new = np.array([[18, 1, 3, 40, 15, 16]])

Y_new = model.predict(X_new)

print(Y_new)
```

Here we define a new NumPy array with values for our features in the right order. Then we use the *predict* method, to calculate the likely final grade for our inputs.

[17.12142363]

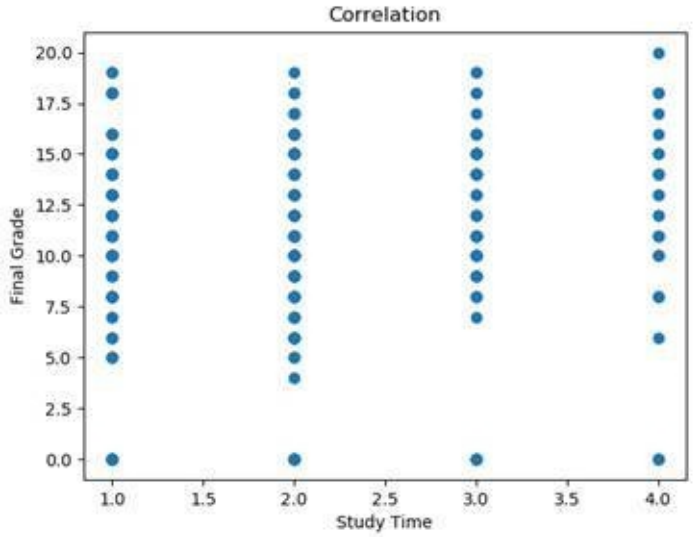
In this case, the final grade would probably be 17.

VISUALIZING CORRELATIONS

Since we are dealing with high dimensions here, we can’t draw a graph of our model. This is only possible in two or three dimensions. However, what we can visualize are relationships between individual features.

```
plt.scatter(data['studytime'], data['G3'])

plt.title("Correlation")

plt.xlabel("Study Time")

plt.ylabel("Final Grade")

plt.show()
```

Here we draw a scatter plot with the function scatter, which shows the relationship between the learning time and the final grade.

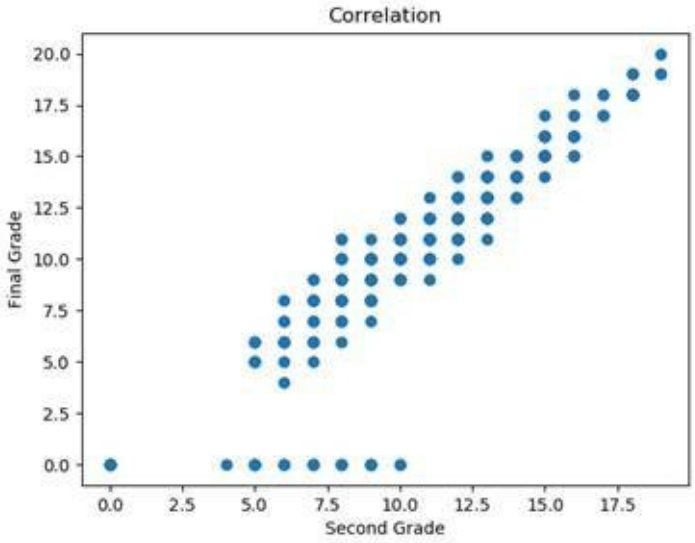


In this case, we see that the relationship is not really strong. The data is very diverse and you cannot see a clear pattern.

```
plt.scatter(data['G2'], data['G3'])

plt.title("Correlation")
```

```
plt.xlabel("Second Grade")

plt.ylabel("Final Grade")

plt.show()
```

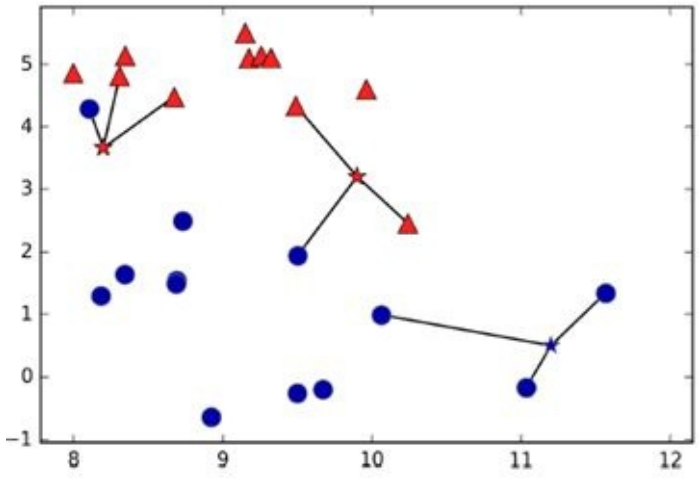
However, if we look at the correlation between the second grade and the final grade, we see a much stronger correlation.



Here we can clearly see that the students with good second grades are very likely to end up with a good final grade as well. You can play around with the different columns of this data set if you want to.

4 – CLASSIFICATION

With regression we now predicted specific output-values for certain given input-values. Sometimes, however, we are not trying to predict outputs but to categorize or classify our elements. For this, we use *classification* algorithms.



In the figure above, we see one specific kind of classification algorithm, namely the *K-Nearest-Neighbor* classifier. Here we already have a decent amount of classified elements. We then add a new one (represented by the stars) and try to predict its class by looking at its nearest neighbors.

CLASSIFICATION ALGORITHMS

There are various different classification algorithms and they are often used for predicting medical data or other real life use-cases. For example, by providing a large amount of tumor samples, we can classify if a tumor is benign or malignant with a pretty high certainty.

K-NEAREST-NEIGHBORS

As already mentioned, by using the K-Nearest-Neighbors classifier, we assign the class of the new object, based on its nearest neighbors. The *K* specifies the amount of neighbors to look at. For example, we could say that we only want to look at the one neighbor who is nearest but we could also say that we want to factor in 100 neighbors.

Notice that *K* shouldn't be a multiple of the number of classes since it might cause conflicts when we have an equal amount of elements from one class as from the other.

NAIVE-BAYES

The *Naive Bayes* algorithm might be a bit confusing when you encounter it the first time. However, we are only going to discuss the basics and focus more on the implementation in Python later on.

Outlook	Temperture	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Rainy	Mild	High	False	No
Rainy	Hot	High	True	No
Overcast	Hot	Normal	True	Yes
Sunny	Hot	Normal	True	Yes
Sunny	Mild	High	True	Yes
Overcast	Cold	Normal	True	No
…	…	…	…	…

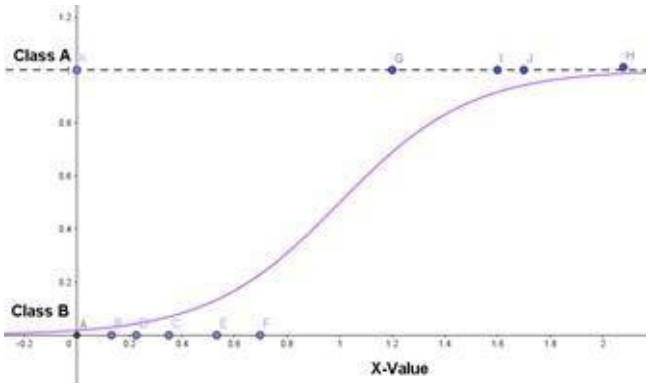
Imagine that we have a table like the one above. We have four input values (which we would have to make numerical of course) and one label or output. The two classes are *Yes* and *No* and they indicate if we are going to play outside or not.

What *Naive Bayes* now does is to write down all the probabilities for the individual scenarios. So we would start by writing the general probability of playing and not playing. In this case, we only play three out of eight times and thus our probability of playing will be 3/8 and the probability of not playing will be 5/8.

Also, out of the five times we had a high humidity we only played once, whereas out of the three times it was normal, we played twice. So our probability for playing when we have a high humidity is 1/5 and for playing when we have a medium humidity is 2/3. We go on like that and note all the probabilities we have in our table. To then get the classification for a new entry, we multiply the probabilities together and end up with a prediction.

LOGISTIC REGRESSION

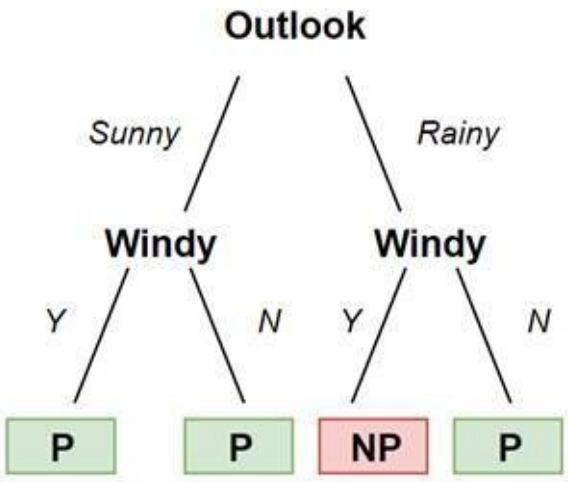
Another popular classification algorithm is called *logistic regression*. Even though the name says *regression*, this is actually a classification algorithm. It looks at probabilities and determines how likely it is that a certain event happens (or a certain class is the right one), given the input data. This is done by plotting something similar to a logistic growth curve and splitting the data into two.



Since we are not using a line (and thus our model is not linear), we are also preventing mistakes caused by outliers.

DECISION TREES

With *decision tree* classifiers, we construct a decision tree out of our training data and use it to predict the classes of new elements.



This classification algorithm requires very little data preparation and it is also very easy to understand and visualize. On the other hand, it is very easy to be *overfitting* the model. Here, the model is very closely matched to the training data and thus has worse chances to make a correct prediction on new data.

RANDOM FOREST

The last classification algorithm of this chapter is the *random forest* classifier. It is based on decision trees. What it does is creating a *forest* of multiple decision trees. To classify a new object, all the various trees determine a class and the most frequent result gets chosen. This makes the result more accurate and it also prevents overfitting. It is also more suited to handle data sets with higher dimensions. On the other hand, since the generation of the forest is *random*, you have very little control over your model.

LOADING DATA

Now let us get into the code. In this example, we will get our data directly from the sklearn module. For the program we need the following imports:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer
```

At the last import, we import a dataset containing data on breast cancer. Also notice that we are only importing the *KNeighborsClassifier* for now.

```
data = load_breast_cancer()


print(data.feature_names)
print(data.target_names)
```

We load the data with the *load_breast_cancer* function and get the names of the features and targets. Our features are all parameters that should help to determine the label or the target. For the targets, we have two options in this dataset: *malignant* and *benign*.

PREPARING DATA

Again, we convert our data back into NumPy arrays and split them into training and test data.

```
X = np.array(data.data)

Y = np.array(data.target)



X_train, X_test, Y_train, Y_test =
train_test_split(X,Y,test_size=0.1)
```

The data attribute refers to our features and the target attribute points to the classes or labels. We again choose a test size of ten percent.

TRAINING AND TESTING

We start by first defining our K-Nearest-Neighbors classifier and then training it.

```
knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(X_train, Y_train)
```

The *n_neighbors* parameter specifies how many neighbor points we want to consider. In this case, we take five. Then we test our model again for its accuracy.

```
accuracy = knn.score(X_test, Y_test)

print(accuracy)
```

We get a pretty decent accuracy for such a complex task.

0.9649122807017544

THE BEST ALGORITHM

Now let's put all the classification algorithms that we've discussed up until now to use and see which one performs best.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
```

Of course, we need to import all the modules first. We can then create five different classifiers to train and test them with the exact same data.

```
clf1 = KNeighborsClassifier(n_neighbors=5)
clf2 = GaussianNB()
clf3 = LogisticRegression()
clf4 = DecisionTreeClassifier()
clf5 = RandomForestClassifier()


clf1.fit(X_train, Y_train)
clf2.fit(X_train, Y_train)
clf3.fit(X_train, Y_train)
clf4.fit(X_train, Y_train)
clf5.fit(X_train, Y_train)


print(clf1.score(X_test, Y_test))
```

```
print(clf2.score(X_test, Y_test))

print(clf3.score(X_test, Y_test))

print(clf4.score(X_test, Y_test))

print(clf5.score(X_test, Y_test))
```

When you run this program a couple of times, you will notice that we can’t really say which algorithm is the best. Every time we run this script, we will see different results, at least for this specific data set.

PREDICTING LABELS

Again, we can again make predictions for new, unknown data. The chance of success in the classification is even very high. We just need to pass an array of input values and use the *predict function*.

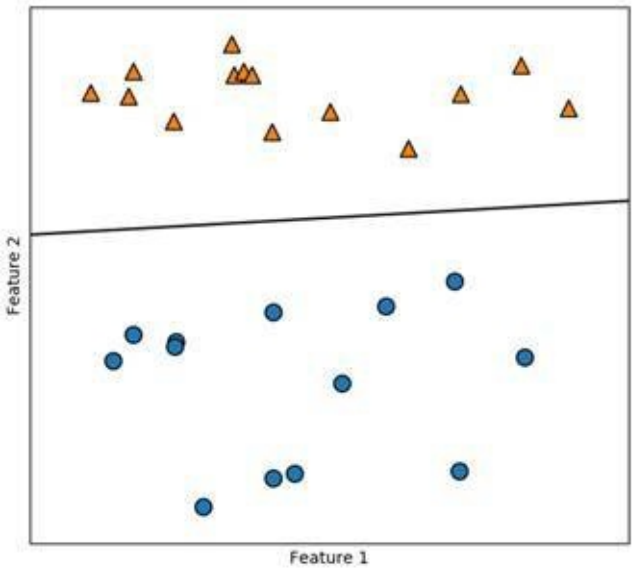
```
X_new = np.array([[...]])

Y_new = clf.predict(X_new)
```

Unfortunately, visualizing the data is not possible here because we have 30 features and cannot draw a 30-dimensional coordinate system.

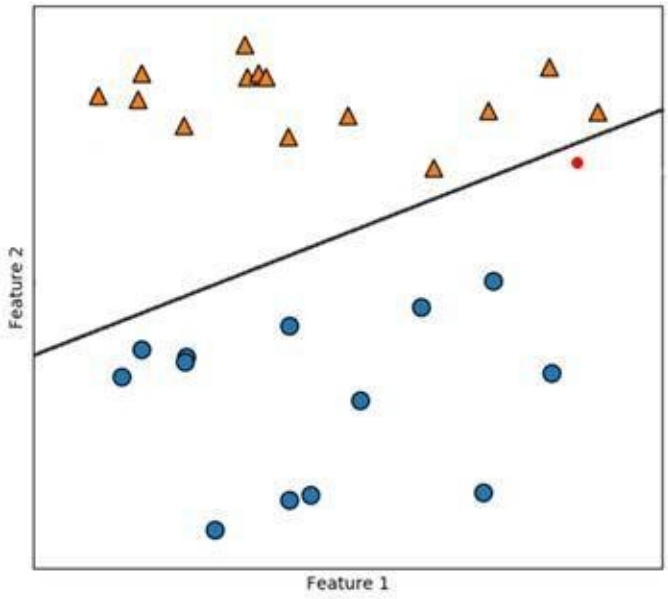
5 – SUPPORT VECTOR MACHINES

Now things get mathematically a bit more demanding. This chapter is about *Support Vector Machines*. These are very powerful, very efficient machine learning algorithms and they even achieve much better results than neural networks in some areas. We are again dealing with classification here but the methodology is quite different.

What we are looking for is a *hyperplane* that distinctly classifies our data points and has the maximum margin to all of our points. We want our model to be as generalized as possible.

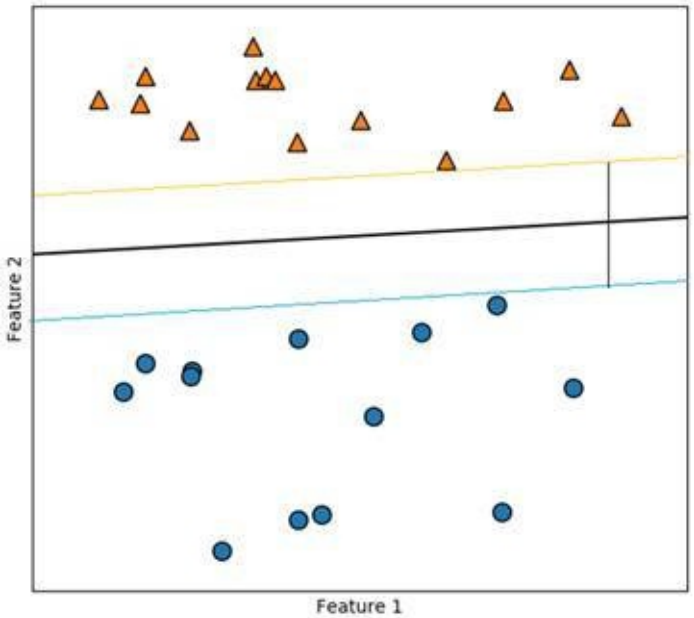


In the graph above the model is very general and the line is the optimal function to separate our data. We can use an endless amount of lines to separate the two classes but we don’t want to overfit our model so that it only works for the data we already have. We also want it to work for unknown data.



Here our model also separates the data we already have perfectly. But we've got a new red data point here. When we just look at this with our intuition it is obvious that this point belongs to the orange triangles. However, our model classifies it as a blue circle because it is overfitting our current data.

To find our perfect line we are using so-called *support vectors*, which are parallel lines.

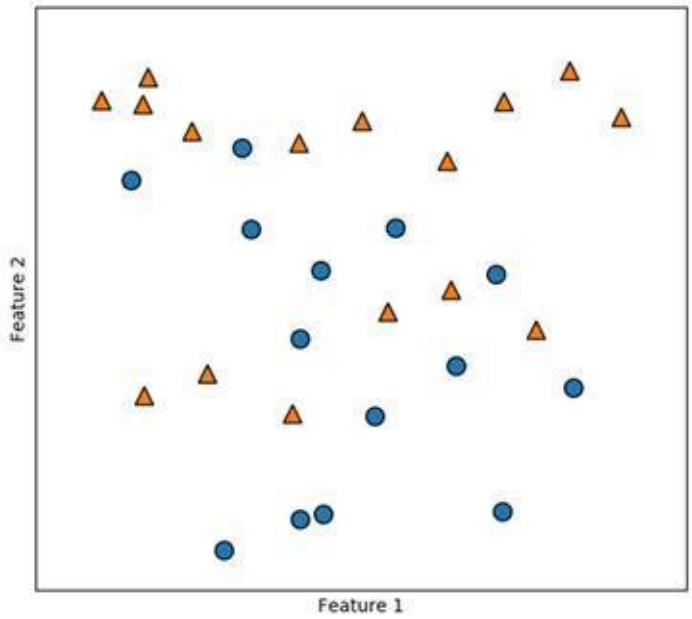


We are looking for the two points that are the farthest away from the other class. In between of those, we draw our hyperplane so that the distance to both points

is the same and as large as possible. The two parallel lines are the support vectors. In between the orange and the blue line there are no data points. This is our margin. We want this margin to be as big as possible because it makes our predictions more reliable.

KERNELS

The data we have looked at so far is relatively easy to classify because it is clearly separated. Such data can almost never be found in the real world. Also, we are oftentimes working in higher dimensions with many features. This makes things more complicated.



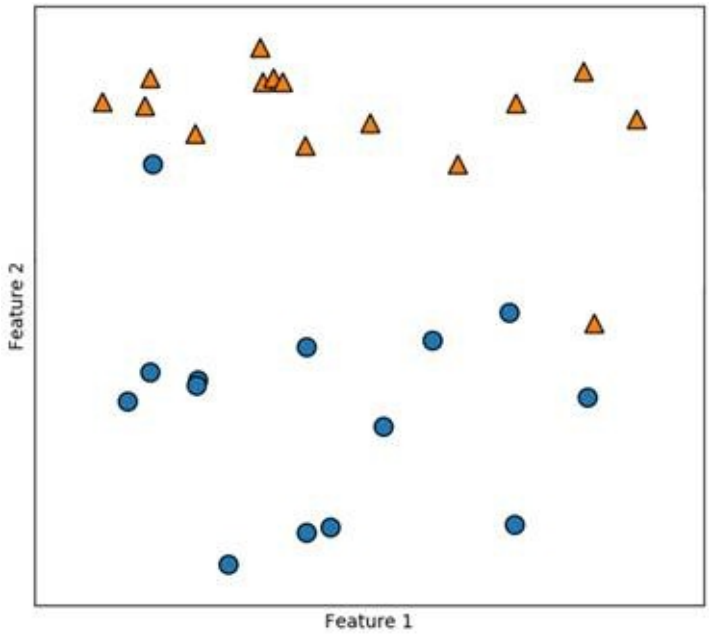
Data taken from the real world often looks like this in figure. Here it is impossible to draw a straight line, and even a quadratic or cubic function does not help us here. In such cases we can use so-called *kernels*. These add a new dimension to our data. By doing that, we hope to increase the complexity of the data and possibly use a hyperplane as a separator.

Notice that the kernel (a.k.a. the additional dimension) should be derived from the data that we already have. We are just making it more abstract. A kernel is not some random feature but a combination of the features we already have.

We can define our kernel to be whatever we want. For example, we could define it as the result of dividing feature one by feature two. But that wouldn’t be reasonable or helpful. Therefore, there are pre-defined and effective kernels that we can choose from.

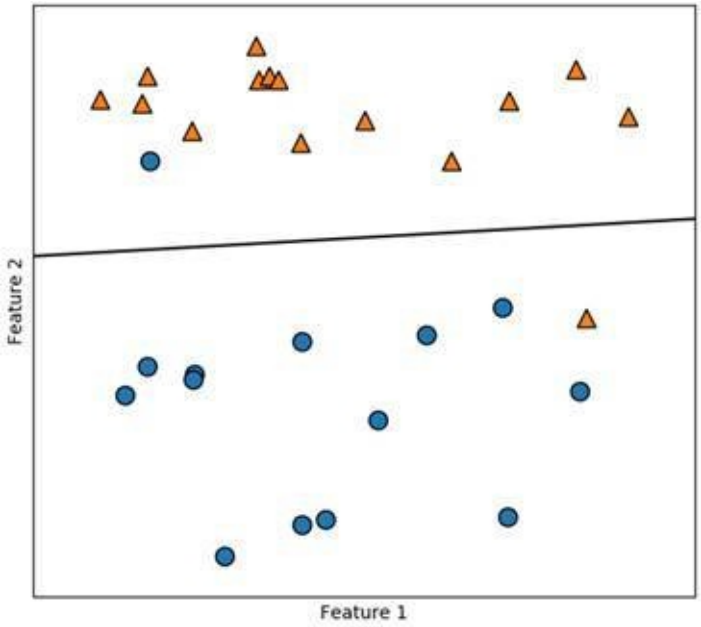
SOFT MARGIN

Sometimes, we will encounter statistical outliers in our data. It would be very easy to draw a hyperplane that separates the data into the classes, if it wasn't for these outliers.



In the figure above, you can see such a data set. We can see that almost all of the orange triangles are in the top first third, whereas almost all the blue dots are in the bottom two thirds. The problem here is with the outliers.

Now instead of using a kernel or a polynomial function to solve this problem, we can define a so-called *soft margin.* With this, we allow for conscious misclassification of outliers in order to create a more accurate model. Caring too much about these outliers would again mean overfitting the model.



As you can see, even though we are misclassifying two data points our model is very accurate.

LOADING DATA

Now that we understand how SVMs work, let's get into the coding. For this machine learning algorithm, we are going to once again use the breast cancer data set. We will need the following imports:

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
```

Besides the libraries we already know, we are importing the *SVC* module. This is the support vector classifier that we are going to use as our model. Notice that we are also importing the *KNeighborsClassifier* again, since we are going to compare the accuracies at the end.

```
data = load_breast_cancer()


X = data.data
Y = data.target


X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.1, random_state=30)
```

This time we use a new parameter named *random_state*. It is a seed that always produces the exact same split of our data. Usually, the data gets split randomly every time we run the script. You can use whatever number you want here. Each number creates a certain split which doesn't change no matter how many times we run the script. We do this in order to be able to objectively compare the different classifiers.

TRAINING AND TESTING

So first we define our support vector classifier and start training it.

```
model = SVC(kernel='linear', C=3)

model.fit(X_train, Y_train)
```

We are using two parameters when creating an instance of the SVC class. The first one is our *kernel* and the second one is *C* which is our soft margin. Here we choose a *linear* kernel and allow for three misclassifications. Alternatively we could choose *poly, rbf, sigmoid, precomputed* or a self-defined kernel. Some are more effective in certain situations but also a lot more time-intensive than linear kernels.

```
accuracy = model.score(X_test, Y_test)

print(accuracy)
```

When we now *score* our model, we will see a very good result.

0.9649122807017544

Now let's take a look at the *KNeighborsClassifier* with the same *random_state*.

```
knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(X_train, Y_train)



knn_accuracy = knn.score(X_test, Y_test)

print(knn_accuracy)
```

The result is only a tiny bit worse but when the data becomes larger and more complex, we might see a quite bigger difference.

0.9473684210526315

Play around with different *random_state* parameters. You will see that most of the time the SVC will perform better.

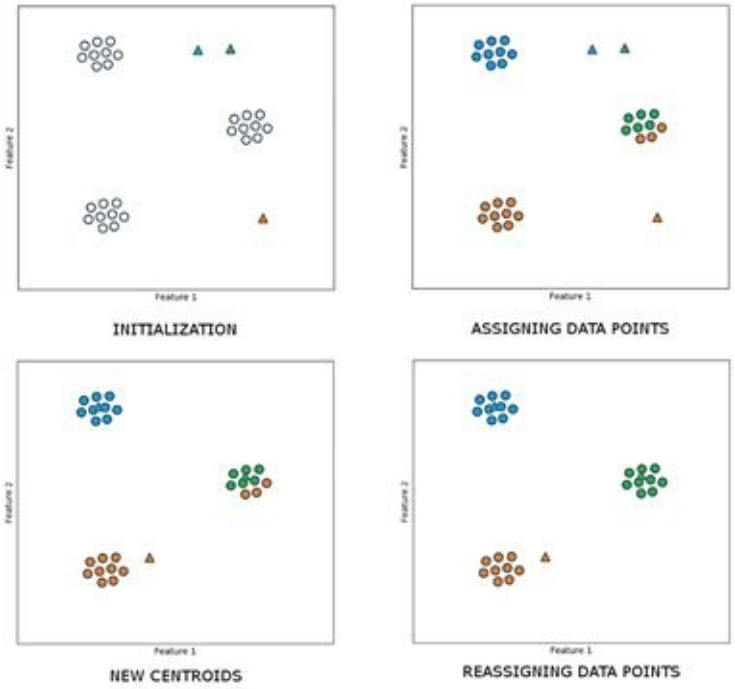
6 – CLUSTERING

Up until now, we have only looked at supervised learning algorithms. *Clustering* however is an unsupervised learning algorithm, which means that we don’t have the results for our inputs. We can’t tell our model what is right and wrong. It has to find patterns on its own.

The algorithm gets raw data and tries to divide it up into *clusters*. K-Means-Clustering is the method that we are going to use here. Similar to K-Nearest-Neighbors, the *K* states the amount of clusters we want.

HOW CLUSTERING WORKS

The clustering itself works with so-called *centroids*. These are the points, which lie in the center of the respective clusters.



The figure above illustrates quite well how clustering works. First, we randomly place the centroids somewhere in our data. This is the *initialization*. Here, we have defined three clusters, which is why we also have three centroids.

Then, we look at each individual data point and assign the cluster of the nearest centroid to it. When we have done this, we continue by realigning our centroids. We place them in the middle of all points of their cluster.

After that, we again reassign the points to the new centroids. We continue doing this over and over again until almost nothing changes anymore. Then we will hopefully end up with the optimal clusters. The result then looks like this:

Of course, you will probably never find data that looks like this in the real world. We are working with much more complex data and much more features (i.e. dimensions).

