

# Python Summer School

🕒 Created	@May 4, 2021 7:59 AM
📋 Property	
🏷️ Tags	
🕒 Updated	@Jun 1, 2021 7:44 AM

## Basics of Python programing through Hydrology

**You wanted to learn coddng, but had no idea where and how to start? Then this guide is exactly for you!**

This script will help you getting started with Python programing with interesting examples from Hydrology and Meteorology. It is meant to be both a simple programming tutorial and a motivational letter with the goal of improving your skills. My idea is that it's easier to understand some concept if you can visualize it, and what is a better example than the weather or some daily phenomena we encounter each day. 😊

## Chapter 1. - How to start programming with Python?

What is Python? According to its creator, Guido van Rossum:

"high-level programming language, and its core design philosophy is all about code readability and a syntax which allows programmers to express concepts in a few lines of code."

First we will take a look how easily to setup an environment for Python programming, and then some basic concepts where and how to start. First, what is an **environment**?

According to a [Quora question](#), "*The environment is quite literally everything installed on your machine which can affect either the development and or testing of your application*", it includes:

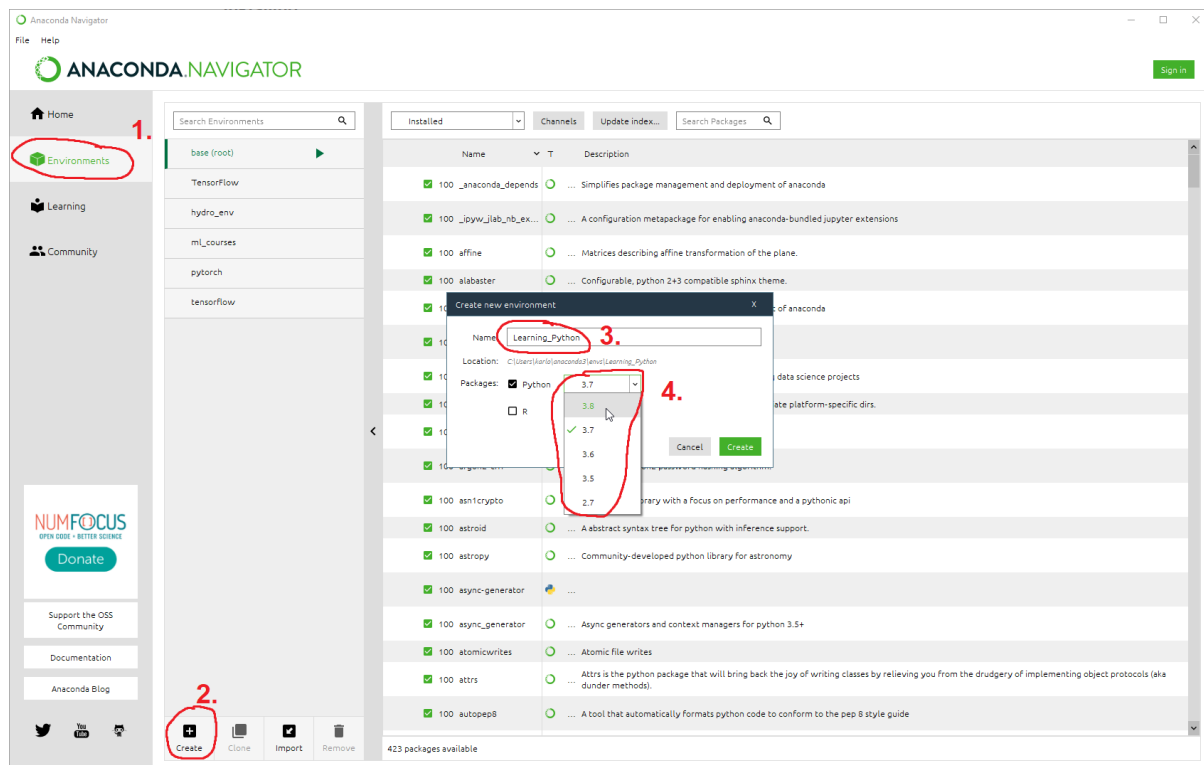
- editors/IDEs (software where u can write and/or run the code)
- compilers/interpreters
- operating system installed on your machine
- environment variables set on your machine
- extra libraries installed on your machine etc.

In an ideal case, the environment should be as simple as possible with only the needed libraries installed, and only the versions of the compiler/interpreter that you are using.

So, how we setup an environment? Well, in my opinion, it's best to download the [Anaconda](#) (or [Miniconda](#) if you have limited space or and older PC). Simply by following the guide on the website you get everything (and even more 😊) than you need to start programming. Basically, after downloading Anaconda we install it as any other application. After the installation is complete, we get the Anaconda Navigator app. It's an easy to use interface where we can manage our programming environments, install libraries, start an IDE etc.

## **Creating the environment**

Before we start with programming, we will create an environment. We can do it two ways: through the Navigator or through an Anaconda prompt.



The first way is through the *Navigator*. When we start the app in the left column we select the "Environments" tab. As I have already have some environments, my list is not empty. On a fresh install you only have the base environment. Second, you select "create", we type in a name we like, i.e. "Learning\_Python" and we select the Python version we want, here we have selected Python 3.8.

The other way is through the *Anaconda prompt*.

First we run an Anaconda prompt, we can do it by searching for it in Start menu. After prompted we need to enter following command:

```
conda create -n Python_Learning python=3.8
```

Conda create is the command to create a new environment, after -n we specify the name, and we can choose the python version, if desired, here I've chosen version 3.8. When prompted, we type **y** to proceed. Then we activate the new environment by typing:

```
conda activate Python_Learning
```

We have now successfully activated out new python environment.

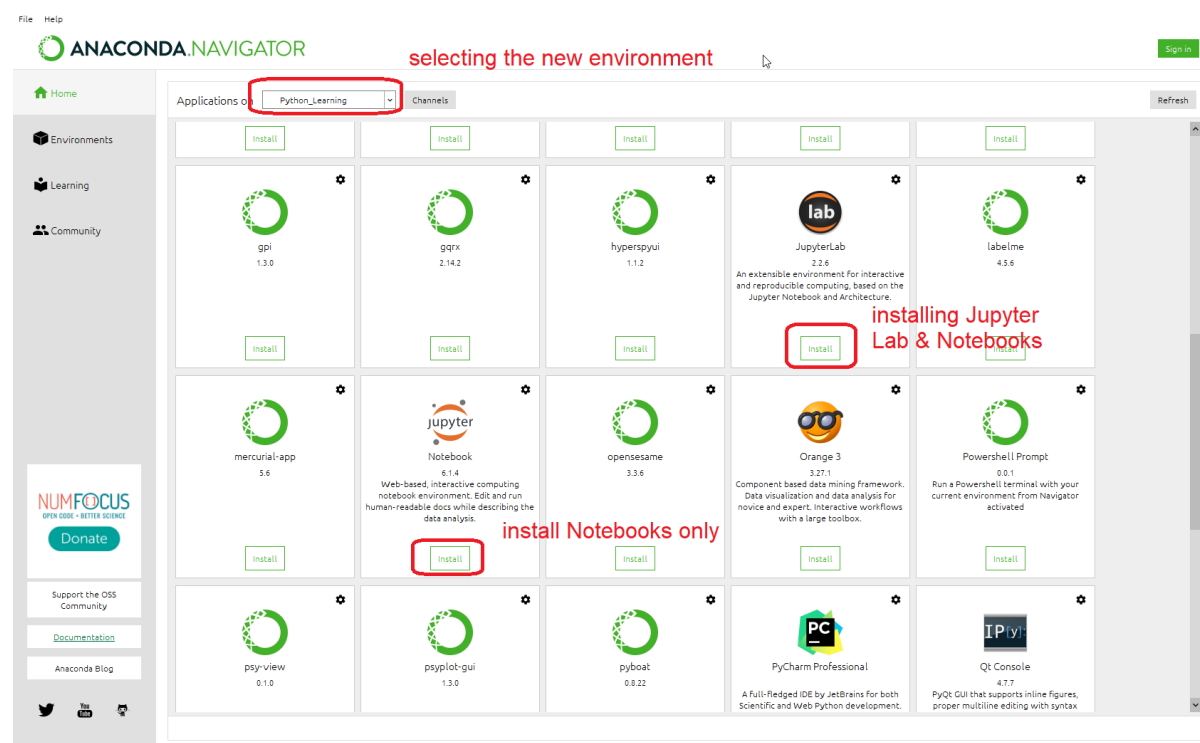
My opinion is that you shouldn't care too much about the editor (or IDE) you are going to use, since at this point, it's pretty much irrelevant. Everybody encountered this situation, when you need to start with something new, I made the mistake of too much thinking and researching what IDE to use, instead of just start typing code. Believe me, when you get to the level that the IDE matters to you, you will figure out by yourself which one to use, you certainly don't need me to tell you. 😊

For start let's stick with Jupyter Notebooks.

## Jupyter Notebook

To learn the basic stuff I would suggest you to start with Jupyter Notebooks. It's basically a web application that allows writing and running code from so called code cells. First we will run Jupyter Notebooks. To do so we have two ways:

### Navigator



First we select the new *environment*, then we install Jupyter, either Lab and Notebooks, or just Notebooks. You can safely hit **Notebook**. (the lower left option) After the installation is done, we just hit **Launch** under *Jupyter Notebook*. A new tab opens up in our browser. To start a new notebook, we just select "New" in the upper right corner, and "Python 3". This opens a new notebook in the next browser tab.

### Prompt

To install the Notebook through the prompt, after creating the environment and activating it, we type the following command:

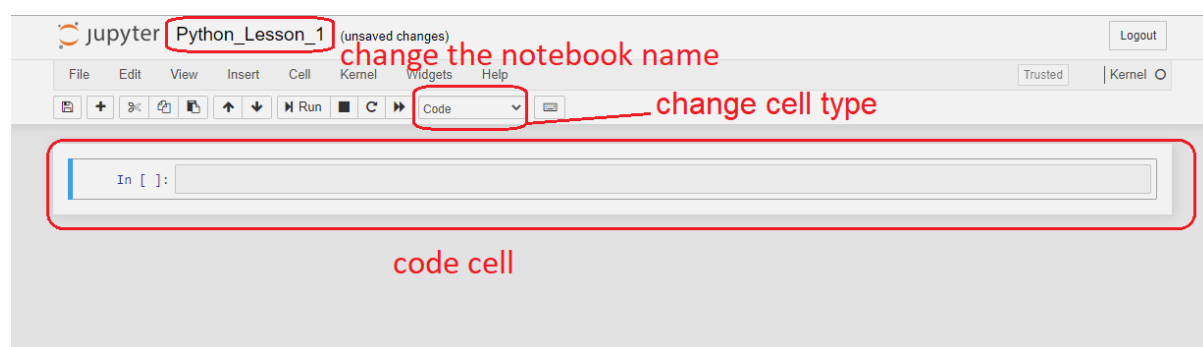
```
conda install -c conda-forge notebook
```

We hit enter, the **Y** to accept and it's done. 😊

We type:

```
jupyter notebook
```

A new tab opens up in our browser. To start a new notebook, we just select "New" in the upper right corner, and "Python 3". This opens a new notebook in the next browser tab.



Now we can start to learn coding with Python 😊

The first command we will learn is called print. As the name suggests it's used to print out, either a sentence or an variable for instance. Let's try it out! Type the following in the first cell:

```
print("Today is a cold day!")
```

We run the cell by hitting: **Shift+Enter**, or by clicking either the **"Play"** button the toolbar, or Cell, Run in the menu bar. The important thing here is that if we want to print out a sentence (string) we need to use quotation marks. They mark the start and the end of the string.

Last, but not least, it's worth mentioning, that in Jupyter Notebook, you can change the type of each cell. It's great for using explanations, or divide your code into chunks. To do so you select **markdown** option, and the cell turns in plain text editor, like notepad++ or MS Word. We can type an explanation or subtitle of this part of the notebook, and hit **Shift+Enter**.

```
In [34]: print("Today is a cold day!")
Today is a cold day!
Today we will learn something about meteorology!
```

## Chapter 2. - Data Types

### Strings

Before we continue we have to learn what types of data we can encounter in Python.

Above I've already mentioned *strings*, it is basically a sequence of characters. We can specify a string by using single ' or double " quotes. In cases of long strings, over two or more lines we use triple quotes """.

Let's see some examples.

```
print('Rain is falling.')
print("It's cold outside.")
print("""If it's very cold outside, rain can freeze
and turn into sleet or snow.""")
```

In the second example we can see why we use double quotes. If we had used a single quote, the ' in "it's" had finished our string and we had an error.

```
In [5]: print('It's cold outside')
File "<ipython-input-5-61e54fb079c0>", line 1
    print('It's cold outside')
          ^
SyntaxError: invalid syntax
```

This is our first error, many more will come. 😊

### Numbers

There are three types of numbers in Python: integers, floating point and complex numbers.

In Python there is no need to define the variable type a priori, and it is allowed to even change the data type later in the program, wherever needed.

Here we show how to print multiple variables, and the `type` function, which is used to return the type of a variable.

```
In [1]: a = 6
        b = 3.5

        print(type(a), type(b))

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

**INT**egers (whole numbers) are used for indexing (more on that later) the arrays (vector, matrix), for counting etc. **FLOAT**s (floating point) are decimal numbers.

## Variables

So what is a variable? Well, it's a container for storing data values (integers, strings, etc). It's created the moment you first assign a value to it. Similar like in basic math, when we say that: ***X equals 10***

```
x = 10
weather = "sunny"
```

We declared that the variable **x** equals to 10. In Python we also can declare that a variable equals a word (string). Above we declared that the variable **weather** contains the string **sunny**.

When defining variables, some "unwritten" rules have to be followed. It's a good practice to select a meaningful name, and to document for what this variable is being used for.

The name can be of arbitrary length, should start with a letter, and can contain numbers as well. If we use a variable name with multiple words, we use **underscores \_**. You can see an example under **Lists**.

Also, some of the keywords are reserved by Python, i.e., **class**, **def**, **dict**, or some predefined function like **print** or **sum** etc, using those should and have to be avoided.

## Chapter 3. - Data structures

Data structures are objects that can hold more than one data entries in it. Some examples in Python are: **lists**, **tuples**, **dictionaries** and **sets**.

### Lists

**Lists** are used for saving larger collections of data. List items are ordered, mutable (changeable), and allow duplicate values. Like i.e. a list of strings, week days or a list of floats (snow depth in centimeters), measurements of daily snow depth during a week, so that means seven entries.

