Introduction to Practical Applications

Aksel Hiorth

University of Stavanger

Dec 22, 2023

Contents

1 One reason Python is so popular: The import statement

It is always hard decide which coding language to learn, it usually depends on what you want to do. If you want to do very fast numerical calculations Fortan or C used to be the most popular languages, and for web or application programming Java. However, in recent years Python has become more and more popular, one of the reasons is its large amount of libraries and communities. If you have an idea of what you want to do, you can almost be certain that there exist a Python library written for that purpose. In the next chapters we will cover some advanced operations in Python and use them to motivate to learn more about the basic operations.

Which library to use?

This is not easy to answer, but here we will introduce you to the most popular libraries. We will suggest to stick to as few libraries as possible and try to achieve what you want with these.

2 Matplotlib: Basic plotting in Python

Visualizing data is a must, the code for plotting two arrays of data is

```
import matplotlib.pyplot as plt
x=[1,2,3,4]
y=[2,4,9,16] # y=x*x
plt.plot(x,y,label='y=x^2')
plt.legend() # try to remove and see what happens
plt.grid()   # try to remove and see what happens
```

Let us go through each line

1. `import matplotlib.pyplot as plt` this line tells Python to import the matplotlib.pyplot library. This library contains a lot of functions that other people have made. We use the `as` statement to indicate that we will name the `matplotlib.pyplot` library as `plt`. Thus we do not need to write `matplotlib.pyplot` every time we want to use a function in this library. We access functions in the library by simply placing a `.` after `plt`.

2. Next, we define two lists `x` and `y`, these lists have to be of equal length

3. The `plt.plot` commands plots `y` vs `x`
4. `plt.legend()` display the legend given inside the `plt.plot` command
5. `plt.grid()` adds grid lines which makes it easier to read the plot

By visiting the official documentation, and view the Matplotlib gallery, you will see a lot of examples on how to visualize your data.

3 Data structures (Basic)

The data we want to e.g. visualize has to be stored or passed around in the code somehow. Data structures provide an interface to your data, that makes it efficient to access them. In the next subsections we give a short overview of the most used data structures, which you should be familiar with.

3.1 Lists

Lists are defined using the square bracket `[]` symbol, e.g.

```
my_list = []      # an empty list
my_list = []*10   # still an empty list ...
my_list = [0]*10  # a list with 10 zeros
my_list = ['one', 'two','three'] # a list of strings
my_list = ['one']*10 # a list with 10 equal string elements
```

Notice.

To get the first element in a list, we do e.g. `my_list[0]`. Notice that the counter start at 0 and not 1. In a list with 10 elements the last element would be `my_list[9]`, the length of a list can be found by using the `len()` function, i.e. `len(my_list)` would give =10. Thus, the last element can also be found by doing `my_list[len(my_list)-1]`. However, in Python you can always get the last element by doing `my_list[-1]`, the second last element would be `my_list[-2]` and so on.

To add stuff to a list, we use the `append` function

```
my_list = []      # an empty list
my_list.append(2) # [2]
my_list.append('dog') # [2, 'dog']
```

You can also remove stuff, using the `pop` function, then you also have to give the index

```
my_list.pop(0) # my_list=['dog']
```

print statement.

As default python will usually write to screen your last statement. But at any time you can use the `print` statement to force python to print out any variable, e.g.

```
print(my_list[0]) # first element
print(my_list[0],my_list[1]) # first and second element
print(my_list) # the whole list
```

List comprehension. Sometimes you do not want to initialize the list with everything equal, and it can be tiresome to write everything out yourself. If that is the case you can use *list comprehension*

```
x = [i for i in range(10)] # a list from 0,1,2,..,9
y = [i**2 for i in range(10)] # a list with elements 0,1,4, ..,81
```

We will cover for loops later, but basically what is done is that the statement `i in range(10)`, gives `i` the value 0, 1, ..., 9 and the first `i` inside the list tells python to use that value as the element in the list. Using this syntax, there are plenty of opportunities to initialize.

Example: use list comprehension to make a plot of $y = x^3$.

- Create one list of $x \in [-3, 3]$ and the corresponding y values
- Use `matplotlib.pyplot` to create a plot.

```
#first we create the x-values
N=100 # 100 points
dx=6/N
x=[-3+i*dx for i in range(N)]
y=[i**3 for i in x]
plt.plot(x,y) # if you want you can add legend and grid lines
```

Solution.