LOADING DATA

For the clustering algorithm, we will use a dataset of handwritten digits. Since we are using unsupervised learning, we are not going to classify the digits. We are just going to put them into clusters. The following imports are necessary:

```
from sklearn.cluster import KMeans

from sklearn.preprocessing import scale

from sklearn.datasets import load_digits
```

Besides the *KMeans* module and the *load_digits* dataset, we are also importing the function *scale* from the *preprocessing* library. We will use this function for preparing our data.

```
digits = load_digits()
data = scale(digits.data)
```

After loading our dataset we use the *scale* function, to standardize our data. We are dealing with quite large values here and by scaling them down to smaller values we save computation time.

TRAINING AND PREDICTING

We can now train our model in the same way we trained the supervised learning models up until now.

```
clf = KMeans(n_clusters=10, init="random", n_init=10)
clf.fit(data)
```

Here we are passing three parameters. The first one (*n_clusters*) defines the amount of clusters we want to have. Since we are dealing with the digits 0 to 9, we create ten different clusters.

With the *init* parameter we choose the way of initialization. Here we chose *random*, which obviously means that we just randomly place the centroids somewhere. Alternatively, we could use *k-means++* for intelligent placing.

The last parameter (*n_init*) states how many times the algorithm will be run with different centroid seeds to find the best clusters.

Since we are dealing with unsupervised learning here, scoring the model is not really possible. You won't be able to really score if the model is clustering right or not. We could only benchmark certain statistics like *completeness* or *homogeneity*.

What we can do however is to predict which cluster a new input belongs to.

```
clf.predict([...])
```

In this case, inputting data might be quite hard, since we would need to manually put in all the pixels. You could either try to write a script what converts images into NumPy arrays or you could work with a much simpler data set.

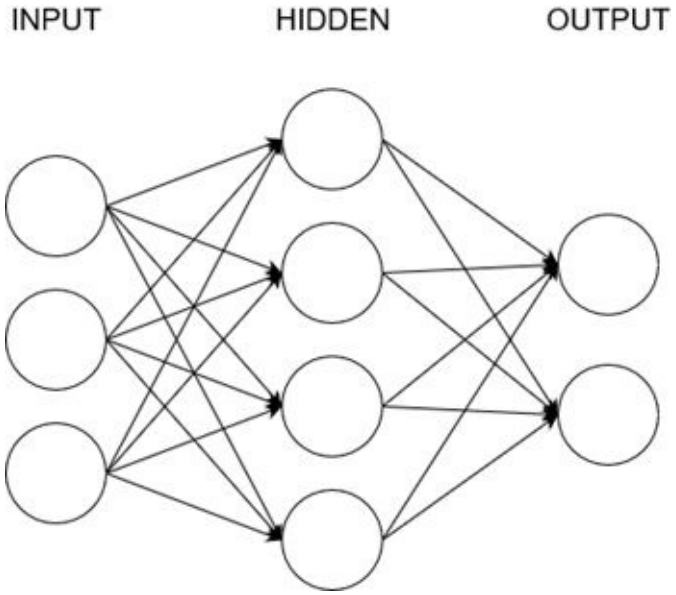
Also, since we are working with huge dimensions here, visualization is quite hard. When you work with two- or three-dimensional data, you can use the Matplotlib knowledge from volume three, in order to visualize your model.

7 – NEURAL NETWORKS

As already mentioned in the beginning, *neural networks* are a very complex and comprehensive topic. Way too comprehensive to cover it in one chapter. For this reason, I will probably write a separate book about neural networks in the future. However, in this chapter, we are going to cover the basics of neural networks and we will build a fully functioning model that classifies handwritten digits properly.

STRUCTURE OF A NEURAL NETWORK

With neural networks we are trying to build our models based on the structure of the human brain, namely with neurons. The human brain is composed of multiple billions of neurons which are interconnected. Neural networks are structures which try to use a similar principle.

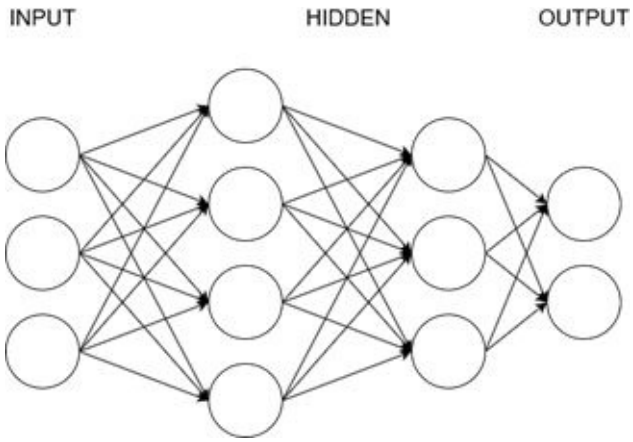


In the figure above, we can see three layers. First the *input layer*, at the end the *output layer* and in between the *hidden layer*.

Obviously the input layer is where our inputs go. There we put all the things which are being entered or sensed by the script or the machine. Basically these are our features.

We can use neural networks to classify data or to act on inputs and the output layer is where we get our results. These results might be a class or action steps. Maybe when we input a high temperature into our model, the output will be the action of cooling down the machine.

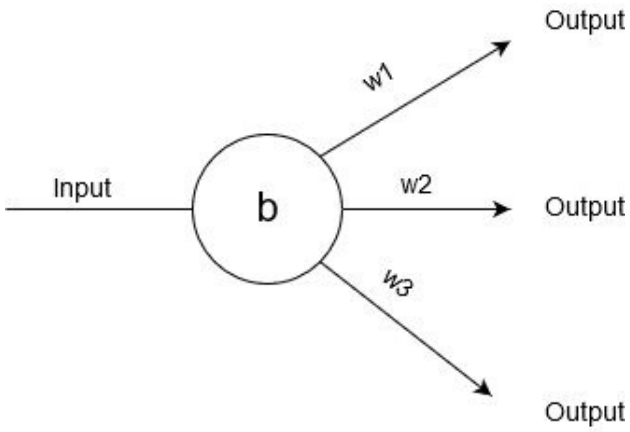
All the layers between input and output are called *hidden layers*. They make the model more abstract and more complex. They extend the internal logic. The more hidden layers and neurons you add, the more sophisticated the model gets.



Here for example we have two hidden layers, one with four neurons and one with three neurons. Notice that every neuron of a layer is connected to every neuron of the next layer.

STRUCTURE OF A NEURON

In order to understand how a neural network works in general, we need to understand how the individual neurons work.



As you can see, every neuron has a certain input, which is either the output of another neuron or the input of the first layer.

This number (which is the input) now gets multiplied by each of the *weights* (*w1, w2, w3…)*. After that, we subtract the *bias b*. The results of these calculations are the outputs of the neuron.

What I have just explained and what you can see on the picture is an outdated version of a neuron called a *perceptron*. Nowadays, we are using more complex neurons like the *sigmoid neurons* which use more sophisticated activation functions to calculate the outputs.

Now you can maybe imagine to some degree how complex these systems get when we combine hundreds of thousands of these neurons in one network.

HOW NEURAL NETWORKS WORK

But what has all this to do with artificial intelligence or machine learning? Since neural networks are structures with a huge amount of parameters that can be changed, we can use certain algorithms so that the model can adjust itself. We input our data and the desired outcome. Then the model tries to adjust its weights and biases so that we can get from our inputs to the respective outputs. Since we are dealing with multiple thousands of neurons, we can’t do all this manually.

We use different algorithms like *backpropagation* and *gradient descent*, to optimize and train our model. We are not going to deep into the mathematics here. Our focus will be on the coding and the implementation.

RECOGNIZING HANDWRITTEN DIGITS

Up until now, we always used the *sklearn* module for traditional machine learning. Because of that all our examples were quite similar. In this chapter, we will use *Tensorflow*. We will need the following imports:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

LOADING DATA

The handwritten digits data that we are going to use in this chapter is provided by the *Tensorflow Keras* datasets. It is the so-called *MNIST* dataset which contains 70,000 images of handwritten digits in the resolution of 28x28 pixels.

```
mnist = tf.keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

The *mnist* class that we import here has the function *load_data*. This function returns two tuples with two elements each. The first tuple contains our training data and the second one our test data. In the training data we can find 60,000 images and in the test data 10,000.

The images are stored in the form of NumPy arrays which contain the information about the individual pixels. Now we need to normalize our data to make it easier to handle.

```
X_train = tf.keras.utils.normalize(X_train)
X_test = tf.keras.utils.normalize(X_test)
```

By normalizing our data with the *normalize* function of the *keras.utils* module, we scale our data down. We standardize it as we have already done in the last chapter. Notice that we are only normalizing the X-values since it wouldn’t make a lot of sense to scale down our results, which are the digits from 0 to 9.

BUILDING THE NEURAL NETWORK

We have prepared our data so that we can now start building our network. Tensorflow offers us a very easy way to construct neural networks. We just create a model and then define the individual layers in it.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(units=128,
                                activation='relu'))
model.add(tf.keras.layers.Dense(units=128,
                                activation='relu'))
model.add(tf.keras.layers.Dense(units=10,
                                activation=tf.nn.softmax))
```

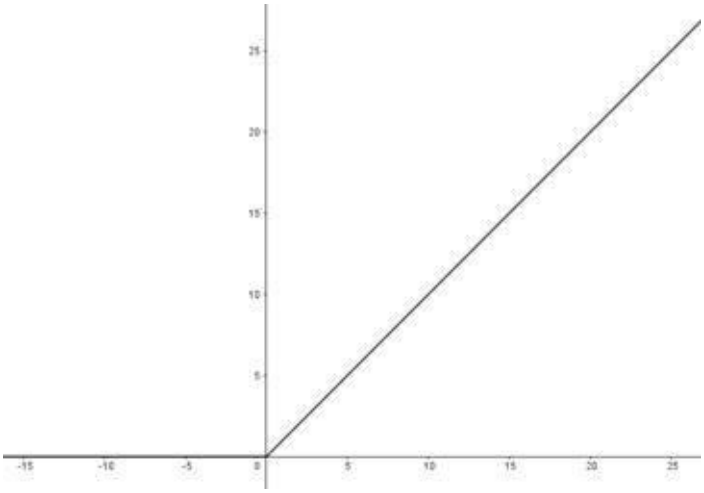
First we define our model to be a *Sequential*. This is a linear type of model where which is defined layer by layer. Once we have defined the model, we can use the *add* function to add as many different layers as we want.

The first layer that we are adding here is the input layer. In our case, this is a *Flatten* layer. This type of layer flattens the input. As you can see, we have specified the input shape of 28x28 (because of the pixels). What *Flatten* does is to transform this shape into one dimension which would here be 784x1.

All the other layers are of the class *Dense*. This type of layer is the basic one which is connected to every neuron of the neighboring layers. We always specify two parameters here. First, the *units* parameter which states the amount of neurons in this layer and second the *activation* which specifies which activation function we are using.

We have two hidden layers with 128 neurons each. The activation function that we are using here is called *relu* and stands for *rectified linear unit*. This is a very fast and a very simple function. It basically just returns zero whenever our input is negative and the input itself whenever it is positive.

$$f(x) = max(0,x)$$



We use it because it is quite simple, quite powerful, very fast and prevents negative values.

For our output layer we only use ten neurons (because we have ten possible digits to classify) and a different activation function. This one is called *softmax* and what it does is it picks output values so that all of our end results add up to one. Because of this we are getting ten different probabilities for each digit, indicating its likelihood.

Our model is now defined but before we can start working with it, we have to compile it first. By doing this we define certain parameters and configure it for training and testing.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Here we define three things, namely the *optimizer*, the *loss function* and the *metrics* that we are interested in. We are not going to go into the math of the *adam* optimizer or the *sparse_categorical_crossentropy* loss function. However, these are very popular choices, especially for tasks like this one.

TRAINING AND TESTING

The training and testing of the model is quite simple and very similar to the way we did it with *sklearn* models.

```
model.fit(X_train, Y_train, epochs=3)

loss, accuracy = model.evaluate(X_test, Y_test)

print('Loss: ', loss)

print('Accuracy: ', accuracy)
```

We use the *fit* function to train our model but this time we have an additional parameter named *epochs*. The number of epochs is the number of times that we feed the same data into the model over and over again. By using the *evaluate* function, we get two values – *loss* and *accuracy*.

The accuracy is the percentage of correctly classified digits and therefore we want to keep it as high as possible. The loss on the other hand is not a percentage but a summation of the errors made that we want to minimize.

When we run our test, we have a pretty high accuracy around 97 percent.

PREDICTING YOUR OWN DIGITS

Now that we have trained such an awesome model, we of course want to play with it. So what we are going to do is to read in our own images of handwritten digits and predict them with our model.

For this, you can either use software like Paint or Gimp and draw 28x28 pixel images or you can scan digits from real documents into your computer and scale them down to this size. But we will need an additional library here that we haven't installed yet.

```
pip install opencv-python
```

This library is called *OpenCV* and is mainly used for computer vision. However, we will use it in order to read in our images. The import looks a little bit different than the installation.

```
import cv2 as cv
```

Now the only thing we need to prepare is a 28x28 image of a handwritten digit. Then we can start coding.

```
image = cv.imread('digit.png')[:,:,0]

image = np.invert(np.array([image]))
```

We use the *imread* method to read our image into the script. Because of the format we remove the last dimension so that everything matches with the input necessary for the neural network. Also, we need to convert our image into a NumPy array and invert it, since it will confuse our model otherwise.

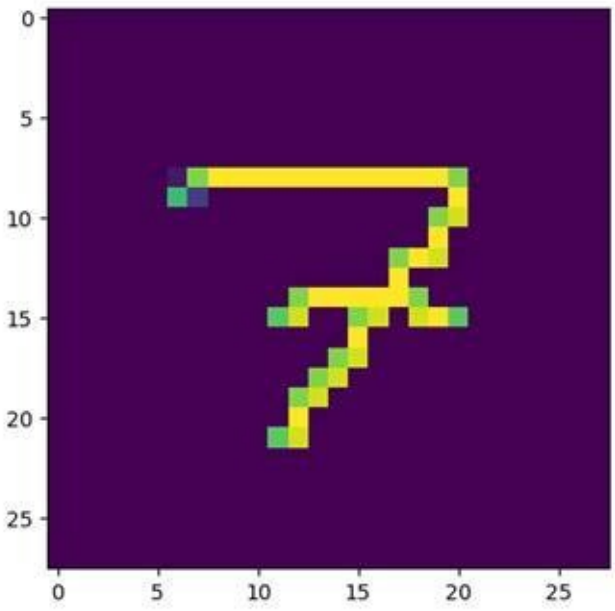
Finally, we just have to use the *predict* function of our model on our image and see if it works.

```
prediction = model.predict(image)

print("Prediction: {}".format(np.argmax(prediction)))

plt.imshow(image[0])

plt.show()
```

The prediction we get is an array of the ten different probabilities calculated by the *softmax* activation function. By using the *argmax* method, we get the index of the highest probability, which at the same time is the respective digit.

Last but not least, we use Matplotlibs *imshow* function, to display the image that we just scanned. The result is very satisfying.

Prediction: 7



If you are looking for a little challenge, you can try to do the same thing by using a Support Vector Machine. Usually it performs better than a neural network at this particular task. Otherwise just play around with some digits and have fun.

8 – OPTIMIZING MODELS

In this final chapter we will talk about saving, loading and optimizing our models. Up until now we always loaded our data, trained and tested our model and then used it. But sometimes (as you have probably noticed with neural networks) the training is very time-intensive and we don’t want to train the model every time we run the script again. Training it one time is enough.

SERIALIZATION

For this reason, there is the concept of *serialization*. We use it to save objects into files during runtime. By doing this, we are not only saving the attributes but the whole state of the object. Because of that, we can load the same object back into a program later and continue working with it.

To work with serialization in Python, we need to import the module *pickle*:

```
import pickle
```

SAVING MODELS

As our example, we will use the breast cancer classification script that uses SVMs.

```
import pickle

from sklearn.svm import SVC

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split


data = load_breast_cancer()


X = data.data

Y = data.target


X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.1)


model = SVC(kernel='linear', C=3)

model.fit(X_train, Y_train)
```

Here we have fully trained our model and we could also go ahead and score it but we don’t want to use it right now. We want to save it so that we can use it in the future whenever we need it.

```
with open('model.pickle', 'wb') as file:
```

```
pickle.dump(model, file)
```

We are opening a file stream in the *write bytes* mode. Then we use the *dump* function of pickle, to save our model into the specified file.

LOADING MODELS

Loading our model is now quite simple. We can write a completely new script and use the model there.

```
import pickle


with open('model.pickle', 'rb') as file:

    model = pickle.load(file)


model.predict([...])
```

Here we open a file stream in the *read bytes* mode and use the *load* function of the pickle module to get the model into our program. We can then just continue and work with it.

OPTIMIZING MODELS

We can use this serialization principle in order to train our model in the best way possible and to optimize it.

```
best_accuracy = 0


for x in range(2500):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1)


    model = SVC(kernel='linear', C=3)
    model.fit(X_train, Y_train)
    accuracy = model.score(X_test, Y_test)
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        print("Best accuracy: ", accuracy)
        with open('model.pickle', 'wb') as file:
            pickle.dump(model, file)
```

The concept is quite simple. We define a variable *best_accuracy* which starts with the value zero. Then we run a loop with 2500 iterations and we train our model over and over again with a different split for our data and different seeds.

When we test the model, we check if the accuracy is higher than the highest measured accuracy (starting with zero). If that is the case, we save the model and update the best accuracy. By doing this, we find the model with the highest accuracy.

Notice that we are still only using our training data and our test data. This means that if we take things too far, we might overfit the model, especially with simple datasets. It is not impossible to reach an accuracy of 100% but the question remains if this accuracy also applies to unknown data.

WHAT'S NEXT?

You've come quite far! You've finally finished volume four of this Python Bible series. We covered a lot of machine learning material, from basic linear regression, over support vector machines to neural networks. You have every right to be proud of yourself that you have made it that far.

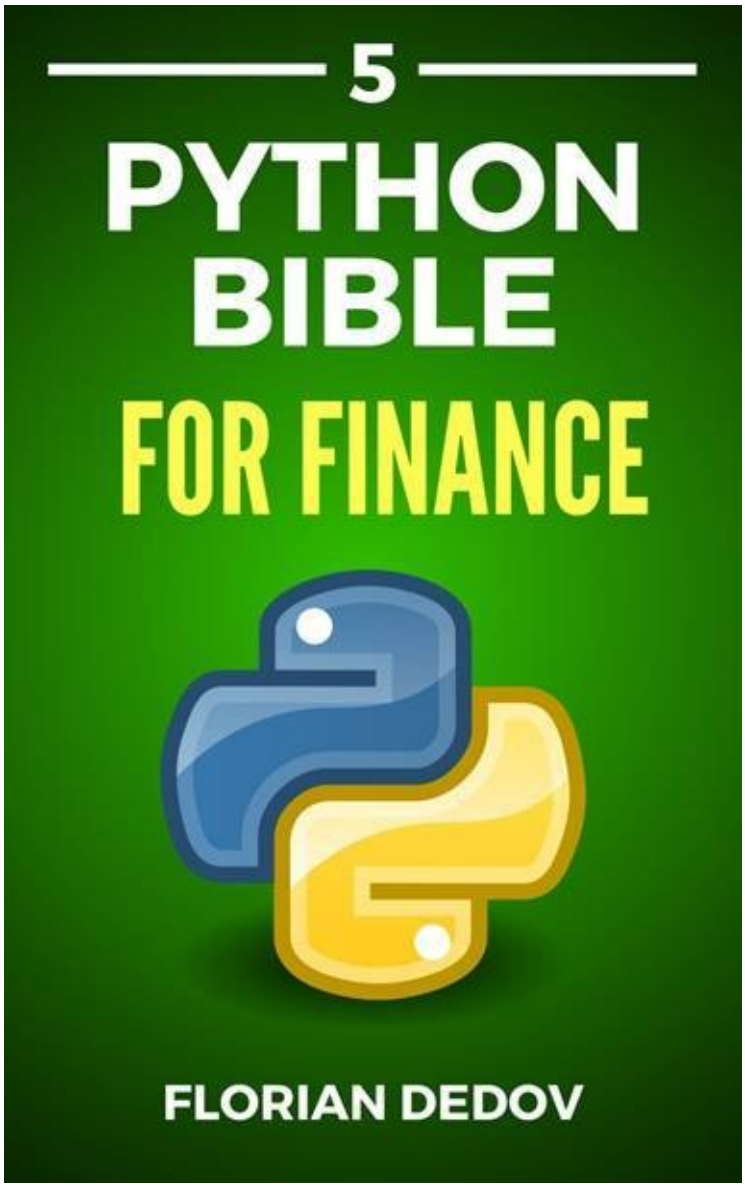
The skills you possess right now are crucial in today's economy. They are rare and very valuable. You are not only capable of developing advanced Python scripts but also of developing awesome and powerful machine learning models that can be applied to complex real-life problems.

I encourage you to continue your journey. Even though you have already learned quite a lot, we don't stop here. There are a lot of topics to be covered yet and you should not miss out on them. Practice what you've learned, play around and experiment!

I wish you a lot of success on your programming journey! Stay tuned and prepare for the next volume!

Last but not least, a little reminder. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!


5
PYTHON
BIBLE
FOR FINANCE
FLORIAN DEDOV

THE
PYTHON BIBLE

VOLUME FIVE

FINANCE

BY

FLORIAN DEDOV

TABLE OF CONTENT

INTRODUCTION

Who wants to build long-term wealth needs to invest his capital. But nowadays investing isn't done in the same way as it was a couple of decades ago. Nowadays everything works with computers, algorithms, data science and machine learning. We already know that Python is the lingua franca of these fields.

In the last volumes we learned a lot about data science and machine learning but we didn't apply these to anything from the real world except for some public datasets for demonstration. This book will focus on applying data science and machine learning onto financial data. We are going to load stock data, visualize it, analyze it and also predict share prices.

Notice however that finance and investing always involves risk and you should be very careful with what you are doing. I am not taking any responsibility for what you are doing with your money. In this book we are only going to talk about the financial analysis with Python.

After reading this book you will be able to apply the advanced Python knowledge and the machine learning expertise that you've already got to the finance industry. Take time while reading this book and code along. You will learn much more that way. I wish you a lot of fun and success with this fifth volume.

Just one little thing before we start. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you have learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

1 – INSTALLING MODULES

For this book we are going to use some libraries we already know from the previous volumes but also some new ones. We will need the following list:

- Pandas
- Pandas-Datareader
- Matplotlib
- MPL-Finance
- NumPy
- Scikit-Learn
- Beautifulsoup4

Now let’s take a look at the individual libraries. We will recap the ones we already know and also explain what we will use the new ones for.

PANDAS

Pandas is a library that we have already used in the past two volumes. It offers us the powerful data structure named data frame. With Pandas we can manage our data in a similar way to SQL tables or Excel sheets.

PANDAS-DATAREADER

The *Pandas-Datareader* is an additional library which we are going to use, in order to load financial data from the internet. It loads data from APIs into data frames.

MATPLOTLIB

Matplotlib is a library that we use to visualize our data and our models. We can choose from a variety of different plotting types and styles.

MPL-FINANCE

MPL-Finance is a library that works together with Matplotlib and allows us to use special visualization for finance. We will use it for plotting candlestick charts.

NUMPY

NumPy is our fundamental module for linear algebra and dealing with arrays. It is necessary for Matplotlib, Pandas and Scikit-Learn.

SCIKIT-LEARN

Scikit-Learn is the module that we have used in the last volume of The Python Bible series. It offers us a lot of different classic and traditional machine learning models. We are going to apply these models to our financial data in order to make predictions.

BEAUTIFULSOUP4

Last but not least, we are using a new module with the name *beautifulsoup4*. I admit that this is a pretty stupid and misleading name but this library is a powerful web scraping library. We are going to use it in order to extract financial data out of HTML files.

INSTALLATION

These are the installation commands (with pip) for the necessary libraries:

```
pip install pandas

pip install pandas-datareader

pip install matplotlib

pip install mpl-finance

pip install scikit-learn

pip install beautifulsoup4
```

2 – LOADING FINANCIAL DATA

Now that we have installed the necessary libraries we are going to start by taking a look at how to load financial data into our script. For this, we will need the following imports:

```
from pandas_datareader import data as web
import datetime as dt
```

We are importing the *data* module of the *pandas_datareader* library with the alias *web*. This module will be used to get our data from the Yahoo Finance API. Also, we are importing the *datetime* module so that we can specify time frames. To do that, we use the *datetime* function.

```
start = dt.datetime(2017,1,1)
end = dt.datetime(2019,1,1)
```

Here we have defined a start date and an end date. This is our timeframe. When we load the data, we want all entries from the 1st of January 2017 up until the 1st of January 2019. Alternatively, we could also use the *datetime.now* function, to specify the present as the end date.

```
end = dt.datetime.now()
```

The next step is to define a data frame and to load the financial data into it. For this we need to know four things. First: The ticker symbol of the stock we want to analyze. Second: The name of the API we want to receive the data from. And last: The start and end date.

```
df = web.DataReader('AAPL', 'yahoo', start, end)
```

We are creating an instance of *DataReader* and we pass the four parameters. In this case, we are using the Yahoo Finance API, in order to get the financial data of the company Apple (AAPL) from start date to end date.

To now view our downloaded data, we can print a couple of entries.

```
                High         Low  ...      Volume   Adj Close
```

```
Date                       ...
2017-01-03  116.330002  114.760002  ...  28781900.0  111.286987
2017-01-04  116.510002  115.750000  ...  21118100.0  111.162437
2017-01-05  116.860001  115.809998  ...  22193600.0  111.727715
2017-01-06  118.160004  116.470001  ...  31751900.0  112.973305
2017-01-09  119.430000  117.940002  ...  33561900.0  114.008080
```

Warning: Sometimes the Yahoo Finance API won't respond and you will get an exception. In this case, your code is not the problem and you can solve the problem by waiting a bit and trying again.

As you can see, we now have a data frame with all the entries from start date to end date. Notice that we have multiple columns here and not only the closing share price of the respective day. Let's take a quick look at the individual columns and their meaning.

Open: That's the share price the stock had when the markets opened that day.

Close: That's the share price the stock had when the markets closed that day.

High: That's the highest share price that the stock had that day.

Low: That's the lowest share price that the stock had that day.

Volume: Amount of shares that changed hands that day.

Adj. Close: The adjusted close value that takes things like stock splits into consideration.

READING INDIVIDUAL VALUES

Since our data is stored in a Pandas data frame, we can use the indexing we already know, to get individual values. For example, we could only print the closing values.

```
print(df['Close'])
```

Also, we can go ahead and print the closing value of a specific date that we are interested in. This is possible because the date is our index column.

```
print(df['Close']['2017-02-14'])
```

But we could also use simple indexing to access certain positions.

```
print(df['Close'][5])
```

Here we would print the closing price of the fifth entry.

SAVING AND LOADING DATA

With Pandas we can now save the financial data into a file so that we don't have to request it from the API every time we run our script. This just costs time and resources. For this we can use a bunch of different formats.

CSV

As we have already done in the previous volumes, we can save our Pandas data frame into a CSV file.

```
df.to_csv('apple.csv')
```

This data can then be viewed by using an ordinary text editor or a spreadsheet application. The default setting is to separate the entries by commas. We can change that by specifying a separator in case our values contain commas.

```
df.to_csv('apple.csv', sep=";")
```

Here we would separate our data by semi-colons.

EXCEL

If we want to put our data into an Excel sheet, we can use the *to_excel* function.

```
df.to_excel('apple.xlsx')
```

When we open that file in Excel, it looks like this:

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2017-01-03 0:00:00	115.8	116.33	114.76	116.15	112.14	28781900
3	2017-01-04 0:00:00	115.85	116.51	115.75	116.02	112.0145	21118100
4	2017-01-05 0:00:00	115.92	116.86	115.81	116.61	112.5841	22193600
5	2017-01-06 0:00:00	116.78	118.16	116.47	117.91	113.8392	31751900
6	2017-01-09 0:00:00	117.95	119.43	117.94	118.99	114.882	33561900
7	2017-01-10 0:00:00	118.77	119.38	118.3	119.11	114.9978	24462100
8	2017-01-11 0:00:00	118.74	119.93	118.6	119.75	115.6157	27588600
9	2017-01-12 0:00:00	118.9	119.3	118.21	119.25	115.133	27086200
10	2017-01-13 0:00:00	119.11	119.62	118.81	119.04	114.9302	26111900
11	2017-01-17 0:00:00	118.34	120.24	118.22	120	115.8571	34439800

We can now analyze it further in the spreadsheet application.