```
days_of_week = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
daily_snow_depth = [4.5, 4.5, 4.6, 4.7, 4.9, 5.4, 6.2, 6.9]

print (daily_snow_depth)
print (days_of_week)
```

To declare a list in Python we use square brackets `[]`, and divide the entries with commas. Lists are also dynamic, which means we can add items later on if we like.

To access an item in the list we use indexing. The indices in Python start with 0. So to access let's say the 3rd item in an list we use the number 2 as index. The index is placed inside **square brackets**:

```
In [10]: print (days_of_week[2])
         Wednesday
```

We can also access the indexes backwards, from last to the first one. In that case, we start from -1, to access the last item, -2 second last and so on.

To add an item to a list ( ) we use the **.append()** method, as follows:

```
daily_snow_depth.append(7.5)
```

As the method suggests, it adds the new item to the last place in the list.

However, if we want to add an item to a specific place, we need to use the **.insert()** method. First we specify the *index*, then the *value* we want to add:

```
In [30]: daily_snow_height.insert(4, 50)
         print (daily_snow_height)
         [4.5, 4.5, 4.6, 4.7, 50, 4.9, 5.4, 6.2]
```

However, when inserting backwards, the indices start with -1, and it means insertin the new value to second last place. If we wan to to add to the last place, we use



the above shown **.append()** method.

To replace an item in the list we specify the desired index, and declare the new value as follows:

```
In [33]: daily_snow_height[4] = 14.0
         print (daily_snow_height)
         [4.5, 4.5, 4.6, 4.7, 14.0, 4.9, 5.4, 6.2]
```

Replacing works also backwards, and here the **-1** index means the **last place**, as it was when accessing items in the list.

## Dictionaries

Dictionaries, are similar to lists, they have indices, but they can be of any type. Tuples are mutable, but unordered and do not allow duplicates. Meaning, they can be changed later, data can be added or removed. Here's an example:

```
snow_depths = {"Monday":4.5, "Tuesday":4.5, "Wednesday":4.6,
               "Thursday":4.7, "Friday":4.9, "Saturday":5.4, "Sunday":6.2}

print(snow_depths["Wednesday"])
```

Often the keys are used as names, for instance, a dictionary could be used to store snow depth on a certain day. Here, I've created a *dictionary* called **snow\_depths**, it is used to store the measured snow depth on each day for one week.

To declare a dictionary we use curly brackets **{}**. First we enter the **key**, followed by a **semicolon :** and then its **value**.

Example: **"Monday": 4.5**

Here, the name of the day, Monday, is a *string*, so we need to use **quotation marks** **""**.

To print out the snow depth on Wednesday, instead of the index like for lists, we enter the *key* (Wednesday) and the function prints the *value*.

## Tuples

A **tuple** is a **immutable** (unchangeable) and **ordered** sequence of values. This means that when created (defined), we cannot change the values, or add new ones to the tuple. A common case for using tuples are geographic coordinates of a meteorological measuring station.

```
meteo_loc = (46.28277778, 16.36388889)

print (meteo_loc)
```

Here we defined a tuple that contain geographical latitude and longitude of a gauging station.

As mentioned, adding new items to the tuple is not possible.

```
In [23]: meteo_loc[2] = 45

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-23-383ff5c53c6a> in <module>
----> 1 meteo_loc[2] = 45

TypeError: 'tuple' object does not support item assignment
```

Similar results are shown when trying to remove an item from a tuple.

## Chapter 4. - Operators

### Arithmetic operators

Arithmetic operators are used for basic mathematical operations, i.e. addition, subtraction, multiplication, division, modulo operation (modulus), exponentiation and floor division.

Let's start with addition. Its operator is the plus sign **+**. It adds the numbers on either side of the sign.

```
In [3]: 4 + 3
Out[3]: 7
```

Subtraction is done using the minus sign **-**. It subtracts the right hand operand from left hand operand.

```
In [4]: 4 - 3
Out[4]: 1
```

Multiplication is done using the asterisk sign \*. It multiplies values on either side of the operator.

```
In [5]: 4 * 3
Out[5]: 12
```

Division is done using the forward slash operator /. It divides the left hand operand by right hand operand. It's important to note that even when two **integers** are divided, the operation returns a **float**.

```
In [6]: 4 / 3
Out[6]: 1.3333333333333333

In [13]: print (type(4/3))
<class 'float'>
```

The modulo operation, or modulus divides left hand operand by right hand operand and returns remainder. It's done using the percent sign %.

```
In [7]: 4 % 3
Out[7]: 1
```

Exponentiation or power operation, performs exponential (power) calculation on operators. It can be thought of as repeated multiplication. It's done using **two asterisk** signs \*\*.

```
In [8]: 4 ** 3
Out[8]: 64
```

Last but not least, we have floor division. It's a division of operands where the result is the quotient in which the digits after the decimal point are removed.

```
In [9]: 4 // 3
Out[9]: 1
```

However, if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity).

```
In [14]: 4 // -3
Out[14]: -2
```

All these examples, can also be done using variables. Here's an example for multiplication.

```
In [11]: a = 5
          b = 2

In [12]: a * b
Out[12]: 10
```

Now we covered the basic of arithmetic operators, we will meet them later, in more complex operations. But let's now see another important group of operators.

## Comparison Operators

As the name suggests, they compare the values on either sides of them and decide the relation among them. They are also called Relational operators. They return a **boolean value**, or in other words, **true** or **false**. Let's jump into some examples. We will use variables now, **a** will hold 14 and **b** will hold 23.

The equal operator checks if the two variables (or numbers) are equal. Its checked by using **two** equal signs **==**.

```
In [15]: a = 14
          b = 23

In [16]: a == b
Out[16]: False
```

If the values of two operands are equal, then the condition becomes true. Since **a** and **b** are **not equal** it returns the boolean value **False**. That means that the not equal operator should return **True**. It's checked by using an exclamation mark and equal sign **!=**.

```
In [17]: a != b
Out[17]: True
```

To check if **a** is bigger than **b** we use the **greater than** operator **>**. It checks if the value of left operand is greater than the value of right operand, then condition becomes true.

```
In [22]: a > b
Out[22]: False
```

Since a is not greater, it returns False.

To prove that **a** is smaller than **b**, let's use the **less than** operator **<**. If the value of left operand is less than the value of right operand, then condition becomes **True**.

```
In [23]: a < b
Out[23]: True
```

Since 14 is lower than 23, it returns **True**.

In addition to greater than and less than, there are **greater than or equal to** and **less than or equal to** operators, checked with **>=** or **<=** signs, respectively. **Greater than or equal to** returns **True** if the value of left operand is greater than or equal to the value of right operand.

```
In [24]: a >= b
Out[24]: False
```

If the left hand operand is smaller than right hand operand the **less than or equal to**, returns true.

```
In [25]: a <= b
Out[25]: True
```

Comparison operators will often be used to check if certain conditions are met, for example in loops which will be covered later.

## Assignment Operators

Assignment operators are used in cases where we want assign a value to a variable. We have already encountered the first assignment operator, the equal sign `=`. It is used when a constant (value) is assigned to a variable.

```
In [15]: a = 14  
        b = 23
```

Or we want to assign the sum of two variables, **a** and **b**, to a new variable, **c**.

```
In [28]: c = a + b  
        print (c)  
37
```

Or, shorter written.

```
In [29]: c += a  
        # equal to: c = c + a  
        print (c)  
51
```

Similarly, we can do any other arithmetic operation mentioned above, combined with an assignment, subtraction, multiplication, division, modulo, exponentiation and/or floor division. Only multiplication is presented, but any other can be done following the example.

```
In [31]: c *= a  
        # equal to: c = c * a  
        print (c)  
518
```

All the mentioned operation, take the left operand, add, subtract, multiply... it with the right operand, and assign the result to the left operand.

## Logical Operators

Logical operators are used in cases where we want to combine conditional statements. I.e., we want to check if two or more conditions are met. There are

three logical operators, **and**, **or** and **not**.

The **and** operator returns **True**, if all checked conditions are **True**. If just **one condition is not met**, it will return **False**.

```
In [45]: a = 14  
         b = 23  
  
In [46]: a > 10 and b > 20  
Out[46]: True
```

The **or** operator, will return **True**, if **at least one** checked condition is **True**. It will only return **False**, when **none of the conditions is met**.

```
In [47]: a < 10 or b < 15  
Out[47]: False
```

Here, both conditions are **False**, hence the **or** operator returned **False**.

The **not** operator returns **True** if all conditions are **False**, or it returns **False**, if all conditions are **True**. Or easily, we can think of it, if we wanted **the opposite of the and** operator.

```
In [49]: not(a < 10 and b < 15)  
Out[49]: True
```

There are a lot more combinations possible, and I highly encourage you to try them by yourself, since it sometimes can get confusing, and the best way is to try it out.

## Bitwise Operators

Bitwise operators are used when we want to compare numbers on the binary level. The integers are first converted to binary format and then the selected operation is performed bit by bit. The result is returned in decimal format. The bitwise operators are **&** (and), **|** (or), **^** (xor), **~** (not), **<<** (zero fill left shift) and **>>** (signed right shift).

Bitwise **and (&)** sets the bit to 1 if both bits are 1. It will be clear, when we see an example.

```
In [56]: a = 23
         b = 25

         print (bin (a))
         print (bin (b))

         0b10111
         0b11001
```

Binary value of **23 is 10111**, while the binary value of **25 is 11001**. So if we position these two binary values one on top of other, we see that the **first** and **last** bit are for both numbers, **1**.

```
In [58]: print (a & b)
         print (bin(a & b))

         17
         0b10001
```

It's important to notice that a bitwise operation returns a decimal value, of the result. And decimal value of the binary 10001 is 17.

Similarly as the logical or, the bitwise or (`|`) returns a 1 if at least one of the values is 1. Again, if we look at the binary values of 23 and 25, 10111 and 11001, respectively, we see that, **a at least one 1 appears on each position (each bit)**. And the decimal value of the binary 11111 is 31.

```
In [60]: print (a | b)
         print (bin(a | b))

         31
         0b11111
```

The logical **XOR** operator (`^`) copies the bit, if it's set in one operand, but not both. It's somewhat similar to bitwise **or**, but in case of **both bits being 1**, it **will not activate** and set the result to 1.

```
In [61]: print (a ^ b)
         print (bin(a ^ b))

         14
         0b1110
```

The first bit is not printed, since it is zero (0). So the result is basically **01110**.



## Membership operators

Membership operators test for membership in a sequence, such as strings, lists, or tuples. I.e., if our goal is to test if a element is contained in a list, tuple or string.

There are two identity operators, **in** and **not in**.

IN evaluates if the variable is contained in the desired sequence, string, list, dictionary...

```
In [4]: x = "The weather is nice"
        y = {"Air temperature": 32.0, "Humidity": 86, "Precipitation": 3}

In [5]: print ("w" in x)
        print ("nice" in x)
        print ("prec" in x)

        True
        True
        False

In [8]: print ("weather" in y)
        print ("Air temperature" in y)
        print (32.0 in y)

        False
        True
        False
```

Similarly, we use **not in**, which return **True** is the variable **IS NOT** in the sequence, and **False** if the variable **IS IN** the sequence.

```
In [9]: print ("rain" not in x)
        print ("weather" not in x)

        True
        False

In [10]: print ("Humidity" not in y)
        print ("Precip" not in y)

        False
        True
```

## Identity Operators

Identity operators compare the memory locations of certain object, meaning they do not test if the two object are equal in value, or size, moreover they test if two objects have same location in the memory. The identity operators are **is** and **is not**.

IS returns True if the objects on the left and right of the operator point to the same object in memory.

```
In [14]: x1 = 5
         y1 = 5

         x2 = "Air_temperature"
         y2 = "Air_temperature"

         x3 = ["rain", "snow", "sleet", "hail"]
         y3 = ["rain", "snow", "sleet", "hail"]

In [16]: print (x1 is not y1)
         print (x2 is y2)
         print (x3 is y3)

False
True
False
```

Since list are mutable objects, meaning, elements can be added, or elements can be removed from the list using the assignment statement. the examples x1, y1 and x2, y2 contain integers and string, respectively, which are immutable, meaning they once assigned, can not be changed, therefore they also share the same place in the memory.

## Chapter 5. - Control flow

### Introduction


First and foremost, what is Control Flow? Well, in simple terms it is the order in which certain operations are executed. Let's take a simple example, say we want to measure the air temperature outside. What do we need to do? First we take the thermometer, then we open the door, we go outside, we close the door, we find a place in the shadow, we measure the temperature, we write it down, etc.

Similar, in programming, if we want to do a specific task or operation, we need to do simple smaller steps. These steps can for example include decision making or repetition of a task for a given number of times.

Let's say we wanted a script that according to some criterion executes differently, for example, if we measured the air temperature is 3°C it prints out "It's cold outside", but if it's 21°C it prints out "It's warm outside". In this case, some

condition is checked, and according to the condition, a task is executed (a certain statement is printed out).

Or let's say we wanted to convert the measured air temperature on each weekday in degrees Celsius to Fahrenheit. Then we need a script that takes the measured temperature on each weekday and executes the following the formula:

  $(^{\circ}\text{C} * 1.8) + 32 = ^{\circ}\text{F}$

This means we do a similar task for a known number of times, precisely seven times. We take seven numbers and multiple each by 1.8 and add 32 to it.

Let's now see some of the examples in more detail and with code provided. 😊

## Conditionals or Conditional statements

Conditional statements, or often called ***if-then*** statements, allow us to execute a piece of code depending on some condition (Boolean condition).

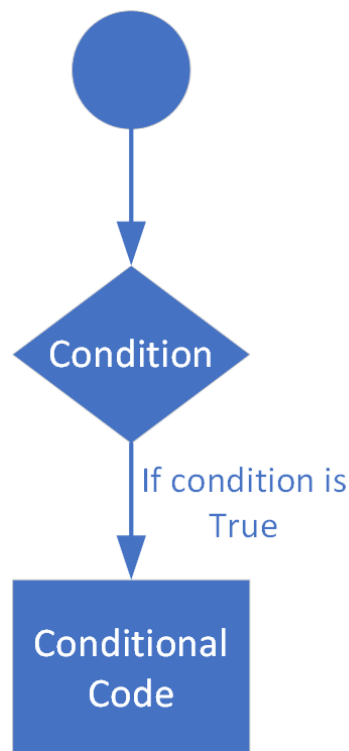
Conditional statements in Python are:

- Simple if
- if-else
- nested if
- if-elif-else

The keywords (symbols) to apply a conditional statement are **if, elif, else and a colon (:)**. It's important to **indent** the new line after a colon.

### Simple if statement:

Let's visualize the simplest case.



If the **if** statement expression evaluates to **True**, then the indented code following the statement is executed. If the expression evaluates to **False** then the indented code following the **if** statement is skipped and the program executes the next line of code which is indented at the same level as the **if** statement.

```
In [1]: x = 10
        if x > 5:
            print ("X is greater than 5!")

X is greater than 5!
```

We declare the variable `x` to be 10. We check if `x` is greater than 5, since this statement is **True** ( $10 > 5$ ), the **print** command gets executed, and the sentence "**X is greater than 5!**" is printed out.

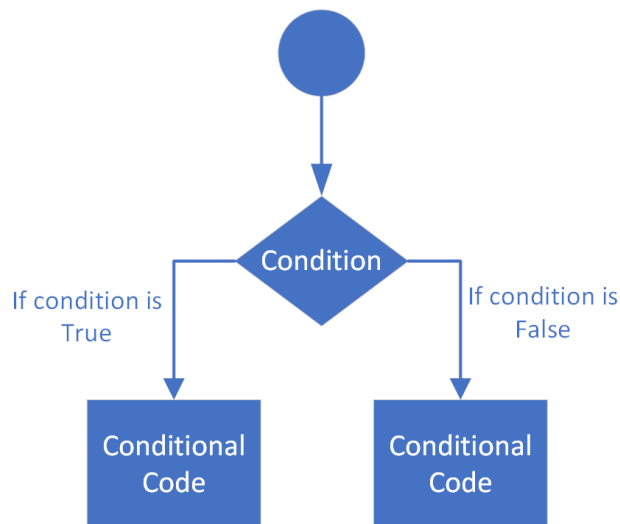
What about the case when this statement is not True? Then, this script would do nothing.

If we wanted a script that does something when the statement is **False** we need to modify it a little.

## if-else statement

As mentioned above, this statement allows us to add a second possibility to the script, what to do when the condition is **False**.

Let's visualize this case.



The indented code for the **if** statement is executed if the expression evaluates to **True**. The indented code immediately following the **else** is executed only if the expression evaluates to **False**. To mark the end of the else block, the code must be unindented to the same level as the starting **if** line.

```
In [4]: air_temp = 11

if air_temp > 15:
    print ("It's warm outside. You don't need a jacket!")
else:
    print ("It's fresh outside, you better take a jacket.")

