E-MOD321 Basic Python coding for subsurface applications

Aksel Hiorth, University of Stavanger

Jan 6, 2024

(Work in progress) Material prepared for MOD321 at University of Stavanger. The documents have been prepared by the use of doconce[a].

[a] `https://github.com/doconce/doconce`

Contents

Preliminaries 1

1.1 Is it possible to learn Python in two days?

You will for sure not master Python in two days, but you will be able to perform many useful tasks, and lay the foundation for further development. In particular with the release of ChatGPT[1], I would say it has never been so easy to get advanced applications up and running with only a basic understanding of Python. If you manage to formulate what your task precisely, then ChatGPT will translate your request into code (see e.g. figure 1.1). This is clearly not foolproof, the code you get back might not work exactly as you want and you will need to modify it, or perhaps you have to break your task into several smaller pieces to get useful answers from ChatGPT (or the web) and then you are left with the task of gluing them together yourself.

1.2 Why should you learn coding?

A quick google search will tell you that you should learn to code because it will lead to job opportunities and boost your career. I would also highlight that it will let you test out ideas much more efficiently. When you have a lot of domain knowledge in a certain area, you will most likely have ideas that can lead to innovation or new insight. As an example, maybe you have made some observation indicating that two phenomena originally

[1] `https://chat.openai.com/auth/login`

believed to be unrelated, actually are related. To prove or support you claim, you would then need to collect data from these phenomena and present them together. Since the phenomena are unrelated the data are most likely located in different places. With some basic knowledge of Python you can easily access different files, folders, web pages, scrap data from them, filter the data, and join them. Once you have collected the data you would then make some plots, and inspect the plots to discover patterns. The next steps is to quantify correlations by e.g. regression analysis or to use more advanced machine learning techniques. All this can be achieved with Python, it will take you time to master this fully, but with basic Python knowledge you can easily run through tutorials yourself and become quite advanced within weeks.

1.3 About this course

With the development of tools that can write code for you and also the large number of libraries in Python, it becomes less important to learn syntax. Rather, you should focus on learning the basic concepts, and to learn the logic of coding. For the examples presented in this course, try to focus on

1. What kind of task do we want to perform?
2. How can this task be broken down into smaller pieces?
3. How are these smaller pieces implemented in order for the computer to understand us?

It is very important to develop an understanding on how to break a big problem into smaller tasks. In the beginning you will copy what others have done, but over time you will develop your own personal style and how to do things.

In this course we start with practical applications and gradually move to basic operations, because we hope it will be more engaging. Once we have achieved our goal or the task, we will explain the logic, and investigate line by line what is happening.

This has the consequence that we will introduce basic programming concepts such as types, lists, dictionaries *when it is needed, and only the minimal amount of information*. The challenge with this approach is that there is always more to learn about the basic programming concepts, thus if you feel that you would like to know more about the different

concepts you should explore this on your own. See the next section for where to find resources.

1.4 Online programming resources

This course is supposed to be self contained, but there are of course plenty of online courses, youtube videos, and books that you should take advantage of to improve your understanding. These resources are extremely valuable if you know exactly what your are looking for. As a complete beginner with little or no knowledge of Python it can be confusing if you do not know what you are looking for. Great online sources that cover much of Python basics are

- w3schools[2], brief description of different functionality in Python, has an extensive index, which makes it is easy to look up different concepts.
- A Whirlwind tour of Python[3]. Basic introduction to Python, from simple to more complex concepts.
- Automate the boring stuff[4], another comprehensive low level, introduction to Python.

These resources explains quite briefly important concepts and give examples, such as

- specific Python syntax,
- data types (float, int, Boolean, etc.),
- data structures (lists, dictionaries, tuples, etc.),
- control flow (if, else, while, for loops etc.),
- functions and classes.