HTML

If for some reason we need our data to be shown in browsers, we can also export them into HTML files.

```
df.to_html('apple.html')
```

The result is a simple HTML table.

JSON

Finally, if we are working with JavaScript or just want to save the data into that format, we can use JSON. For this, we use the *to_json* function.

```
df.to_json('apple.json')
```

LOADING DATA FROM FILES

For every file format we also have a respective loading or reading function. Sometimes we will find data in HTML format, sometimes in JSON format. With pandas we can read in the data easily.

```
df = pd.read_csv("apple.csv", sep=";")
df = pd.read_excel("apple.xlsx")
df = pd.read_html("apple.html")
df = pd.read_json("apple.json")
```

3 – GRAPHICAL VISUALIZATION

Even though tables are nice and useful, we want to visualize our financial data, in order to get a better overview. We want to look at the development of the share price. For this, we will need Matplotlib.

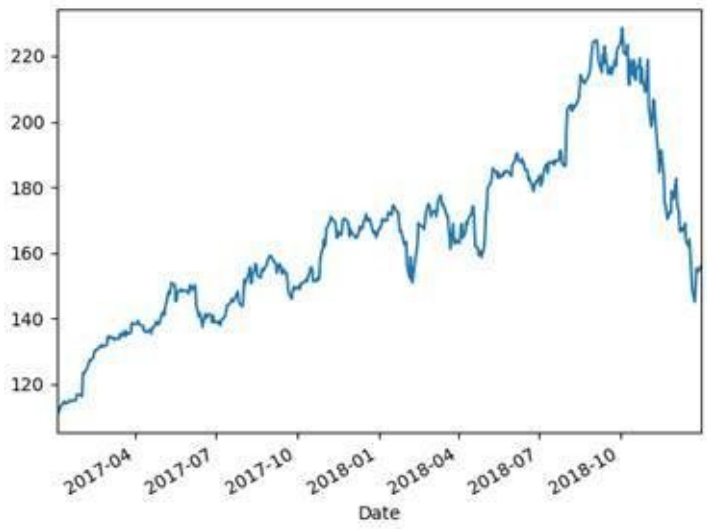
```
import matplotlib.pyplot as plt
```

PLOTTING DIAGRAMS

Actually plotting our share price curve with Pandas and Matplotlib is very simple. Since Pandas builds on top of Matplotlib, we can just select the column we are interested in and apply the *plot* method.

```
df['Adj Close'].plot()
plt.show()
```

The results are amazing. Since the date is the index of our data frame, Matplotlib uses it for the x-axis. The y-values are then our adjusted close values.



As you can see, with just two lines of code we plotted the two-year development of the Apple share price.

PLOTTING STYLE

Now we can improve the style of our plot. For this, let's start by choosing a style. At the following page you can take a look at the different pre-defined Matplotlib styles.

Styles: https://bit.ly/2OSCTdm

But before we can apply one, we will need to import the *style* module from Matplotlib.

```
from matplotlib import style
```

For our case, *ggplot* is probably the best suited style. It has a grid, nice colors and it looks smooth.

```
style.use('ggplot')
```

The next thing is our labeling. Whereas our x-axis is already labeled, our y-axis isn't and we are also missing a title.

```
plt.ylabel('Adjusted Close')
plt.title('AAPL Share Price')
```

Let's take a look at our graph now.

This looks much better. It is now way easier to understand what these values mean. However, there is a much better way to plot financial data. But this will be the topic of the next chapter.

COMPARING STOCKS

As we already know, we can plot multiple graphs into one figure. We can use this in order to compare the share price development of two companies.

```
style.use('ggplot')

start = dt.datetime(2017,1,1)
end = dt.datetime(2019,1,1)

apple = web.DataReader('AAPL', 'yahoo', start, end)
facebook = web.DataReader('FB', 'yahoo', start, end)

apple['Adj Close'].plot(label="AAPL")
facebook['Adj Close'].plot(label="FB")
plt.ylabel('Adjusted Close')
plt.title('Share Price')
plt.legend(loc='upper left')
plt.show()
```

Here we load the financial data of Apple into one data frame and the data of Facebook into another one. We then plot both curves. Notice that we are specifying a *label*, which is important for the legend that helps us to distinguish between the two companies. By using the *legend* function, we can activate the legend and specify its location. The result looks like this:

It looks pretty good. The problem here is that this only works because the share prices are quite similar here. If we would compare Apple to a company like Amazon or Alphabet, which shares cost around 1000 to 2000 dollars each, the

graph wouldn’t give us much information. In that case, we could work with subplots.

```
apple = web.DataReader('AAPL', 'yahoo', start, end)
amazon = web.DataReader('AMZN', 'yahoo', start, end)

plt.subplot(211)
apple['Adj Close'].plot(color='blue')
plt.ylabel('Adjusted Close')
plt.title('AAPL Share Price')

plt.subplot(212)
amazon['Adj Close'].plot()
plt.ylabel('Adjusted Close')
plt.title('AMZN Share Price')

plt.tight_layout()
plt.show()
```

What we do here is creating two subplots below each other instead of plotting the two graphs into one plot. We define a subplot for Apple and one for Amazon. Then we label them and at the end we use the *tight_layout* function, to make things prettier. This is what we end up with:

Now even though the share prices are radically different, we can compare the development of the two stocks by looking at the two graphs.

4 – CANDLESTICK CHARTS

The best way to visualize stock data is to use so-called *candlestick charts*. This type of chart gives us information about four different values at the same time, namely the high, the low, the open and the close value. In order to plot candlestick charts, we will need to import a function of the MPL-Finance library.

```
from mpl_finance import candlestick_ohlc
```

We are importing the *candlestick_ohlc* function. Notice that there also exists a *candlestick_ochl* function that takes in the data in a different order.

Also, for our candlestick chart, we will need a different date format provided by Matplotlib. Therefore, we need to import the respective module as well. We give it the alias *mdates*.

```
import matplotlib.dates as mdates
```

PREPARING THE DATA

Now in order to plot our stock data, we need to select the four columns in the right order.

```
apple = apple[['Open','High','Low','Close']]
```

Now, we have our columns in the right order but there is still a problem. Our date doesn't have the right format and since it is the index, we cannot manipulate it. Therefore, we need to reset the index and then convert our *datetime* to a number.

```
apple.reset_index(inplace=True)
apple['Date'] = apple['Date'].map(mdates.date2num)
```

For this, we use the *reset_index* function so that we can manipulate our *Date* column. Notice that we are using the *inplace* parameter to replace the data frame by the new one. After that, we map the *date2num* function of the *matplotlib.dates* module on all of our values. That converts our dates into numbers that we can work with.

PLOTTING THE DATA

Now we can start plotting our graph. For this, we just define a subplot (because we need to pass one to our function) and call our *candlestick_ohlc* function.

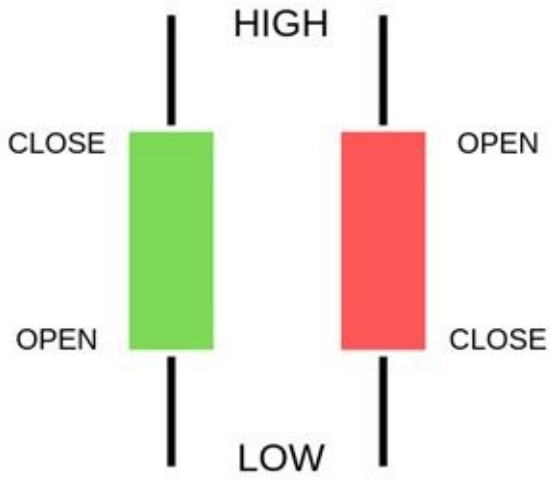
```
ax = plt.subplot()
candlestick_ohlc(ax, apple.values,
                 width=5,
                 colordown='r', colorup='g')
ax.grid()
ax.xaxis_date()
plt.show()
```

Besides the subplot, we are also passing a couple of other parameters here. First of all, our four financial values. We access these by referring to the *values* attribute of our data frame. Additionally, we define the width of the individual candlesticks and the colors for upward movements and downward movements. Last but not least, we turn on the grid and we define the x-axis as the date axis and our numerical values get displayed as dates again. This is our result:

This might look a bit confusing right now but don’t worry we will take care about that in a minute.

THE CANDLESTICK

First of all, we need to understand what a candlestick is and how it works.



As you can see, we have two types of candlesticks – the green one and the red one. If the share price went up on that particular day, the candlestick is green. Otherwise it is red.

One candlestick gives us the information about all four values of one specific day. The highest point of the stick is the *high* and the lowest point is the *low* of that day. The colored area is the difference between the *open* and the *close* price. If the stick is green, the close value is at the top and the open value at the bottom, since the close must be higher than the open. If it is red, it is the other way around.

PLOTTING MULTIPLE DAYS

Another thing that we can do with this kind of plot is to plot the open, high, low and close of multiple days. For this, we can take one value like the *adjusted close* and calculate the four values for a specific amount of time.

```
apple = web.DataReader('AAPL', 'yahoo', start, end)
apple_ohlc = apple['Adj Close'].resample('10D').ohlc()

apple_ohlc.reset_index(inplace=True)
apple_ohlc['Date'] = apple_ohlc['Date'].map(mdates.date2num)
```

By using the *resample* function we stack the data in a specific time interval. In this case, we take ten days (10D). At the end we apply the *ohlc* function, to get the four values out of our entries. Then we again have to convert our date into a

numerical format.

Additionally, we are going to create a second subplot this time, which displays the volume for these days.

```
apple_volume = apple['Volume'].resample('10D').sum()
```

This time however we are using the *sum* function, since we don't want to plot another candlestick chart but only the volume of these ten days.

Now we need to define the two subplots. For this, we are using the function *subplot2grid* which makes it easier to align the plots.

```
ax1 = plt.subplot2grid((6,1),(0,0),
                       rowspan=4, colspan=1)
ax2 = plt.subplot2grid((6,1),(4,0),
                       rowspan=2, colspan=1,
                       sharex=ax1)
ax1.xaxis_date()
```

The first tuple here (6,1) states the amount of rows and columns of the window. Here we define six rows and one column. The second tuple defines at which point the subplots start. The first one takes row one and column one, whereas the second one takes row four and column one. The parameters *rowspan* and *colspan* define across how many rows and columns our plots shall stretch. Also notice that we define that both subplots share the x-axis.

```
candlestick_ohlc(ax1, apple_ohlc.values, width=5,
                 colorup='g', colordown='r')
ax2.fill_between(apple_volume.index.map(mdates.date2num),
                 apple_volume.values)

plt.tight_layout()
plt.show()
```

We again just use the same function to plot our candlestick charts but this time we use our ten day values. Also, we plot our volumes on the second subplot by using the *fill_between* function. This creates a type of chart that fills the area below the graph. Our x-values here are the converted dates and our y-values are the volumes. This is the result:

Since we only have one tenth the amount of values now, things are way more readable. We can see how the share price develops in a ten day interval.

5 – ANALYSIS AND STATISTICS

Now let's get a little bit deeper into the numbers here and away from the visual. From our data we can derive some statistical values that will help us to analyze it.

100 DAY MOVING AVERAGE

For this chapter, we are going to derive the *100 day moving average*. It is the arithmetic mean of all the values of the past 100 days. Of course this is not the only key statistic that we can derive, but it is the one we are going to use now. You can play around with other functions as well.

What we are going to do with this value is to include it into our data frame and to compare it with the share price of that day.

For this, we will first need to create a new column. Pandas does this automatically when we assign values to a column name. This means that we don't have to explicitly define that we are creating a new column.

```
apple['100d_ma'] = apple['Adj Close'].rolling(window = 100, min_periods = 0).mean()
```

Here we define a new column with the name *100d_ma*. We now fill this column with the mean values of every 100 entries. The *rolling* function stacks a specific amount of entries, in order to make a statistical calculation possible. The *window* parameter is the one which defines how many entries we are going to stack. But there is also the *min_periods* parameter. This one defines how many entries we need to have as a minimum in order to perform the calculation. This is relevant because the first entries of our data frame won't have a hundred previous to them. By setting this value to zero we start the calculations already with the first number, even if there is not a single previous value. This has the effect that the first value will be just the first number, the second one will be the mean of the first two numbers and so on, until we get to a hundred values.

By using the *mean* function, we are obviously calculating the arithmetic mean. However, we can use a bunch of other functions like *max, min* or *median* if we

like to.

NAN-VALUES

In case we choose another value than zero for our *min_periods* parameter, we will end up with a couple of *NaN-Values*. These are *not a number* values and they are useless. Therefore, we would want to delete the entries that have such values.

```
apple.dropna(inplace=True)
```

We do this by using the *dropna* function. If we would have had any entries with *NaN* values in any column, they would now have been deleted. We can take a quick look at our data frame columns.

```
print(apple.head())
```

```
                 High        Low  ...   Adj Close     100d_ma
Date                              ...
2017-01-03  116.330002  114.760002  ...  111.286987  111.286987
2017-01-04  116.510002  115.750000  ...  111.162437  111.224712
2017-01-05  116.860001  115.809998  ...  111.727715  111.392380
2017-01-06  118.160004  116.470001  ...  112.973305  111.787611
2017-01-09  119.430000  117.940002  ...  114.008080  112.231705
```

VISUALIZATION

To make this statistic more readable and in order to compare it to our actual share prices, we are going to visualize them. Additionally, we are also going to plot our volumes again. This means that we will end up with an overview of the share price compared to our 100 day moving average and of how many shares changed their owners. For this, we will again use two subplots.

```
ax1 = plt.subplot2grid((6,1),(0,0),
                       rowspan=4, colspan=1)
ax2 = plt.subplot2grid((6,1),(4,0),
                       rowspan=2, colspan=1,
                       sharex=ax1)
```

Again we use the same proportions here. Our main plot will take up two thirds of

the window and our volume plot will take up one third. Now we just need to plot the values on the axes.

```
ax1.plot(apple.index, apple['Adj Close'])
ax1.plot(apple.index, apple['100d_ma'])
ax2.fill_between(apple.index, apple['Volume'])

plt.tight_layout()
plt.show()
```

The result is a very nice overview over price, volume and our statistical value.

ADDITIONAL KEY STATISTICS

Of course there are a lot of other statistical values that we can calculate. This chapter was focusing on the way of implementation. However, let us take a quick look at two other statistical values.

PERCENTAGE CHANGE

One value that we can calculate is the percentage change of that day. This means by how many percent the share price increased or decreased that day.

```
apple['PCT_Change'] = (apple['Close'] - apple['Open']) / apple['Open']
```

The calculation is quite simple. We create a new column with the name *PCT_Change* and the values are just the difference of the closing and opening values divided by the opening values. Since the open value is the beginning value of that day, we take it as a basis. We could also multiply the result by 100 to get the actual percentage.

HIGH LOW PERCENTAGE

Another interesting statistic is the high low percentage. Here we just calculate the difference between the highest and the lowest value and divide it by the closing value. By doing that we can get a feeling of how volatile the stock is.

```
apple['HL_PCT'] = (apple['High'] - apple['Low']) / apple['Close']
```

These statistical values can be used with many others to get a lot of valuable information about specific stocks. This improves the decision making.

6 – S&P 500 INDEX

When we talk about how the markets are doing, we are usually looking at indices. One of the most important indices is the so-called *S&P 500* index which measures the stock performance of the 500 largest companies listed on the US stock exchanges.

Up until now, we always downloaded financial data for individual stocks from the internet. But when we are doing larger calculations many times, it would be preferable to not need to bother the Yahoo Finance API every time.

For this reason, we can download the stock data of the 500 companies which are represented in the S&P 500 right now and save them into files. We can then use these files instead of making requests to the API all the time.

WEBSCRAPING

The Yahoo Finance API doesn't offer any possibilities to request all the companies of the S&P 500 index. Therefore, we will need to get the information about which companies are represented from somewhere else. And for this, we will need something called *webscraping*.

With webscraping we are reading the HTML files of a website, in order to extract some specific information we are looking for. In this case, we are going to use the Wikipedia page of the list of S&P 500 companies to get the information we need.

Link: https://en.wikipedia.org/wiki/List_of_S%26P_500_companies

On this page, we can find a table with all the 500 companies and different information about them. We can see the name of the company, the industry, the headquarters location and some more things. What we need however is the ticker symbol, which we can find in the first column (make sure you take a look at the page yourself, since the structure of the table might change from time to time).

To now understand how we can extract the information out of this website, we need to look at the HTML code. For this, we go into our browser, make a right

click and view the source code of the page.

```
<table class="wikitable sortable" id="constituents">

<tbody><tr>
<th><a href="/wiki/Symbol" title="Symbol">Symbol</a>
</th>
<th>Security</th>
<th><a href="/wiki/SEC_filing" title="SEC filing">SEC f
<th><a href="/wiki/Global_Industry_Classification_Stand
<th>GICS Sub Industry</th>
<th>Headquarters Location</th>
<th>Date first added</th>
<th><a href="/wiki/Central_Index_Key" title="Central In
<th>Founded
</th></tr>
<tr>
```

There we can see a *table* with table rows (*tr*) and within these rows we have table data (*td*). So we can find a way to filter the elements.

EXTRACTING THE DATA

For webscraping with Python we will need the library *beautifulsoup4*, which we installed in the beginning of this book. Also, we will need the library *requests*, which is built-in into Core-Python. We will use requests to make HTML requests and beautifulsoup4 to extract data out of the responses.

```
import bs4 as bs
import requests
```

First we need to get the HTML code into our program. For this, we make a request.

```
link = 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
response = requests.get(link)
```

We use the *get* function to make a HTTP request to the website and it returns a response which we save into a variable.

Now we need to create a *soup object*, in order to parse the content of the response.

```
soup = bs.BeautifulSoup(response.text, 'lxml')
```

We use the *BeautifulSoup* constructor to create a new soup object. As the first parameter we pass the *text* attribute of the response. The second parameter defines the parser that we choose. In this case, we pick *lxml* which is the default choice. However, it may be the case that it is not installed on your computer.

Then you just need to install it using pip as always.

When we have done that, we define a *table* object which filters the HTML file and returns only the table we are looking for.

```
table = soup.find('table', {'class': 'wikitable sortable'})
```

We use the *find* function of our soup object to find a *table* element. Also, we pass a dictionary with the requirements. In this case, we want a table, which has the classes *wikitable* and *sortable*.

```
68 <table class="wikitable sortable" id="constituents">
```

If you want to exclude other tables on this side, you can also define the *id* if you want.

What we now do is creating an empty list for our ticker symbols. We then fill this list with the entries from the table.

```
for row in table.findAll('tr')[1:]:
    ticker = row.findAll('td')[0].text[:-1]
    tickers.append(ticker)
```

By using the *findAll* method we get all elements which are a table row. We then select every element except for the first one, since it is the header. Then we use the same function to get all the table data elements of the first column (index zero). Notice that we are using the `[:-1]` notation here to cut of the last two letters, since they contain a new line escape character. Finally, we save our tickers into our array.

To make our script more readable and modular let’s put all of our code into a function.

```
def load_sp500_tickers():

    link = 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
    response = requests.get(link)

    soup = bs.BeautifulSoup(response.text, 'lxml')

    table = soup.find('table', {'class': 'wikitable sortable'})

    tickers = []

    for row in table.findAll('tr')[1:]:
```

```
    ticker = row.findAll('td')[0].text[:-1]
    tickers.append(ticker)

  return tickers
```

Now we have our code in a function, which we can call whenever we need it. This is useful because we are going to extend our program a bit further.

SERIALIZING TICKERS

So that we do not have to scrape our ticker list over and over again, we will save it locally and then read it out whenever we need it. We do that by serializing IT. During serialization, we save an object, including the current state, in a file. Then we can reload it in exactly that state whenever we want. For the serialization, we use pickle, which we know from previous volumes.

```
import pickle
```

We now add two lines to our function before the return statement. These are responsible for serializing the ticker object.

```
with open("sp500tickers.pickle", 'wb') as f:
  pickle.dump(tickers, f)
```

Now when we scrape our tickers from the Wikipedia page once, we can save them in a file, to reload them whenever we want. But since the list is changing from time to time, we might have to update it.

LOADING SHARE PRICES

Up until now we only have the ticker symbols and nothing more. But of course we want all the financial data as well. So now we are going to download the stock data for each ticker symbol from the Yahoo Finance API. This will take up a couple of hundred megabytes (depending on the time frame). But first we will need three additional imports.

```
import os
import datetime as dt
import pandas_datareader as web
```

The *datetime* and the *pandas_datareader* module should be obvious here. But we are also importing the *os* library which provides us with basic functions of the operating system. We will use it for directory operations.

Now we are going to create a second function for loading the actual share prices. We start by getting our ticker symbols into the function.

```
def load_prices(reload_tickers=False):

    if reload_tickers:
        tickers = load_sp500_tickers()
    else:
        if os.path.exists('sp500tickers.pickle'):
            with open('sp500tickers.pickle', 'rb') as f:
                tickers = pickle.load(f)
```

Here we have the function *load_prices*. It has one parameter which's default is *False*. This parameter decides if we are going to scrape the tickers anew or if we are going to load them from our serialized file. If we want to scrape it again, we call our first function. Otherwise we check if our pickle file exists and if yes we load it. You can also define an else-tree which defines what happens when it doesn't exist. Maybe you want to also call the first function in that case.

The next thing we need to do is to create a directory for our files. We will create a CSV file for every single ticker symbol and for these files we want a new directory.

```
if not os.path.exists('companies'):
    os.makedirs('companies')
```

We again use the function *os.path.exists* to check if a directory named *companies* exists (you can choose any name you like). If it doesn't exist, we use the *makedirs* method to create it.

Now let's get to the essence of the function, which is the downloading of our data.

```
start = dt.datetime(2016,1,1)
end = dt.datetime(2019,1,1)

for ticker in tickers:
  if not os.path.exists('companies/{}.csv'.format(ticker)):
    print("{} is loading...".format(ticker))
    df = web.DataReader(ticker, 'yahoo', start, end)
    df.to_csv('companies/{}.csv'.format(ticker))
  else:
    print("{} already downloaded!".format(ticker))
```

Here we defined a pretty narrow time frame. Three years are not enough for a

decent analysis. However, you can adjust these values as you want but the broader your time frame, the more time it will take and the more space you will need.

Basically, what this function does is just checking if a CSV file for a specific ticker symbol already exists and if it doesn't it downloads and saves it.

Now when you run this script and it starts downloading, you may notice that it takes quite a while. One interesting idea would be to implement a faster way to download the data using multithreading and queues. This would be a nice exercise for you. If you need some help for doing this, check out the second volume, which is for intermediates.

COMPILING DATA

All good things come in threes. Therefore we are going to write a third and last function that compiles our data. We will take the data out of each of the 500 CSV files and combine it into one data frame. Then we will export that data frame into a new final CSV file.

Let's start by loading the ticker symbols into our function.

```
with open('sp500tickers.pickle', 'rb') as f:
    tickers = pickle.load(f)

main_df = pd.DataFrame()
```

As you can see, we create a new empty data frame here. This *main_df* will be our main data frame which contains all values. We are now going to extract the *adjusted close* value from every CSV file and add this column to our main data frame. This means that in the end, we will have a CSV file with the adjusted close value for all companies.

```
print("Compiling data...")
for ticker in tickers:
  df = pd.read_csv('companies/{}.csv'.format(ticker))
  df.set_index('Date', inplace=True)

  df.rename(columns = {'Adj Close': ticker}, inplace=True)
  df.drop(['Open', 'High', 'Low', 'Close'], 1, inplace=True)

  if main_df.empty:
    main_df = df
```

```
    else:
        main_df = main_df.join(df, how='outer')
```

Here we have a for loop that iterates over all ticker symbols. For every ticker symbol we load the respective CSV file into a data frame. Then we set the index of this data frame to be the *Date* column, since we will need a common index. We then rename the *Adj Close* column to the ticker symbol. This is because we will have one big CSV files with 500 columns and they should not all have the same name. Then we drop all the other columns except for *Date*. Last but not least we check if our main data frame is empty or not. If it is empty, our first data frame becomes the main data frame. Otherwise, we join the data frame onto our main data frame using an outer join. We've discussed joins in previous volumes.

```
main_df.to_csv('sp500_data.csv')
print("Data compiled!")
```

At the end we save our main data frame into a new CSV file and we're done. We can now run our functions.

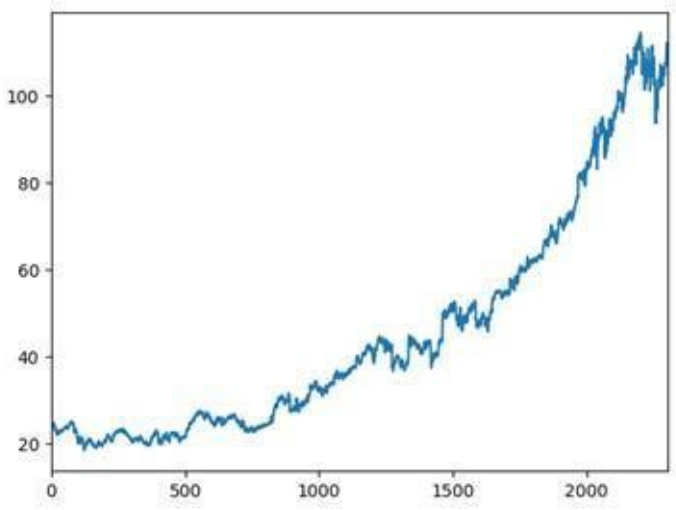
```
load_prices(reload_tickers=True)
compile_data()
```

VISUALIZING DATA

Now we have a local CSV file with all the S & P 500 Index tickers. So, for a while, we don't need to ask the Yahoo Finance API for past data.

```
sp500 = pd.read_csv('sp500_data.csv')
sp500['MSFT'].plot()
plt.show()
```

We load our CSV file into a DataFrame and can then, simply indicate our desired ticker symbol in the square brackets. In this case, we draw the graph of Microsoft.



CORRELATIONS

Finally, for this chapter, let's look at a very interesting feature of Panda's data frames. This function is called *corr* and stands for *correlation*.

```
correlation = sp500.corr()
```

Here we create a correlation data frame, which contains the values of the correlations between individual share prices. In a nutshell this means that we can see how the prices influence each other.

```
print(correlation)
```

	MMM	ABT	ABBV
MMM	1.000000	0.919446	0.928664
ABT	0.919446	1.000000	0.907837
ABBV	0.928664	0.907837	1.000000
ABMD	0.812892	0.888733	0.888560
ACN	0.963421	0.952070	0.936506

The numbers we see here show us how "similar" the change in the prices of the individual stocks is. The stocks MMM and MMM have a correlation of 100% because they are the same stock. On the other hand, ABBV and MMM have only about 93% correlation, which is still a lot.

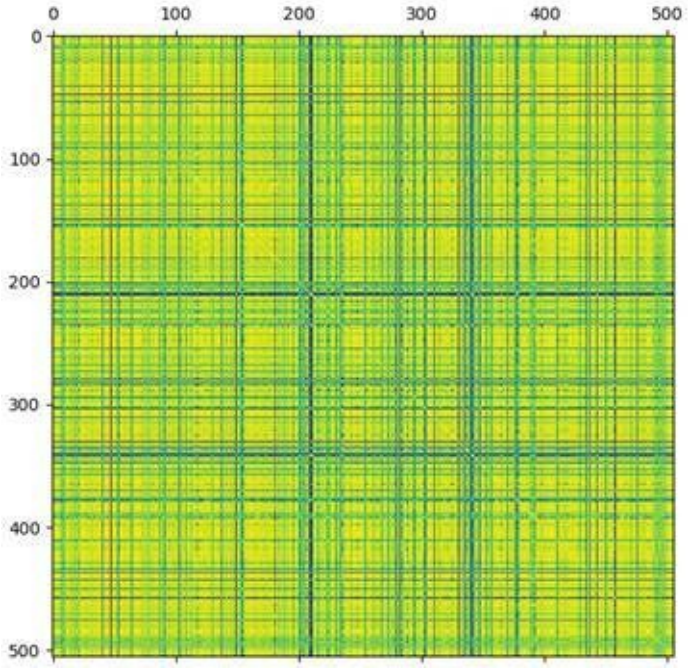
If you look at the whole table, you will find that there are some correlations that are less than 1% and even some that are negative. This means that if stock A

falls, stock B rises and vice versa. They are indirectly proportional.