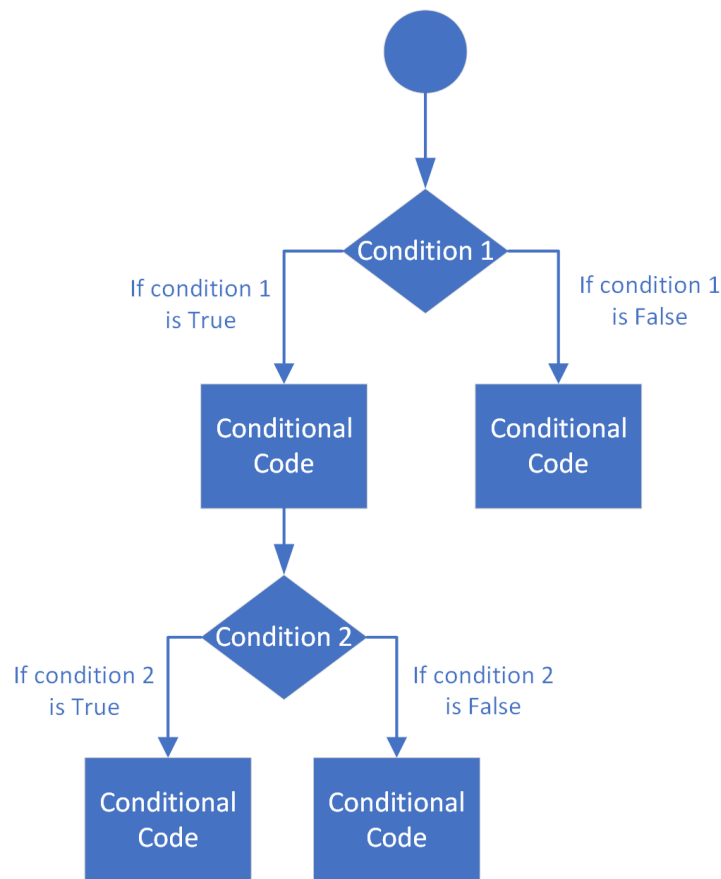
It's fresh outside, you better take a jacket.
```

The above code checks if the entered temperature is above 15, if the statement is **True**, it prints out that we don't need a jacket, but if the air temperature is below 15, the script suggests to take a jacket.

Now we know that if it's above 15 °C we don't need a jacket, but what shirt to take, short or long sleeved? In order to answer this question with our script we need to use a nested if statement.

## nested if statement

Basically that's an if statement inside another if statement. Let's see it visualized, to get a better understanding.



```
In [12]: air_temp = 22

if air_temp > 15:
    print ("It's warm outside. You don't need a jacket!")
    if air_temp <= 20:
        print ("Take a long sleeved shirt!")
    else:
        print("Take a short sleeved shirt!")
else:
    print ("It's fresh outside, you better take a jacket.")
```

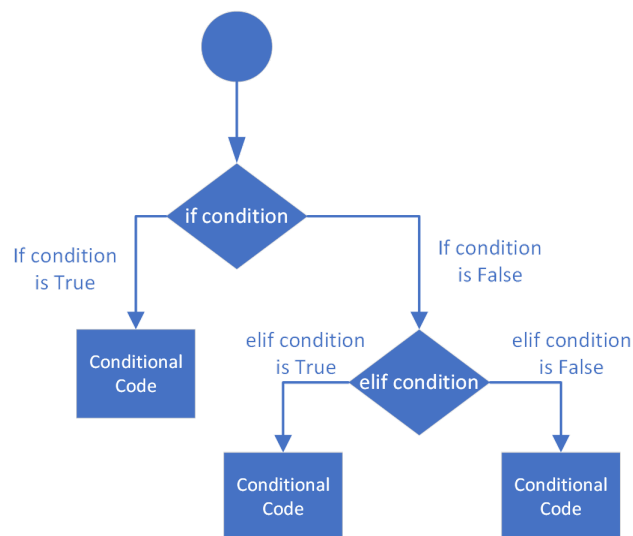
It's warm outside. You don't need a jacket!  
Take a short sleeved shirt!

As before, the code checks if the air temperature is above 15°C, if the statement is **True**, it suggests that we don't need a jacket. Next it checks if it's below or equal to 20°C, if that's **True**, it suggests a long sleeved shirt. If that's **False**, so it's warmer than 20°C, it suggests a short sleeved shirt.

This example can also be solved using a **if-elif-else** statement. Let's see how.

## if-elif-else statement

The **elif** keyword can be thought as *else if*, we used it if we want a more distinct division between **if** and **else**. The Python **elif** statement allows for continued checks to be performed after an initial **if** statement. An **elif** statement differs from the **else** statement because another expression is provided to be checked, just as with the initial **if** statement.



If the expression is **True**, the indented code following the **elif** gets executed. If the expression evaluates to **False**, the code can continue to an optional **else** statement. Multiple **elif** statements can be used following an initial **if** to perform a series of checks. Once an **elif** expression evaluates to **True**, no further **elif** or the **else** statement is being executed.

```
In [13]: air_temp = 22

if air_temp < 15:
    print ("It's fresh outside, you better take a jacket.")
elif air_temp <= 20:
    print ("It's warm outside. You don't need a jacket!")
    print ("Take a long sleeved shirt!")
else:
    print ("It's warm outside. You don't need a jacket!")
    print("Take a short sleeved shirt!")

It's warm outside. You don't need a jacket!
Take a short sleeved shirt!
```

The code first checks if the air temperature is bellow 15°C, if **True**, it suggests taking a jacket. If **False**, it checks if it's bellow or equal to 20°C, if **True**, it suggests a long sleeved shirt. If both statements are **False**, the **else** statement gets executed, and it suggests taking a short sleeved shirt.

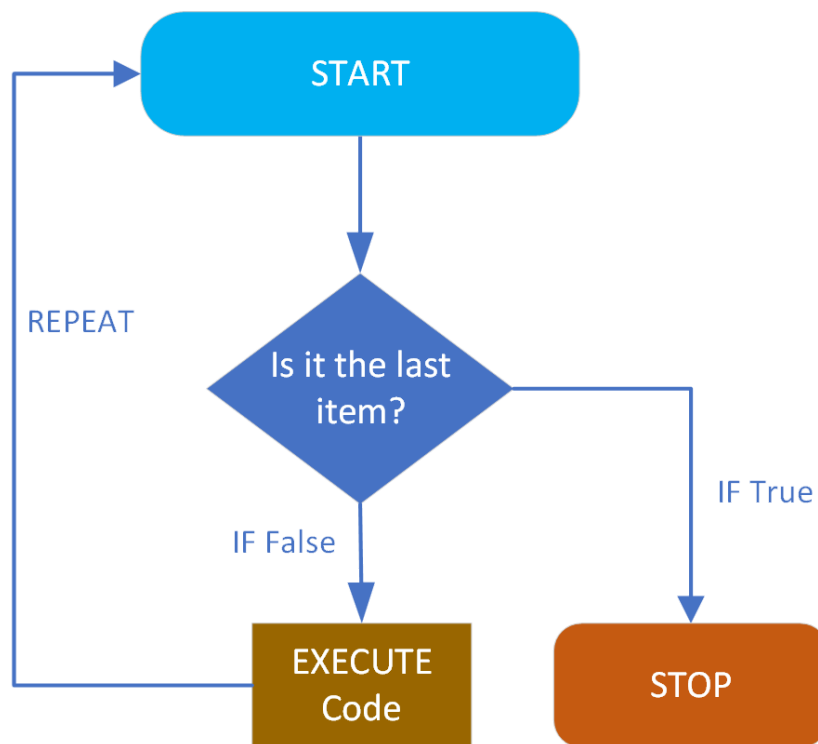
# Loops

Or often referred as repetition statements, are used to repeat a block of instructions. In Python, there are two types of loops:

- for loops
- while loops

## for loops

For loops are used when we iterate over a sequence of data, i.e. a list, tuple, dictionary, string etc. They are used when we iterate for a known (or desired) number of times, since we know how many elements there are in a list or string. The keywords to apply a **for** loop are **for** and **in**.



This visualization shows a simple **for** loop workflow. We initialize a sequence (list, string, tuple...), the loop checks if the item is the last, if **True** the loop stops, if **False** the code gets executed and the loop repeats on next item in the sequence. Until the code gets executed for the last item, the loop will repeat itself.

Let's see an example. We will calculate air temperature in Fahrenheit for an week of Celsius degrees measurements. Therefore we declare a python **dictionary** which holds the weekday as **key** and the measured daily air temperature as **value**.




```
In [28]: air_temps = {"Monday": 16.4,
                    "Tuesday": 17.3,
                    "Wednesday": 17.2,
                    "Thursday": 14.5,
                    "Friday": 18.9,
                    "Saturday": 20.1,
                    "Sunday": 20.4}


for k, v in air_temps.items():
    temp_f = (v * 1.8 ) + 32
    print ("The air temperature on {} was {:.2f}°F.".format(k, temp_f))

The air temperature on Monday was 61.52°F.
The air temperature on Tuesday was 63.14°F.
The air temperature on Wednesday was 62.96°F.
The air temperature on Thursday was 58.10°F.
The air temperature on Friday was 66.02°F.
The air temperature on Saturday was 68.18°F.
The air temperature on Sunday was 68.72°F.
```

When we loop over a dictionary we use the **.items()** method, which returns the **k (key)** and **v (value)**. Next, we create a new variable called **temp\_f** which holds the calculated temperature in °F. To print out the calculated temperatures we use the **string format method**. This method converts the given variable to a string and allows us to use it in a sentence. We have to use **curly braces {}** on the wanted location in the sentence.

 "The air temperature on {} was {:.2f}°F."

After the sentence (string), notice the quotation marks, we apply the **.format () method**. We provide the variables we want to print out in the sentence, in this case, the **k (weekday)** and **temp\_f (calculated temperature in Fahrenheit)**. The **:.2f** means we convert the calculated value (float) to a string, with 2 decimal places after the decimal point. [Check the official sites for more insights.](#)

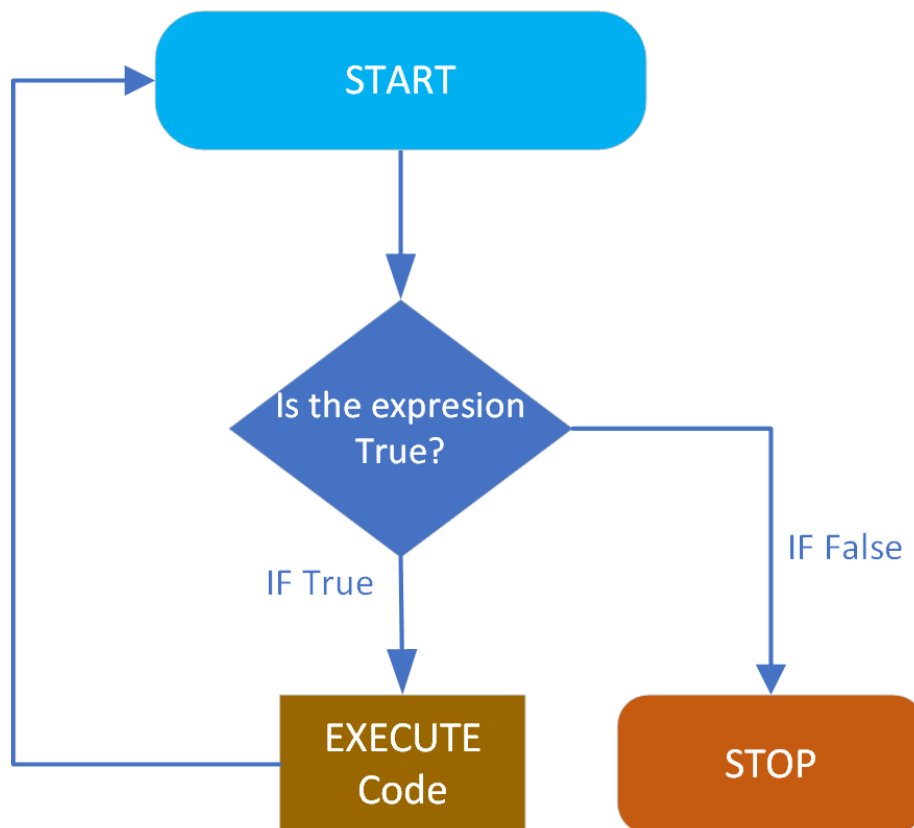
 .format(k, temp\_f)

The great thing about loops is, we need to do this once, but the loop will repeat for each weekday and execute the wanted task.

## while loops

In Python, while loops are used to iterate until a certain condition is satisfied. Basically, the loop is executed as many times as the condition remains True. when

if becomes False, the loop stops. Let's clarify this statement with a simple visualization.



A simple while loop works as follows. The argument gets evaluated, according to a expression, the code gets executed until the argument is evaluated as True. When it becomes False, the loop stops.



**VERY IMPORTANT:** In every step of the while loop we need to change the argument in order to avoid an endless loop.

Let's see an example with counting days.

```
In [37]: day = 1
         while day <= 8:
             print(day)
             day += 1

1
2
3
4
5
6
7
8
```

We start with `day` equal to one. The argument (value of the variable `day`) is checked if it's lower or equal to eight, if **True**, it gets printed out. After printing we **increase the value of the variable `day` by one to avoid an endless loop**. If we hadn't increased `day` by one in each loop, the variable `day` would have stayed equal to one, and so less than eight and the loop would have been endless.

## Controlling loops with Breaks and Continues

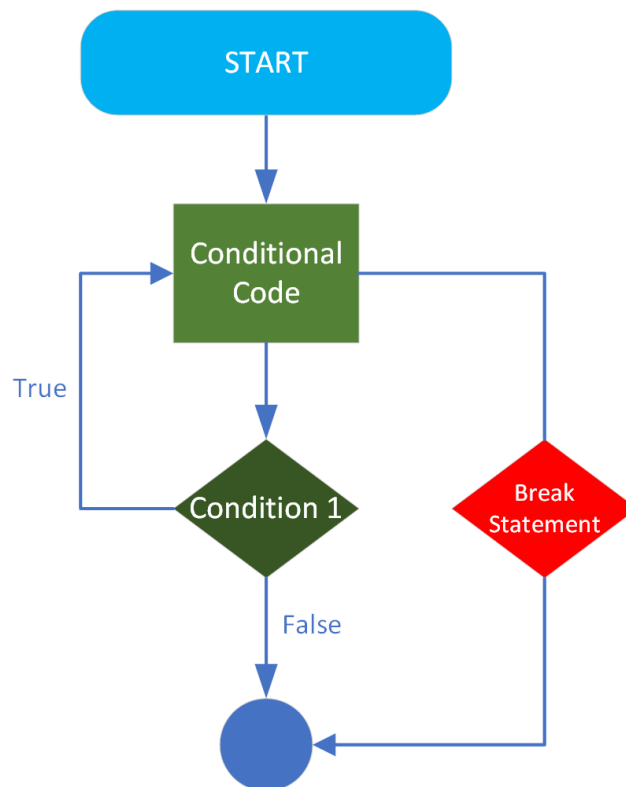
The **break** and **continue** statements help us fine tune our loops and their execution.

- The **break** statement breaks-out of the loop entirely
- The **continue** statement skips the remainder of the current loop, and goes to the next iteration

Both of them can be used in both for and while loops.

### break statement

The **break** statement is used in situations where we want to break out of the loop, even if the condition has not become **False** or we have iterated over the entire sequence. Also, if **break** is used, any following else blocks are not executed.



Let's take a simple example with our air temp measurements in a week. Let's say we want to stop printing out temperature values, if the air temperature is equal to or higher than 18°C.