I would also like to highlight Real Python[5]. Real Python is comprehensive, it offers different learning paths from python basics[6], to machine learning with python[7]. Whenever I want to understand certain Python concepts in depth, I often end up at Real Python, I find it to be precise and not too lengthy.

[2] `https://www.w3schools.com/python/default.asp`
[3] `https://jakevdp.github.io/WhirlwindTourOfPython/`
[4] `https://automatetheboringstuff.com/`
[5] `https://realpython.com/`
[6] `https://realpython.com/learning-paths/python-basics/`
[7] `https://realpython.com/learning-paths/machine-learning-python/`

1.4.1 ChatGPT

ChatGPT[8], developed by OpenAI[9] is perhaps one of the best online sources to help you write code. So far all the examples in this course can be generated from ChatGPT. Just type in "Show me how to plot sin(x) in Python", you will get the output in figure 1.1.

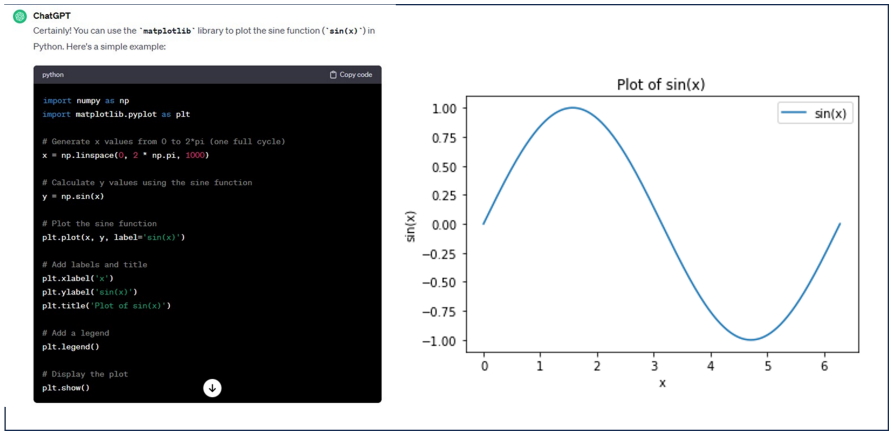

Fig. 1.1 Output from ChatGPT and the result after running the code.

I would encourage you to use ChatGPT actively in your coding, you will be more efficient. The code generated is generally good, and if there are parts you do not understand it is possible to get additional help from ChatGPT.

1.5 Stuff you need to do

1. You need to install Python, even if you have installed Python before we recommend you to install the Anaconda distribution[10]. It is straight forward to install, just follow the instructions and choose default options that are suggested.
2. Install an integrated development environment (IDE). An IDE is simply where you write the Python code. After installing Anaconda you should already have Spyder installed, if not you can install it

[8] `https://chat.openai.com/auth/login`
[9] `https://openai.com/`
[10] `https://www.anaconda.com/download`

by opening the Anaconda Navigator. You will find the Anaconda Navigator in the start menu in the Anaconda folder, but most likely there will already be a program called Spyder in your program folder. Another IDE is Visual Studio Code[11] or VS Code for short, see figure 1.2 for two examples. An IDE will help you to write code, because it will give information about the code you write and also help you to find errors.

3. Sign up for an account for ChatGPT[12]. This is not mandatory, but it will help you write code faster.

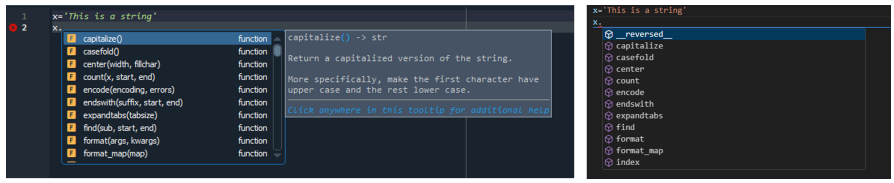

Fig. 1.2 Two IDEs to write Python code (left) Spyder (right) VS Code.