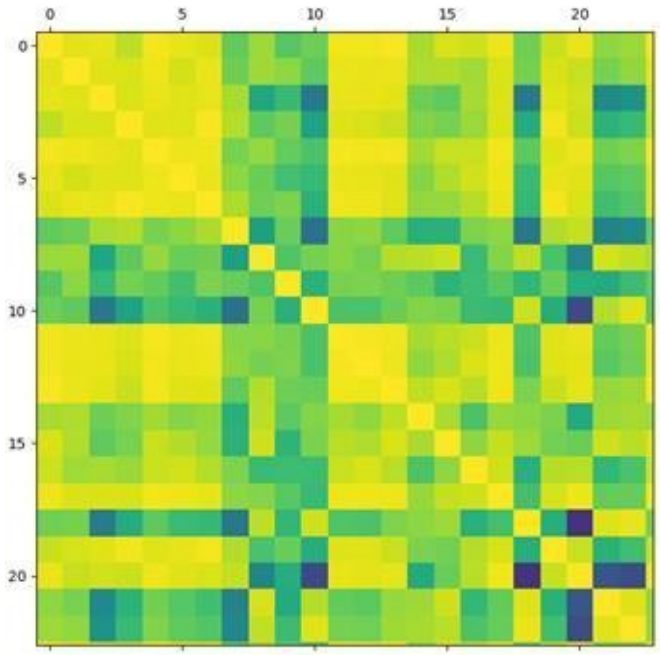
VISUALIZING CORRELATIONS

This table can be very helpful in analyzing and predicting prices. We can also visualize the correlations using a special Matplotlib function.

```
plt.matshow(correlation)
plt.show()
```



You can use the Matplotlib window to zoom into the correlations. The more yellow a point is, the higher the correlation.



This is quite a nice way to visualize correlations between share prices of different companies.

7 – REGRESSION LINES

In this chapter, we are going to use linear regression in order to plot regression lines. These indicate in which direction the share price is going in a specific time frame. For this chapter we are going to need NumPy.

```
import numpy as np
```

First, we are going to load our financial data again.

```
start = dt.datetime(2016,1,1)
end = dt.datetime(2019,1,1)

apple = web.DataReader('AAPL','yahoo', start, end)
data = apple['Adj Close']
```

In this case, we again choose the company Apple. As a next step, we need to quantify our dates in order to be able to use them as x-values for our algorithm. The y-value will be the adjusted close.

```
x = data.index.map(mdates.date2num)
```

Here we again use the function *date2num* and we map it onto our *Date* column. The next step is to use NumPy to create a linear regression line that fits our share price curve.

```
fit = np.polyfit(x, data.values, 1)
fit1d = np.poly1d(fit)
```

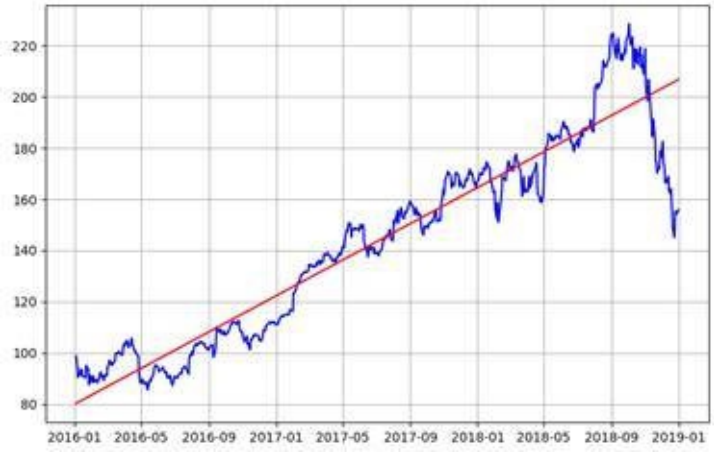
You will notice that this implementation of linear regression is quite different from the one, we already used in the last volume with scikit-learn. Here we use NumPy. First we call the *polyfit* method to fit a model for the x-values (our dates) and the y-values (the prices). The last parameter (one) is the degree of the function. In this case, it is linear. What this function returns to us is a list of coefficients. To now use this list and make an actual function of it, we need the second method *poly1d*. It takes the list and constructs a function for *x*. So our variable *fit1d* is actually a callable function.

We can now use what we have in order to plot our share price graph and the

regression line for it.

```
plt.grid()
plt.plot(data.index, data.values, 'b')
plt.plot(data.index, fit1d(x),'r')
plt.show()
```

First we just plot our price graph in blue color. Then we plot our regression line. Here our x-values are also the dates but the y-values are the result of our *fit1d* function for all input values, which are our numerical dates. The result looks like this:



Now we just need to be able to choose the time frame for which we want to draw the regression line.

SETTING THE TIME FRAME

So first we need to define two dates in between of which we want to draw the regression line. We do this as always with the *datetime* module.

```
rstart = dt.datetime(2018, 7, 1)
rend = dt.datetime(2019, 1, 1)
```

In this case, we want to look at the six months from the 1st of June 2018 to the 1st of January 2019. What we now need to do may be a bit confusing. We will create a new data frame and cut off all other entries.

```
fit_data = data.reset_index()
pos1 = fit_data[fit_data.Date >= rstart].index[0]
pos2 = fit_data[fit_data.Date <= rend].index[-1]

fit_data = fit_data.iloc[pos1:pos2]
```

Here we create the data frame *fit_data* which starts by copying our original data frame and resetting its index. Then we calculate two positions by querying data from our new data frame. We are looking for the first position (index zero) in our data frame, where the *Date* column has a value greater or equal to our start date. Then we are looking for the last position (index negative one) where our *Date* column has a value less or equal to our end date. Finally, we cut out all other entries from our data frame by slicing it from position one to position two.

Now we of course need to rewrite our *fit* functions a little bit.

```
dates = fit_data.Date.map(mdates.date2num)

fit = np.polyfit(dates, fit_data['Adj Close'], 1)
fit1d = np.poly1d(fit)
```

We again create a new variable *dates* which contains the dates from our time frame in numerical format. Then we fit the regression model with our data again.

```
plt.grid()
plt.plot(data.index, data.values, 'b')
plt.plot(fit_data.Date, fit1d(dates),'r')
plt.show()
```

At the end, we again plot our two graphs. But this time we refer to the *Date* column specifically since it is no longer the index of the *fit_data* data frame. This is the result:

This time we can clearly see that the slope is negative, since the prices go down in that time frame.

8 – PREDICTING SHARE PRICES

Now in this last chapter, we will use machine learning to predict our share price development. However, this prediction won't be reliable, since it is quite simplistic and it is generally very hard to predict the markets. This is more about learning how to apply machine learning to financial data. For this chapter, we will need the following libraries in addition to the ones we already used:

```
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

You should be familiar with these from the previous volume. We have one library for preprocessing our data, one for splitting our data into training and testing data and one library which provides the machine learning algorithm itself.

LOADING AND PREPARING DATA

We are going to use linear regression again. But first, of course, we will need to load our data again.

```
start = dt.datetime(2016,1,1)
end = dt.datetime(2019,1,1)

apple = web.DataReader('AAPL','yahoo', start, end)
data = apple['Adj Close']
```

Now how are we going to predict our share prices? Our approach will be quite simple. We are going to choose an amount of days and then shift our prices by that amount. Then we can look at the data and see how they have developed in past times and predict how they are going to do it in future times.

```
days = 50
data['Shifted'] = data['Adj Close'].shift(-days)
data.dropna(inplace=True)
```

Here we defined 50 days. We create a new column *Shifted* which contains our

prices shifted upwards by 50 days using the *shift* function. At the end, we drop the *Nan* values which we have in this new column.

The next step is to prepare our data so that our model can learn from it. For this, we will need to convert it into NumPy arrays.

```
X = np.array(data.drop(['Shifted'],1))
Y = np.array(data['Shifted'])
X = preprocessing.scale(X)
```

Our x-value will be the adjusted close share price. For this, we drop the shifted column. As a y-value we choose only the shifted values. In order to make computations more efficient, we scale our x-values down. We normalize them.

TRAINING AND TESTING

Now we are going to split our data into training data and into test data.

```
X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size=0.2)
```

We are using a test size of 20%, which means that we use 80% of the data for training and 20% for calculating the accuracy.

```
clf = LinearRegression()
clf.fit(X_train, Y_train)
accuracy = clf.score(X_test, Y_test)
print(accuracy)
```

We create a linear regression model and then fit it to our training data. After that we use the *score* method to calculate our accuracy.

0.8326191580805993

Most of the time the result will be around 85% which is actually not that bad for predicting share prices based on share prices.

PREDICTING DATA

We can now use our trained model to make predictions for future prices. Notice however that this is not a tool or model that you should be using for real trading. It is not accurate enough.

```
X = X[:-days]
X_new = X[-days:]
```

Here we cut out the last 50 days and then create a new array *X_new* which takes the last 50 days of the remaining days. We will use these for predicting future values. For this, we will use the *predict* function of our model.

```
prediction = clf.predict(X_new)
print(prediction)
```

The results we get are our predicted prices for the next upcoming days:

```
[185.41161298 185.06584397 184.52124063 187.43442155 188.80890817
 190.08825219 190.66744184 190.21792733 188.69649258 188.29022104
 189.19786639 187.83204987 187.91852237 186.22418294 186.13775077
 183.50119313 184.20140126 183.30238569 182.8355713  180.4583505
 182.41201333 182.17861958 183.33694511 182.99983276 184.78920724
 181.97114204 183.25051294 185.4721155  187.72833126 187.52951038
 185.39433999 188.11731637 188.37665321 188.01359776 188.48038526
 187.57273991 188.85211081 188.47174204 188.61870362 189.82028557
 191.39357963 190.86626272 188.07410028 187.14914161 187.47763764
 197.16804184 202.25960924 202.77828293 203.71189826 202.01758572]
```

WHAT'S NEXT?

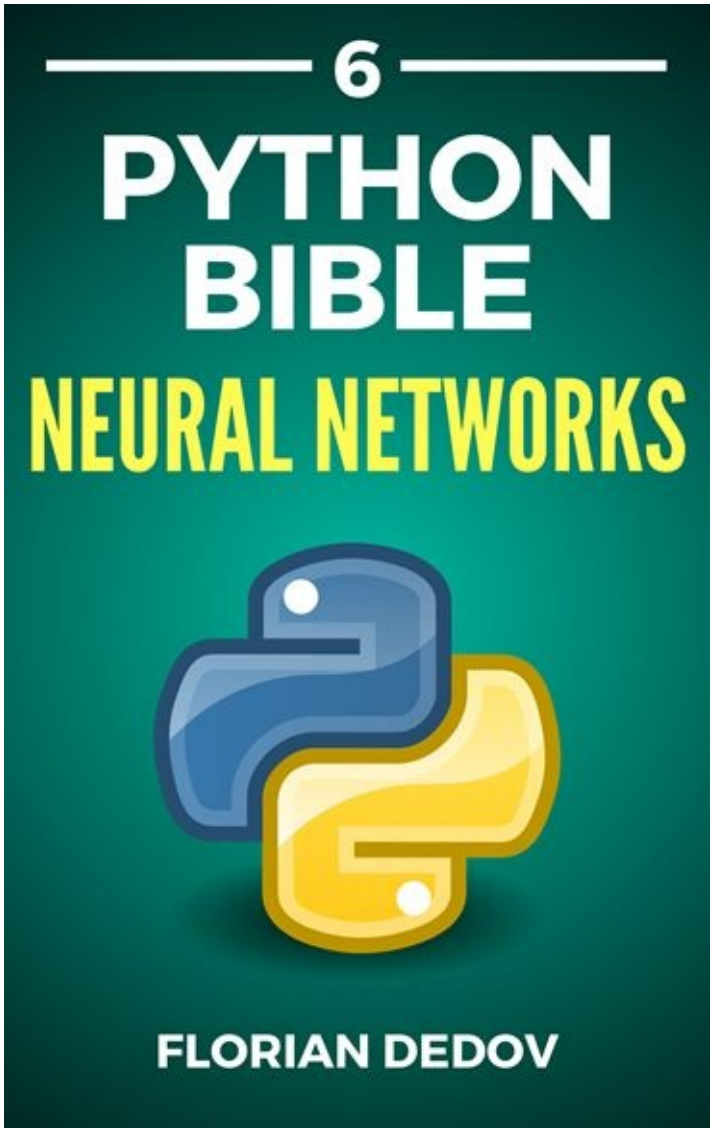
With this volume finished, you can consider yourself a very advanced Python programmer. You are able to write high level code and apply it to real-world problems and use cases. With the knowledge gained from this book, you can even develop your own portfolio analysis or management tool. You can use data science and machine learning to program your own financial software.

The skills you possess right now are crucial in today's economy. They are rare and very valuable. I encourage you to continue your journey. Even though you have already learned quite a lot, we don't stop here. There are a lot of topics to be covered yet and you should not miss out on them. Practice what you've learned, play around and experiment!

I wish you a lot of success on your programming journey! Stay tuned and prepare for upcoming volumes.

Last but not least, a little reminder. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!


6
PYTHON
BIBLE
NEURAL NETWORKS
FLORIAN DEDOV

THE
PYTHON BIBLE

VOLUME SIX

NEURAL NETWORKS

BY

FLORIAN DEDOV

TABLE OF CONTENT

INTRODUCTION

Machine learning is one of the most popular subjects of computer science at the moment. It is fascinating how computers are able to learn to solve complex problems, without specific instructions but just by looking at training data. With machine learning we made computers able to recognize and reproduce human voices, to recognize objects and to drive cars. Wherever big advances and achievements are made, we hear the term *deep learning*. Rarely do we hear that we made some radical progress with linear regression or with a basic classification algorithm. The technologies used are almost always neural networks.

Deep learning is a sub-field of machine learning, which is all about neural networks. These neural networks are inspired by the human brain and produce extraordinary results. They beat professional chess players, drive cars and outperform humans even in complex video games like Dota 2.

Furthermore, the speed of progress that we make in these fields makes it almost impossible to predict where all of this is going over the next couple of decades. But one thing is for sure: Those who understand machine learning and neural networks will have huge advantages over those, who will be overrun by these developments. Therefore, it makes a lot of sense to educate yourself on these subjects.

THIS BOOK

In this book, you will learn what neural networks are and how they work from scratch. You will also learn how to build and use them in the programming language Python. We will work with impressive examples and you will be astonished by some of the results.

What you will need for this book are advanced Python skills and a basic understanding of machine learning. Fundamental math skills are also beneficial. If you lack these skills, I recommend reading the previous volumes of this Python Bible Series, before proceeding. The first couple of volumes focus on the basic and intermediate concepts of Python. Then we learn some things about data science, machine learning and finance programming. This book is the sixth volume and a lot of knowledge from the previous volumes is required. However, you can also try to learn these skills from other sources. In this book though, there won't be any explanations of basic Python syntax or fundamental machine learning concepts.

Amazon Author Page: https://amzn.to/38D209r

HOW TO READ THIS BOOK

Fundamentally, it is your own choice how you go about reading this book. If you think that the first few chapters are not interesting to you and don't provide any value, feel free to skip them. You can also just read the whole book without ever writing a single line of code yourself. But I don't recommend that.

I would personally recommend you to read all the chapters in the right order because they built on top of each other. Of course, the code works without the theoretical understanding of the first chapter. But without it you will not be able to understand what you are doing and why it works or why it doesn't.

Also I highly recommend you to actively code along while reading this book. That's the only way you will understand the material. In the later chapters, there will be a lot of code. Read through it, understand it but also implement it yourself and play around with it. Experiment a little bit. What happens when you change some parameters? What happens when you add something? Try everything.

I think that's everything that needs to be said for now. I wish you a lot of success and fun while learning about neural networks in Python. I hope that this book will help you to progress in your career and life!

Just one little thing before we start. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you have learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

If you are interested in free educational content about programming and machine learning, check out: https://www.neuralnine.com/

1 – BASICS OF NEURAL NETWORKS

WHAT ARE NEURAL NETWORKS?

Before we start talking about how neural networks work, what types there are or how to work with them in Python, we should first clarify what neural networks actually are.

Artificial neural networks are mathematical structures that are inspired by the human brain. They consist of so-called *neurons*, which are interconnected with each other. The human brain consists of multiple billions of such neurons. Artificial neural networks use a similar principle.

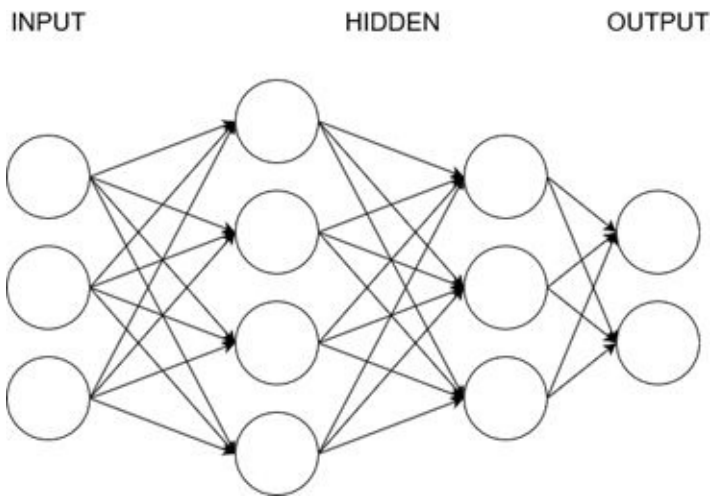


Fig. 1.1: Structure of an artificial neural network

The structure of a neural network is quite simple. In the figure above, we can see multiple layers. The first one is the *input layer* and the last one is the *output layer*. In between we have multiple so-called *hidden layers*.

The input layer is for the data that we want to feed into the neural network in order to get a result. Here we put all of the things that are “perceived” by or put into the neural network. For example, if we want to know if a picture shows a cat or a dog, we would put all the values of the individual pixels into the input layer. If we want to know if a person is overweight or not, we would enter parameters like height, weight etc.

The output layer then contains the results. There we can see the values generated

by the neural network based on our inputs. For example the classification of an animal or a prediction value for something.

Everything in between are abstraction layers that we call hidden layers. These increase the complexity and the sophistication of the model and they expand the internal decision making. As a rule of thumb we could say that the more hidden layers and the more neurons we have, the more complex our model is.

STRUCTURE OF NEURONS

In order to understand how a neural network works, we need to understand how the individual neurons work.

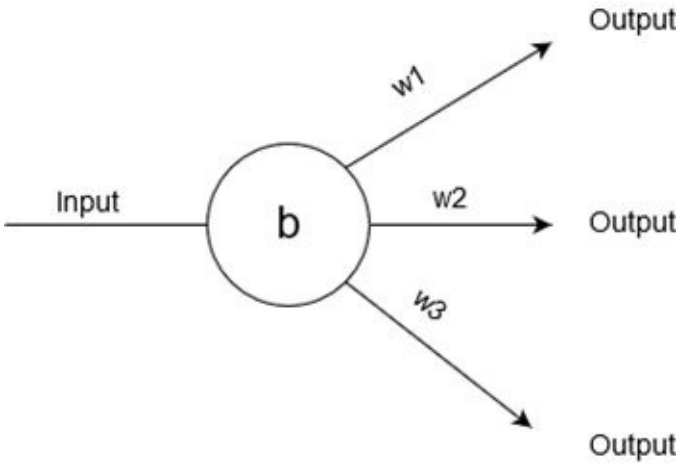


Fig. 1.2: Structure of an artificial neuron

As you can see every neuron gets a certain input, which is either the output of a previous neuron or the raw input of the input layer.

This input is a numerical value and it then gets multiplied by each individual *weight (w1, w2, w3…)*. At the end we then subtract the *bias (b)*. The result is the output of that particular connection. These outputs are that forwarded to the next layer of neurons.

What I just explained and what you can see at the figure above is an outdated version of a neuron, called the *perceptron*. Nowadays we are using much more complex neurons like the sigmoid neurons, which use mathematical functions to calculate the result for the output.

You can probably imagine how complex a system like this can get when all of the neurons are interconnected and influence each other. In between our input and output layer we oftentimes have numerous hidden layers with hundreds or thousands of neurons each. These abstraction layers then lead to a final result.

ACTIVATION FUNCTIONS

There are a lot of different so-called *activation functions* which make everything more complex. These functions determine the output of a neuron. Basically what we do is: We take the input of our neuron and feed the value into an activation function. This function then returns the output value. After that we still have our weights and biases.

SIGMOID ACTIVATION FUNCTION

A commonly used and popular activation function is the so-called *sigmoid activation function*. This function always returns a value between zero and one, no matter what the input is. The smaller the input, the closer the output will be to zero. The greater the input, the closer the output will be to one.

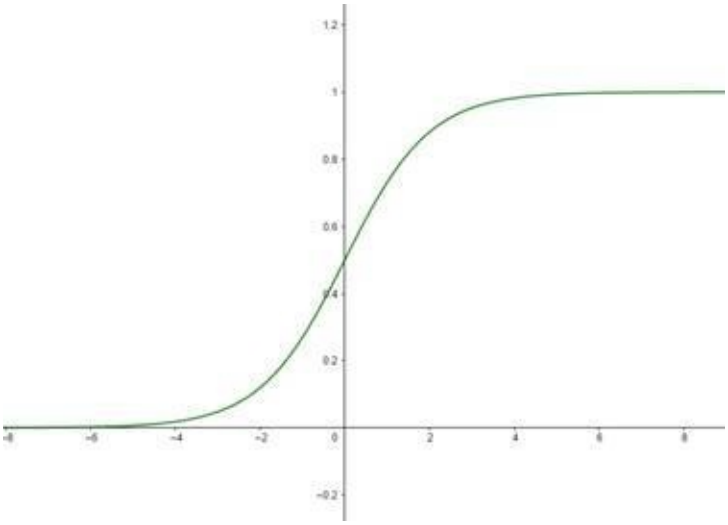


Fig 1.3: Sigmoid Activation Function

The mathematical formula looks like this:

$$f(x) = \frac{1}{1 + e^{-x}}$$

You don’t have to understand this function 100% if you don’t want to. But what you can easily see is that the one in the numerator indicates that the output will always lie in between zero and one, since the denominator is always positive. This function is much smoother than the basic perceptron.

RELU ACTIVATION FUNCTION

The probably most commonly used activation function is the so-called *ReLU function*. This stands for *rectified linear unit*. This function is very simple but also very useful. Whenever the input value is negative, it will return zero. Whenever it is positive, the output will just be the input.

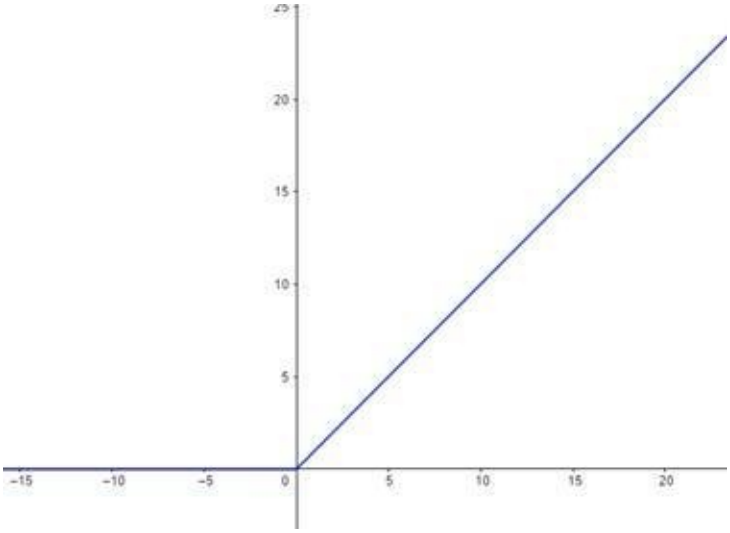


Fig 1.4: ReLU activation function

The mathematical formula looks like this:

$$f(x) = \max(0, x)$$

Even though that function is that simple, it oftentimes fulfils its purpose and it is commonly used as the go-to activation function for hidden layers.

Of course there are also a lot of other activation functions. But the purpose of this chapter is not to show you all of them but to give you a basic understanding of what activation functions are and how they work. We will talk about more activation functions in later chapters, when we are using them.

TYPES OF NEURAL NETWORKS

Neural networks are not only different because of the activation functions of their individual layers. There are also different types of layers and networks. In this book we are going to take a deeper look at these. For now, we will get a first overview of them in this chapter.

FEED FORWARD NEURAL NETWORKS

The so-called *feed forward neural networks* could be seen as the *classic* neural networks. Up until now we have primarily talked about these. In this type of network the information only flows into one direction – from the input layer to the output layer. There are no circles or cycles.

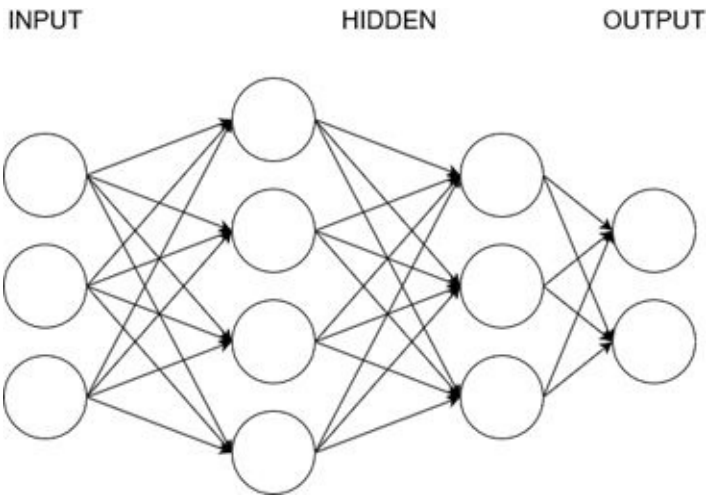


Fig 1.5: Feed Forward Neural Network

Above you can see the figure that we already talked about. If you look closer, you will see that the connections are pointed into one direction only. The information flows from left to right.

RECURRENT NEURAL NETWORKS

So-called *recurrent neural networks* on the other hand work differently. In these networks we have layers with neurons that not only connect to the neurons next layer but also to neurons of the previous or of their own layer. This can also be called *feedback*.

If we take the output of a neuron and use it as an input of the same neuron, we are talking about *direct feedback*. Connecting the output to neurons of the same

layer is called *lateral feedback*. And if we take the output and feed it into neurons of the previous layer, we are talking about *indirect feedback*.

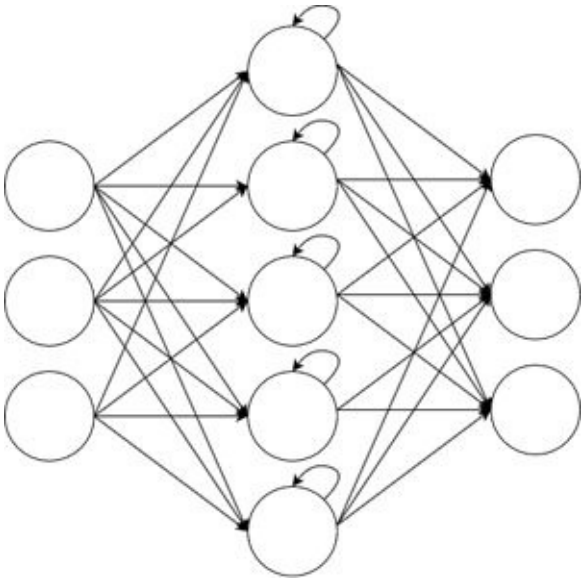


Fig. 1.6: Recurrent neural network (direct feedback)

The advantage of such a recurrent neural network is that it has a little memory and doesn’t only take the immediate present data into account. We could say that it “looks back” a couple of iterations.

This kind of neural networks is oftentimes used when the tasks requires the processing of sequential data like text or speech. The feedback is very useful in this kind of tasks. However it is not very useful when dealing with image recognition or image processing.

CONVOLUTIONAL NEURAL NETWORKS

For this purpose we have the so-called *convolutional neural networks*. This type is primarily used for processing images and sound. It is especially useful when pattern recognition in noisy data is needed. This data may be image data, sound data or video data. It doesn’t matter.

We are going to talk about how this type of neural network works in the respective chapter, but for now let us get a superficial quick overview.

Fig. 1.7: Xs and Os for classification

Let’s look at a simple example. Here we have multiple Xs and Os as examples in a 16x16 pixels format. Each pixel is an input neuron and will be processed. At the end our neural network shall classify the image as either an X or an O.

Of course this example is trivial and could probably even be solved with a K-Nearest-Neighbors classification or a support vector machine. However, we are going to use it just to illustrate the principle.

For us humans it is very easy to differentiate between the two shapes. All of the Xs and all of the Os are quite similar. We easily spot the patterns. For a computer however, these images are totally different because the pixels do not match exactly. In this case, this is not a problem, but when we try to recognize cats and dogs, things get more complex.

What convolutional neural networks now do is: Instead of just looking at the individual pixels, they look for *patterns* or *features*.

Most of the Xs for example have a similar center with four pixels which splits up into four lines. Also they have long diagonal lines. Os on the other hand have an empty center and shorter lines. If we were classifying cats (in comparison with dogs) we could look for pointy ears or whiskers.

As I already said, this explanation is quite superficial and we are going to get into the details in the respective chapter. But what we basically do is just looking for the most important features and classifying the images based on these.

TRAINING AND TESTING

In order to make a neural network produce accurate results, we first need to train and test it. For this we use already classified data and split it up into training and testing data. Most of the time we will use 20% of the data for testing and 80% for training. The training data is the data that we use to optimize the performance. The testing data is data that the neural network has never seen before and we use it to verify that our model is accurate.

If we take the example of pictures of cats and dogs, we could take 8000 images that were classified by human experts and then show these to the neural network (we are going to talk about the technical details in a second). Then we could use 2000 images as testing data and compare the results of our neural network with the answers that we know to be true.

ERROR AND LOSS

When evaluating the accuracy or the performance of our model, we use two metrics – *error* and *loss*.

I am not going to get too deep and theoretical into the definition of these terms, since especially the concept of loss confuses a lot of people. Basically you could say that the error indicates how many of the examples were classified incorrectly. This is a relative value and it is expressed in percentages. An error of 0.21 for example would mean that 79% of the examples were classified correctly and 21% incorrectly. This metric is quite easy to understand for humans.

The loss on the other hand is a little bit more complex. Here we use a so-called *loss function* to determine the value. This value then indicates how bad our model is performing. Depending on the loss function, this value might look quite different. However, this is the value that we want to minimize in order to optimize our model.

GRADIENT DESCENT

The minimization of this value is done with the help of the so-called *gradient descent algorithm*. The mathematics behind this algorithm is quite confusing for a lot of people but I will do my best to explain it to you as simple as possible.

Imagine a loss function that looks like this (trivial example):

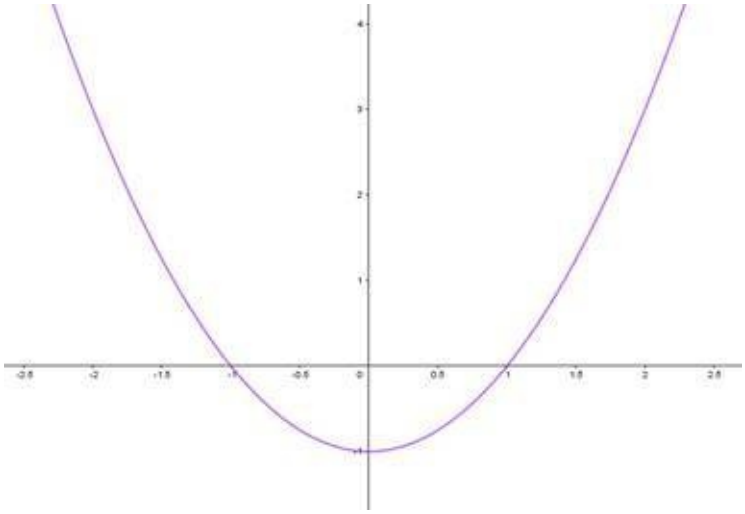


Fig 1.8: Trivial example for a loss function

Of course no loss function on earth would look like this because this doesn’t make any sense at all. But we are going to use this function as an example to illustrate the concept of the gradient descent.

As I already said, our goal is to minimize the output of that function. This means that we are looking for the x-value which returns the lowest y-value. In this example, it is easy to see that this value is zero, which returns negative one. But how can our algorithm see this?

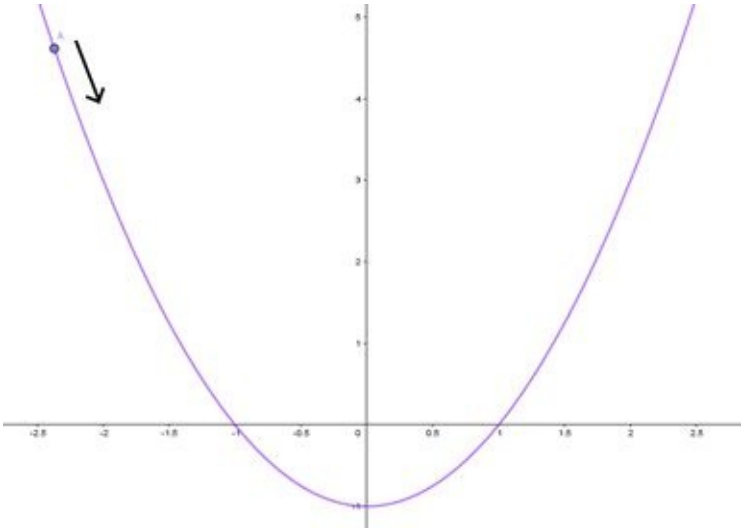


Fig. 1.9: Visualization of gradient descent

Since our computer doesn’t see the graph like we do, it will just start with a random initial point A. We then calculate the gradient of the function in that particular point. This tells us in which direction we need to move in order to

increase the output of the function the most. Since we want to minimize the output, we take a small step into the opposite direction. At the new point, in which we end up, we repeat this process. We continue to do this until we reach the valley where the gradient will be zero. This is the local minimum.

You can imagine it to be like a ball that rolls down the function's graph. It will inevitably roll into the local minimum. The emphasis is on *local*. Let's take a look at another function.

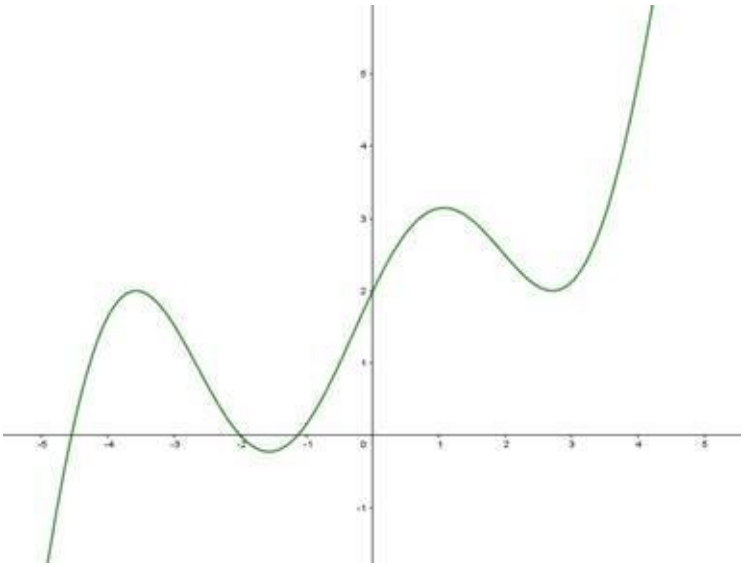


Fig. 1.10: Another trivial loss function example

The problem with this function is that it has multiple local minima. There is more than one valley. In which one we land depends on the initial starting point. In this case it might be easy to figure out which one is the best, but we will see in a second why this random approach doesn't work that easily.

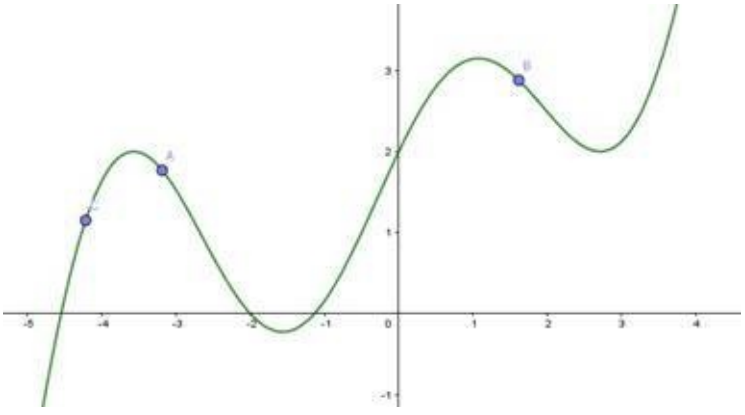


Fig. 1.11: Three different initial starting points

Here we can see three different starting points in the same function. When we start at point B, we will get into a local minimum but this local minimum is not even close to the minimum that we would get when starting at A. Also we have the starting point C, which might lead into an even lower minimum (if it leads into a minimum at all).

Multiple Features

Up until now the functions that we looked at, all had one input value and one output value. In a neural network however, we have thousands, if not more, weights, biases and other parameters, which we can tweak in order to change the outputs. The actual loss function gets the weights and biases as parameters and then returns the loss, based on the estimated and the actual results.

To get a better intuition about all of this, let us first take a look at a three-dimensional function.

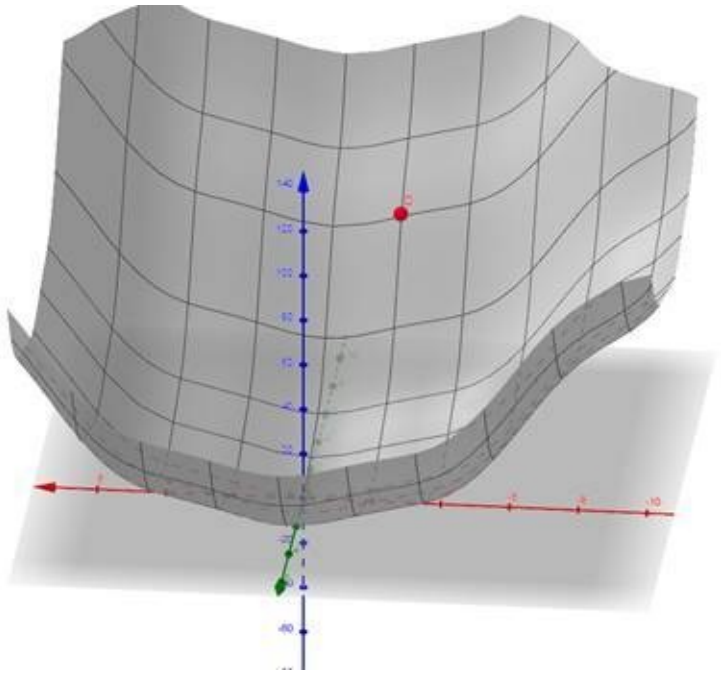


Fig. 1.12: Three-dimensional function

Here we apply the same principle. We have one point (this time in three dimensions) and this point shall roll down to the local minimum. The difference here is that we cannot only go left or right but into all directions of the plane. Therefore we need to determine the gradient for each axis. Mathematically speaking we need to work with partial derivatives.

First of all we look at the point and how the resulting value (vertical axis) changes, when we tweak the value of the first axis. Which direction is the one that causes the greatest increase of the output? We find this direction and then negate it, since we want to go the opposite way. We repeat the same process for the second input axis. At the end we take a tiny step into the resulting direction. This is the one with the steepest descent. Thus we inevitably roll into the local minimum.

Now try to imagine that process in multiple thousands of dimensions. Each weight and each bias would be one axis and therefore one additional dimension. Our algorithm needs to tweak all of these parameters in order to produce the best possible output. Of course as humans we can't imagine anything that is higher than three or four dimensions, let alone visualize it.

The Mathematics

We are not going to get too deep into the mathematics here but it might be quite beneficial for you to get a little bit more familiar with the mathematical notations.

Let's imagine our loss function to be the following one:

$$C(w,b) \equiv \frac{1}{2n} * \sum_{x} ||f(x) - y||^2$$

What this function basically does is the following: We pass the weights and biases as parameters. Then we calculate all the differences between the predictions of the models and the actually desired results. In this case *f(x)* is the prediction of the network and *y* is the actual result. We calculate the absolute value of the difference so that we are dealing with a positive value. Then we square that difference. We do this for every single example and we add all the differences up so that we get the sum. At the end we then divide it by twice the amount of examples.

Let me explain it again a little bit simpler. We take all the differences, square them, sum them up and divide it by twice the amount of examples in order to get the *mean squared error*. This is also the name of this loss function.

Now we want to minimize the output of this function, by tweaking the parameters with the gradient descent.

$$\nabla C := \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, \cdots, \frac{\partial C}{\partial v_n}\right)$$

We want to calculate the gradient of this function. This gradient is composed of all the partial derivatives of the loss function *C* (stands for the alternative name of *cost function*). The vectors v1, v2 etc. are the vectors of the individual weights and biases.

The only thing that we need to do now is to make a tiny step into the opposite direction of that gradient.

$$-\nabla C * \varepsilon$$

We take the negative gradient and multiply it with a minimal value.

Don’t panic if you don’t understand everything about the mathematics. The focus of this book is the application of these principles. Therefore you can continue reading even if you don’t understand the mathematics at all.

BACKPROPAGATION

We now understand what is needed to optimize our neural network. The question remains though, how we are going to implement all of that. How are we going to calculate the gradient? How are we going to tweak the parameters? For this we are going to use the *backpropagation algorithm*. Here I will also try to explain everything as simple as possible.

Basically backpropagation is just the algorithm that calculates the gradient for the gradient descent algorithm. It determines how and how much we need to change which parameters in order to get a better result.

First of all we take the prediction of the model and compare it to the actually desired result.

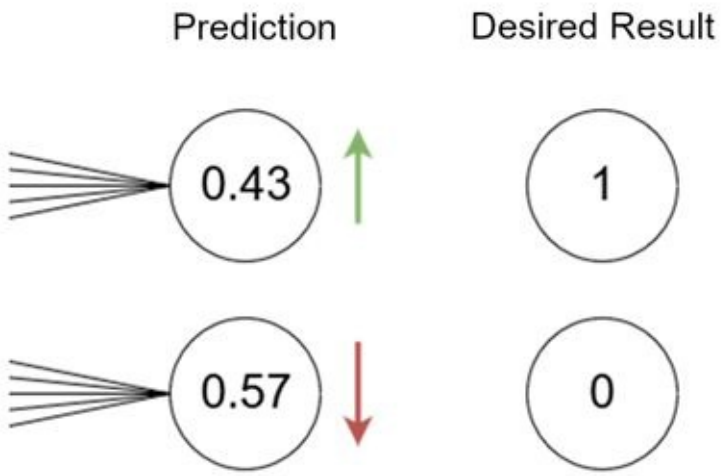


Fig. 1.13: Comparing the outputs

What we see in the figure above is the comparison of the output layer (consisting of two neurons) and the desired results. Let’s say the first neuron is the one that indicates that the picture is a cat. In this case the prediction would say that the picture is a dog (since the second neuron has a higher activation) but the picture is actually one of a cat.

So we look at how the results need to be changed in order to fit the actual data. Notice however that we don’t have any direct influence on the output of the neurons. We can only control the weights and the biases.

We now know that we want to increase the value of the first neuron and decrease the value of the second neuron. For this we will need to look back one layer.

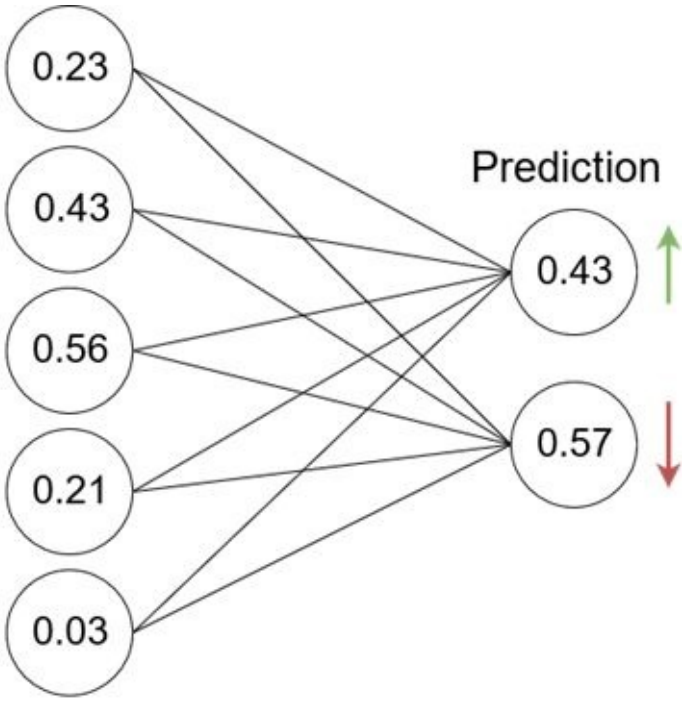


Fig. 1.14: Looking back one layer

In order to change the value of neurons we can either tweak the weights and biases or we can try to change the inputs.

Let’s take a look at how the value of the final output neurons gets calculated. We will notice that some connections increase this value, whereas some connections decrease it. Put differently: There are some neurons in the previous layer that will, when tweaked into a certain direction, change the value of the output neurons towards the desired results.

So we now again think, how the value of each neuron should be changed in order to move towards the desired result. Keep in mind that we still cannot directly influence these values. We can only control the weights and biases but it still makes sense to think about the ideal changes that we would like to make.

Also keep in mind that up until now everything we are doing is just for a single training example. We will have to look at every single training example and how this example will want to change all these values. This has to be done for multiple thousands of examples. What we are actually doing is therefore determining the change that is the best for all the training examples at once. Basically the mean or average.

For example if 7000 examples want a certain neuron to have a higher value and

3000 examples want it to have a lower value, we will increase it but not as much as we would increase it if 10,000 examples demanded it.

When we do this for every single neuron of that layer, we know how we need to change each of those. Now we need to go back one more layer and repeat the same process. We do this until we get to the input layer.

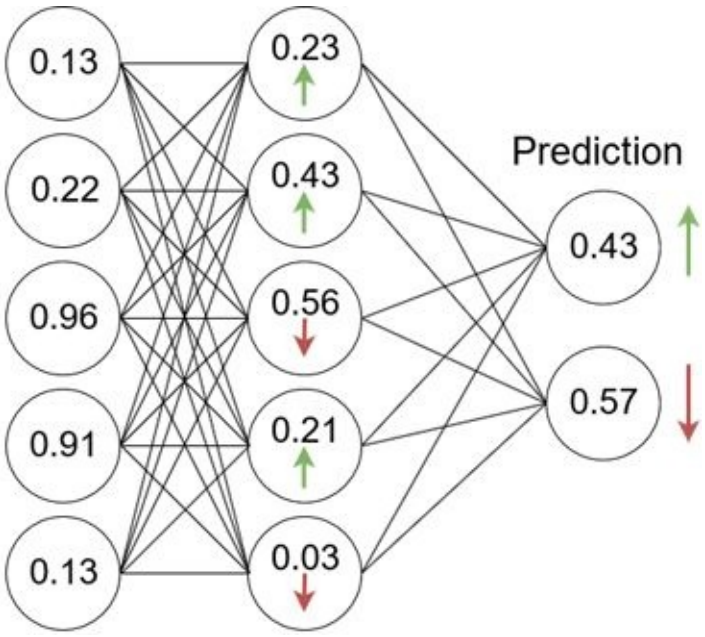


Fig. 1.15: One more layer back

When we continue to do this process until we reach the input layer, we will get the on average demanded changes for all weights and biases. This is the exact same thing as the negative gradient of the gradient descent algorithm. We then take a tiny step and repeat the whole process. Depending on the machine we are working on and some other parameters this might take a while. We do this over and over again until we are satisfied with the results.

SUMMARY

Since we covered a lot of different theoretical topics in this chapter, let us summarize the essential things:

- Activation functions determine the activation of a neuron which then influences the outputs.
- The classic neural networks are feed forward neural networks. The information only flows into one direction.
- In recurrent neural networks we work with feedback and it is possible to take the output of future layers as the input of neurons. This creates something like a memory.
- Convolutional neural networks are primarily used for images, audio data and other data which requires pattern recognition. They split the data into features.
- Usually we use 80% of the data we have as training data and 20% as testing data.
- The error indicates how much percent of the data was classified incorrectly.
- The loss is a numerical value which is calculated with a loss function. This is the value that we want to minimize in order to optimize our model.
- For the minimization of the output we use the gradient descent algorithm. It finds the local minimum of a function.
- Backpropagation is the algorithm which calculates the gradient for the gradient descent algorithm. This is done by starting from the output layer and reverse engineering the desired changes.

2 – INSTALLING LIBRARIES

DEVELOPMENT ENVIRONMENT

Now after all that theory, let us get into the implementation. First of all, we are going to set up our development environment. Actually it is your choice which IDE you are going to use. However I highly recommend using a professional environment like PyCharm or Jupyter Notebook.

PyCharm: https://www.jetbrains.com/pycharm/download/

Anaconda: https://www.anaconda.com/distribution/

LIBRARIES

Also, we are going to need some external libraries for our projects. It is important that we understand all the theory behind neural networks but it is a waste of time to reinvent the wheel and code everything from scratch. Therefore we will use professional libraries that do most of the work for us.

For this book we will need Tensorflow, which is the most popular library for working with neural networks. We will use *pip* for the installation:

pip install tensorflow

In the course of this book we will also oftentimes use some of the external libraries that we have already used in the previous volumes. If we use additional libraries, we are going to mention that an installation is needed in the respective chapter. The following libraries are yet pretty important and should be part of the stack of every Python programmer.

pip install numpy

pip install matplotlib

pip install pandas

All these libraries will do a lot of the work for us. We almost don’t have to deal with any mathematics or theory at all. With these libraries we are operating at the use-case level.

3 – HANDWRITTEN DIGIT RECOGNITION

Let us now finally get into some real programming. In this chapter our goal is to build and train a neural network, which recognizes handwritten digits with a mind-blowing accuracy. It will be able to recognize the digits from 0 to 9.

REQUIRED LIBRARIES

For this chapter we will need the following imports:

```
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

Tensorflow is the main library here. We will use it to load data sets, build neural networks, train them etc. The other three libraries are not necessary for the functionality of the neural network. We are only using them in order to load our own images of digits at the end.

Numpy will be used for reformatting our own images and *Matplotlib* will be used for their visualization.

CV2 is the *OpenCV* library and it will allow us to load our images into the script. You will need to install this module separately:

```
pip install opencv-python
```

LOADING AND PREPARING DATA

Before we start building and using our neural network, we need to first get some training data and prepare it.

For this chapter we are going to use the *MNIST dataset* which contains 60,000 training examples and 10,000 testing examples of handwritten digits that are already classified correctly. These images have a resolution of 28x28 pixels. We will use the *keras* module, in order to load the dataset.

```
mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

In order to get the dataset, we access the *mnist* object from the *keras.datasets*. Then we call the *load_data* function. This function automatically splits the data appropriately and returns a tuple with the training data and a tuple with the testing data.

In order to make the whole data easier to process, we are going to *normalize* it. This means that we scale down all the values so that they end up between 0 and 1.

```
X_train = tf.keras.utils.normalize(X_train, axis=1)
X_test = tf.keras.utils.normalize(X_test, axis=1)
```

For this we use the *normalize* function of *keras.utils*. We have now structured and normalized our data so that we can start building our neural network.

BUILDING THE NEURAL NETWORK

Let's think about what kind of structure would make sense for our task. Since we are dealing with images, it would be reasonable to build a convolutional neural network. When we take a look at the official website of the MNIST dataset we will find a table of the various different types and structures of neural networks and how well they perform at this task.

MNIST Website: http://yann.lecun.com/exdb/mnist/

There we can see that in fact convolutional neural networks are one of the best ways to do this. However we are going to use an ordinary feed forward neural network for this task. First of all, because it is certainly enough and second of all, because we will come back to convolutional neural networks in a later chapter. It makes sense to start with the fundamental structures first.

```
model = tf.keras.models.Sequential()
```

We use the *models* module from *keras* to create a new neural network. The *Sequential* constructor does this for us. Now we have a model, which doesn't have any layers in it. Those have to be added manually.

```
model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
```

We start out by adding a so-called *Flatten* layer as our first layer. In order to add a layer to our model, we use the *add* function. Then we can choose the kind of layer that we want from the *layers* module. As you can see, we specified an input shape of 28x28 which represents the resolution of the images. What a flattened layer basically does is it flattens the input and makes it one dimensional. So instead of a 28x28 grid, we end up with 784 neurons lined up. Our goal is now to get to the right result based on these pixels.

```
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
```

In the next step we now add two *Dense* layers. These are our hidden layers and increase the complexity of our model. Both layers have 128 neurons each. The activation function is the *ReLU* function (see chapter 1). Dense layers connect every neuron of this layer with all the neurons of the next and previous layer. It is basically just a default layer.

```
model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
```

Last but not least we add an output layer. This one is also a dense layer but it only has ten neurons and a different activation function. The values of the ten neurons indicate how much our model believes that the respective number is the right classification. The first neuron is for the zero, the second for the one and so on.

The activation function that we use here is the *softmax* function. This function scales the output values so that they all add up to one. Thus it transforms the absolute values into relative values. Every neuron then indicates how likely it is that this respective number is the result. We are dealing with percentages.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(28,28)))
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
```

In a nutshell, we have a flattened input layer with 784 neurons for the input pixels, followed by two hidden layers and one output layer with the probabilities for each digit.

COMPILING THE MODEL

Before we start training and testing our model, we need to compile it first. This optimizes it and we can also choose a loss function.

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

We are not going to get too deep into the optimizers and the loss functions here. The parameters that we chose here are pretty good for our task. Also, we define the metrics that we are interested in. In this case, we only care about the accuracy of our model.

TRAINING AND TESTING

Now we get to the essential part of the whole project – the training and testing. For this, we just have to use the *fit* function of our model.

```
model.fit(X_train, y_train, epochs=3)
```

Here we pass our x- and y-values as the training data. Then we also define the number of epochs that we want to go through. This number defines how many times our model is going to see the same data over and over again.

```
loss, accuracy = model.evaluate(X_test, y_test)
print(loss)
print(accuracy)
```

After that we use the *evaluate* method and pass our testing data, to determine the accuracy and the loss. Most of the time we get an accuracy of around 95% (try it yourself). This is pretty good if you take into account that mere guessing would give us a 10% chance of being right. Our model performs quite well.

```
model.save('digits.model')
```

Instead of training the model over and over again every single time we run the script, we can save it and load it later on. We do this by using the *save* method and specifying a name.

```
model = tf.keras.models.load_model('digits.model')
```

If we now want to load the model, we can just use the *load_model* function of *keras.models* and refer to the same name.