```

In [42]: air_temps = {"Monday": 16.4,
                      "Tuesday": 17.3,
                      "Wednesday": 17.2,
                      "Thursday": 14.5,
                      "Friday": 18.9,
                      "Saturday": 20.1,
                      "Sunday": 20.4}

for k, v in air_temps.items():
    if v >= 18:
        break
    print ("The air temperature on {} was {:.1f}°C.".format(k, v))
  
```

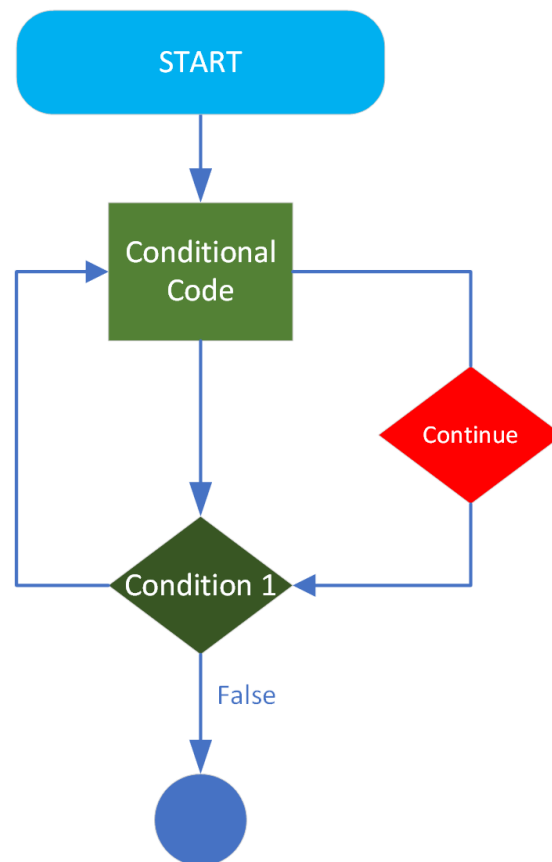
The air temperature on Monday was 16.4°C.  
 The air temperature on Tuesday was 17.3°C.  
 The air temperature on Wednesday was 17.2°C.  
 The air temperature on Thursday was 14.5°C.

The code prints out temperature values, when we reach Friday, with a temperature of 18.9°C, the loop breaks, and stops.

## Continue statement

The continue statement is somewhat similar to the break statement, but instead of breaking the loop, it will start the next iteration. Let's see the visualization, to get a

clearer understanding.



Let's say we have a case where we want to print out temperatures but exclude those that are lower than 15°C. This would mean, we print out all except the values for Thursday. For such a task we would use a continue statement. So we loop through the dictionary, and when we reach a temperature that is lower than 15°C, the continue statement is activated, and it avoids the execution of the print statement, instead, it jumps to the next iteration. Let's see the code.

```
In [47]: air_temps = {"Monday": 16.4,
                    "Tuesday": 17.3,
                    "Wednesday": 17.2,
                    "Thursday": 14.5,
                    "Friday": 18.9,
                    "Saturday": 20.1,
                    "Sunday": 20.4}

for k, v in air_temps.items():
    if v < 15:
        continue
    print ("The air temperature on {} was {:.1f}°C.".format(k, v))

The air temperature on Monday was 16.4°C.
The air temperature on Tuesday was 17.3°C.
The air temperature on Wednesday was 17.2°C.
The air temperature on Friday was 18.9°C.
The air temperature on Saturday was 20.1°C.
The air temperature on Sunday was 20.4°C.
```

The continue statement is great for discarding or excluding tasks, where our goal is to avoid a value, group of values or a certain condition.

## Chapter 6. - Numpy

Why to use Numpy? When printed, a Python list of integers or floats, looks exactly the same as a Numpy ndarray. Both can do mathematic operations on a bunch of numbers, both can do statistical calculations and comparisons can go on... So you could think Numpy is just a mathematical library with similar functionality as lists, but is it? Let me explain...

The data in Numpy arrays is of homogeneous type, meaning all the data in an array is of same type, while lists are just pointers to objects, even though all the data is of the same type. As a consequence, the Numpy arrays use much less memory than regular lists. Also, most of the Numpy operations is implemented in the C language, meaning, the cost of Python loops and dynamic checking of the data type is avoided. This, yields a significant increase in processing speed when comparing Numpy to a Python list.

More often than one we encounter large datasets with tens of thousands rows of data, just think of hourly air temperature measurements for a county or region since the measurements beginning in this region. If your weather service is measuring hourly air temperature for last 50 years, that's more that 400 000 rows of data, just for one station.

### How to install Numpy?

Well, if using Anaconda, Numpy is preinstalled in the base environment. However, more often than not, it's good practice to create new environments for new projects. TO install Numpy, we run an Anaconda prompt and type:

```
conda install numpy  
  
or  
  
conda install -c anaconda numpy
```

If pip is being used, Numpy can be installed by typing:

```
pip install numpy
```

## How to import Numpy?

When importing certain libraries, including Numpy, we follow a convention, basically this means we use well established abbreviations for libraries. In the case of Numpy we use "np".

```
import numpy as np
```

The goal is that our code is reproducible, and every Python programmer in the World, knows what the following line does:

```
a = np.array([3,4])
```

Congrats, if you have imported Numpy, and used the above command, you have successfully created your first Numpy array. Let's see what happens if we print it out.

Print gives us something that looks like a list, but it's not. When we check the type we see that's a "**numpy.ndarray**".

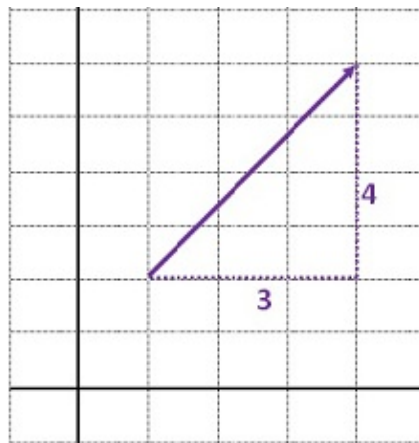
```
In [9]: a = np.array([3,4])  
  
        print (a)  
        print (type(a))  
  
[3 4]  
<class 'numpy.ndarray'>
```

## Vectors?

In the example we saw how we can create a 1-dimensional array. If you remember vectors from Math, well a 1-dimensional Numpy array is basically a vector. Since we gave two numbers, 3 and 4, this vector lies in the 2-dimensional space (geometric plane). It's same as in math when you had a vector:



$$v = 3i + 4j$$



In Computer Science, vectors are just lists, where the length of the list (in our case, 2) is the number of dimensions of the vector. And in Data Science terms, a vector represents one or multiple features of an object. Think of meteorological measurements on Monday, you could measure air temperature, precipitation, wind speed, snow depth, etc. To learn more about vectors, I highly recommend [this video](#) by 3Blue1Brown.

## Creating arrays

Above we already saw how to create a simple 1-D array in Numpy. Often, our data comes in more dimensions, we have multiple features (like above), but also have measurements for multiple days in the week. In this case, we need to add a second dimension to our arrays. Let's see some 2-D arrays.

```
In [25]: weather_data = np.array([[13.4, 0.3, 1.6, 0], [14.8, 0., 1.1, 0]])
        weather_data
Out[25]: array([[13.4, 0.3, 1.6, 0. ],
               [14.8, 0. , 1.1, 0. ]])
```



To create a 2-D array, we provide a list, containing two lists. Think of this array as measurements on Monday (first list/row) and Tuesday (second row/list) where 1st column is air temperature, 2nd column precipitation, 3rd column wind speed and 4th column snow depth. The excel screenshot should clarify things.

	A	B	C	D	E
1		T_air	P	WS	SD
2	Monday	13.4	0.3	1.6	0
3	Tuesday	14.8	0	1.1	0
4					

Numpy also provides some useful functions to create arrays of **zeros** or **ones**. Try out the following commands by yourself, and print out the results.

```
zeros = np.zeros([2,3])

ones = np.ones([3,4])
```

To demonstrate how to get the number of **dimensions** of your newly created array, I will use the np.ones function together with the **ndim** attribute.

```
ones = np.ones([2,3,4,5])
ones.ndim

4
```

So, our array has four dimensions, but how does a 4-dimensional array look like?

```
In [67]: print (ones)

[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]]

[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]]

[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]]]]
```

If we take a closer look, we can determine the number of dimension if we count the square brackets on the start or the end of the array also, a handy hack 😊

Another useful method is **arange**. It is used to get an evenly spaced array. We need to specify the end number (int or float).

```
range_a = np.arange(7)
range_a

array([0, 1, 2, 3, 4, 5, 6])
```

Numpy then assumes that starting point is zero. We can also provide the starting and ending point.

```
range_b = np.arange(7.,12.)
range_b

array([ 7.,  8.,  9., 10., 11.])
```

And, we can specify the step as follows:

```
range_c = np.arange(7,12,2)
range_c

array([ 7,  9, 11])
```

Similarly, with the method `linspace` we can create an array, but instead of the step, **`linspace`** takes the number of elements in the array. Here we create an array with five elements between 7 and 12. Also, contrary to `np.arange` and most of Python methods, the last number (ending number) here is **including**.

```
range_d = np.linspace(7,12,5)
range_d

array([ 7. ,  8.25,  9.5 , 10.75, 12.  ])
```

## Shape and Reshape

Before, we checked how many dimensions (or axes) our *ones* array had. But what if we are interested in how many elements are in each of the dimensions? The **`shape`** attribute comes in handy. Since we have 4 dimension, we get a tuple of 4 numbers.

```
In [69]: ones.shape
Out[69]: (2, 3, 4, 5)
```

To count the number of elements in the whole array we use the **`size`** attribute.

```
In [70]: ones.size
Out[70]: 120
```

In order to change the shape of an array, we use the **`.reshape()`** method. Care has to be taken though, the newly reshaped array has to be of same size as the old one. Let me explain..

```
In [77]: zeros_new = zeros.reshape(3,2)
print (zeros, zeros.shape)
print(zeros_new, zeros_new.shape)

[[0. 0. 0.]
 [0. 0. 0.]] (2, 3)
[[0. 0.]
 [0. 0.]
 [0. 0.]] (3, 2)
```

The original zeros array had the shape of (2, 3), and we can reshape it into (3,2), (6, 1) or (1,6), since it has a size of 6 elements. I shall mention, that in case of reshaping it to (6, 1) or (1, 6) we change the number of dimensions, from a 2-D array, to a 1-D array, but as long we take care of the array size, we are on the safe side.

A handy "shortcut" to a 1-D array are the **flatten()** and **ravel()** methods. The difference is that **flatten** creates a 1-D copy of the original array, while **ravel** creates a reference to the original array. So, using **ravel()** has the consequence that changing for example some of the data in the newly created array while also change the data in the original array.

The usage depends on the specific task, most of the time I've used the **flatten()** method.

```
In [79]: ones_flat = ones.flatten()
print (ones.shape)
print (ones_flat.shape)

(2, 3, 4, 5)
(120,)
```

Last but not least, let's not forget the **transpose()** method. This method simply swaps the rows and columns of an array.

```
In [85]: zeros_transposed = zeros.T
print (zeros_transposed, zeros_transposed.shape)

[[0. 0.]
 [0. 0.]
 [0. 0.]] (3, 2)
```

In this case, the result is same as before with reshape. In case of a multidimensional array, all dimensions get swapped, let's see.

```
In [87]: ones_transposed = ones.T
print (ones_transposed.shape)

(5, 4, 3, 2)
```

## Accessing elements and slicing an Array

Until now, we saw how to create, find proportions and reshape or flatten an array. Let's turn our focus now on data extraction from an array using **indexing** and **slicing**. To **slice** an array means to access its elements by providing the desired elements **index**.

The default syntax of slicing involves the array name and square brackets, like for Python lists, as follows:

```
| array_name[start_index : end_index : step_size]
```

```
In [32]: hourly_air_temp = [0, 1, 1, 2, 3, 3, 4, 6, 7, 9, 9, 10,
                           12, 14, 16, 16, 14, 11, 9, 8, 7, 4, 3, 2]

print ("Temperature very 2 hours - afternoon:", hourly_air_temp[11:20:2])

Temperature very 2 hours - afternoon: [10, 14, 16, 11, 8]
```

In our array the temperature measurements start from 1:00. To clarify the things, we printed out hourly temperatures at 12:00, 14:00, 16:00, 18:00 and 20:00.

If we don't define the step size, every element in the specified range gets returned. For example if we need the hourly temperatures from 7:00 to 12:00.

```
In [35]: print ("Temperatures from 7:00 to 12:00 - morning:", hourly_air_temp[6:12])

Temperatures from 7:00 to 12:00 - morning: [4, 6, 7, 9, 9, 10]
```

As usual in Lists, so in Numpy also, the `start_index` is including, while the `end_index` is not including. Also, the first index in an array is always zero. So to access the temperature at 7:00, we input 6-th index. And since we need the measurements until 12 (which is at index no. 11), we provided 12th index (it's excluding).

## Slicing 2-D arrays

When slicing a 2-D array, we need to specify the row and column of the element we desire. It can be a little tricky at first, but when tried on few examples, it's soon gets really easy.

The syntax to slice a 2-D array is as follows:

```
array_name[row_start_index : row_end_index : row_step_size,  
           column_start_index : column_end_index :  
           column_step_size]
```

To see how slicing a 2-D array works, I will first extend our weather\_data array to a full week, so we shall get an array of shape (7,4).

```
In [39]: weather_data = np.array([[13.4, 0.3, 1.6, 0],  
                                  [14.8, 0. , 1.1, 0],  
                                  [14.9, 0.1, 1.2, 0],  
                                  [13.2, 8.6, 3.7, 0],  
                                  [15.7, 0. , 0.4, 0],  
                                  [15.8, 0. , 0.8, 0],  
                                  [16.4, 0. , 0. , 0]])  
weather_data.shape  
Out[39]: (7, 4)
```

Let's say we want to know the weekly precipitation. So we need to slice out all rows, and the 2nd column.

```
In [46]: weekly_precip = weather_data[:,1]  
weekly_precip  
Out[46]: array([0.3, 0. , 0.1, 8.6, 0. , 0. , 0. ])
```

In this case, to select all the rows we use a **colon** sign (:). To select the 2nd column we use the **index value of 1** (remember, **indices start from zero**).

Feel free to try out other possibilities, I would compare slicing with integrals in Math, there are certain rules to follow, *but practice makes perfect*.

Negative slicing is also allowed, and works in same manner as with python lists. The last item in an array has the index of -1.

## "Finding" data in an array

Another way of finding data in an array is by using a very popular function inside of Numpy, **np.where()**. The function returns the indices of elements that meet a condition. Commonly, it's used when finding elements that are greater, equal or less than a number. The basic syntax of np.where() is as follows:

```
np.where(condition [, x, y])
```

**x** and **y** are parameters which can be used to **replace** the value in the array that meets the given condition. Either **we don't provide x, y** (we just need to **find indices or values** that meet the condition), or we **provide both x, y** then the values at the found index get changed by **x if True**, or **y if condition is False**. Very similar like **IF() function** in MS Excel.

Let's say we want to print out the indices of days that were warmer than 14.5 degrees °C.

```
In [6]: print ("Warmer then 14.5 °C: ", np.where(weather_data[:,0] > 14.5))  
Warmer then 14.5 °C: (array([1, 2, 4, 5, 6], dtype=int64),)
```

We have two important things here: firstly, we use the above learned slicing to select the first column (all rows, since we search all weekdays), and then we set the condition > 14.5.

Let's say we want to convert all temperature values greater than 14.5 to Fahrenheit degrees, and store the resulting array to the variable **weather\_b**.