3.2 Dictionaries

Dictionaries is useful if your data fits the template of key:value pairs. A very good mental image to have is an excel sheet where data are organized in columns. Each column has a header name, or a *key*. Assume we have the following table

x	y	z
1.0	1.0	3.0
2.0	4.0	
3.0	9.0	
4.0	16.0	

This could be represented as a dictionary as

```
my_dict={'x':[1.,2.,3.,4.],'y':[1.,4.,9.,16.],'z':[3.]}
```

The syntax is `{key1:values, key2:values2, ...}`. We access the values in the dictionary by the key i.e. `print(my_dict['x'])` would print `[1.,2.,3.,4.]`.

Example: Add numbers to dictionary and plot.

- Add two numbers to the list of `x` and `y` in `my_dict` and plot `y` vs `x`

```
import matplotlib.pyplot as plt
my_dict={'x':[1.,2.,3.,4.],'y':[1.,4.,9.,16.],'z':[3.]}
# add two numbers
my_dict['x'].append(-3)
my_dict['y'].append(-10)
plt.plot(my_dict['x'],my_dict['y'])
#alternatively only points
plt.plot(my_dict['x'],my_dict['y'],'*')
```

Solution.

3.3 Tuples

A tuple is a data structure that in many respects is equal to a list. It can contain various Python objects, *but the elements cannot be changed after the tuple has been created*. It is created by using round parenthesis, `()` as opposed to the square parenthesis, `[]`, used in list creation.

```
my_tuple=(1,2,3,4)
print(my_tuple[0]) # would give 1
print(my_tuple[-1]) # would give 4
my_tuple[0]=2 # would give an error message
```

Tuples vs Lists.

The obvious difference between tuples and lists is that you cannot change tuples after they have been created, in Python an object where you cannot change the state after it has been created is called *immutable* as opposed to *mutable*. Since tuples are immutable, you can also use them as keys in dictionaries. This can be useful if you need a lookup value that contains more information, e.g.

```
my_dict={}
my_dict[('perm','mD')]=[50,100,250]
```

4 Numpy: Working with numerical arrays in Python

In the above example we used lists to store values that we wanted to plot. Lists are one of the basic data structures in Python, but since they are so flexible they are not well suited for mathematical operations. If you are only working with arrays that contain numbers you should use the Numpy library. The above example in Numpy would be

```
import numpy as np
x=np.linspace(-3,3,100) # vector of 100 points from -3 to 3
y=x*x # multiply each number in x by itself
plt.plot(x,y)
```

Numpy has built in functions that allows you to calculate e.g. the logarithm, sine, exponential of arrays

```
np.log(x) # log of all elements in x
np.exp(x) # exp of all elements in x
np.sin(x) # sin of all elements in x
```

If you have a list, it can easily be converted to a Numpy array

```
x=[1,4,7] # x is a list
x+x # x+x would give [1,4,7,1,4,7]
x*x # would give an error message
x=np.array([1,4,7]) # now x is a Numpy array
x+x # would give [2,8,14]
x*x # would give [1,16,49]
x/x # would give [1.,1.,1.]
```

5 Boolean masking

In Python we also have a Boolean type, which is `True` or `False` (note the big letters), it takes up the smallest amount of memory (one byte). It is a subclass of integer, `int`, and if you add or subtract them, `True` and `False` will be given the value of 1 and 0 respectively. In many applications you would like to pick out only a part of the elements of an array. If we work with Numpy arrays, it is extremely easy achieve this using Boolean masking or Boolean indexing. We use the word "mask" to indicate which bits we want to keep, and which we want to remove. It is best demonstrated on some examples (remember: it will not work on lists)

```
x=np.array([4,5,7,8,9]) #create numpy array from a list
print(x>5) # [False, False, True, True, True]
print(np.sum(x>5)) # 3
x[x>5] # [7,8,9]
```

6 Pathlib: Working with files and folders in Python

When you want to open a file in a Python script, you first have to locate it. If you have downloaded all the files, there will be a `data` folder in which there are several data sets. Sometimes we would like to list all files in a folder, or files of a certain type, and maybe create a unique file name that does not already exists.

To access files one can use strings, but in practice this can be very tiresome, especially as a path in Windows has a different syntax than e.g. Linux. In Windows directories and sub directories are indicated with a backslash, `\`, whereas in Linux it is a forward slash `/`.

It is much better to use the Pathlib library and work with *Path objects*, then your code would work regardless of operating system.