1.6 If you never have coded before

Here I have collected stuff that will make your life easier, and increase the speed of understanding. I have tried to explain some concepts below, if this is too little information there are plenty of online resources that you can check out. The purpose of this section is to introduce you to some concepts that are key to any programming language, but can take some time to master or to get under your skin. If you understand these concepts, coding will be easier. Do not focus on how we use these concepts in coding, that is what the course is all about, rather try to understand the meaning of the concepts.

1.6.1 Variable

Coding is very much about passing information around and do something with that information. In Python we can easily import an Excel sheet,

[11] `https://code.visualstudio.com/`
[12] `https://chat.openai.com/auth/login`

then we typically pass the Excel sheet around in the code and do some mathematical operations on the different columns. To pass data around in our code we use *variables*, below are two variables called `x` and `y`

```
x=13
y='Dog'
```

We use the equal sign, `=`, to create a binding between what is on the left and right side. Here we have assigned the value 13 and the string `Dog` to `x` and `y`, respectively. The value 13 (or the string `Dog`) is stored somewhere in the computer memory. In many ways you can consider the operation of creating a variable as to pick a box, put something in it, and labeling it, as illustrated in figure 1.3.

Fig. 1.3 A visualization of a variable.

The illustration in figure 1.3 also indicate that the size of the box may vary dependent on the content. Note that `x` and `y` are labels, it does not matter what kind of label we use, it is the content of the box that is important not the label you put on it. Normally you would use a more descriptive name than `x` or `y` to simply help other humans to better understand your code.

1.6.2 Functions

A function is several lines of code that perform a specific task. We can think of a function as a recipe, e.g. a cake recipe. To make a cake we need a certain input, eggs, flour, sugar, chocolate, then we follow a specific set of operations to produce the cake. A function in Python operates in the same way, it takes something as input (different variables), follow certain steps and returns a product (the cake).

Functions are useful because it allows us to wrap several lines of code that we believe we will use many times into reusable functions. Thus, we write the function (recipe) once and every time we want to make the same cake, we invoke the function to produce the output (cake), see figure 1.4.

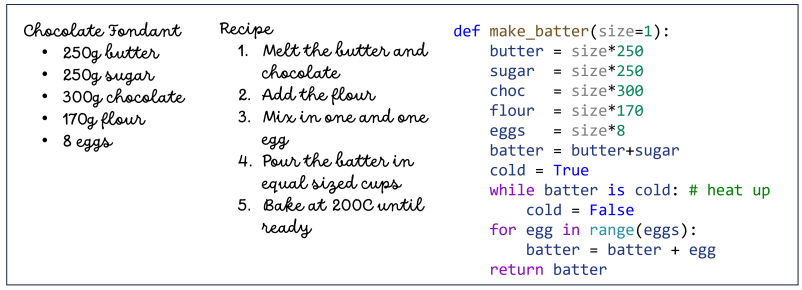

Fig. 1.4 My favorite chocolate fondant recipe and a Python code.

1.6.3 Objects and Classes

In Python everything is an object. An object is a variable (a box) that contains data and functions. That means that the boxes in figure 1.3 is more than just pieces of memory. To continue the with the recipe example above, we can think of an object as a cookbook that also contains ingredients, in Norwegian ”matkasse” or in English a ”meal kit”. There will be many recipes in this meal kit and many ingredients. In Python the syntax for accessing the functions (recipes) or data (ingredients) is by using the `.` syntax. In figure 1.2, this is illustrated. When we write `x='This is a string'`, we can e.g. do `x.capitalize()`, which will (not surprisingly) transform all the small letters to capital letters, `'THIS IS A STRING'`.

Thus in Python there will be a lot of ready made functions that you can use to quickly perform simple operations on your variables.