```
In [11]: weather_b = np.where(weather_data[:,0] > 14.5, weather_data[:,0]*1.8+32, weather_data[:,0])  
weather_b  
Out[11]: array([13.4 , 58.64, 58.82, 13.2 , 60.26, 60.44, 61.52])
```

Again, we first provide a condition ( >14.5 °C), then we give what value (multiply the value by 1.8 and add 32) to use if **True**, and what value (use the existing value) to use if **False**. Notice, that we always slice the array, since we work only on the first column.

We can now create a new array called **weather\_f** (identical to **weather\_data**) but the temperature values will be replaced with values in Fahrenheits.

```
In [19]: weather_f = weather_data.copy()  
weather_f[:,0] = weather_b  
weather_f  
Out[19]: array([[13.4 , 0.3 , 1.6 , 0. ],  
                [58.64, 0. , 1.1 , 0. ],  
                [58.82, 0.1 , 1.2 , 0. ],  
                [13.2 , 8.6 , 3.7 , 0. ],  
                [60.26, 0. , 0.4 , 0. ],  
                [60.44, 0. , 0.8 , 0. ],  
                [61.52, 0. , 0. , 0. ]])
```

First, we make a **copy** of `weather_data` (remember **`flatten()`** and **`ravel()`** methods ), to avoid changes in original `weather_data` array, and then we slice the new `weather_f` (first column), and replace the values with the ones calculated in `weather_b`.

## Maths and Statistics with Numpy

Finally we've come to my favorite part of Numpy, mathematical and statistical operations. This is in my eyes what makes Numpy so great, and superior to same operations with common lists. It's the simplicity and speed advantage when dealing with a great amount of numerical data. Let's first take a look to mathematical operations. I will provide an example with division, but the general syntax is the same for other operations, and can be looked up at the official [Numpy pages](#).

Our `weather_data` array contains precipitation data in millimetres, lets convert those to metres. To convert millimetres to metres, we need to divide the value by 1000.

```
In [16]: precip_metres = weather_data[:, 1]/1000
         precip_metres

Out[16]: array([0.0003, 0.      , 0.0001, 0.0086, 0.      , 0.      , 0.      ])
```

Again we use slicing, to select the second column of the array, and divide the values by 1000. For practice, try to replace precipitation values from the array `weather_f` with the ones converted to metres. (you can do the change on the `weather_f` array directly)

As for the statistics example, I shall use the most common case, we need to calculate the average temperature, precipitation, wind speed and snow depth values for the week. The Numpy function to calculate the average values is called **`np.mean()`**. The basic syntax of `np.mean()` function is as follows:

```
np.mean(a, axis=None, dtype=None, out=None, keepdims=<no
value>, *, where=<no value>)
```

For our case, the important part is the **`axis`**. Since our goal is to calculate the mean values for each column, we need to set the **`axis`** parameter to **`0`**. Setting the axis to 1, would yield the result row-wise.



```
In [24]: mean_values = np.mean(weather_data, axis=0)
         mean_values

Out[24]: array([14.88571429,  1.28571429,  1.25714286,  0.        ])
```

Other statistical functions retain an equal or similar syntax, and can be looked up at the [Statistics section](#) of the official Numpy site.

## Bonus content - Speed advantages of Numpy

I've mention that Numpy also has some speed advantages, this probably got you tempted. Let's see that in action. Is Numpy really that faster than an for loop?

First, we will create an random array of floats, let's say its hourly air temperature measured at some location in th US. The length of the array is 30 000.

```
In [57]: hourly_temp_F = np.random.uniform(low=-20., high=100., size=30000)
         len(hourly_temp_F)
         hourly_f_list = hourly_temp_F.tolist()
```

We want to convert those numbers to Celsius degrees. Let's measure the time needed using a for loop and Python list, and then using Numpy.

```
In [60]: %%time
         hourly_temp_C = list()
         for temp in hourly_f_list:
             hourly_temp_C.append((temp-32)*(5/9))

         Wall time: 4.99 ms

In [61]: %%time
         hourly_temp_C = (hourly_temp_F-32)*(5/9)

         Wall time: 997 µs
```

So, the time taken using an loop over a Python list took around 5 ms, while using an Numpy array the same operation took less than 1 ms. So Numpy is in this specific case around 5 times faster. Also, please consider that this test is not completely applicable, it really depends on the speed of your computer (CPU) and the chosen task. Since this is not the main topic of the article, I'll leave it to you to

check other articles that are covering speed benefits of Numpy, and hopefully, try it out yourself, on your specific task with your data.

## Chapter 7. - Pandas

What is Pandas? Pandas is an extremely popular library built upon Numpy, for handling tabular data, data manipulation and analysis. Probably the best thing about Pandas is that it stores data as a Python object with rows and columns, very similar to data stored in Excel files. Also, this way we can easily visualize our data, making our job a lot easier than handling data in form of lists or dictionaries. Also, the advantage over Numpy is that it handles multiple data types (i.e., strings), not only numerical data, although I need to mention the downside, this makes it slower in comparison to Numpy.

### How to install Pandas?

Well, if using Anaconda, Pandas is preinstalled in the base environment. However, more often than not, it's good practice to create new environments for your new projects. To install Pandas in a new environment we activate the environment and then we type:

```
conda install pandas
```

If pip is being used, Pandas can be installed by typing:

```
pip install pandas
```

### How to import Pandas?

When importing certain libraries, including Pandas, we follow a convention, basically this means we use well established abbreviations for libraries. In the case of Pandas we use "pd".

```
import pandas as pd
```

## Pandas Series and Pandas Dataframe

Before we start dealing with some of Pandas' tools, we need mention the two data structures Pandas uses to store data, the Pandas **Series** and the Pandas **Dataframe**. Think of Pandas **Series** as an *1 column Excel spreadsheet*, with an

additional *index* column, or even better, if you are familiar with Numpy think of an one dimensional array.

Imagine daily meteorological observations at a point location, for example wind speed for the period of two weeks. Let's see an example.

```
In [4]: wind_speed = pd.Series([0.5, 1.2, 2.1, 1.1, 1.6, 1.8, 1.4,
                                0.6, 0.4, 0.3, 0.9, 2.5, 2.1, 2.8])

print (wind_speed)

0    0.5
1    1.2
2    2.1
3    1.1
4    1.6
5    1.8
6    1.4
7    0.6
8    0.4
9    0.3
10   0.9
11   2.5
12   2.1
13   2.8
dtype: float64
```

We use the **pd.Series()** command, and provide a list, in this case a list of floats with a length of 14.

When printed, we also get the indices of each element in the Series.

We also can create a Series from an 1-dim Numpy array. For the sake of efficiency, I shall use the same list of wind speeds.

```
array = np.array([0.5, 1.2, 2.1, 1.1, 1.6, 1.8, 1.4,
                  0.6, 0.4, 0.3, 0.9, 2.5, 2.1, 2.8])

wind_speed_b = pd.Series(array)
```

Try printing out the newly created **Series**, and compare it with the previous result.

You may think now, what if we have multiple **Series**, what if we also have for example precipitation measurements, well then, Pandas stores our data in a **Dataframe**. I'm pretty sure, you get the point, a **Dataframe** is just a container for multiple **Series**. Think of it as an Excel spreadsheet with multiple columns, and one index column, or as an N-dimensional Numpy array. Let's see an example.

```

array_ws = np.array([0.5, 1.2, 2.1, 1.1, 1.6, 1.8, 1.4,
                     0.6, 0.4, 0.3, 0.9, 2.5, 2.1, 2.8])
array_precip = np.array([0., 0., 5.8, 4.3, 2., 1.1, 0.,
                        0.7, 0.9, 1.3, 1.9, 6.5, 8.2, 1.8])
array_temp = np.array([11., 12.1, 12.2, 12.7, 10.8, 9.7, 13.2,
                      12.8, 12.6, 14.1, 14.8, 15.3, 14.4, 16.2])

weather_df = pd.DataFrame({"wind_speed": array_ws,
                           "precipitation": array_precip,
                           "temperature": array_temp})

```

To create a DataFrame we use the **pd.DataFrame()** command, and provide a **dictionary** where the **keys** represent our **column names**, and the **values** are the before created **Numpy** arrays (our data).

	wind_speed	precipitation	temperature
0	0.5	0.0	11.0
1	1.2	0.0	12.1
2	2.1	5.8	12.2
3	1.1	4.3	12.7
4	1.6	2.0	10.8
5	1.8	1.1	9.7
6	1.4	0.0	13.2
7	0.6	0.7	12.8
8	0.4	0.9	12.6
9	0.3	1.3	14.1
10	0.9	1.9	14.8
11	2.5	6.5	15.3
12	2.1	8.2	14.4
13	2.8	1.8	16.2

Looks kind of similar to an Numpy N-dimensional array. Each Pandas DataFrame consists of following components: **index, columns and data (values)**. The **indices** are the labels ("names") of each **row**, while the **columns** also have their labels ("names"). The indices always start with a zero (0) up to  $n-1$ , where  $n$  is the number of rows.

```

index = weather_df.index
column_names = weather_df.columns
data = weather_df.values

index
RangeIndex(start=0, stop=14, step=1)

column_names
Index(['wind_speed', 'precipitation', 'temperature'], dtype='object')

data
array([[ 0.5,  0. , 11. ],
       [ 1.2,  0. , 12.1],
       [ 2.1,  5.8, 12.2],
       [ 1.1,  4.3, 12.7],
       [ 1.6,  2. , 10.8],
       [ 1.8,  1.1,  9.7],
       [ 1.4,  0. , 13.2],
       [ 0.6,  0.7, 12.8],
       [ 0.4,  0.9, 12.6],
       [ 0.3,  1.3, 14.1],
       [ 0.9,  1.9, 14.8],
       [ 2.5,  6.5, 15.3],
       [ 2.1,  8.2, 14.4],
       [ 2.8,  1.8, 16.2]])

```

This DataFrame contains only floats (numbers), but any data type is allowed (integers, strings, Booleans..), although if combining multiple data types inside a column, care has to be taken.

To avoid, unexpected behaviour, we can also specify the column data type as follows:

```

weather_df = pd.DataFrame({"wind_speed": pd.Series(array_ws, dtype="float"),
                           "precipitation": pd.Series(array_precip, dtype="float"),
                           "temperature": pd.Series(array_temp, dtype="float")})

```

You can also use the **pd.Series()** command with a dictionary, if you want to name the created Series.

There are a lot more options and possibilities of creating Series and DataFrames, you can find those on the [official Pandas sites](#). But When working on Data Science or Data Analysis tasks often we have to deal with large datasets, stored as comma separated values (.csv) or Excel spreadsheets (.xlsx). Let's see how to read such files into a Pandas Dataframe .

## Reading data

To read a .csv file, we use the **pd.read\_csv()** command. When files are properly formatted, Pandas figures out which separator, column names, data types etc. are present in our data. All of the mentioned can also be provided, and many more arguments can be added to **pd.read\_csv()**, in order to widen the possibilities of reading differently formatted files. Also, providing the dtype argument for each (or some) column(s), significantly improves the speed of reading the file but also lowers the memory usage by quite a margin, find out more in this [great article](#).

To see how **pd.read\_csv()** works, I've created a dataset with some meteorological observations for entire year 2018 which we are going to read in as a Pandas DataFrame and work on. To visualize the newly created DataFrame Pandas offers some handy methods, **df.head()** and **df.tail()** (df stands for the name of our dataframe, in our case, weather\_data).

```
weather_data = pd.read_csv("test_data.csv")
print (weather_data.head())
print (weather_data.tail())
```

Here, few things are important: we need to make sure that our test\_data file is in the **same folder as our Jupyter notebook (or .py file)** and second that we correctly spell the name of the file, and also provide the extension **.csv**.

	date	precipitation	min_temp	max_temp	wind_speed
0	1/1/2018	0.0	5.2	11.2	4.2
1	1/2/2018	13.7	0.2	6.9	2.4
2	1/3/2018	0.0	-2.4	8.6	5.3
3	1/4/2018	0.4	-1.5	7.1	3.7
4	1/5/2018	0.0	5.3	13.8	5.3

	date	precipitation	min_temp	max_temp	wind_speed
360	12/27/2018	0.0	-4.4	8.8	1.9
361	12/28/2018	0.0	-2.2	6.4	1.3
362	12/29/2018	0.0	-3.8	8.4	1.9
363	12/30/2018	0.0	-2.5	5.5	2.4
364	12/31/2018	0.1	-0.6	7.3	4.8

The **.head()** method prints out the first 5 rows (by default, we can change by entering any integer in the brackets), while **.tail()** prints out the last 5 rows (also by default).

In same manner as for .csv files, we can also read Excel files.

```
weather_excel = pd.read_excel("test_data.xlsx")
```

Sometimes, an error can occur, "***XLRDError:Excel xlsx file; not supported***", then make sure to install the **openpyxl** library to your environment of choice, and use the command:

```
weather_excel = pd.read_excel("test_data.xlsx", engine="openpyxl")
```

I've used a basic example of reading comma separated value files, or Excel files, but make sure that you check the official documents ([for .csv here](#), and [for .xlsx here](#)) of Pandas to find out all the arguments we can use with those two methods. Also, you can check one of my previous articles on [opening and processing not so nicely formatted files with Pandas](#).

## Inspecting data

After we have successfully read the data, we need to take a closer look at our DataFrame. We already printed first 5 and last five rows, but what about the rest? It's always good practice to check the shape of your dataframe, similarly, like we did for Numpy ndarrays. Therefore, we use the **df.shape** method, which returns the number of rows, and columns of the DataFrame.

```
In [9]: weather_data.shape
Out[9]: (365, 5)
```

The dataframe has 365 rows, and 5 columns, since the year 2018 had 365 days, and we have observations for each day. Also, we have a date column, and 4 different observations.

We can get even more information about our dataframe by using the `df.info()` method, which yields the length, number and data type of the columns, and the size in the memory.

```
In [16]: weather_data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 365 entries, 0 to 364
Data columns (total 5 columns):
#   Column          Non-Null Count  Dtype
---  -
0   date             365 non-null   object
1   precipitation     365 non-null   float64
2   min_temp         365 non-null   float64
3   max_temp         365 non-null   float64
4   wind_speed       365 non-null   float64
dtypes: float64(4), object(1)
memory usage: 14.4+ KB
```

In Hydrological (and Meteorological) tasks, we are often interested in statistical parameters of our dataset, Pandas has a solution for this problem to. It provides the most commonly used statistical parameters as methods, mean, median, minimum, maximum, skewness etc. Let's see a few examples.

```
In [11]: weather_data.mean()

Out[11]: precipitation    2.462466
         min_temp         7.313699
         max_temp        17.281370
         wind_speed       3.523562
         dtype: float64
```

```
In [12]: weather_data.median()

Out[12]: precipitation    0.0
         min_temp         8.4
         max_temp        19.3
         wind_speed       3.0
         dtype: float64
```

```
In [13]: weather_data.max()

Out[13]: date            9/9/2018
         precipitation    33.4
         min_temp        20.0
         max_temp        33.4
         wind_speed      11.3
         dtype: object
```

Interestingly, **df.max()** also evaluates the object type column with the dates. Since we have not converted the dates to the datetime format, the maximum value is not 12/31/2018, but rather the maximum of a string, and that's 9, so 9/9/2018. In order



to convert the dates to the datetime format (which Python then evaluates as real dates) we use the **pd.to\_datetime()** functions, as follows:

```
weather_data["date"] = pd.to_datetime(weather_data["date"], format="%m/%d/%Y")
```

Try now, to print out the first 5 rows of the dataframe, and compare it to the previous example, before we converted the dates to datetime format. Also, we can see a change in the data type of the columns.