6.1 Pathlib cwd(): Current working directory

How can you know which directory your are currently in? Using Pathlib we can do

```
import pathlib as pt
p=pt.Path('.') # we create a Path object
print(p.cwd())
```

What happens here?

1. `import pathlib as pt` imports Pathlib which we name `pt`. Functions in Pathlib is accessed with the `.` syntax.
2. We create a Path object, by `p=pt.Path('.')` and store it in the variable `p`. The `'.'` argument is the current directory.

3. Now we have access to build-in functions int the Path object by using the `.` syntax, and we can print current working directory `print(p.cwd())`

6.2 Our first for-loop: List all files and folder in current directory

Here we will encounter or first for-loop, a for-loop is a way to tell the computer to do something until a certain condition has been met. In Python it is very common to combine a for-loop with a the `in` command. Let say we have a list of strings: `my_list=['dog','cat','rock']` if we want to print out everything in this list, we can do

```
my_list=['dog','cat','rock']
for x in my_list:
    print(x)
```

The variable `x` will then recursively take the value `dog`, `cat`, and `rock`.

Meaningful variable names.

The specific name we give `x` (or `my_list`), is not important for the computer. But if you choose a more descriptive name, it makes the code easier to read for humans, e.g.

```
words=['dog','cat','rock']
for word in words:
    print(word)
```

Indentation matters!

The for-loop above ends with `:`, and then Python uses indentation to indicate a block of code. *You have to use the same amount of spaces in the same block of code.* The following code will give an *error*.

```
my_list=['dog','cat','rock']
for word in my_list:
print(word) #Error because no intendation
```

Anyway, here is the code for listing all files and directories in a folder

```
import pathlib as pt
p=pt.Path('.') # the directory where this python file is located
for x in p.iterdir():
    if x.is_dir():#NB Indentation, all below belongs to p.iterdir()
        print('Found dir: ', x) #NB Indentation, belongs to if x.is_dir()
    elif x.is_file():
        print('Found file: ', x) #NB Indentation, belongs to if x.is_file()
```

`p.iterdir()` is a *generator*, and for now you can simply think of it as generating a list of all files and folders in the `'.'` (current) directory. To view the elements in `p.iterdir()`, you can do

```
print(list(p.iterdir()))
```

Let us go through each line

1. `p=pt.Path('.')` we create a Path object and name it `p`.
2. The next part is the for-loop, the variable `x` takes on the value of each element in the list generated by `p.iter_dir()`.
3. `x.is_dir()`, gives True if `x` is a directory and False if not.
4. `x.is_file()`, gives True if `x` is a file and False if not.

List all files of a type: Below we use `p.rglob()`, and not `p.iterdir()`, the difference is that `rglob()` also lists recursively the sub directories, and files within.

```
p=Path('.')
for p in p.rglob("*.xlsx"):# rglob means recursively, searches sub directories
    print(p.name)
```

If you want to print the full path do `print(p.absolute())`.

```
Path('tmp_dir').mkdir()
```

Create a directory, files and joining paths:

If you run the code twice it will produce an error, because the directory exists, then we can simply do `Path('tmp_dir').mkdir(exist_ok=True)`.

Furthermore, the forward slash, `/`, can be used combine paths

```
p=Path('.')
new_path = p / 'tmp_dir' / 'my_file.txt'
print(new_path.absolute())
print(p.exists()) # gives False, because my_file.txt does not exists
new_path.touch() # touch creates file, tmp_dir must exists
print(p.exists()) # gives True
```

7 Pandas: Working with tabulated data (Excel files)

Pandas is a Python package that among many things are used to handle data, and perform operations on groups of data. It is built on top of Numpy, which makes it easy to perform vectorized operations. Pandas is written by Wes McKinney, and one of it objectives is according to the official website " providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python". Pandas also has excellent functions for reading and writing excel and csv files. An excel file is read directly into memory in what is called a `DataFrame` in Pandas. A DataFrame is a two dimensional object where data are typically stored in column or row format.

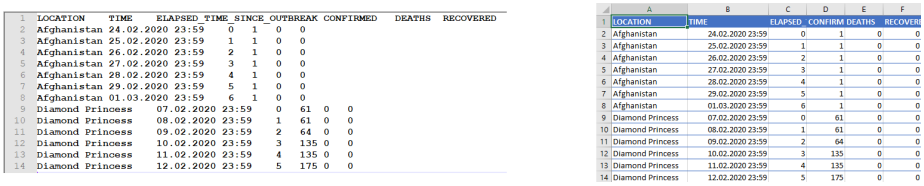
Pandas has a lot of functionality.