Objects vs Classes

A class is a blueprint and objects are an instance of the class. We create a class by writing lines of code, to create objects we execute the code in the class. In many ways you can say that objects are

Fig. 1.5 A visualization of an object, containing functions (recipes) and variables (ingredients).

physical whereas a class is logical. For the food kit case a class can be a description of the food kit on paper, describing how many recipes, how much potatoes, meat etc. should be included in each food kit, whereas all the physical food kits delivered to the customers are the objects.

1.6.4 Library

Python has a lot of libraries, which is one of the reasons why Python is so popular. These libraries are free and you can import them into your code. You can think of a library as a collection of cookbooks or meal kits as they also contains data. A Python library consists of a collection of objects, functions and or variables. In figure 1.1 two libraries are imported `numpy` and `matplotlib.pyplot`, we will return to these libraries later.

1.7 Exercise: Install a Python library in a separate environment

Hopefully you have managed to install Anaconda Python. Next, you want to open a terminal, if you are working on Linux or Mac, you open the terminal window. On Windows you open Anaconda Powershell Prompt from the Windows start menu. On my machine the prompt is a black screen, with the following text

Terminal

```
(base) PS C:\Users\Aksel Hiorth>
```

The `(base)` to the left indicates that we are in the base environment in Anaconda, `C:\Users\Aksel Hiorth`, show me that I am at the `C` disk, in the `Aksel Hiorth` folder, which is a sub directory of the `User` folder.

As a general rule we do not want to install new packages or libraries in the `base` environment, this is simply because different packages are not always internally consistent. If you are unlucky you will install two packages that requires two different version of a third package, this will then break your installation and suddenly code that used to run will no longer work.

A good practice is for each new project you start on that requires some special packages or libraries that you have not used before is to create a new environment.

Step 1 (Update conda, if you did not recently installed Anaconda):

- Open your anaconda powershell prompt (on Windows) and terminal (Mac or Linux)

- Make sure that conda is updated (conda is the package manager of Anaconda), by enter the following command.

Terminal
```
(base) Aksel Hiorth>conda update -n base -c conda-forge conda
```

This may take some time.

Step 2 (Create environment):

- Create a new environment by the following command

Terminal
```
(base) Aksel Hiorth>conda create -n MOD321
```

Here I tell python create a new environment called MOD321, which is the course code of this course. After you have accepted everything, you will get sometime like this

Terminal
```
#
# To activate this environment, use
#
#     $ conda activate MOD321
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

- Execute the above command

Terminal
```
(base) Aksel Hiorth>conda activate MOD321
```

Delete environments

If something goes wrong you can always delete the environment and create it once more. You simply has to deactivate it

Terminal

```
(MOD321) Aksel Hiorth>conda deactivate
```

Then you delete it

Terminal

```
(base) Aksel Hiorth>conda remove --name MOD321 --all
```

Step 3 (Install packages): There are several packages we will need and if we list all packages simultaneously conda will make sure that they are internally consistent. Enter the following command

Terminal

```
(base) Aksel Hiorth>conda activate MOD321
(MOD321) Aksel Hiorth>conda install matplotlib pandas jupyter scipy \
numpy ipykernel pathlib numba openpyxl
```

(NB: Due to formatting reasons I have added a back slash in the above command, but you can put everything on one line: `conda install matplotlib pandas jupyter scipy numpy ipykernel pathlib numba openpyxl`.) If you later find out you need additional packages, you just can just open a terminal window, activate the correct environment and then do `conda install PACKAGE_NAME`. But, as already mentioned if you install more and more very specialized packages, you might get inconsistencies. However, then you can just delete your environment and install everything once more.

pip install vs conda install

If you find a package you would like to install, the documentation may in many cases say run the command `pip install PACKAGE_NAME`. That may very well work, but I would always advice you to do

`conda install PACKAGE_NAME` first, because I believe conda is better at checking for internal consistency. If `conda install` fails, you can do `pip install` and 99 out 100 times this will work out just fine.

Introduction to practical applications

2

2.1 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.

2.2 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[1] 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[2], you will see a lot of examples on how to visualize your data.

2.3 Exercise 1: Reproduce a plot

- Reproduce figure 2.1 as closely as possible

2.3.1 Solution:

```
import matplotlib.pyplot as plt
x=[1,2,3,4]
y=[1,2,3,4]
plt.title('A linear function')
plt.plot(x,y,'-o',label='y=x')
plt.xlabel('x-values')
plt.ylabel('y-values')
plt.grid() #minor grid lines for readability
plt.legend()
```

[1] https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html
[2] https://matplotlib.org/stable/gallery/index.html

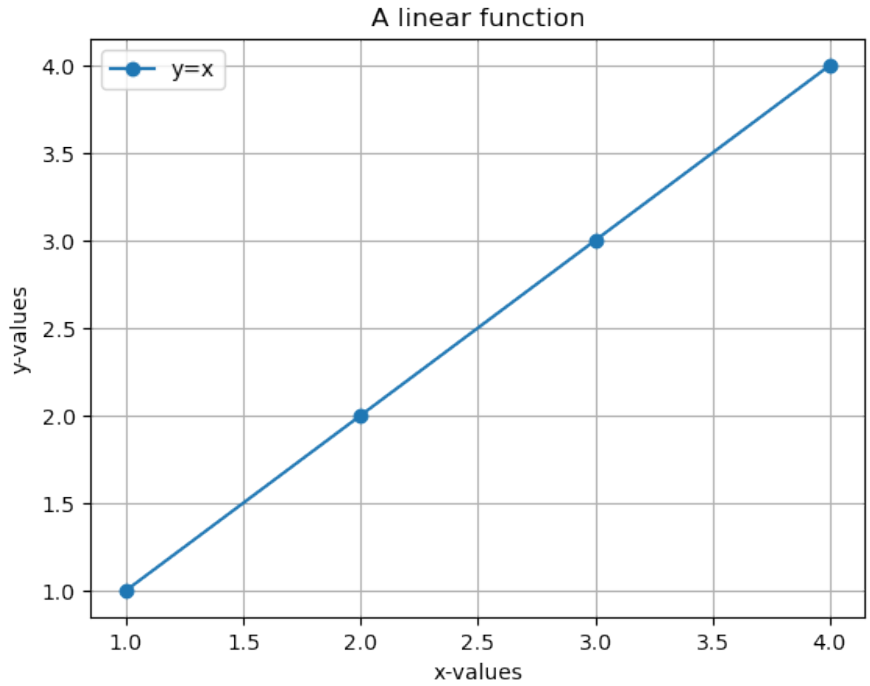

Fig. 2.1 A plot of a linear function.

2.4 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.

2.4.1 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.

```
my_list=['dog','cat',2]
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.

2.5 Exercise 2: Make a plot of $y = x^3$

- Use $x \in [-3, 3]$ and make a plot of $y = x^3$, similar to the one in figure 2.2