```
In [20]: weather_data["date"] = pd.to_datetime(weather_data["date"], format="%m/%d/%Y")
         weather_data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 365 entries, 0 to 364
Data columns (total 5 columns):
#   Column          Non-Null Count  Dtype
---  ---
0   date            365 non-null   datetime64[ns]
1   precipitation    365 non-null   float64
2   min_temp        365 non-null   float64
3   max_temp        365 non-null   float64
4   wind_speed      365 non-null   float64
dtypes: datetime64[ns](1), float64(4)
memory usage: 14.4 KB
```

Let's now go a step further, and learn to select certain parts of the data from our weather\_data dataframe.

## Selecting data

To select data from a DataFrame, we have several options, like i.e. using **square brackets []**, the **.loc[]** or the **.iloc[]** functions. There are certainly more possibilities, but those three should cover most of our needs.

### Square brackets

Using the square brackets, we can either select one (returns a Pandas Series) or multiple columns (returns a Pandas DataFrame). Let's say we are only interested in the precipitation data.

```
In [33]: weather_data["precipitation"]
Out[33]: 0      0.0
         1     13.7
         2      0.0
         3      0.4
         4      0.0
         ...
        360     0.0
        361     0.0
        362     0.0
        363     0.0
        364     0.1
        Name: precipitation, Length: 365, dtype: float64
```

Or as mentioned, we can also provide a **list** of column names inside the square brackets, which then returns a new DataFrame.

```
In [35]: weather_data[["min_temp", "max_temp"]]
Out[35]:
```

	min_temp	max_temp
0	5.2	11.2
1	0.2	6.9
2	-2.4	8.6
3	-1.5	7.1
4	5.3	13.8
...	...	...
360	-4.4	8.8
361	-2.2	6.4
362	-3.8	8.4
363	-2.5	5.5
364	-0.6	7.3

365 rows × 2 columns

It is also possible to select data by providing the indices to the square brackets, but this method is not very often used.

## **.loc[]**

The `.loc` method works in a bit different way. It's meant to select data by the index label, also, the data type depends on the index data type of the DataFrame on which we work on. We use it like follows:

```
In [53]: weather_data.loc[13]
Out[53]: date                2018-01-14 00:00:00
precipitation                1.2
min_temp                    -0.6
max_temp                     2.0
wind_speed                   5.9
Name: 13, dtype: object
```

It also works with DataFrames with other types of indices (string, datetimes, etc.).

Selecting only one index label, returns a Series. Selecting two label, or a subset, returns a new DataFrame.

```
In [54]: weather_data.loc[[13, 14]]
Out[54]:
```

	date	precipitation	min_temp	max_temp	wind_speed
13	2018-01-14	1.2	-0.6	2.0	5.9
14	2018-01-15	0.3	-2.4	-0.6	3.7

```
In [56]: weather_data.loc[13:19]
Out[56]:
```

	date	precipitation	min_temp	max_temp	wind_speed
13	2018-01-14	1.2	-0.6	2.0	5.9
14	2018-01-15	0.3	-2.4	-0.6	3.7
15	2018-01-16	0.1	-1.0	10.1	10.3
16	2018-01-17	4.9	0.0	8.5	2.4
17	2018-01-18	3.9	-4.1	7.8	8.2
18	2018-01-19	1.3	3.6	7.5	5.1
19	2018-01-20	9.3	-0.5	6.1	4.7

Please notice, that when providing multiple labels, we need to store them in a list (hence the double square brackets). Also, **.loc** includes the last values, opposite to indexing a list for example.

With **.loc** we can also select multiple rows and columns!

```
In [59]: weather_data.loc[[10, 11, 12], ["min_temp", "max_temp"]]
```

```
Out[59]:
```

	min_temp	max_temp
10	5.0	6.1
11	3.5	6.0
12	1.6	4.7

So, **.loc** can only be used with labels (integers, strings, datetimes...), it can select rows, but also columns and we can provide the selection as single label, subset or a list.

## **.iloc[]**

The **.iloc** method works very similar to the **.loc** method, but it only supports integer locations of the row. (whatever the data type the index of the DataFrame is, **.iloc** is selecting data by the integer location) Using only one value (index) will return a series, while using multiple indices or a subset will return a DataFrame.

```
In [62]: weather_data.iloc[4]
```

```
Out[62]: date                2018-01-05 00:00:00
precipitation                0.0
min_temp                     5.3
max_temp                     13.8
wind_speed                   5.3
Name: 4, dtype: object
```

```
In [64]: weather_data.iloc[4:9]
```

```
Out[64]:
```

	date	precipitation	min_temp	max_temp	wind_speed
4	2018-01-05	0.0	5.3	13.8	5.3
5	2018-01-06	0.0	8.4	15.8	3.7
6	2018-01-07	0.0	5.9	16.0	4.7
7	2018-01-08	0.0	4.1	6.6	1.1
8	2018-01-09	0.0	5.2	6.8	2.5

```
In [66]: weather_data.iloc[[7,8,9,10]]
```

```
Out[66]:
```

	date	precipitation	min_temp	max_temp	wind_speed
7	2018-01-08	0.0	4.1	6.6	1.1
8	2018-01-09	0.0	5.2	6.8	2.5
9	2018-01-10	6.0	4.4	7.7	1.9
10	2018-01-11	2.5	5.0	6.1	1.9

.iloc also can be used to select columns simultaneously, but in the same manner as for rows, by their integer location. Let's take a look.

```
In [68]: weather_data.iloc[[7,8,9,10], 1:3]
```

```
Out[68]:
```

	precipitation	min_temp
7	0.0	4.1
8	0.0	5.2
9	6.0	4.4
10	2.5	5.0

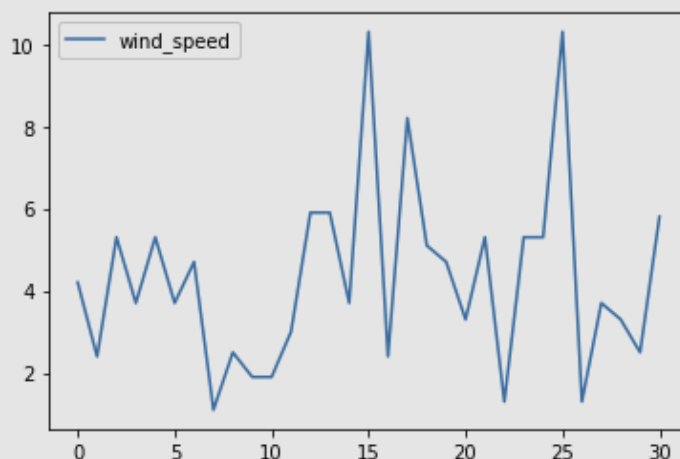
So in the above example we have selected the 7th, 8th, 9th and 10th rows, and 1st to 3rd column. (both the counting for rows and columns start from zero)

## Simple plots with Pandas

It's good to get the statistics for the data, and to know how to select certain parts of the DataFrame, but it's certainly nice to see some graphical interpretation of the dataset. So, as a final step of data exploration, let's take a look to simple plots with Pandas. For this example, let's plot out the wind speed for January 2018.

```
In [71]: weather_data.loc[0:30, ["wind_speed"]].plot()
```

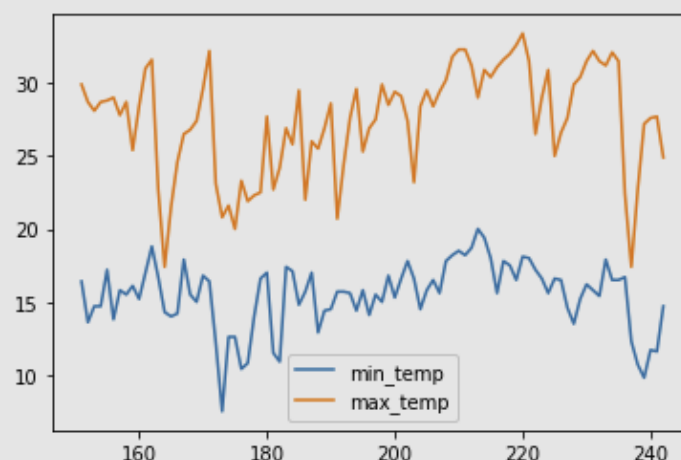
```
Out[71]: <AxesSubplot:>
```



We can also plot multiple columns, let's now see how warm (or cold) was the summer of 2018.

```
In [76]: weather_data.loc[151:242, ["min_temp", "max_temp"]].plot()
```

```
Out[76]: <AxesSubplot:>
```

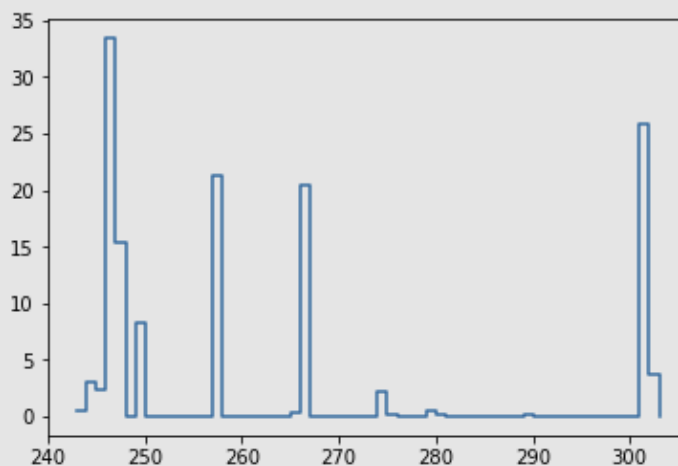


It is worth mentioning, that those plots, can be done by other ways of selecting data, can you think of any? Try it out, by yourself.

Precipitation is mostly plotted, as a bar plot. Since for this location, the Fall is in most cases pretty rainy, let's see how rainy it was 2018, by plotting the precipitation in September and October as a bar plot.

```
In [85]: weather_data.loc[243:303, "precipitation"].plot(drawstyle="steps-post")
```

```
Out[85]: <AxesSubplot:>
```



We are not plotting as a bar plot, but the "step" argument is a nice workaround to get nice barplot like plots, without messing around with the labels. 😊

## Chapter 8. - Matplotlib

Matplotlib is one of the most popular data visualization libraries in Python. It allows us to create figures and plots, and makes it very easy to produce static raster or vector files without the need for any GUIs.

### Installing Matplotlib

If you have [Anaconda](#), you can simply install Matplotlib from your terminal or command prompt using:

```
conda install matplotlib
```

If you do not have Anaconda on your computer, install Matplotlib from your terminal using:

```
pip install matplotlib
```

Now that you have Matplotlib installed, let's begin by understanding the anatomy of a plot.

We will begin by importing Matplotlib using:

```
import matplotlib.pyplot as plt
```

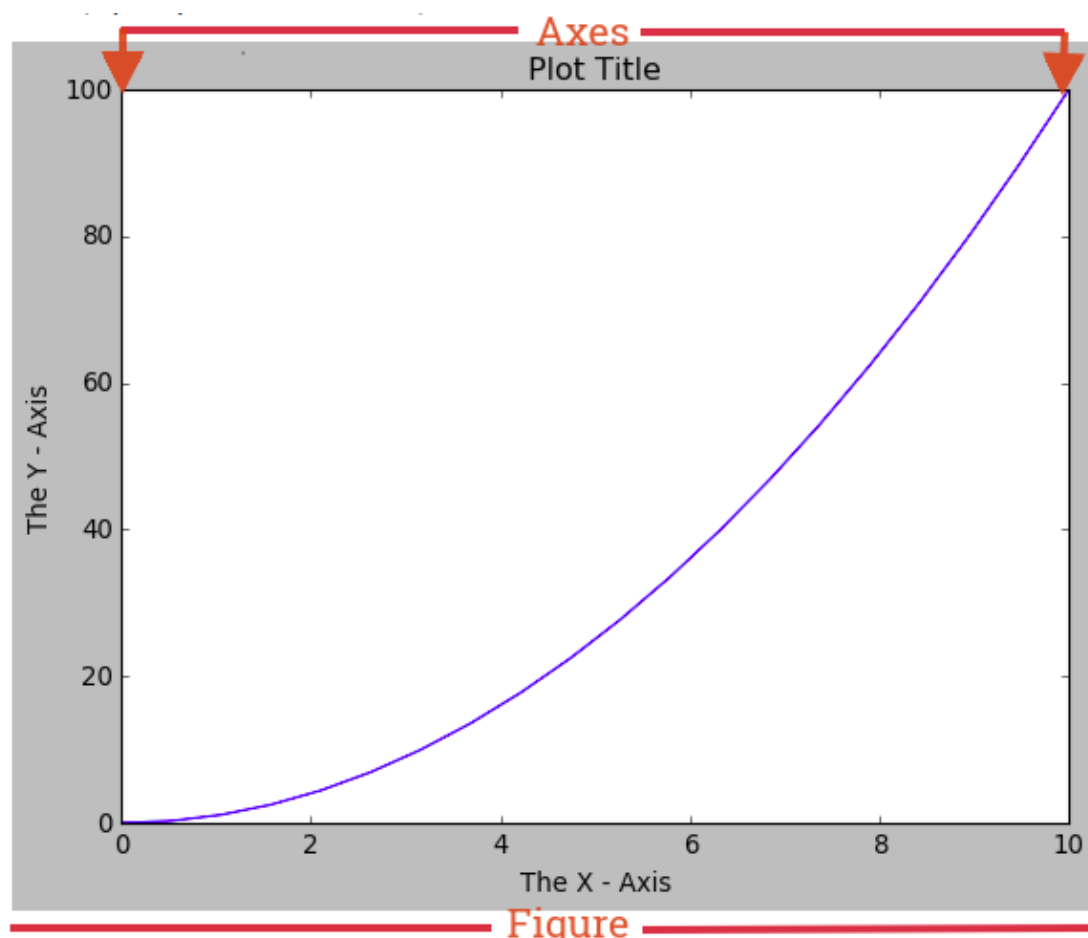
Now that we have Matplotlib imported, we need to be able to display the plots as it's being created. If you're using the Jupyter notebook we can easily display plots using:

```
%matplotlib inline
```

However, if you're using Matplotlib from within a Python script, you have to add **plt.show()** method inside the file to be able display your plot.

## Anatomy of a plot

There are two key components in a Plot; namely, Figure and Axes.





The Figure is the top-level container that acts as the window or page on which everything is drawn. It can contain multiple independent figures, multiple Axes, a subtitle (which is a centered title for the figure), a legend, a color bar, etc.

The **Axes** is the area on which we plot our data and any labels/ticks associated with it. Each **Axes** has an **X-Axis** and a **Y-Axis** (like in the image above).

## Functional approach to plotting

It's the more basic approach to plotting in Matplotlib. To see a simple plot, we will load a .csv file with some weather data.

```
weather_data = pd.read_csv("test_data.csv")
print (weather_data)
```

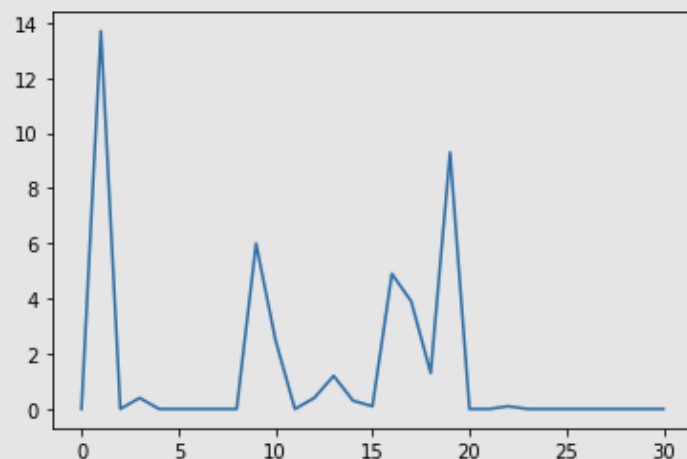
Our data contains daily precipitation, minimum and maximum air temperature and wind speed. For the sake of this example, we will print precipitation values for January 2018.