Pandas has so much functionality that it is almost like a programming language. Here we will only use it to read and write excel files, and do some basic filtering of data. However, there is a high probability that whatever you would like to do with your data, e.g. clean, filter, mathematical operations, there is already a Pandas command to achieve your goal.

7.1 DataFrame: The basic object in Pandas

What is a DataFrame?

You should think of a DataFrame as a single sheet in Excel with tabulated data. A DataFrame will typically have data stored with a header name and data in an array associated with that header, as illustrated in 1.



LOCATION	TIME	ELAPSED_TIME_SINCE_OUTBREAK	CONFIRMED	DEATHS	RECOVERED
Afghanistan	24.02.2020 23:59	0	1	0	0
Afghanistan	25.02.2020 23:59	1	1	0	0
Afghanistan	26.02.2020 23:59	2	1	0	0
Afghanistan	27.02.2020 23:59	3	1	0	0
Afghanistan	28.02.2020 23:59	4	1	0	0
Afghanistan	29.02.2020 23:59	5	1	0	0
Afghanistan	01.03.2020 23:59	6	1	0	0
Diamond Princess	07.02.2020 23:59	0	61	0	0
Diamond Princess	08.02.2020 23:59	1	61	0	0
Diamond Princess	09.02.2020 23:59	2	64	0	0
Diamond Princess	10.02.2020 23:59	3	135	0	0
Diamond Princess	11.02.2020 23:59	4	135	0	0
Diamond Princess	12.02.2020 23:59	5	175	0	0

	A	B	C	D	E	F
1	LOCATION	TIME	ELAPSED	CONFIRM	DEATHS	RECOVERI
2	Afghanistan	24.02.2020 23:59	0	1	0	0
3	Afghanistan	25.02.2020 23:59	1	1	0	0
4	Afghanistan	26.02.2020 23:59	2	1	0	0
5	Afghanistan	27.02.2020 23:59	3	1	0	0
6	Afghanistan	28.02.2020 23:59	4	1	0	0
7	Afghanistan	29.02.2020 23:59	5	1	0	0
8	Afghanistan	01.03.2020 23:59	6	1	0	0
9	Diamond Princess	07.02.2020 23:59	0	61	0	0
10	Diamond Princess	08.02.2020 23:59	1	61	0	0
11	Diamond Princess	09.02.2020 23:59	2	64	0	0
12	Diamond Princess	10.02.2020 23:59	3	135	0	0
13	Diamond Princess	11.02.2020 23:59	4	135	0	0
14	Diamond Princess	12.02.2020 23:59	5	175	0	0

Figure 1: Official Covid-19 data, and example of files (left) tab separated (right) excel file.

If we have file in the `data` directory, we can import them into a DataFrame as follows

```
df=pd.read_excel('../data/corona_data.xlsx') # excel file
df2=pd.read_csv('../data/corona_data.dat',sep='\t') # csv tab separated file
```

If the excel file has several sheets, you can give the sheet name directly, e.g. `df=pd.read_excel('file.xlsx',sheet_name="Sheet1")`, for more information see the documentation.

We can easily save the data frame to excel format and open it in excel

```
df.to_excel('covid19.xlsx', index=False) # what happens if you put index=True?
```

Index column.

Whenever you create a DataFrame Pandas by default create an index column, it contains an integer for each row starting at zero. It can be accessed by `df.index`, and it is also possible to define another column as index column.

7.2 Create DataFrame from dictionary

A DataFrame can be quite easily be generated from a dictionary. A dictionary is a special data structure, where an unique key is associated with a data type (key:value pair). In this case, the key would be the title of the column, and the value would be the data in the columns.

```
my_dict={'ints':[0,1,2,3], 'floats':[4.,5.,6.,7.],
'tools':['hammer','saw','rock','nail']
}
df=pd.DataFrame(my_dict)
print(df) # to view
```

7.3 Accessing data in DataFrames

Selecting columns. If we want to pick out a specific column we can access it in the following ways

```
# following two are equivalent
time=df['TIME'] # by the name, alternatively
time=df[df.columns[1]]
# following two are equivalent
time=df.loc[:,['TIME']] # by loc[] if we use name
time=df.iloc[:,1] # by iloc, pick column number 1
```

The `loc[]` and `iloc[]` functions also allows for list slicing, one can then pick e.g. every second element in the column by `time=df.iloc[::2,1]` etc.