CLASSIFYING YOUR OWN DIGITS

Now that we know that our model works and performs quite well, let us try to predict our own handwritten digits. For this you can either use a program like Paint, and set the resolution to 28x28 pixels, or you can actually use a scanner, scan a real digit and scale the picture down to that format.

```
img = cv2.imread('digit.png')[:,:,0]

img = np.invert(np.array([img]))
```

In order to load our image into the script, we use the *imread* function of OpenCV. We specify the file name and use the index slicing at the end in order to choose just one dimension, in order to fit the format. Also we need to invert the image and convert it into a NumPy array. This is necessary because otherwise it will see the image as white on black rather than black on white. That would confuse our model.

```
prediction = model.predict(img)

print("Prediction: {}".format(np.argmax(prediction)))

plt.imshow(img[0])

plt.show()
```

Now we use the *predict* method to make a prediction for our image. This prediction consists of the ten activations from the output neurons. Since we need to generate a result out of that, we are going to use the *argmax* function. This function returns the index of the highest value. In this case this is equivalent to the digit with the highest probability or activation. We can then visualize that image with the *imshow* method of Matplotlib and print the prediction.

Prediction: 7

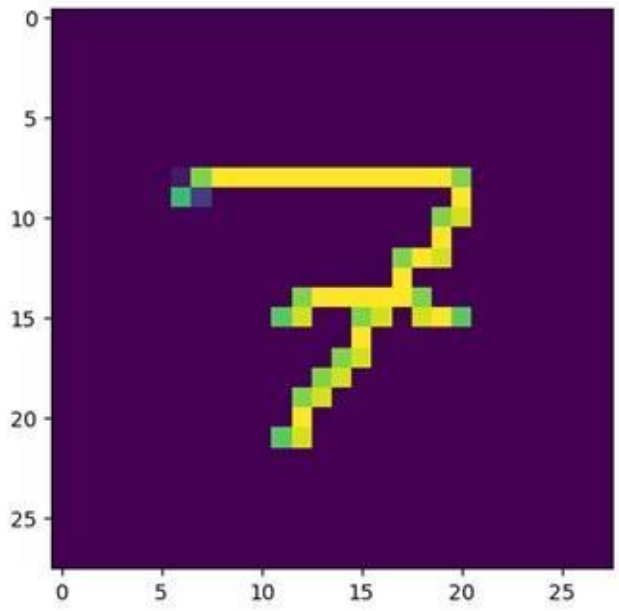


Fig. 2.1: Classified Digit 7

Even though our model is pretty accurate it might still make some mistakes, especially if you tend to write digits in a very unusual way. However this example should have helped you to better understand neural networks and also to get a feeling of how to work with them in Python.

4 – GENERATING TEXTS

In machine learning we are oftentimes dealing with sequential data. Not every input is looked at separately but in context of the previous data.

RECURRENT NEURAL NETWORKS

We already mentioned that for these kinds of tasks, recurrent neural networks are the best choice. Remember: Recurrent layers not only forward their output to the next layer but can also send it to their own or to the previous layer.

This type of network is therefore especially effective when we are dealing with time-dependent data. Examples for this are weather data, stock prices and other values that change sequentially.

In this chapter however we are going to focus on the generation of texts. Texts can also be seen as sequential data, since after every combination of letters a certain “next letter” follows. So here we are not just looking at the last one letter but at the last 20 or 30 letters.

What we want to do here is to get our neural network to generate texts similar to those of the famous poet Shakespeare. For this we are just going to show our model original texts and then train it to generate similar texts itself.

LONG-SHORT-TERM MEMORY (LSTM)

When we talked about recurrent neural networks in the first chapter, we looked at the most basic version of recurrent neurons. These were ordinary neurons that were just connected to themselves or to neurons of the previous layer.

These are definitely useful but for our purposes another type is better suited. For this task we are going to use so-called *LSTM neurons*. This stands for *Long-Short-Term Memory* and indicates that these neurons have some sort of retention.

The problem with ordinary recurrent neurons is that they might forget important information because they don't have any reliable mechanisms that prioritize information based on relevance. Let's look at a quick example. Read the following review of a product:

Awesome! This drink tastes wonderful and reminds me of a mixture of kiwis and strawberries. I only drank half of it but I will definitely buy it again!

When you read this review and you want to tell a friend about it in a couple of days, you will definitely not remember every single word of it. You will forget a lot of words like "I", "will" or "this" and their position, unless you read the text multiple times and try to memorize it.

Primarily you will remember the terms like "awesome", "wonderful", "mixture of kiwis and strawberries" and "definitely buy it again". This is because these are the essential words. And an LSTM network does the same thing. It filters out the unimportant information and only remembers the essential content. These words and phrases are the most important things to look at, in order to determine if this review is positive or negative. Words like "I", "will" or "this" may also appear in a negative review.

We are not going to talk too much about the detailed workings of LSTMs. This would be too much for this chapter and this book. For our purposes it is important to understand that LSTMs have mechanisms that focus on the retention of the essential data. Therefore we use these neurons instead of the default recurrent neurons.

LOADING SHAKESPEARE'S TEXTS

As I already mentioned, we are going to need a decent amount of Shakespeare's texts, in order to train our model. Thus we will now start to load this data into the script. We are going to use the file that is also used by the official Tensorflow Keras tutorials.

Link: https://bit.ly/37IjtMs

This is a shortened link to the file. However, don't bother downloading it manually, since we are going to load it directly into the script, using the full link.

Of course, you can also use all kinds of different text files here. You can export WhatsApp chats and use them as training data, you can download speeches of Donald Trump and use these. Feel free to use whatever you like. The output will then be similar to the training data.

For this chapter we will need the following imports:

```
import random
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.layers import Activation, Dense, LSTM
```

We are going to talk about the individual classes and modules, when we use them. The first step is to now download the file.

```
filepath = tf.keras.utils.get_file('shakespeare.txt',
'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')

text = open(filepath, 'rb')\
   .read().decode(encoding='utf-8').lower()
```

We use the *get_file* method of *keras.utils* in order to download the file. The first parameter is the filename that we choose and the second one is the link. After that we open the file, decode it and save it into a variable. Notice that we are using the *lower* function at the end. We do this because it drastically increases the performance, since we have much less possible characters to choose from. And for the semantics of the text the case is irrelevant.

PREPARING DATA´

The problem that we have right now with our data is that we are dealing with text. We cannot just train a neural network on letters or sentences. We need to convert all of these values into numerical data. So we have to come up with a system that allows us to convert the text into numbers, to then predict specific numbers based on that data and then again convert the resulting numbers back into text.

Also I am not going to use the whole text file as training data. If you have the capacities or the time to train your model on the whole data, do it! It will produce much better results. But if your machine is slow or you have limited time, you might consider just using a part of the text.

```
text = text[300000:800000]
```

Here we select all the characters from character number 300,000 up until 800,000. So we are processing a total of 500,000 characters, which should be enough for pretty decent results.

CONVERTING TEXT

Now we need to start building a system, which allows us to convert characters into numbers and numbers into characters. We are not going to use the ASCII codes but our own indices.

```
characters = sorted(set(text))

char_to_index = dict((c, i) for i, c in enumerate(characters))
index_to_char = dict((i, c) for i, c in enumerate(characters))
```

We create a sorted set of all the unique characters that occur in the text. In a set no value appears more than once, so this is a good way to filter out the characters. After that we define two structures for converting the values. Both are dictionaries that enumerate the characters. In the first one, the characters are the keys and the indices are the values. In the second one it is the other way around. Now we can easily convert a character into a unique numerical representation and vice versa. Notice that the *enumerate* function doesn’t use the ASCII values but just indexes every single character.

SPLITTING TEXT

In this next step, we will need to split our text into sequences, which we can use for training. For this, we will define a sequence length and a step size.

```
SEQ_LENGTH = 40
STEP_SIZE = 3

sentences = []
next_char = []
```

Our goal is to create a list of multiple sequences and another list of all the “next characters” that follow these sequences. In this case, we chose a sequence length of 40 and a step size of three. This means that our base sentences will be 40 characters long and that we will jump three characters from the start of one sentence in order to get to the start of the next sentence. A step size that is too small might result in too many similar examples where as a step size that is too large might cause bad performance. When choosing the sequence length we must also try to find a number that produces sentences that are long enough but also not too long so that our model doesn’t rely on too much previous data.

Now we are going to fill up the two empty lists that we just created. These will be the features and the targets of our training data. The text sequences will be the input or the features and the next characters will be the results or the targets.

```
for i in range(0, len(text) - SEQ_LENGTH, STEP_SIZE):
    sentences.append(text[i: i + SEQ_LENGTH])
    next_char.append(text[i + SEQ_LENGTH])
```

Here we run a for loop and iterate over our text with the given sequence length and step size. The control variable *i* gets increased by *STEP_SIZE* with each iteration.

Additionally, in every iteration, we add the sequence from *i* up to *i* plus the sequence length, to our list. In our case we start with the first 40 characters, save them, then shift the start by three characters, save the next 40 characters and so on. Also we save every “next character” into our second list.

CONVERT TO NUMPY FORMAT

This training data now needs to be converted into numerical values and then into NumPy arrays.

```
x = np.zeros((len(sentences), SEQ_LENGTH,
            len(characters)), dtype=np.bool)
```

```
y = np.zeros((len(sentences),
              len(characters)), dtype=np.bool)
```

For this we first create two NumPy arrays full of zeroes. These zeroes however are actually *False* values, because our data type is *bool* which stands for *Boolean*. The *x* array is three-dimensional and it shapes is based on the amount of sentences, the length of those and the amount of possible characters. In this array we store the information about which character appears at which position in which sentence. Wherever a character occurs, we will set the respective value to one or *True*.

The *y* array for the targets is two-dimensional and its shape is based on the amount of sentences and the amount of possible characters. Here we also work with bools. When a character is the next character for a given sentence, we set the position to one or *True*. Now we need to fill up these two arrays with the proper values.

```
for i, satz in enumerate(sentences):
    for t, char in enumerate(satz):
        x[i, t, char_to_index[char]] = 1
    y[i, char_to_index[next_char[i]]] = 1
```

This code does exactly what I just described above. We use the enumerate function two times so that we know which indices we need to mark with a one. Here we use our *char_to_index* dictionary, in order to get the right index for each character.

To make all of that a little bit more clear, let us look at an example. Let’s say the character ‘g’ has gotten the index 17. If this character now occurs in the third sentence (which means index two), at the fourth positon (which means index three), we would set *x[2,3,17]* to one.

BUILD RECURRENT NEURAL NETWORK

Now our training data is perfectly prepared and has the right format that our neural network can work with. But this network has to be built yet. Let's look at the needed imports again and talk a little bit about their role:

```
import random
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.layers import Activation, Dense, LSTM
```

The library *random* will be used later on in a helper function. We have already used *numpy*. We used the basic *tensorflow* library was used to load the data from the internet.

For building our neural network, we will once again need the *Sequential* model from Keras. This time however, we will use a different optimizer, namely the *RMSprop*. And of course we also import the layer types, which we are going to use.

```
model = Sequential()
model.add(LSTM(128,
          input_shape=(SEQ_LENGTH,
                       len(characters))))
model.add(Dense(len(characters)))
model.add(Activation('softmax'))
```

Our model is actually quite simple. The inputs go directly into an *LSTM* layer with 128 neurons. We define the input shape to be the sequence length times the amount of possible characters. We already talked about how this layer works. This layer is the memory of our model. It is then followed by a *Dense* layer with as many neurons as we have possible characters. This is our hidden layer. That adds complexity and abstraction to our network. And then last but not least we have the output layer, which is an *Activation* layer. In this case it once again uses the *softmax* function that we know from the last chapter.

```
model.compile(loss='categorical_crossentropy',
          optimizer=RMSprop(lr=0.01))

model.fit(x, y, batch_size=256, epochs=4)
```

Now we compile our model and optimize it. We choose a learning rate of 0.01. After that we *fit* our model on the training data that we prepared. Here we choose a *batch_size* of 256 and four *epochs*. The batch size indicates how many sentences we are going to show the model at once.

HELPER FUNCTION

The model is now trained and ready to generate some predictions. However, the output that we get is not really satisfying. What our network gives us as a result is actually just the next character in the numerical format.

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

I copied this function from the official Keras tutorial.

Link: https://keras.io/examples/lstm_text_generation/

This function will later on take the predictions of our model as a parameter and then choose a “next character”. The second parameter *temperature* indicates how risky or how unusual the pick shall be. A low value will cause a conservative pick, whereas a high value will cause a more experimental pick. We will use this helper function in our final function.

GENERATING TEXTS

The neural network that we have returns as a result an array with bool values. From those we need to extract the choice using our helper function. But then we still end up with a numerical value which represents just one character. So we need to convert this number into a readable representation and we also need to produce not just one “next character” but multiple.

```
def generate_text(length, temperature):
    start_index = random.randint(0, len(text) - SEQ_LENGTH - 1)
    generated = ''
    sentence = text[start_index: start_index + SEQ_LENGTH]
    generated += sentence
    for i in range(length):
        x_predictions = np.zeros((1, SEQ_LENGTH, len(characters)))
        for t, char in enumerate(sentence):
            x_predictions[0, t, char_to_index[char]] = 1

        predictions = model.predict(x_predictions, verbose=0)[0]
        next_index = sample(predictions,
                            temperature)
        next_character = index_to_char[next_index]

        generated += next_character
        sentence = sentence[1:] + next_character
    return generated
```

This is the function that we are going to use for that. It looks more complicated than it actually is. So don’t panic. We will analyze it step-by-step. First we generate a random entry point into our text. This is important because our network needs some starting sequence in order to generate characters. So the first part will be copied from the original text. If you want to have text that is completely generated, you can cut the first characters out of the string afterwards.

We then convert this initial text again into a NumPy array. After that we feed these x-values into our neural network and predict the output. For this we use the *predict* method. This will output the probabilities for the next characters. We then take these predictions and pass them to our helper function. You have probably noticed that we also have a *temperature* parameter in this function. We directly pass that to the helper function.

In the end we receive a choice from the *sample* function in numerical format. This choice needs to be converted into a readable character, using our second dictionary. Then we add this character to our generated text and repeat this process until we reach the desired length.

RESULTS

Let’s take a look at some samples. I played around with the parameters, in order to diversify the results. I am not going to show you all of the results, but just some snippets that I found interesting.

```
print(generate_text(300, 0.2))
print(generate_text(300, 0.4))
print(generate_text(300, 0.5))
print(generate_text(300, 0.6))
print(generate_text(300, 0.7))
print(generate_text(300, 0.8))
```

Settings: *Length: 300, Temperature: 0.4 (Conservative)*

ay, marry, thou dost the more thou dost the mornish,
and if the heart of she gentleman, or will,
the heart with the serving a to stay thee,
i will be my seek of the sould stay stay
the fair thou meanter of the crown of manguar;
the send their souls to the will to the breath:
the wry the sencing with the sen

Settings: *Length: 300, Temperature: 0.6 (Medium)*

warwick:
and, thou nast the hearth by the createred
to the daughter, that he word the great enrome;
that what then; if thou discheak, sir.

clown:
sir i have hath prance it beheart like!

Settings: *Length: 300, Temperature: 0.8 (Experimental)*

i hear him speak.
what! can so young a thordo, word appeal thee,
but they go prife with recones, i thou dischidward!
has thy noman, lookly comparmal to had ester,
and, my seatiby bedath, romeo, thou lauke be;
how will i have so the gioly beget discal bone.

clown:
i have seemitious in the step--by this low,

As you can see, the results are far away from perfection. But considering the fact that our computer doesn’t even understand what words or sentences are, this is quite impressive nevertheless. Most sentences don’t make a lot of sense and you can find some made up words but the texts are unique and it learned to generate those just by reading some Shakespeare literature.

Now it is your turn! Experiment around with that code. Tweak the parameters. Use different training texts. Maybe you want to export your WhatsApp chats and train the model on those. The text you use for training drastically influences the final generated texts.

5 – IMAGE AND OBJECT RECOGNITION

In the first chapter we talked about convolutional neural networks and the fact that this type of neural network is especially effective when processing image and sound data. They excel at recognizing patterns and segmenting the data into so-called *features*. Thus they perform much better than ordinary feed forward neural networks.

WORKINGS OF CNNS

Convolutional neural networks derive their name from the fact that they consist of *convolutional layers*. These oftentimes followed by *pooling layers*, which filter the resulting information and simplify it. A convolutional neural network may consist of multiple such combinations. For example, we could have an input layer, followed by a convolutional layer, followed by a pooling layer and this combination repeats three times. After that we add a couple of dense layers at the end and a final output layer.

CONVOLUTIONAL LAYER

Similar to an ordinary layer, convolutional layers also get their input from previous neurons, process it and send the result to the next layer. The processing of convolutional layers is called *convolution*. We are going to talk about this in a second.

Most of the time convolutional layers are two- or three-dimensional. When we load black-and-white images and classify those, we are dealing with two dimensions. Working with colored images happens in three dimensions. The individual values represent the pixels.

Let’s take a superficial look at how such a processing could look like.

Fig. 5.1: Image of a car

When you look at this picture, you immediately recognize that it shows a car. This is because you have already seen numerous cars throughout your whole life and you now the label “car”. For a computer that isn’t so obvious. If we have ten possible objects for classification (car, plane, cat, dog, table etc.), it will be

impossible for it, to classify these objects by nature.

With ordinary feed forward neural networks it would just look at all the pixels, try to make some connections and then make a prediction, which will probably be inaccurate. What works when it comes to handwritten digits doesn't work as easily in more complicated examples. Handwritten digits are just clear shapes, black on white. Images of a car or a dog can be shot from different perspectives on different backgrounds with different colors. Sometimes the pictures will be brighter and sometimes darker. This will inevitably confuse an ordinary neural network.

What stays the same though, are all the features, attributes and patterns of those objects. A car has tires, wheels, side mirrors, a windshield etc. The perspective might always be different but most of the features can always be found. And this is exactly what convolutional neural networks do. They extract the relevant features from the images.

Fig. 5.2: Example of feature extraction

When you think about it, this is the exact same process that we humans do. We don't look at every single "pixel" of our field of view. Instead we look at the big picture and recognize the features of a car. We see wheels, a license plate, the shape and before we even have time to think about it, we know that what we see is a car.

Let's get a little bit deeper into the technical details of convolutional layers. Fundamentally a convolutional layer is just a matrix that we apply onto our data.

0.762	0.561	0.022
0.675	0.132	0.982
0.111	0.671	0.231

We will take this 3x3 Matrix as an example. This matrix is now our filter or our convolutional layer. Additionally we can also imagine a picture which is encoded in the same way. Each pixel would have a value in between 0 and 1. The higher the value, the brighter the pixel and the lower the value, the darker the pixel would be. Zero would then equal black and one would equal white.

What we now do is we apply our matrix onto each 3x3 field of our image. Applying it means calculating the scalar product of both matrices. We take the first 3x3 pixels of our image and calculate the scalar product with our filter. Then we shift our selection by one column and apply the same operation to the next 3x3 pixels. The scalar product is then the new resulting pixel.

Now you are probably asking yourself two questions. First of all: Why are we doing all of that? And second of all: Where do we get the values for our filter from? The answer to both of this question is kind of the same.

Initially we use random and irrelevant values. Therefore the filtering has no real effect in the beginning. But we are operating in the world of machine learning. So we start filtering our images with random values. Then we look at the results and evaluate the accuracy of our model. Of course it is going to be quite low. Thus we tweak the parameters using backpropagation over and over again, so that our results approve. This works because our filters become pattern detectors over time because of all this training. When certain patterns occur over and over again in different pictures, the respective values in our filters and channels will be accordingly high.

Most of the time these filters are not 3x3 matrices but 64x64 matrices or even bigger ones. And a convolutional layer consists of multiple such filters. Also, we oftentimes line up many convolutional layers in a row. Thus, everything gets kind of complex and sophisticated and very effective.

POOLING LAYER

Pooling layers are the layers that usually follow convolutional layers. Their primary role is to simplify the output of those. Roughly speaking we could say that these layers make sure that our model is focusing on the essential parts before we forward our data even further.

The most popular type of pooling is the so-called *max-pooling*. Here we take 2x2 matrices of our images and only take on the one highest value for further

processing.

Pooling reduces the required space, increases the computational speed and counteracts overfitting. In general it saves resources without causing worse performance.

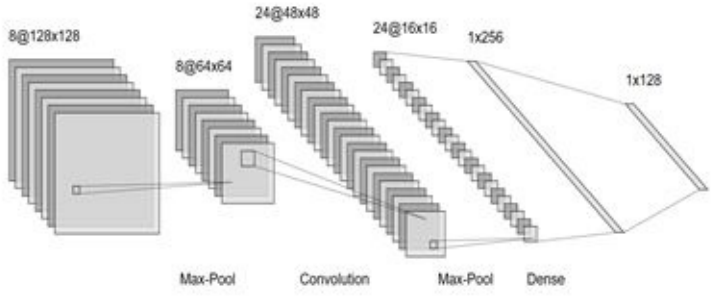


Fig. 5.3: Structure of a convolutional neural network

Here we see what a convolutional neural network could look like. Initially the inputs flow into a convolutional layer which uses eight 128x128 filters. The result is then forwarded into a max-pooling layer which reduces it to its essential parts. After that we repeat the process with two more layers in a smaller resolution. Then we feed the information into two dense layers and end up with a final classification.

LOAD AND PREPARE IMAGE DATA

Let us now get to the implementation part of the chapter. In this chapter we are going to use another Keras dataset, which contains numerous images of ten different categories. These are the following:

['Plane', 'Car', 'Bird', 'Cat', 'Deer',
'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

This dataset contains tens of thousands of images of different objects with their respective class. Our goal here is to train a convolutional neural network on that data, in order to then classify other images that the model has never seen before.

For this we will need the following libraries:

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras import datasets, layers, models
```

If haven't installed OpenCV yet, you need to do this. For this just open your command line and enter:

```
pip install opencv-python
```

We again receive the data already split up into two tuples, when we load it from Keras.

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
train_images, test_images =
train_images / 255.0, test_images / 255.0
```

This time we load the *cifat10* dataset with the *load_data* method. We also normalize this data immediately after that, by dividing all values by 255. Since we are dealing with RGB values, and all values lie in between 0 and 255, we end up with values in between 0 and 1.

Next, we define the possible class names in a list, so that we can label the final numerical results later on. The neural network will again produce a softmax result, which means that we will use the *argmax* function, to figure out the class name.

```
class_names = ['Plane', 'Car', 'Bird', 'Cat', 'Deer',
```

```
'Dog', 'Frog', 'Horse', 'Ship', 'Truck']
```

Now we can visualize a section of the data, to see what this dataset looks like.

```
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i][0]])

plt.show()
```

For this we run a for loop with 16 iterations and create a 4x4 grid of subplots. The x-ticks and the y-ticks will be set to empty lists, so that we don't have annoying coordinates. After that, we use the *imshow* method, to visualize the individual images. The label of the image will then be the respective class name.

Fig. 5.4: Images of the Cifar10 dataset with labels

This dataset contains a lot of images. If your computer is not high-end or you don't want to spend too much time on training the model, I suggest you only use a part of the data for training.

```
train_images = train_images[:20000]
train_labels = train_labels[:20000]
test_images = test_images[:4000]
test_labels = test_labels[:4000]
```

Here for example we only use the first 20,000 of the training images and the first 4,000 of the test images. Of course your model will be way more accurate if you use all the images. However, for weak computers this might take forever.

BUILDING NEURAL NETWORK

Now that we have prepared our data, we can start building the neural network.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Here we again define a *Sequential* model. Our inputs go directly into a convolutional layer (*Conv2D*). This layer has 32 filters or channels in the shape of 3x3 matrices. The activation function is the ReLU function, which we already know and the input shape is 32x32x3. This is because we our images have a resolution of 32x32 pixels and three layers because of the RGB colors. The result is then forwarded into a *MaxPooling2D* layer that simplifies the output. Then the simplified output is again forwarded into the next convolutional layer. After that into another max-pooling layer and into another convolutional layer. This result is then being flattened by the *Flatten* layer, which means that it is transformed into a one-dimensional vector format. Then we forward the results into one dense hidden layer before it finally comes to the softmax output layer. There we find the final classification probabilities.

TRAINING AND TESTING

Now we are almost done. We just need to train and test the model before we can use it.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Here we again use the *adam* optimizer and the *sparse categorical crossentropy* loss function.

```
model.fit(train_images,
          train_labels,
          epochs=10,
          validation_data=(test_images, test_labels))
```

We now train our model on our training data in ten epochs. Remember: This means that our model is going to see the same data ten times over and over again.

```
test_loss, test_acc = model.evaluate(test_images,
                                     test_labels,
                                     verbose=2)
```

We use the *evaluate* function to test our model and get the *loss* and *accuracy* values. We set the parameter *verbose* to 2, so that we get as much information as possible.

```
- 1s - loss: 0.8139 - acc: 0.7090
```

Your results are going to slightly differ but in this case I got an accuracy of around 70%. This is quite impressive when you keep in mind that we have ten possible classifications and the chance to be right by guessing is 10%. Also this task is way more complicated that classifying handwritten digits and we also have some similar image types like *car* and *truck* or *horse* and *deer*.

CLASSIFYING OWN IMAGES

However, the interesting part starts now. Since our model is trained, we can now go ahead and use our own images of cars, planes, horses etc. for classification. These are images that the neural network has never seen before. If you don’t have your own images, you can use Google to find some.

Fig. 5.5: Car and horse

I chose these two pictures from Pixabay, since they are license-free.

The important thing is that we get these images down to 32x32 pixels because this is the required input format of our model. For this you can use any software like Gimp or Paint. You can either crop the images or scale them.

Fig. 5.6: Images in 32x32 pixels resolution

Now we just have to load these images into our script, using OpenCV.

```
img1 = cv.imread('car.jpg')
img1 = cv.cvtColor(img1, cv.COLOR_BGR2RGB)
img2 = cv.imread('horse.jpg')
img2 = cv.cvtColor(img2, cv.COLOR_BGR2RGB)
plt.imshow(img1, cmap=plt.cm.binary)
plt.show()
```

The function *imread* loads the image into our script. Then we use the *cvtColor* method, in order to change the default color scheme of BGR (blue, green, red) to

RGB (red, green, blue).

```
plt.imshow(img1, cmap=plt.cm.binary)
plt.show()
```

With the *imshow* function, we can show the image in our script, using Matplotlib.

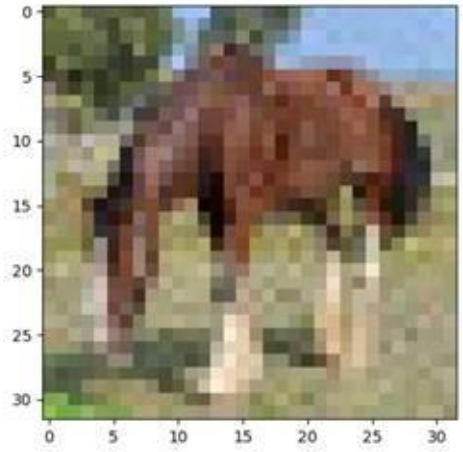


Fig. 5.7: Horse in Matplotlib

We can now use the loaded images as the input for our model, in order to get a prediction.

```
prediction = model.predict(np.array([img1]) / 255)
index = np.argmax(prediction)
print(class_names[index])
```

First we use the *predict* function to get the softmax result. Notice that we are converting our image into a NumPy array and dividing it by 255. This is because we need to normalize it, since our model was trained on normalized values. Then we use the *argmax* function to get the index of the highest softmax activation value. Finally, we print the class name of that index as a result.

Car Horse

The results speak for themselves. These pictures were classified absolutely correct. Of course this will not always be the case. Sometimes a deer will be classified as a horse and vice versa. But the performance of our model is still pretty impressive.

Again I encourage you to experiment and play around with that model. Try to

change the parameters of the network. Try to classify other images. Maybe use completely different training data with different possible class names. Make sure that you feel comfortable using convolutional neural networks.

6 – REVIEW AND RESOURCES

We have learned quite a lot in these five chapters and some of the topics were quite complex. Thus we are going to quickly review all the concepts in this final chapter and add some additional information and resources.

REVIEW: BASICS

The first chapter was probably the most confusing one, since it was highly theoretical and mathematical. It is your choice how deep you want to go into that matter.

If you really want to become a machine learning expert, innovate and develop new technologies in this field, a solid understanding of the detailed mathematics is probably necessary.