```
precipitation_data = weather_data[["precipitation"]][:31]
```

This means, we need to use slicing on the Pandas Dataframe to select the first 31 entries.

```
plt.plot(precipitation_data["precipitation"])
plt.plot()
```

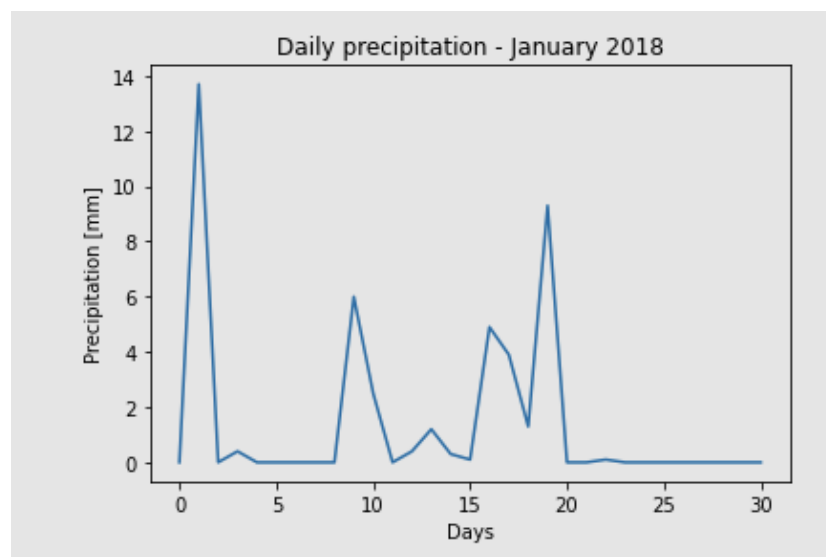
Out[14]: [



We select the desired column for the dataframe, in this case, *precipitation*. We could have just plot the entire dataframe, in this case, since we only had 1 column, the result would've remained the same.

Let's modify our plot a bit. We will add a title, and axis labels.

```
plt.plot(precipitation_data["precipitation"])
plt.title("Daily precipitation - January 2018")
plt.xlabel("Days")
plt.ylabel("Precipitation [mm]")
plt.plot()
```



The functional approach is often used when we need a fast and simple plot, for the sake of data visualization. But more often than not, we need specific, plots, with predefined resolution, sizes etc.

## Object oriented Interface

This is the best way to create plots. The idea here is to create Figure objects and call methods off it.

```
fig, ax = plt.subplots()
ax.plot(precipitation_data, label="Precipitation")
ax.plot()
```

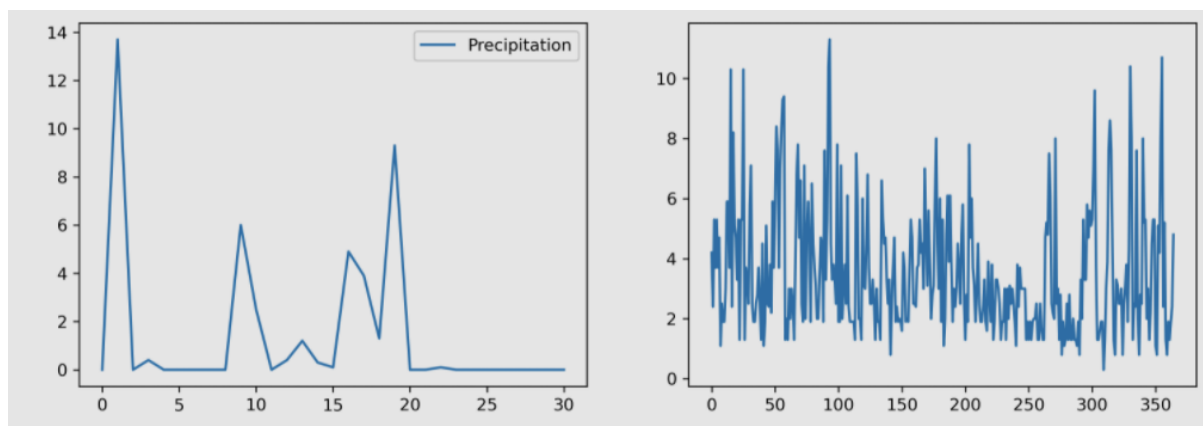
Fig defines the figure, where one (or more) axes are plotted. `.subplots()` method can take multiple arguments, for example we can define the size of figure, by specifying the **figure size**, aspect ratio, and **DPI** by simply specifying the **figsize**

and **dpi** arguments. The **figsize** is a tuple of the width and height of the figure (in **inches**), and **dpi** is the **dots-per-inch** (pixel-per-inch).

```
fig, ax = plt.subplots(figsize=(10,6), dpi = 300)
ax.plot(precipitation_data, label="Precipitation")
ax.plot()
```

Also, by using the **.subplots()** method, we can define the number of axes we need.

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,4), dpi = 300)
ax[0].plot(precipitation_data, label="Precipitation")
ax[1].plot(weather_data["wind_speed"])
ax[0].legend()
```

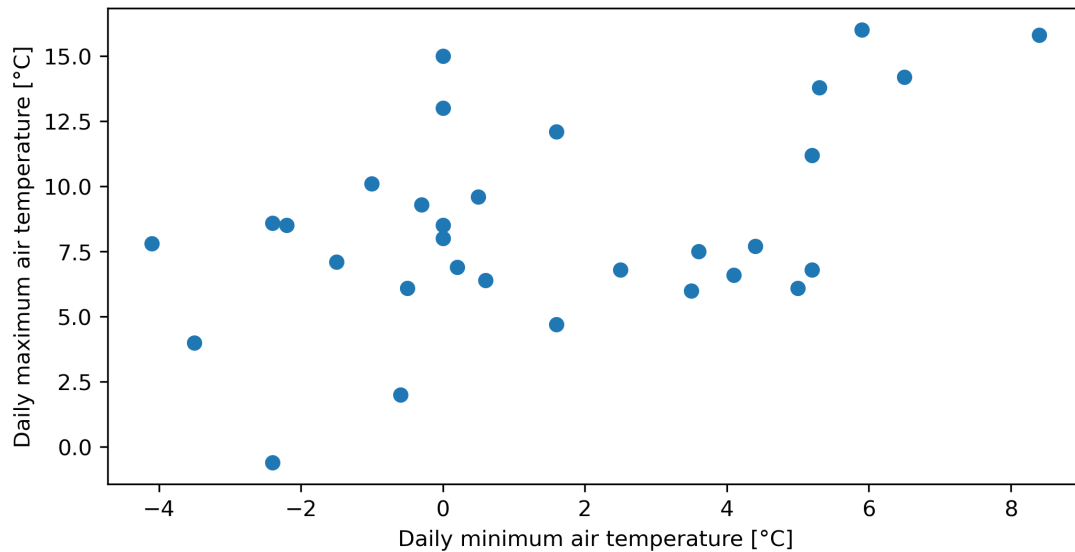


We have defined our figure to have 1 row, and 2 columns, meaning, we have now 2 axes. Also, we have introduced the **.legend()** method, which is used to display the legend on the desired axes. Before using the **.legend()** method, we need to provide the labels for the data we plot.

## Scatter plots

Scatterplots offer a convenient way to visualize how two numeric values are related in your data. It helps in understanding relationships between multiple variables. Using **.scatter()** method, we can create a scatter plot:

```
fig, ax = plt.subplots(figsize=(8,4), dpi = 300)
ax.scatter(weather_data["min_temp"][:31], weather_data["max_temp"][:31])
ax.set_xlabel("Daily minimum air temperature [°C]")
ax.set_ylabel("Daily maximum air temperature [°C]")
```



## Bar plots

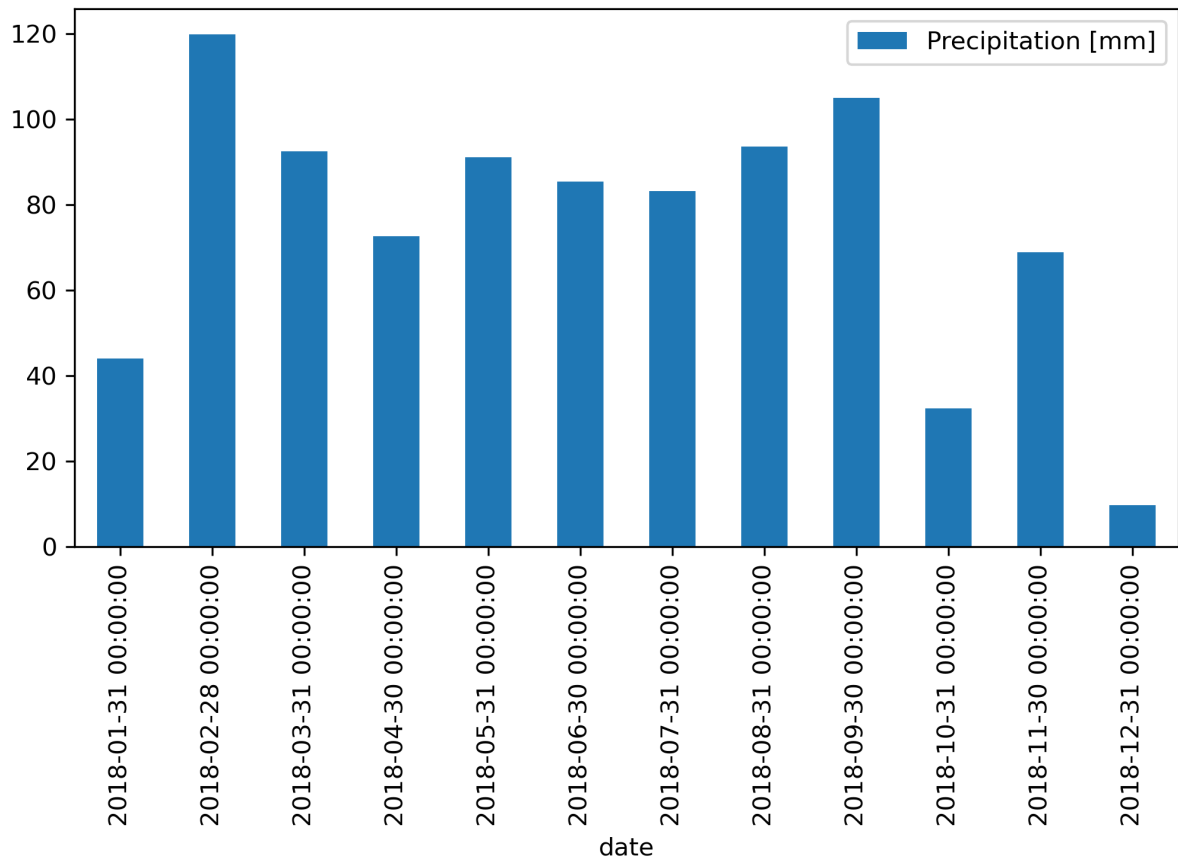
Bar graphs are convenient for comparing numeric values of several groups. First, we will summarize our precipitation data for the year of 2018, across months, then we will plot the monthly sums as a bar chart.

First, we need to resample our daily precipitation data, to monthly sums. To do so, we will first set the index of the dataframe as **DatetimeIndex**. This means, we will no longer have numerical indices (0, 1, 2, 3, ...), instead, we will have Dates. This, allows us to use the `.resample()` method from Pandas.

```
precip_data = weather_data[["date", "precipitation"]]
precip_data.index = pd.to_datetime(precip_data["date"])
precip_data = precip_data["precipitation"]
precip_monthly_sums = precip_data.resample("M").sum()
```

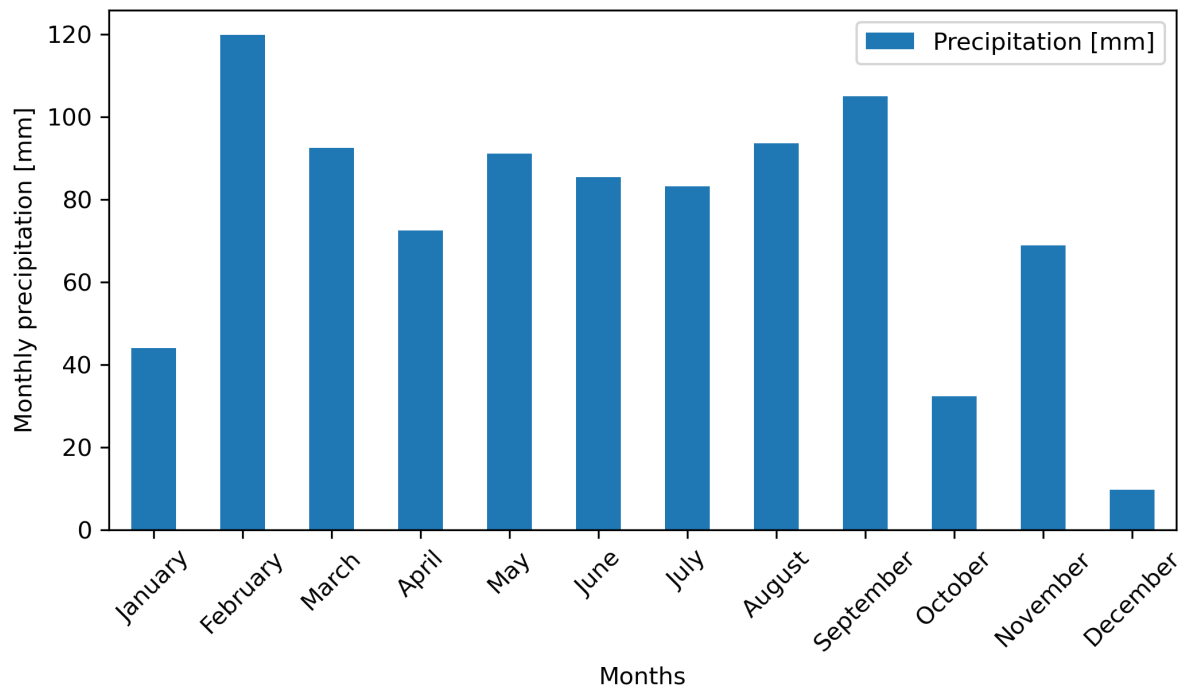
Using **plot.bar()** method, we can create a bar graph:

```
fig, ax = plt.subplots(figsize=(8,4), dpi = 300)
precip_monthly_sums.plot.bar(label="Precipitation [mm]")
ax.legend()
```



We can also modify the X-axis labels, a good practice for this kind of plot would be to display the months.

```
fig, ax = plt.subplots(figsize=(8,4), dpi = 300)
precip_monthly_sums.plot.bar(label="Precipitation [mm]")
ax.set_xticklabels([x.strftime("%B") for x in precip_monthly_sums.index], rotation=45)
ax.set_xlabel("Months")
ax.set_ylabel("Monthly precipitation [mm]")
ax.legend()
```



To alternate the Dates to display, we use **.strftime()** method, which converts our Date, to a String, according to the provided format. I.e. **"%B"**, to display is the formatter for long month names, **"%b"** would be for short month names. For more information regarding date formats take a look to the [official sites of Matplotlib](#). Here we use a list **comprehension**, to do this in one line. Another way to do this would have been to first create a list called months.

## Saving a plot

To save a plot from Matplotlib as a figure (i.e., .png or .jpeg) we use the **.savefig()** method.

```
fig, ax = plt.subplots(figsize=(8,4), dpi = 300)
precip_monthly_sums.plot.bar(label="Precipitation [mm]")
ax.set_xticklabels([x.strftime("%B") for x in precip_monthly_sums.index], rotation=45)
ax.set_xlabel("Months")
ax.set_ylabel("Monthly precipitation [mm]")
ax.legend()
plt.savefig("Precip_bar_plot_M.jpg", bbox_inches="tight")
```

The **.savefig()** method takes the filename keyword. Also, we added **bbox\_inches="tight"** argument, to prevent part of the figure to be missing.