The difference is that `loc[]` uses the name, and `iloc[]` the index (usually an integer).

Why several ways of doing the same operation? It turns out that although we are able to extract what we want with these operations, they are of different type

```
print(type(df['TIME']))
print(type(df.loc[:,['TIME']]))
```

Selecting rows. When selecting rows in a DataFrame, we can use the `loc[]` and `iloc[]` functions

```
# pick column number 0 and 1
time=df.loc[0:1,:] # by loc[]
time=df.iloc[0:2,:] # by iloc
```

`pandas.DataFrame.loc` vs `pandas.DataFrame.iloc`.

When selecting rows `loc` and `iloc` they behave differently, `loc` includes the endpoints (in the example above both row 0 and 1), whereas `iloc` includes the starting point and up to 1 minus the endpoint.

Challenges when accessing columns or rows.

Special characters.

Sometimes when reading files from excel, headers may contains invisible characters like newline `\n` or tab `\t` or maybe Norwegian special letters that have not been read in properly. If you have problem accessing a column by name do `print(df.columns)` and check if the name matches what you would expect.

If the header names has unwanted white space, one can do

```
df.columns = df.columns.str.replace(' ', '') # all white spaces
df.columns = df.columns.str.lstrip() # the beginning of string
df.columns = df.columns.str.rstrip() # end of string
df.columns = df.columns.str.strip()  # both ends
```

Similarly for unwanted tabs

```
df.columns = df.columns.str.replace('\t', '') # remove tab
```

If you want to make sure that the columns does not contain any white spaces, one can use `pandas.Series.str.strip()`

```
df['LOCATION']=df['LOCATION'].str.strip()
```

7.4 Datetime: Time columns not parsed properly

If you have dates in the file (as in our case for the `TIME` column), you should check if they are in the `datetime` format and not read as `str`.

`datetime`.

The `datetime` library is very useful for working with dates. Data types of the type `datetime` (or equivalently `timestamp` used by Pandas) contains both date and time in the format `YYYY-MM-DD hh:mm:ss`. We can initialize a variable, `a`, by `a=datetime.datetime(2022,8,30,10,14,1)`, to access the hour we do `a.hour`, the year by `a.year` etc. It also easy to increase e.g. the day by one by doing `a+datetime.timedelta(days=1)`.

```
import datetime as dt
time=df['TIME']
# what happens if you set
# time=df2['TIME'] #i.e df2 is from pd.read_csv ?
print(time[0])
print(time[0]+dt.timedelta(days=1))
```

The code above might work fine or in some cases a date is parsed as a string by Pandas, then we need to convert that column to the correct format. If not, we get into problems if you want to plot data vs the time column.

Below are two ways of converting the `TIME` column

```
df2['TIME']=pd.to_datetime(df2['TIME'])
# just for testing that everything went ok
time=df2['TIME']
print(time[0])
print(time[0]+dt.timedelta(days=1))
```

Another possibility is to do the conversion when reading the data:

```
df2=pd.read_csv('../data/corona_data.dat',sep='\t',parse_dates=['TIME'])
```

If you have a need to specify all data types, to avoid potential problems down the line this can also be done. First create a dictionary, with column names and data types

```
types_dict={"LOCATION":str,"TIME":str,"ELAPSED_TIME_SINCE_OUTBREAK":int,"CONFIRMED":int,"DEATHS":in
df2=pd.read_csv('../data/corona_data.dat',sep='\t',dtype=types_dict,parse_dates=['TIME']) # set dat
```

Note that the time data type is `str`, but we explicitly tell Pandas to convert those to `datetime`.

7.5 Pandas: Filtering and visualizing data

Boolean masking. Typically you would select rows based on a criterion, the syntax in Pandas is that you enter a series containing `True` and `False` for the rows you want to pick out, e.g. to pick out all entries with Afghanistan we can do

```
df[df['LOCATION'] == 'Afghanistan']
```

The innermost statement `df['LOCATION'] == 'Afghanistan'` gives a logical vector with the value `True` for the five last elements and `False` for the rest. Then we pass this to the DataFrame, and in one go the unwanted elements are removed. It is also possible to use several criteria, e.g. only extracting data after a specific time

```
df[(df['LOCATION'] == 'Afghanistan') & (df['ELAPSED_TIME_SINCE_OUTBREAK'] > 2)]
```

Note that the parenthesis are necessary, otherwise the logical operation would fail.