However, if you are not interested in the details of machine learning theory and you just want to apply the technologies, this is not necessary. For example you won’t need any higher mathematical skills for developing a basic fitness app which uses machine learning. You only need to know how to use Tensorflow and other libraries properly.

As soon as you go into research or innovation though, there is no way around math.

At this point check if you really understand the following concepts:

- Neural networks and their structure
- Structure of perceptrons
- Activation functions
- Training and testing models
- Error and loss
- Gradient descent algorithm
- Backpropagation

If there is anything that you feel you didn’t quite understand, read through the first chapter one more time. Also learn to google and research problems and questions properly. Every programmer encounters a lot of errors, mistakes and confusions while coding. You cannot cover all of these in one book. Therefore, don’t be afraid to use Google, StackOverflow, Documentations and YouTube. Professional developers do this as well.

REVIEW: NEURAL NETWORKS

In the third chapter we talked about the basics of Tensorflow. We built a first simple neural network and used it to classify handwritten digits. If you didn't understand something at this point, you probably also had some problems with the following chapters, since they build on this one. So make sure that you really master this chapter.

Ask yourself if you understand the following concepts:

- Loading datasets from Keras
- Splitting training and testing data
- Building neural networks with Tensorflow
- Compiling models
- Training and testing models in code
- Loading and preparing your own images

Also take some time to be creative and think about ways in which you could use all of that knowledge. What can you use neural networks for? What problem can you solve? What data could you predict? We did just one example in this chapter. Research some datasets and play around with them.

Keras Datasets: https://keras.io/datasets/

Scikit-Learn Datasets:
https://scikit-learn.org/stable/datasets/index.html

REVIEW: RECURRENT NEURAL NETWORKS

Here the topics got more interesting. We started using recurrent neural networks and thus adding a memory to our model. We used the model to generate poetic texts like those of Shakespeare.

A big part of the work in this chapter was preprocessing and preparing our data and also working with the output in a proper way.

The challenge wasn’t really with building recurrent neural networks. It was pretty easy to add an LSTM layer. But converting our text into a numerical format, then transforming it into NumPy arrays, training the network, then again converting the output and generating text was a challenge. But you need to get used to that. Oftentimes the machine learning model is very easy to build. But when we need to work with real-world data instead of perfect datasets from Keras, things become harder. In the real world the data is chaotic and not at all preprocessed or prepared. So it is very crucial that you master this part as well.

Ask yourself if you understand the following concepts:

- Loading online data into the Python script
- Preprocessing and structuring data so that it can be further processed by neural networks
- Basic advantages of LSTM layers
- Workings of recurrent neural networks

Of course, here you should also experiment a little bit. Use different data. Everything that is sequential will work in some way. Maybe you want to use different texts or a different network structure. Maybe you want to predict weather data or stock prices. Be creative and work on your own mini-projects to master the skills.

REVIEW: CONVOLUTIONAL NEURAL NETWORKS

Last but not least we also used convolutional neural networks. These are especially useful when we need to recognize patterns like in images or audio files.

We used a Keras dataset with numerous pictures of ten different possible object types. Our neural network was able to classify these objects with an accuracy of 70%, which is pretty impressive.

Ask yourself if you understand the following concepts:

- Convolutional layers
- Pooling layers and pooling functions
- Filtering or channels and matrices

And once again: Experiment! Maybe create your own set of pictures and labels. Then use it as training data and build a model that classifies the people of your family or the different tools on your desk. Use other datasets and research how to use convolutional neural networks for audio processing. Maybe you can build a voice or speech recognition bot.

NEURALNINE

One place where you can get a ton of additional free resources is *NeuralNine*. This is my brand and it has not only books but also a website, a YouTube channel, a blog, an Instagram page and more. On YouTube you can find high quality video tutorials for free. If you prefer text, you might check out my blog for free information. The *neuralnine* Instagram page is more about infographics, updates and memes. Feel free to check these out!

YouTube: https://bit.ly/3a5KD2i

Website: https://www.neuralnine.com/

Instagram: https://www.instagram.com/neuralnine/

Can’t wait to see you there! J

WHAT'S NEXT?

When you have understood the concepts in this book and are able to apply them, you have progressed in your programming career a lot. You have made a huge step. These skills are invaluable in today's world but even more so in the future.

You are able to digitalize data from the real world, process it, forward it into a neural network and then make predictions or decisions based on that. Some might argue that this is a little bit like magic.

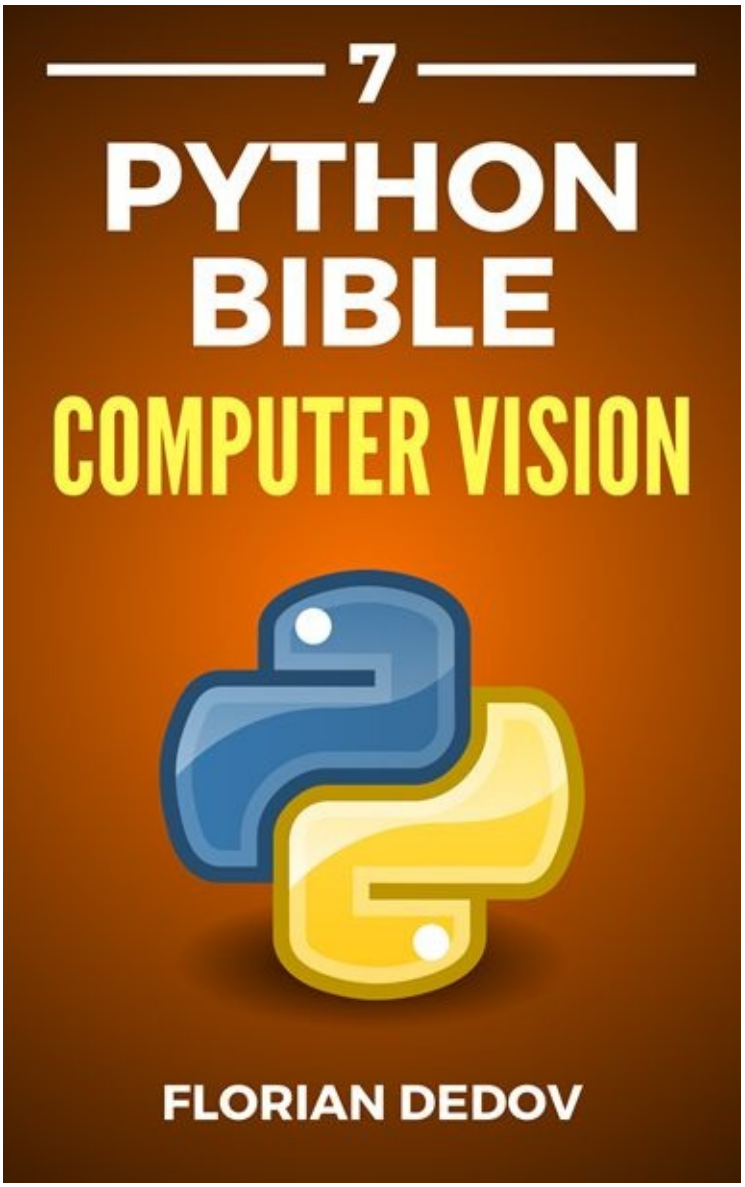
Depending on the field of application that you are interested in, you will need to learn some additional skills. No book on earth can teach you everything. If you want to apply that knowledge in the finance field, you will need to learn about stocks, insurances etc. If you want to use it for medical or sports purposes, you will need to educate yourself on that. Computer science and mathematics alone will probably not get you very far, unless you go into research. However, it is the basis of everything and you should now have this basis. Whether you choose to predict the weather or build the next space exploration company is up to you.

If you are interested in finance programming however, check out my Amazon author page. There I have some additional books and one of them (volume five) is about finance programming in Python. Alternatively you can also find them on the NeuralNine website.

Books: https://www.neuralnine.com/books/

Last but not least, a little reminder. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!


7
PYTHON
BIBLE
COMPUTER VISION
FLORIAN DEDOV

THE
PYTHON BIBLE

VOLUME SEVEN

COMPUTER VISION

BY

FLORIAN DEDOV

TABLE OF CONTENT

INTRODUCTION

Computer vision is one of the most exciting and interesting topics in computer science. This field focuses on how computers perceive and process image and video data. The technologies of this area are fundamental for our future, with virtual reality and internet of things becoming more and more important. With computer vision we can make unreadable, fuzzy and poorly lit pictures readable. We can also recognize objects and faces in real-time. And we can apply filters, transformations and numerous awesome effects.

In the programming language Python we can use the library OpenCV (also available for other programming languages), which allows us to see a lot of impressive results, with very few lines of code.

These skills are essential for many applications like surveillance and security systems. The whole field of robotics is largely based on computer vision as well. But also in medicine, image processing, filmmaking, industry and automation, computer vision is very important.

THIS BOOK

In this book you are going to learn, how to make interesting computer vision applications, using Python and OpenCV. Each chapter will start with a little bit of theory, followed by practical examples and applications.

For this book however, you will need some advanced Python skills already and a basic understanding of data science. You should be comfortable using NumPy, Matplotlib and Pandas. If you are not, I recommend reading my data science book (volume three) or learning the material from other sources. We will use some of these libraries in this book, but I am not going to explain their basic workings in detail here.

Also, if you are missing some Python skills, you can take a look at my Amazon author page. There you will find the full Python Bible series which contains two volumes for the basics of Python, followed by volumes about data science, machine learning and financial analysis with Python. Of course you can also learn these things from other sources. However, in this book there won't be any explanations of basic Python syntax or the data science libraries mentioned above.

My Amazon Author Page: https://amzn.to/38yY9cG

HOW TO READ THIS BOOK

Essentially it is up to you how you are going to read this book. If you think that the initial chapters are not interesting to you or that you already know everything, you can skip them. Also you can read the book from cover to cover without ever writing a single line of code yourself. But I don't recommend all of this.

I personally recommend you to read all the chapters in the right order, since they build on top of each other. The code samples also work without the previous chapters but then you will lack the understanding of the material and you will have no clue why something works or not.

Additionally it is tremendously important that you code along while reading. That's the only way you will really understand the topics of this book. There will be a lot of code in the individual chapters. Read through it, understand it but also implement it on your own machine and experiment around. What happens when you tweak individual parameters? What happens when you add something? Try everything!

That's all that needs to be said for now. I wish you a lot of success and fun learning about computer vision in Python. I hope that this book will help you to progress in your programming career.

Just one little thing before we start. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you have learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

If you are interested in free educational content about programming and machine learning, check out: https://www.neuralnine.com/

INSTALLING LIBRARIES

For this book we will need a couple of libraries that are not part of the default core stack of Python. This means that we need to install them separately using *pip*.

First of all, we are going to install the data science stack:

pip install numpy

pip install matplotlib

pip install pandas

As mentioned in the introduction, you should be comfortable using these libraries already. They are not the main focus of this book but they are going to help us a lot with certain tasks. The main library that we will need for this book is OpenCV. We also install it using pip:

pip install opencv-python

All these libraries will do a lot of the work for us that we would otherwise have to do ourselves.

1 – LOADING IMAGES AND VIDEOS

Before we start with the processing of images and videos, we will need to learn how to load the respective data into our script. This is what we are going to learn in this first chapter.

For this chapter we will need the following imports:

```
import cv2 as cv
import matplotlib.pyplot as plt
```

Notice that OpenCV was named *opencv-python* in the installation but in order to use it we import *cv2*. I chose to use the *cv* alias here, so that the code is also compatible with eventual future versions. Also, we import Matplotlib, which is useful when working with images.

LOADING IMAGES

In order to load an image, we first need to prepare one. Here you can just download images from the internet or use your own photos. For this book I will use license-free images from Pixabay or images that I made myself.

```
img = cv.imread('car.jpg', cv.IMREAD_COLOR)
```

To now load the image into our script, we use the *imread* function of OpenCV. First we pass our file path, followed by the color scheme that we want to use.

In this case we choose *IMREAD_COLOR* because we want to work with the colored version of the image. We can now go ahead and show the image.

```
cv.imshow('Car', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

For this we use the *imshow* function, which accepts the title of the images (the identifier) and the image itself. After that you can see two more commands, which we are going to use quite often. The first one is the *waitKey* function, which waits for a key to be pressed, before the script continues. The parameter it takes is the delay. In this case we chose zero. After that, we have the *destroyAllWindows* function, which does what the name says.

Fig. 1.1: Image of a car in OpenCV

If we choose to use the *IMREAD_GRAYSCALE* color scheme, instead of the *IMREAD_COLOR* color scheme, our image would look like this.

Fig. 1.2: Grayscale version of the image

Of course if you are reading a black-and-white version of this book, you will not see a difference. So run the code yourself to see the difference.

SHOWING IMAGES WITH MATPLOTLIB

Another way of showing our images is by using Matplotlib. Here we also have and *imshow* function, which visualizes images.

```
plt.imshow(img)
plt.show()
```

The problem that we will encounter here however is, that Matplotlib uses a different color scheme from the one OpenCV uses. This results in the following image.

Fig. 1.3: Image of a car in Matplotlib

Again if you are reading a non-color version of this book, you might not see the clear difference here. In this case, you need to execute the code.

CONVERTING COLOR SCHEMES

While OpenCV uses the RGB color scheme (red, green, blue), Matplotlib uses the BGR color scheme (blue, green, red). This basically means that the values for blue and red are swapped in our image. In order to change that, we can convert the color scheme.

```
img = cv.cvtColor(img, cv.COLOR_RGB2BGR)
```

By using the *cvtColor* function, we can convert our image. For this we first pass the image itself, followed by the conversion that we want to happen. In this case we choose *COLOR_RGB2BGR*, since we want to convert our image from RGB to BGR. After that we can see the right colors in Matplotlib.

LOADING VIDEOS

Besides images, we can also load videos into our script, using OpenCV.

```
import cv2 as cv

video = cv.VideoCapture('city.mp4')

while True:
    ret, frame = video.read()

    cv.imshow('City Video', frame)

    if cv.waitKey(30) == ord('x'):
        break

video.release()
cv.destroyAllWindows()
```

Here we use the *VideoCapture* object and pass the file path of our video. Then we run an endless loop which constantly reads one frame after the other, using the *read* function. We then show this frame with the *imshow* method. At the end of our loop, we then use the *waitKey* function that checks if the ‘x’ key was pressed (replacable). If we press it, the script terminates. As a delay we choose 30, which means that we will wait 30 milliseconds before we show the next frame. One second has 1000 milliseconds. When we show one frame every 30 milliseconds, we end up with an FPS rate of 33 frames per second. Of course you can change the values if you want. Last but not least, we then call the *release* function to release our capture. This is like closing a stream.

Fig. 1.4: Screenshot of the video

Now we have one problem. When the video is finished and we don’t terminate it manually, our script crashes and we get an error. We can easily fix this with a little if-statement, which checks for return values.

```
while True:
   ret, frame = video.read()

   if ret:
      cv.imshow('City Video’, frame)

      if cv.waitKey(30) == ord(‘x’):
         break
   else:
      break
```

Alternatively we can also run the same video over and over again in the loop. This is done by replacing the *break* with a line that resets the video.

```
while True:
   ret, frame = video.read()

   if ret:
      cv.imshow('City Video', frame)

      if cv.waitKey(30) == ord('x'):
         break
   else:
      video = cv.VideoCapture('city.mp4')
```

Now every time the video is finished, it will start all over again.

LOADING CAMERA DATA

Last but not least let us talk about getting our camera data into the script. Instead of specifying a file path in our *VideoCapture* object, we can specify a number (index of the camera), in order to view the data of our camera in real-time.

```
import cv2 as cv

video = cv.VideoCapture(0)

while True:
    ret, frame = video.read()

    if ret:
        cv.imshow('City Video', frame)

        if cv.waitKey(1) == ord('x'):
            break
    else:
        video = cv.VideoCapture('city.mp4')

video.release()
cv.destroyAllWindows()
```

Here we choose zero as the video source, which will be the primary camera. If you have two, three or more cameras, you can change that index, to select those.

We can also tweak the delay of the *waitKey* function, in order to adjust the FPS. If you are installing a camera for surveillance, you might want to choose a lower FPS rate (and thus a higher delay) because the cameras are running 24/7 and you don't want to waste too much disk space. But if you want to play around with some filters or effects, you will choose a delay of around one second.

2 – FUNDAMENTAL EDITING

Now that we know how to load images, videos and camera data, we can start talking about some fundamental editing.

DRAWING

Let’s first take a look at how to draw on our images. With OpenCV we can paint simple shapes like lines, rectangles and circles onto our pictures.

```
cv.line(img, (50,50), (250,250), (255,255,0), 15)
cv.rectangle(img, (350,450), (500,350), (0,255,0), 5)
cv.circle(img, (500, 200), 100, (255,0,0), 7)
```

For this we use the functions *line*, *rectangle* and *circle*. But of course we also have others like *fillPoly* to fill polygons.

The parameters vary from function to function. To *line* and *rectangle* we pass the two points right after the image in form of a tuple. These are the starting point and the end point. The two values are the x- and the y-coordinates. Then we also pass a triple with the RGB values for the color. Finally, we specify the thickness of the line.

For the circle on the other hand, we first need to specify one point (which is the center) and then the radius. At the end, we again specify the line thickness.

Fig. 2.1: Drawing shapes with OpenCV

In the figure above you can see what this looks like. This primitive drawing is primarily useful when we want to highlight something specific. For example when we want to but a box around a certain object when it is recognized. We are going to do this in a later chapter.

DRAWING WITH MATPLOTLIB

It is also possible to use Matplotlib for our drawings. I am not going to get into the details of Matplotlib plotting here, since I covered it in volume three already. However, let's look at a quick example of plotting a function onto our image.

```
x_values = np.linspace(100,900,50)
y_values = np.sin(x_values) * 100 + 300

plt.imshow(img, cmap='gray')
plt.plot(x_values,y_values, 'c', linewidth=5)
plt.show()
```

Here for example we plot a modified sine function onto our car. Of course this doesn't make any sense in this particular case but there are definitely scenarios in which that might be useful.

Fig. 2.2: Plotting function over the image

COPYING ELEMENTS

What we can also do with OpenCV is to copy or cut certain parts of the image and then use them elsewhere. This is done with index slicing.

```
img[0:200, 0:300] = [0, 0, 0]
```

In this example we replace all the pixels from 0 to 200 on the y-axis and from 0 to 300 on the x-axis with black pixels. We assign a list with three zeros that represent the RGB color codes.

Fig. 2.3: Replaced upper-left corner

With index slicing, as we just used it, we can also move or copy various parts of our image.

```
copypart = img[300:500, 300:700]

img[100:300, 100:500] = copypart
```

With this code we store all the pixels from 300 to 500 on the y-axis and from 300 to 700 on the x-axis in a temporary variable. Then we assign these values to another part of the image, which has the same resolution.

```
img[300:500, 300:700] = [0,0,0]
```

If you want to move the part, instead of copying it, you should replace the initial pixels with black color. However, be careful with intersections because you might overwrite the part you just moved. It is probably a better idea to first black out the initial part and then paste the copied piece.

Fig. 2.4: Moving and copying of image parts

The difference is obvious. On the one picture we just copied and pasted the section, whereas we replaced it on the other one.

SAVING IMAGES AND VIDEOS

When we are done with the processing of our images and videos, we will want to export them. For images this is quite easy.

```
cv.imwrite('car_new.jpg', img)
```

We just use the *imwrite* method of OpenCV. Depending on the file type we specify, the image will be encoded accordingly.

SAVING VIDEOS

Saving videos is also not really complex. But here we need to define some additional things.

```
capture = cv.VideoCapture(0)
fourcc = cv.VideoWriter_fourcc(*'XVID')
writer = cv.VideoWriter('video.avi', fourcc, 60.0, (640,480))
```

Besides the *VideoCapture* we also need to specify the so-called *FourCC*. This is the codec that we are going to use to encode the video data and it specifies the format. In this case we pick XVID. This is an open-source variation of the MPEG-4 codec.

Additionally, we also need to define a *VideoWriter*. To this we pass the file name, the codec, the frame rate and the desired resolution. In this case we save our video into the *video.avi* file with 60 FPS and a resolution of 640x480 pixels.

```
while True:
    ret, frame = capture.read()

    writer.write(frame)

    cv.imshow('Cam', frame)

    if cv.waitKey(1) == ord('x'):
        break
```

In our endless loop we then call the *write* function with each iteration, in order to write our frames into the file.

```
capture.release()
writer.release()
cv.destroyAllWindows()
```

Last but not least, we release all the components and close all windows. That is how we save video data into a file with OpenCV.

3 – THRESHOLDING

Now we are getting into one of the most interesting topics of computer vision, which is *thresholding*. Here everything is about segmenting our image data. This is important fr technologies like object and face recognition but also for the filtering of information and the optimization of poorly taken images.

INSERT LOGO

In the following section we are going to use thresholding in order to make a logo partly transparent and then insert it into an image.

Fig. 3.1: Image of a workspace

This license-free image is the main background and we want to insert the logo in the upper-left corner.

Fig. 3.2: Sample logo

This will be our sample logo. It is a white M on blue background. What we now want is the M to be transparent, so that we can look through it onto our background. For this we will use thresholding. We will find the white area with OpenCV and then make it transparent.

```
img1 = cv.imread('laptop.jpg')
img2 = cv.imread('logo.png')
```

For this we first load both of the images into our script. In the next step we get to thresholding.

```
logo_gray = cv.cvtColor(img2, cv.COLOR_RGB2GRAY)
ret, mask = cv.threshold(logo_gray, 180, 255, cv.THRESH_BINARY_INV)
```

We are converting the logo into the grayscale color scheme because we are only interested in the while color. For this we use the color mode *COLOR_RGB2GRAY*. Then we use the *threshold* function. To it we pass the grayscale logo and also from which color value, we want to change to which color value. In this example, we choose to convert every pixel that has a higher value than 180 (light gray) to 255 (totally white). For this we use the *THRESH_BINARY_INV* procedure.

As one of the return values we get the mask of this image, which we can visualize.

```
cv.imshow('Mask', mask)
```

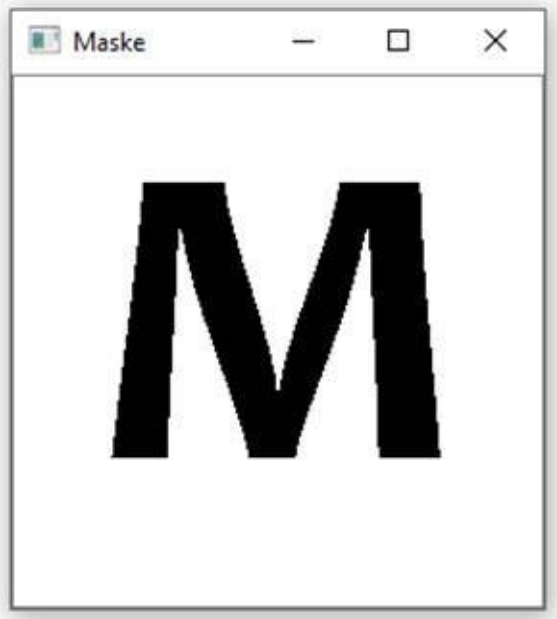


Fig. 3.3: Mask after thresholding

In order to understand the next step here, we first need to understand what we are actually trying to accomplish here. We want to get the blue background unchanged and we want to completely remove the white area. For this we can use the bitwise AND. This is a logical operation, which returns True, when both operands are True. In our context the color black means False and the color

white means True. When we perform a logical AND with the white color, we get as a result the other operand, since white accepts everything. So when we apply a logical AND operation to our background and the white color, the result will be the background itself.

With the black color it is the exact opposite. Since this number represents False, the result will be adapted to zero percent and nothing changes.

If you have understood this basic principle, you should recognize what is wrong with our mask. It is the exact opposite of what it should be. We need a white M so that it can become transparent and we need a black background so that it doesn't change. Therefore, we will invert the mask.

```
mask_inv = cv.bitwise_not(mask)
mask_inv = np.invert(mask) # Alternative way
```

Here we can either use the *bitwise_not* function of OpenCV or the *invert* function of NumPy. Just don't use both because then you invert it twice and end up with the original mask.

Now let's get to the actual inserting of the logo. First we need to determine how big this logo is and select the proper region for our AND operation.

```
rows, columns, channels = img2.shape
area = img1[0:rows, 0:columns]
```

We use the *shape* attribute of our logo in order to get the resolution and the channels. Then we save the respective area of our background into a variable.

```
img1_bg = cv.bitwise_and(area, area, mask=mask_inv)
img2_fg = cv.bitwise_and(img2, img2, mask=mask)
```

Now we apply the bitwise operations. We define two parts of the upper-left corner, which we then combine in the end. First we define the background of the initial picture, by applying the inverted mask to the selected area. This makes the M transparent. The second line of code then adds the mask with the blue background.

```
result = cv.add(img1_bg, img2_fg)
img1[0:rows, 0:columns] = result
```

Last but not least, we use the *add* function to combine both layers. Then we assign the result to the upper-left corner of the final image.

```
cv.imshow('Result', img1)
```

Fig. 3.4: Result of the thresholding

The image is now exactly the way we wanted it to be.

MAKING POORLY LIT IMAGES READABLE

Let us now get to an example that is a little bit more impressive than just adding a logo.

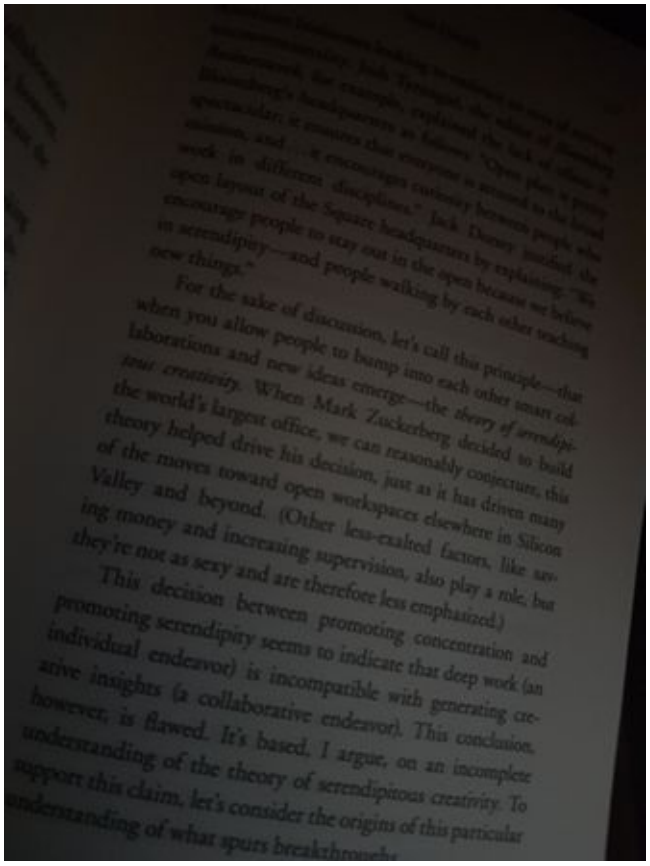


Fig. 3.5: Poorly lit book page

Can you clearly read what is written on that book page? It’s not that it’s impossible but it is pretty hard. The lighting conditions are pretty bad. With thresholding however, we can fix that problem. We can make that text easily readable.

One first idea would be to just convert the image to grayscale and apply the binary thresholding.