## Chapter 9. - scikit.learn - Basics of Machine Learning

If we take a closer look to the scatter plot of MIN vs. MAX daily temperature, in last chapter, we can notice a somewhat linear correlation between the data. And, indeed, we know that with the rise of maximum daily air temperature, during spring and summer, also the minimum daily temperature rises. It certainly would be interesting to see if we can use Linear Regression to describe the daily maximum temperature according to the daily minimum temperature.

### What is Linear Regression, and why is it so popular?

Well, first and foremost it's a very **SIMPLE** algorithm. It attempts to describe a relationship between two variables with a straight line (a linear equation). If we again look at the above plot, we can see that maximum is the **dependent** variable, of the **explanatory** variable minimum temperature. Such a scatter plot is often used as first step before investigating a relationships between two variables. So, the approach assumes that every value of Y (maximum temperature) can be described as a linear function of X (minimum temperature) following the simple equation:

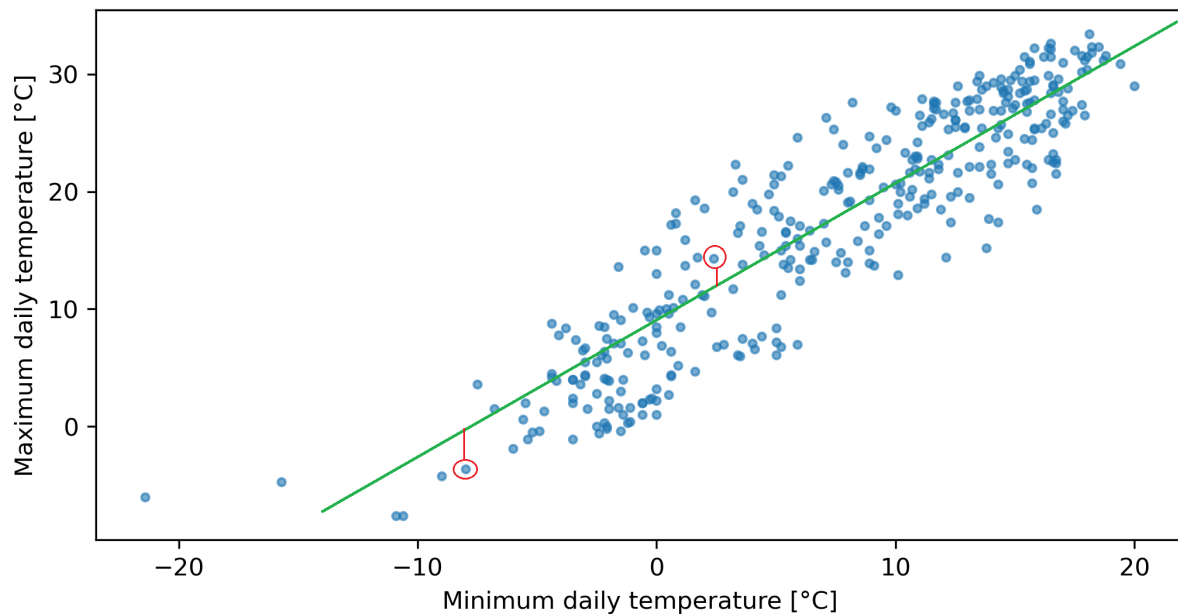
$$Y = w * X + b$$

Where **w** is the slope of the line, and **b** is the y-axis intercept. Or in Machine Learning terms, more often **w** is referred as **weight** and **b** for **bias**.

However, some assumption have to be taken into account:

1. **Linear relationship:** A linear relationship between x and y has to exist.
2. **Independence:** no correlation between consecutive residuals in time series data
3. **Homoscedasticity:** constant variance of residuals for every x
4. **Normality:** the residual are normally distributed

What are these so called residuals? Well, a residual is the value (or deviation) of the observed value from the fitted line. Basically, it is the distance (red line) of the blue point from the fitted line (green line).



## Least-squares error

By now, you probably ask yourself, how we calculate the  $\mathbf{w}$  and  $\mathbf{b}$  parameters? In simple terms, the algorithm calculates the "loss", the summarized values of all the residuals. Here the Least-squares error comes into play. It's a common approach where the squared difference from the observed point to the fitted line gets calculated, all these squared distances get summarized. The goal of the approach is to find the  $\mathbf{w}$  and  $\mathbf{b}$  that yield the **lowest possible sum of squared distances**. The sum of squared distances is often called the "**Mean Squared Error**" or **MSE**.

Let's see the example with air temperatures.

## Dataset loading and description

First, we will import the dependencies, load the test weather data data and take a look at the dataframe.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# we load the dataset
weather = pd.read_csv("test_data.csv")
weather.head()
```



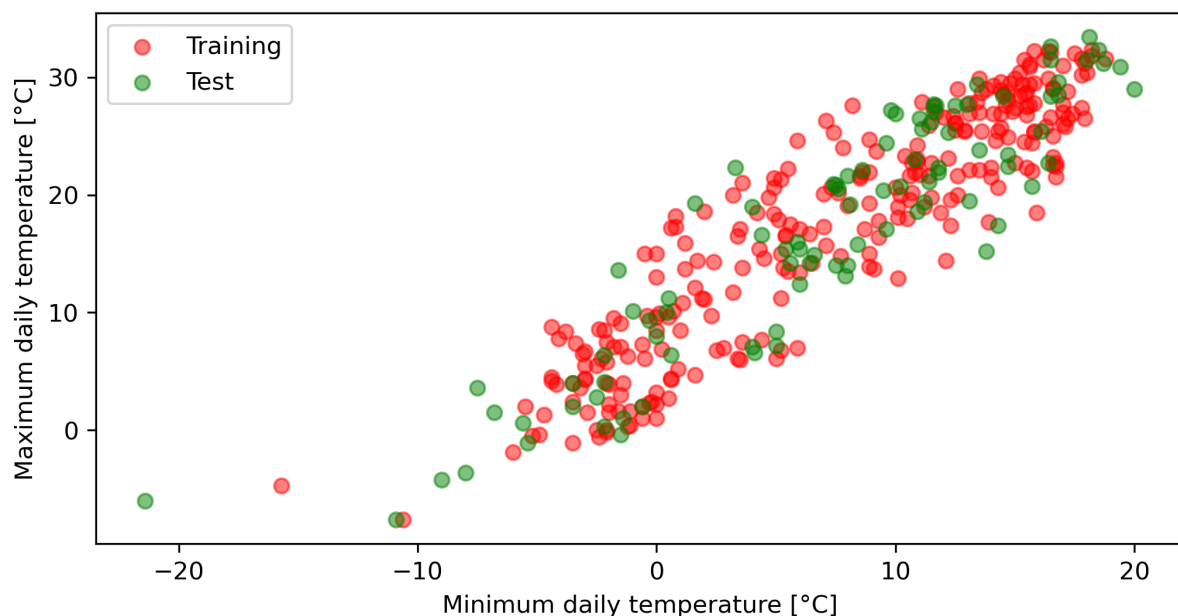
	date	precipitation	min_temp	max_temp	wind_speed
0	1/1/2018	0.0	5.2	11.2	4.2
1	1/2/2018	13.7	0.2	6.9	2.4
2	1/3/2018	0.0	-2.4	8.6	5.3
3	1/4/2018	0.4	-1.5	7.1	3.7
4	1/5/2018	0.0	5.3	13.8	5.3

First, we select and reshape the explanatory variable (min temp), due to Scikit-learn requirements of a 2D array, and since we only have 1 input feature, **.reshape(-1, 1)** is used. Then the data is splitted into training and test samples, using Scikit-learn's function `train_test_split`.

```
# split the data into training and test sets
# default is 75% / 25% train-test split

X = np.array(weather["min_temp"]).reshape(-1,1)
y = np.array(weather["max_temp"])
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

fig, ax = plt.subplots(figsize=(8,4), dpi = 300)
ax.scatter(X_train, y_train, label="Training", color = "r", alpha=0.5)
ax.scatter(X_test, y_test, label="Test", color = "g", alpha=0.5)
ax.set_xlabel("Minimum daily temperature [°C]")
ax.set_ylabel("Maximum daily temperature [°C]")
ax.legend()
plt.show()
```



In the next step we prepare our model. Therefore, the LinearRegression function is used, and fitted with the training data to train the model.

```
# Create the linear model
linreg = LinearRegression()

# Train the linear model
linreg.fit(X_train, y_train)
```

We can check the **w** and **b** parameters of the regression function by calling the **.coef\_** and **.intercept\_** attributes. Also, the R-squared score is calculated by calling the **.score()** method.

Next, let's check the regression parameters, training and test precision.

```
# Check the regression parameters, training and test precision
print('Linear model coeff (w): {}'.format(linreg.coef_))
print('Linear model intercept (b): {:.3f}'.format(linreg.intercept_))
print('R-squared score (training): {:.3f}'.format(linreg.score(X_train, y_train)))
print('R-squared score (test): {:.3f}'.format(linreg.score(X_test, y_test)))
```

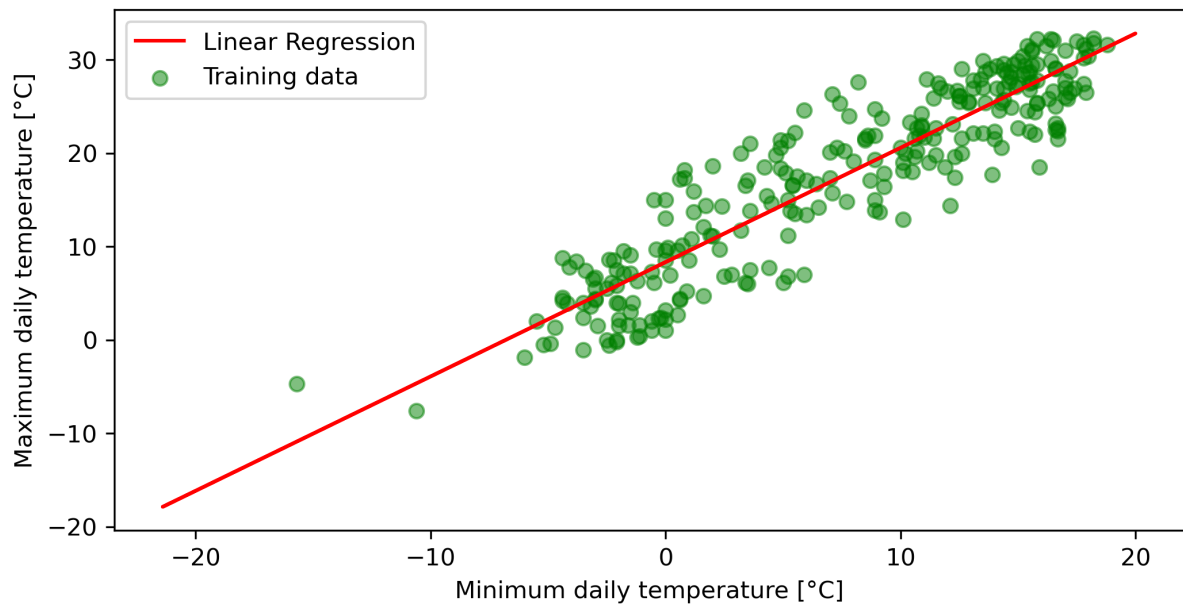
```
Linear model coeff (w): [1.22420566]
Linear model intercept (b): 8.328
R-squared score (training): 0.837
R-squared score (test): 0.836
```

According to our model if we wanted to calculate maximum air temperature from minimum temperature, we need to apply the following equation:

$$Y = 1.2242 * X + 8.328$$

And let us plot the resulting linear regression on both training and test data.

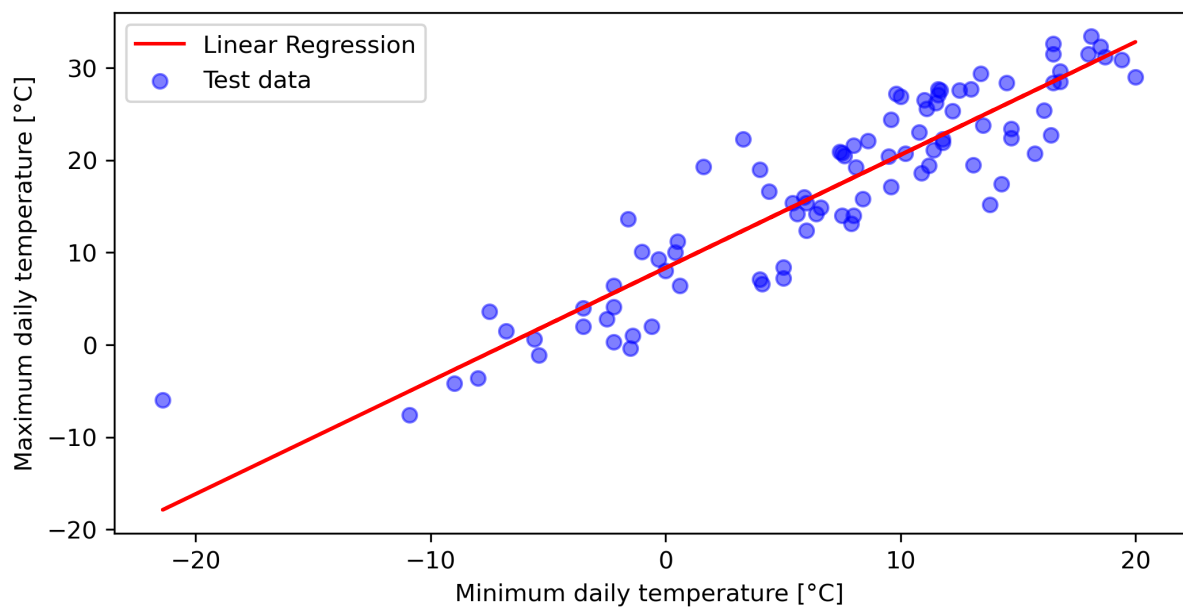
```
# Evaluation and plot of train data
fig, ax = plt.subplots(figsize=(8,4), dpi = 300)
ax.scatter(X_train, y_train, label="Training data", color = "g", alpha=0.5)
ax.plot(X, linreg.coef_ * X + linreg.intercept_, color = 'r', label = "Linear Regression")
ax.set_xlabel("Minimum daily temperature [°C]")
ax.set_ylabel("Maximum daily temperature [°C]")
ax.legend()
plt.show()
```



We calculate the predicted values according to test data by calling the **.predict** on the test sample. And then, we evaluate and plot the test sample.

```
# Evaluation and plot of test data
prediction = linreg.predict(X_test)

fig, ax = plt.subplots(figsize=(8,4), dpi = 300)
ax.scatter(X_test, y_test, label="Test data", color = "g", alpha = 0.5)
ax.plot(X_test, prediction, label = "Linear Regression", color = "r")
ax.set_xlabel("Minimum daily temperature [°C]")
ax.set_ylabel("Maximum daily temperature [°C]")
ax.legend()
plt.show()
```



Now try to fit a minimum today's minimum temperature, and calculate, today's maximum air temperature, and around 14:00 we shall see if our model is correct.



You just provide a numpy array with today's minimum temperature to **linreg.predict**.

```
linreg.predict([[today's_min_temperature]])
```

For temperatures around the middle of the distribution, our model should provide more accurate results, but if we go towards the extremes, the model accuracy slightly decreases.

## Conclusion

So, we are on the end of our training. I hope you had fun and liked the provided examples. This training should provide you a foundation on which you can continue your journey in programming. Hopefully I could show you that Data Science can be interesting and helpful in daily tasks.