Plotting a DataFrame. Pandas has built in plotting, by calling `pandas.DataFrame.plot`.

```
df2=df[(df['LOCATION'] == 'Afghanistan')]
df2.plot()
#try
#df2=df2.set_index('TIME')
#df2.plot() # what is the difference?
#df2.plot(y=['CONFIRMED','DEATHS'])
```

7.6 Performing mathematical operations on DataFrames

When performing mathematical operations on DataFrames there are at least two strategies

- Extract columns from the DataFrame and perform mathematical operations on the columns using Numpy, leaving the original DataFrame intact
- To operate directly on the data in the DataFrame using the Pandas library

Speed and performance.

Using Pandas or Numpy should in principle be equally fast. The advice is to not worry about performance before it is necessary. Use the methods you are confident with, and try to be consistent. By consistent, we mean that if you have found one way of doing a certain operation stick to that one and try not to implement many different ways of doing the same thing.

We can always access the individual columns in a DataFrame by the syntax `df['column_name']`.

Example: mathematical operations on DataFrames.

1. Create a DataFrame with one column (`a`) containing ten thousand random uniformly distributed numbers between 0 and 1 (checkout `np.random.uniform`)
2. Add two new columns: one which all elements of `a` is squared and one where the sine function is applied to column `a`
3. Calculate the inverse of all the numbers in the DataFrame
4. Make a plot of the results (i.e. `a` vs `a*a`, and `a` vs `sin(a)`)

Solution.

1. First we make the DataFrame

```
import numpy as np
import pandas as pd
N=10000
a=np.random.uniform(0,1,size=N)
df=pd.DataFrame() # empty DataFrame
df['a']=a
```

If you like you could also try to use a dictionary. Next, we add the new columns

```
df['b']=df['a']*df['a'] # alternatively np.square(df['a'])
df['c']=np.sin(df['a'])
```

1. The inverse of all the numbers in the DataFrame can be calculated by simply doing

```
1/df
```

Note: you can also do `df+df` and many other operations on the whole DataFrame.

1. To make plots there are several possibilities. Personally, I tend most of the time to use the `matplotlib` library, simply because I know it quite well, but Pandas has a great deal of very simple methods you can use to generate nice plots with very few commands.

```
import matplotlib.pyplot as plt
plt.plot(df['a'],df['b'], '*', label='$a^2$')
plt.plot(df['a'],df['c'], '^', label='$\sin(a)$')
plt.legend()
plt.grid() # make small grid lines
plt.show()
```

Matplotlib:

Pandas plotting: First, let us try the built in plot command in Pandas

```
df.plot()
```

If you compare this plot with the previous plot, you will see that Pandas plots all columns versus the index columns, which is not what we want. But, we can set `a` to be the index column

```
df=df.set_index('a')
df.plot()
```

We can also make separate plots

```
df.plot(subplots=True)
```

or scatter plots

```
df=df.reset_index()
df.plot.scatter(x='a',y='b')
df.plot.scatter(x='a',y='c')
```

Note that we have to reset the index, otherwise there are no column named `a`.

7.7 Grouping, filtering and aggregating data

Whenever you have a data set, you would like to do some exploratory analysis. That typically means that you would like to group, filter or aggregate data. Perhaps, we would like to plot the covid data not per country, but the data as a function of dates. Then you first must sort the data according to date, and then sum all the occurrences on that particular date. For all of these purposes we can use the `pd.DataFrame.groupby()` function. To sort our DataFrame on dates and sum the occurrences we can do

```
df.groupby('TIME').sum()
```

Another case could be that we wanted to find the total number of confirmed, deaths and recovered cases in the full database. As always in Python it can be done in different ways, by e.g. splitting the database into individual countries and do `df[['CONFIRMED','DEATHS','RECOVERED']].sum()` or accessing each column individually and sum each of them e.g. `np.sum(df['CONFIRMED'])`. However, with the `groupby()` function (see figure 2 for final result)

```
df.groupby('LOCATION').sum()
```

Here Pandas sum all columns with the same location, and drop columns that cannot be summed. By doing `df.groupby('LOCATION').mean()` or `df.groupby('LOCATION').std()` we can find the mean or standard deviation (per day).

LOCATION	ELAPSED_TIME_SINCE_OUTBREAK	CONFIRMED	DEATHS	RECOVERED
Afghanistan	21	7	0	0
Diamond Princess	15	631	0	0

Figure 2: The results of `df.groupby('LOCATION').sum()`.

References

Index