```
img = cv.imread('bookpage.jpg')
img_gray = cv.cvtColor(img, cv.COLOR_RGB2GRAY)
ret, threshold = cv.threshold(img_gray, 32, 255, cv.THRESH_BINARY)
```

Every pixel that is whiter than 32 (dark gray) is now being converted to 255

(completely white) and every value below is converted to 0 (completely black).

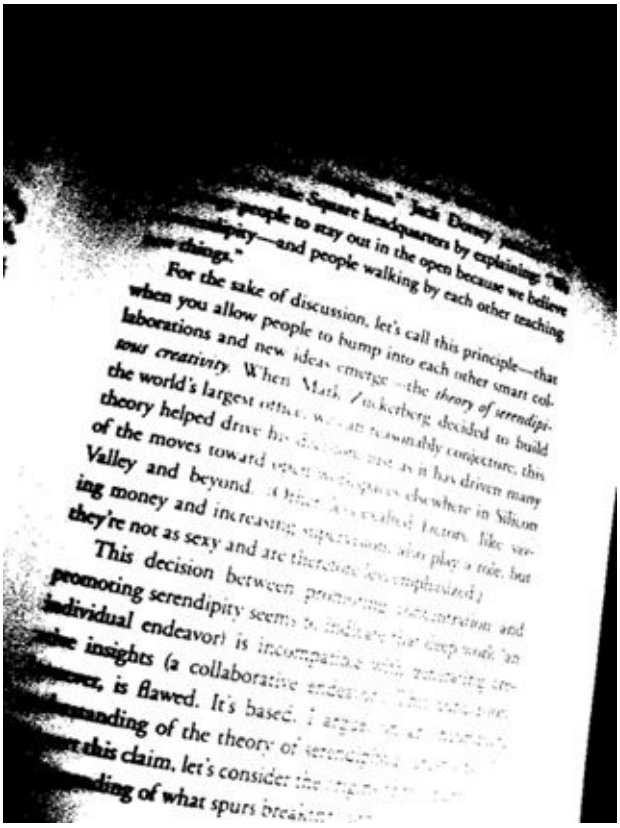


Fig. 3.6: Result after binary thresholding

As you can see the result isn't that good. Parts of the image that we want to be white are black and vice versa. We obviously need a more dynamic way of thresholding.

Here the *adaptive Gaussian thresholding* comes into play. This algorithm allows us the use the binary thresholding more dynamically.

```
gaus = cv.adaptiveThreshold(img_gray, 255,
                            cv.ADAPTIVE_THRESH_GAUSSIAN_C,
                            cv.THRESH_BINARY, 81, 4)
```

We first pass our gray picture, followed by the maximum value (255 for white). Then we choose the adaptive algorithm which is the Gaussian one in this case (*ADAPTIVE_THRESH_GAUSSIAN_C)*. After that we choose the thresholding that we want to use, which is still the binary thresholding. Now the last two parameters are essential. The first one is the block size and specifies how large (in pixels) the blocks used for thresholding shall be. The larger this value is, the more will be taken into the calculations. Smaller details might not be valued as

that important then. This value needs to be odd and for this case 81 is a good choice. The last parameter is called *C* and it specifies how much shall be subtracted from the median value. With this parameter we oftentimes need to experiment a little bit. It sharpens and smooths the image.

```
cv.imshow('Gaus', gaus)
```

Now let us look at our final result.

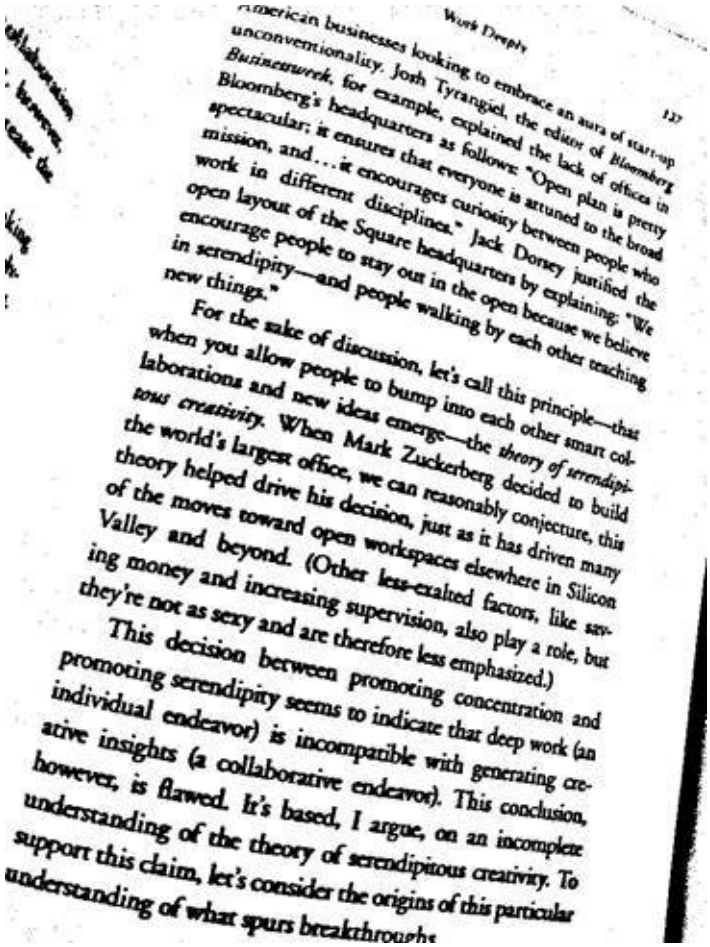


Fig. 3.6: Results after adaptive thresholding

This result is actually quite impressive. It is not perfect but we can easily read the whole text without any problems.

By the way the book page that you see here is from the book *Deep Work* from *Cal Newport*.

As you can see many things are possible with thresholding. Before you continue with the next chapter, try to apply what you have just learned onto your own

images. Maybe you could make some poorly lit images or come up with new creative ideas for the application of thresholding. Experiment a little bit and play around with these technologies.

4 – FILTERING

In this chapter we are going to talk about *filtering*. However we are not talking about the filters that people use on social media to make their pictures prettier. We are talking about actually filtering specific information out of pictures.

For example we might want to extract all the red objects from a video or an image. Or we might be interested in those parts that are brighter than a specific limit.

Fig. 4.1: Image of a parrot

Let’s use this simple image as our example. Here we have a red parrot in front of a blue-green background. With filtering algorithms we can now try to extract and highlight the feathers.

This might be useful for a number of reasons. You might want to recognize specific object or patterns. This particular example is trivial but it will illustrate the concept.

CREATING A FILTER MASK

The first step is to now load our image into the script. At the end of this chapter, we are also going to talk about filtering videos and camera data.

```
img = cv.imread('parrot.jpg')
hsv = cv.cvtColor(img, cv.COLOR_RGB2HSV)
```

What's important here is that we won't work with the RGB color scheme. We are going to convert it into HSV. RGB stands for *Red*, *Green*, *Blue*, whereas HSV stands for *Hue*, *Saturation*, *Value*.

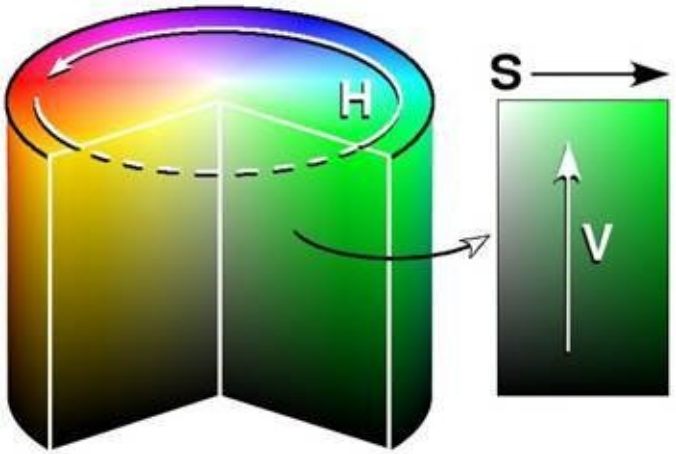


Fig. 4.2: HSV color scheme (Source: Wikipedia)

The H-value determines the color we are choosing from the spectrum. The S-value indicates how high our saturation is or how "strong" the colors are. And the V-value states how bright the chosen color shall be.

We are choosing this color scheme because it is much easier to filter images, using these three parameters.

```
minimum = np.array([100, 60, 0])
maximum = np.array([255, 255, 255])
```

This takes us to the next step. Now we are going to define two limits or boundaries – the minimum and the maximum. The color value than has to be in between of those, in order to be extracted. In our case, the color value has to be in between 100 and 255. This gives us all values that represent orange or red colors. Also we demand a saturation of at least 60, in order to not get any gray values. For this example we will ignore the brightness and therefore allow for

the full spectrum from 0 to 255.

```
mask = cv.inRange(hsv, minimum, maximum)
result = cv.bitwise_and(img, img, mask = mask)
```

Next we are going to define a mask by using the *inRange* function. This function sets all pixels that match our requirements to white pixels and all the others to black pixels. Then we use the logical *bitwise_and* function, in order to AND the original image with itself and apply the mask.

```
cv.imshow('Mask', mask)
```

Let’s take a quick look at the mask we created.

Fig. 4.3: Resulting filter mask

The white pixels are the ones that we are extracting for our result. For this, we will look at the actual result.

```
cv.imshow('Result', result)
```

Fig. 4.4: Resulting image

Actually, that is a quite a nice result already. But we can still see individual pixels that shouldn’t be there. Especially when you are filtering camera data in real-time you will notice a lot of background noise.

BLURRING AND SMOOTHING

To now optimize the result we will use *blurring* and *smoothing*. We are going to make our result less sharp but also reduce the background noise and the unwanted pixels. The first step for this is to create an array for the averages.

```
averages = np.ones((15, 15), np.float32) / 225
```

Here we create an array full of ones that has the shape 15x15. We then divide this array by 225 (the product of 15x15). Then we end up with a factor for each pixel that calculates the average.

The goal here is to correct individual unwanted pixels by looking at the average pixels in 15x15 pixel fields.

```
smoothed = cv.filter2D(result, -1, averages)
```

With the *filter2D* function we apply this averages kernel onto our image. The second parameter, which is set to -1, specifies the depth of the image. Since we choose a negative value, we just copy the depth of the original image.

```
cv.imshow('Smoothed', smoothed)
```

Fig. 4.5: Resulting smoothed image

As you can see, most of the unwanted pixels are gone. But the picture is now pretty blurry.

Notice that the order of these steps is very relevant. Here we first applied the mask onto our image and then smoothed the result. We could also do it the other

way around and first make the mask smoother. Then we get a different result.

```
smoothed2 = cv.filter2D(mask, -1, averages)
smoothed2 = cv.bitwise_and(img, img, mask=smoothed2)

cv.imshow('Smoothed2', smoothed2)
```

Fig. 4.6: Resulting image after smoothing the mask

In this particular example, the second order is useless. We have many more unwanted pixels than before the smoothing. However, this will be useful with other algorithms.

GAUSSIAN BLUR

Another method which we could apply here is the Gaussian blur.

```
blur = cv.GaussianBlur(result, (15, 15), 0)
```

Here we also pass the size of the blocks to blur. The result is a little bit less blurry than the previous one.

MEDIAN BLUR

The probably most effective blur is the *median blur*. This one processes each channel of an image individually and applies the median filter.

```
median = cv.medianBlur(result, 15)

cv.imshow('Median', median)
```

Here we only pass the image and the block size which is quadratic. In this case we again have 15x15 pixels.

Fig. 4.7: Resulting image after median blur

This result is definitely impressive, since there is no background noise left. However, we can do even better. As you can see the image is still pretty blurry and this can be changed by changing the order of the methods.

```
median2 = cv.medianBlur(mask, 15)
median2 = cv.bitwise_and(img, img, mask=median2)

cv.imshow('Median2', median2)
```

Fig. 4.8: Result after changing the order

This is by far the best result. We don’t have any unwanted pixels and the picture is not blurry at all. For this particular example, the median blur turns out to be the best choice.

FILTERING CAMERA DATA

As I already mentioned, what we just did with the parrot image we can also do with the camera data in real time.

```
import cv2 as cv
import numpy as np

camera = cv.VideoCapture(0)

while True:
    _, img = camera.read()
    hsv = cv.cvtColor(img, cv.COLOR_RGB2HSV)

    minimum = np.array([100, 60, 0])
    maximum = np.array([255, 255, 255])

    mask = cv.inRange(hsv, minimum, maximum)

    median = cv.medianBlur(mask, 15)
    median = cv.bitwise_and(img, img, mask=median)

    cv.imshow('Median', median)

    if cv.waitKey(5) == ord('x'):
        break

cv.destroyAllWindows()
camera.release()
```

Again we just create a capturing object and read the frames in an endless loop. Then, in every iteration, we applied the filter on the frame before actually showing it. If you try this at home with your own camera, you will notice that everything that is not red or orange will be invisible.

As always, I encourage you to play around with these filters yourself. Change the HSV settings. Use different images. Tweak all the parameters. Learn how the filters work and which are the most effective for different scenarios.

5 – OBJECT RECOGNITION

Now we get into one of the most interesting subfields of computer vision, which is *object recognition*. In the beginning of this chapter we are going to cover topics like *edge detection*, *template and feature matching* and *background subtraction.* At the end we will then use cascading to recognize actual objects in images and videos.

EDGE DETECTION

Let’s start talking about edge detection. Oftentimes it is quite useful to reduce our images and videos to the most essential information. In this book we are not going to write our own object recognition algorithms from scratch. But if we wanted to do it, detecting edges would be a very useful tool.

Fig. 5.1: Image of a room

For this example we will use this image of an ordinary room. A computer is not really interested in details like shadows and lighting. Therefore we are going to use an algorithm, in order to filter out the essential edges.

```
import cv2 as cv

img = cv.imread('room.jpg')
edges = cv.Canny(img, 100, 100)
cv.imshow('Edges', edges)

cv.waitKey(0)
cv.destroyAllWindows()
```

The *Canny* function of OpenCV does this for us. We pass the original image followed by two tolerance values. The result looks quite interesting.

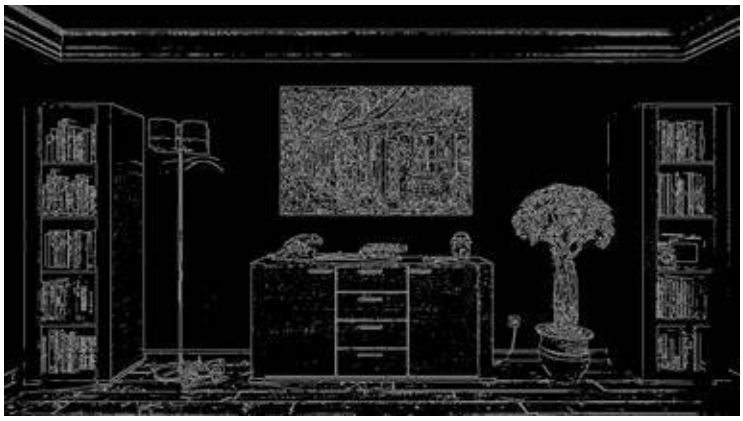


Fig. 5.2: Image after edge detection

As you can see, all the essential information is still there but the image was reduced to its edges.

TEMPLATE MATCHING

Another interesting technique is *template matching*. Here we give our script some templates for specific objects, which it then has to find in the images. The difference to object recognition however is that the matching has to be pretty accurate. It is looking for almost exact matches rather than similar patterns. Thus, this technique is not optimal for the general recognition of faces or clocks for example.

Fig. 5.3: Image of a workspace

Let’s take this image of a workspace as our example. Basically, we could choose any object we want, but the only one that occurs multiple times in this image are they keys of the laptop keyboard.

Fig 5.4: F-key as template

So what we do is we crop out one of the keys using a program like Gimp or Photoshop. Then we use this key as our template image.

```
img_bgr = cv.imread('workspace.jpg')
img_gray = cv.cvtColor(img_bgr, cv.COLOR_BGR2GRAY)
```

```
template = cv.imread('key.jpg', 0)
width, height = template.shape[::-1]
```

First we load the main picture and the template into our program. Then we make a copy of the image and convert it into grayscale. Also we save the width and the height of the template in pixels.

```
result = cv.matchTemplate(img_gray, template,
                          cv.TM_CCOEFF_NORMED)
```

The main work will now be done by the *matchTemplate* function. To it we pass our image, our template and we specify the method that we are going to use for matching. In this case *TM_CCOEFF_NORMED*.

What we get as a result is an array with the respective activations of the image parts, in which this template occurs.

```
threshold = 0.8
area = np.where(result >= threshold)
```

In the next step we define a certain threshold. It specifies how far the pixels of the grayscale image are allowed to deviate from the template. In this case 0.8 means that the area needs to have at least 80% similarity with our template to be recognized as a match. You can tweak this value as you like and experiment around. The function *where* from NumPy returns the indices of the pixels, which are close enough.

```
for pixel in zip(*area[::-1]):
    cv.rectangle(img_bgr, pixel,
                 (pixel[0] + width, pixel[1] + height),
                 (0, 0, 255), 2)
```

Now we run a for loop over the zipped version of our area. For each pixel in this area, which has a high enough activation, we will then draw a red rectangle of the width and height of the template. This will then indicate that our algorithm has found a match there.

```
cv.imshow('Result', img_bgr)
cv.waitKey(0)
cv.destroyAllWindows()
```

Last but not least, we can look at our result with all the matches.

Fig. 5.5: Workspace after feature matching

Again, if you are reading a grayscale version of this book, you will probably only notice the gray rectangles around some of the keys. When you execute the script yourself, you will see that they are red.

As you can see, our algorithm recognized quite a lot of keys. If you want to find more keys, you will have to reduce the threshold. In this case however, you also increase the likelihood of misclassifications.

FEATURE MATCHING

Imagine having two pictures that show the exact same objects but from a different perspective.

Fig. 5.6: Workspace from another perspective

Here for example we have the workspace from a different perspective. For us humans it is not hard to recognize that these objects are the same and match them in both pictures. But for our computer these are completely different pixels.

What's going to help us here is the so-called *feature matching*. Here, algorithms extract all the essential points and descriptors of our images. Then it looks for the same points in the other image and connects them.

```
img1 = cv.imread('workspace1.jpg', 0)
img2 = cv.imread('workspace2.jpg', 0)

orb = cv.ORB_create()

keypoints1, descriptors1 = orb.detectAndCompute(img1, None)
keypoints2, descriptors2 = orb.detectAndCompute(img2, None)
```

After loading our images, we create an *orientational BRIEF* (short ORB). This will help us to determine the essential points. We call the *detectAndCompute* function of this object and apply it to both our images. The results we get are the key points and the descriptors for both images.

In this example, we import the images in black and white (thus the zero). We do this because we can then better see the colored connections in the end. But you

can also load the original images if you want.

```
matcher = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
matches = matcher.match(descriptors1, descriptors2)
matches = sorted(matches, key = lambda x: x.distance)
```

Now we create an instance of the *BFMatcher* class and choose the *NORM_HAMMING* algorithm. This matcher allows us to combine the key points and determine where the key points of one picture can be found in the other one. The result will then be sorted by distance in the last line, so that we have the shortest distances first. This is important because in the next step we are only going to filter out a couple of the best results.

```
result = cv.drawMatches(img1,keypoints1,
                        img2,keypoints2,
                        matches[:10], None, flags=2)
```

Last but not least, we visualize these matches with the *drawMatches* function. For this we pass both images, their key points and specify which matches shall be visualized. In this case we pick the first ten matches, which have the shortest distance.

```
result = cv.resize(result, (1600,900))
cv.imshow('Result',result)
cv.waitKey(0)
```

Now we scale the final result so that we can show the images without problems.

Fig. 5.7: Feature matching of both images

It is very hard to actually see the lines in this book. Even if you have a colored version, you will struggle because OpenCV draws very thin lines here. Therefore, go ahead and execute the code on your own machine to see the

results.

Basically what is happening is that the most important points in the left image are being connected to the most important points in the right image. This works pretty well most of the time but we still have some mistakes in our result. It’s not perfect!

MOVEMENT DETECTION

Before we finally get to object recognition we will take a look at a very interesting technique when it comes to movement detection. This technique is called *background subtraction*. Here we use an algorithm that looks at the changes in pixels, in order to determine what the background is. Then we only focus on the foreground or at those pixels that are changing or moving.

Fig. 5.8: Screenshot of a video

For this I will use a video in which many people are walking at a public place. Of course you can use your own videos or even the camera data as input.

```
# Alternative: video = cv.VideoCapture(0)
video = cv.VideoCapture('people.mp4')
subtractor = cv.createBackgroundSubtractorMOG2(20, 50)
```

As always we load our video into our script. Then we create a new subtractor by using the function *createBackgroundSubtractorMOG2*. To it we (optionally) pass two parameters. The first one is the length of the *history*, which specifies how far back our subtractor shall look for movements or changes. The second one is the *threshold*. Here again you have to play around with those values until you get optimal results.

```
while True:
    _, frame = video.read()
    mask = subtractor.apply(frame)

    cv.imshow('Mask', mask)

    if cv.waitKey(5) == ord('x'):
        break
```

```
cv.destroyAllWindows()
video.release()
```

Now we can once again run our endless loop and show our results. The only thing that we do here is to *apply* our subtractor to the current frame. The result is our mask.

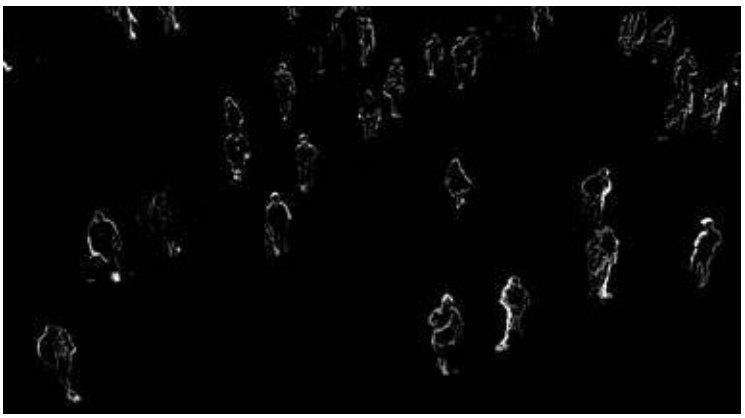


Fig. 5.9: Result as mask

Even though the result is not perfect and we can probably choose better parameters, the subtractor fulfills its purpose. We can only see those parts of the video that are changing. This is the foreground. Try it yourself. It is fun!

OBJECT RECOGNITION

Now we finally get to *object recognition*. Here we try to work as generalized as possible. This is the opposite strategy of template matching. Our goal is not to find the same type of key or the same face but to generally recognize faces, computers, keys etc. For this however, we need some advanced models and training data. Since it takes quite a lot of effort to build those on our own, we will use resources from the internet.

LOADING RESSOURCES

We will make use of so-called *HaarCascades*, which have the format of XML files. In this book we are going to use one of these cascades for recognizing faces and one of them to recognize clocks. For this, we will use the following links:

Face Cascade: https://bit.ly/3bkHNHs

Clock Cascade: https://bit.ly/3bdLQF8

If for some reason the links don’t work in the future, you can just look up some *HaarCascades* for the objects that you want to recognize.

However, take care of the licensing of these files. The first file is from the company Intel and has a copyright. As long as you use it for your private learning, this should not be a problem though.

```
faces_cascade = cv.CascadeClassifier('haarcascade_frontalface_default.xml')
clock_cascade = cv.CascadeClassifier('clock.xml')
```

We now create two *CascadeClassifiers* for both objects and pass both XML-files as parameters.

RECOGNIZING OBJECTS

Fig. 5.10: Group of people

First we are going to use this picture of people in order to recognize some faces.

```
img = cv.imread('people.jpg')
img = cv.resize(img, (1400, 900))

gray = cv.cvtColor(img, cv.COLOR_RGB2GRAY)
faces = faces_cascade.detectMultiScale(gray, 1.3, 5)
```

We scale the image and convert it into grayscale. Then we use the *detectMultiScale* function of our classifier, in order to recognize faces with the help of our XML-file. Here we pass two optional parameters. First the scaling factor of the image which is going to be higher the better our image quality is. And second the minimum amount of neighbor classification for a match. That's actually it. We now just have to visualize everything.

```
for (x,y,w,h) in faces:
   cv.rectangle(img, (x, y),
                (x + w, y + h),
                (255, 0, 0), 2)
   cv.putText(img, 'FACE',
              (x,y+h+30),
              cv.FONT_HERSHEY_SIMPLEX, 0.8,
              (255,255,255), 2)
```

We iterate over each recognized face and get the two coordinates, the width and the height. Then we draw a rectangle and put a text below it.

Fig. 5.11: Classified faces

As you can see, the result is pretty amazing. We will now do the same thing with clocks.

Fig. 5.12: Image of a room with a clock

In this room you can see a clock, which we want to recognize with our script. We repeat the procedure and add some parts to our script.

In diesem Zimmer befindet sich eine Wanduhr, welche wir von unserem Skript erkennen lassen möchten. Wir wiederholen also die Vorgehensweise und fügen Teile zu unserem Skript hinzu.

```
faces = faces_cascade.detectMultiScale(grau, 1.3, 5)
clock = clock_cascade.detectMultiScale(grau, 1.3, 10)
```

For the clock we use ten as the third parameter, because otherwise it makes some

misclassifications.

```
for (x,y,w,h) in clocks:
    cv.rectangle(img, (x, y),
                 (x + w, y + h),
                 (0, 0, 255), 2)
    cv.putText(img, 'CLOCK',
               (x, y + h + 30),
               cv.FONT_HERSHEY_SIMPLEX, 0.8,
               (255, 255, 255), 2)
```

Here we also draw a rectangle and put a text below the clocks. Of course we need to also change the file path of the image that we are loading in the beginning.

Fig. 5.12: Classified clock

As you can see this also works pretty well. Unfortunately I was not able to find a license-free picture with people and clocks on the wall at the same time. However, in such a case our script would recognize both and draw rectangle with different colors and different texts on the right places. This also works with videos and with camera data.

As always experiment around with the concepts you have learned about in this chapter. Use different images, try different cascades and work with your camera. Be creative and try new things because that is how you learn.

WHAT’S NEXT?

If you have understood the concepts in this book and you learned how to apply them, you have made a huge step towards becoming a master programmer and computer scientist. The skills you learned are invaluable in today’s economy but also in the future.

You are able to process image and video data in a very complex way and extract important information. In combination with machine learning and data science, this is a very powerful tool.

Depending on the application field, you will need to learn some extra skills, since no book in the whole world could teach you everything. If you go into natural sciences, you will need to learn the respective skills there. It’s the same for medicine, for sports and for every other field. Computer science and mathematics alone are not going to get you very far, unless you go into theoretical research. However, you should now have a solid basis in programming and computer science, so that you continue to progress further on your journey. If you choose to build surveillance systems, analyze medical data or do something completely different, is up to you.

If you are interested in more machine learning, take a look at my Amazon author page. There you can find the other volumes of that series that are about machine learning, data science and neural networks.

NEURALNINE

One place where you can get a ton of additional free resources is *NeuralNine*. This is my brand and it has not only books but also a website, a YouTube channel, a blog, an Instagram page and more. On YouTube you can find high quality video tutorials for free. If you prefer text, you might check out my blog for free information. The *@neuralnine* Instagram page is more about infographics, updates and memes. Feel free to check these out!

YouTube: https://bit.ly/3a5KD2i

Website: https://www.neuralnine.com/

Instagram: https://www.instagram.com/neuralnine/

Books: https://www.neuralnine.com/books/

Can’t wait to see you there! J

Last but not least, a little reminder. This book was written for you, so that you can get as much value as possible and learn to code effectively. If you find this book valuable or you think you learned something new, please write a quick review on Amazon. It is completely free and takes about one minute. But it helps me produce more high quality books, which you can benefit from.

Thank you!

If you are interested in free educational content about programming and machine learning, check out https://www.neuralnine.com/

There we have free blog posts, videos and more for you! Also, you can follow the ***@neuralnine*** Instagram account for daily infographics and memes about programming and AI!

Website: https://www.neuralnine.com/

Instagram: @neuralnine

YouTube: NeuralNine

Books: https://www.neuralnine.com/books/

If this book benefited your programming life and you think that you have learned something valuable, please consider a quick review on Amazon.

Thank you!

If you are interested in free educational content about programming and machine learning, check out https://www.neuralnine.com/

There we have free blog posts, videos and more for you! Also, you can follow the ***@neuralnine*** Instagram account for daily infographics and memes about programming and AI!

Website: https://www.neuralnine.com/

Instagram: @neuralnine

YouTube: NeuralNine

Books: https://www.neuralnine.com/books/