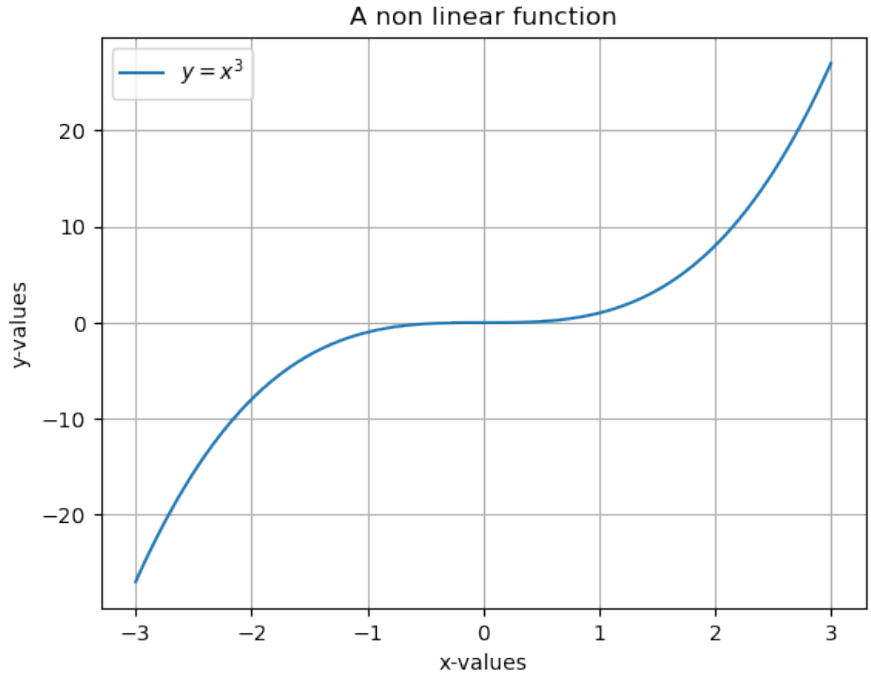

Fig. 2.2 A plot of a non linear function.

2.5.1 Solution

```
#first we create the x-values
N=100 # 100 points
dx=6/(N-1) #N-1 to include end-points
x=[-3+i*dx for i in range(N)] #List comprehension
y=[i**3 for i in x]
plt.title('A non linear function')
plt.plot(x,y,label='$y=x^3$') # if you want you can add legend and grid lines
plt.xlabel('x-values')
plt.ylabel('y-values')
```

```
plt.grid() #minor grid lines for readability
plt.legend()
```

2.5.2 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'],'*')
```

2.5.3 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]
```

2.6 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[3] 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
```

[3] https://numpy.org/

```
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.]
```

2.7 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]
```

2.8 Exercise 3: Make a plot of $y = x^3$ using Numpy

- Use $x \in [-3, 3]$ and make a plot of $y = x^3$, similar to the one in figure 2.2 as you did before, but this time use Numpy.

2.8.1 Solution

```
import numpy as np
N=100
x=np.linspace(-3,3,N)
y=x**3 #element wise operation
#Done, rest is just to make the plot
plt.title('A non linear function')
```

```
plt.plot(x,y,label='$y=x^3$') # if you want you can add legend and grid lines
plt.xlabel('x-values')
plt.ylabel('y-values')
plt.grid()
plt.legend()
```

2.9 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.

2.9.1 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())`

2.9.2 List all files and folder in current directory

As mentioned before a for-loop is a way to tell the computer to do something until a certain condition has been met. The code for listing all files and directories in a folder is then

```
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 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=pt.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())`.

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

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=pt.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
```

2.10 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"[4]. 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.

2.10.1 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

[4] `https://pandas.pydata.org/`

header name and data in an array associated with that header, as illustrated in 2.3.

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	

LOCATION	TIME	ELAPSED	CONFIRM	DEATHS	RECOVER
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

Fig. 2.3 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

```
import pandas as pd
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[5].

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.

2.10.2 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

[5] `https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html`

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']
}
my_df=pd.DataFrame(my_dict)
print(my_df) # to view
```

2.10.3 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
df=pd.read_excel('../data/corona_data.xlsx')
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()`[6]

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

2.10.4 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

[6] `https://pandas.pydata.org/pandas-docs/version/1.2.4/reference/api/pandas.Series.str.strip.`

`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
df=pd.read_excel('../data/corona_data.xlsx')
time=df['TIME']
# what happens if you set
# df2=pd.read_csv('../data/corona_data.dat',sep='\t') # csv tab separated file
# 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":i
df2=pd.read_csv('../data/corona_data.dat',sep='\t',dtype=types_dict,parse_dates=['TIME']) # set da
```

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

2.10.5 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`[7].

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

2.10.6 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

[7] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html`

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`[8])
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`[9] library, simply because I know it

[8] `https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html`
[9] `https://matplotlib.org/`

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()
```

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`.

2.10.7 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()`[10] function. To sort our DataFrame on dates and sum the occurrences we can do

[10] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html`

```
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 2.4 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

Fig. 2.4 The results of `df.groupby('LOCATION').sum()`.

2.10.8 Simple statistics in Pandas

At the end it is worth mentioning the built in methods `pd.DataFrame.mean`, `pd.DataFrame.median`, `pd.DataFrame.std` which calculates the mean, median and standard deviation on the columns in the DataFrame where it make sense (i.e. avoid strings and dates). To get all these values in one go (and a few more) on can also use `pd.DataFrame.describe()`

```
df.describe()
```

The output is shown in figure 2.5

```
df.describe()
✓ 0.1s
```

	CONFIRMED	DEATHS	RECOVERED	ELAPSED_TIME_SINCE_OUTBREAK
count	7.000000	7.0	7.0	7.000000
mean	18.142857	0.0	0.0	3.000000
std	29.277002	0.0	0.0	2.160247
min	1.000000	0.0	0.0	0.000000
25%	1.000000	0.0	0.0	1.500000
50%	1.000000	0.0	0.0	3.000000
75%	31.000000	0.0	0.0	4.500000
max	61.000000	0.0	0.0	6.000000

Fig. 2.5 Output from the describe command.

2.10.9 Joining two DataFrames

Appending DataFrames. The DataFrame with the Covid-19 data in the previous section could have been created from two separate DataFrames, using `concat()`[11]. First, create two DataFrames

```
import datetime as dt
a=dt.datetime(2020,2,24,23,59)
b=dt.datetime(2020,2,7,23,59)
my_dict1={'LOCATION':7*['Afghanistan'],
'TIME':[a+dt.timedelta(days=i) for i in range(7)],
'ELAPSED_TIME_SINCE_OUTBREAK':[0, 1, 2, 3, 4, 5, 6],
'CONFIRMED':7*[1],
'DEATHS':7*[0],
'RECOVERED': 7*[0]}
my_dict2={'LOCATION':6*['Diamond Princess'],
'TIME':[b+dt.timedelta(days=i) for i in range(6)],
'ELAPSED_TIME_SINCE_OUTBREAK':[0, 1, 2, 3, 4, 5],
'CONFIRMED':[61, 61, 64, 135, 135, 175],
'DEATHS':6*[0],
'RECOVERED': 6*[0]}
df1=pd.DataFrame(my_dict1)
df2=pd.DataFrame(my_dict2)
```

Next, add them row wise (see figure 2.6)

```
df=pd.concat([df1,df2])
print(df) # to view
```

[11] https://pandas.pydata.org/docs/reference/api/pandas.concat.html

	LOCATION	TIME	ELAPSED_TIME_SINCE_OUTBREAK	CONFIRMED	DEATHS	RECOVERED
0	Afghanistan	2020-02-24 23:59:00	0	1	0	0
1	Afghanistan	2020-02-25 23:59:00	1	1	0	0
2	Afghanistan	2020-02-26 23:59:00	2	1	0	0
3	Afghanistan	2020-02-27 23:59:00	3	1	0	0
4	Afghanistan	2020-02-28 23:59:00	4	1	0	0
5	Afghanistan	2020-02-29 23:59:00	5	1	0	0
6	Afghanistan	2020-03-01 23:59:00	6	1	0	0
0	Diamond Princess	2020-02-07 23:59:00	0	61	0	0
1	Diamond Princess	2020-02-08 23:59:00	1	61	0	0
2	Diamond Princess	2020-02-09 23:59:00	2	64	0	0
3	Diamond Princess	2020-02-10 23:59:00	3	135	0	0
4	Diamond Princess	2020-02-11 23:59:00	4	135	0	0
5	Diamond Princess	2020-02-12 23:59:00	5	175	0	0

Fig. 2.6 The result of `concat()`.

If you compare this DataFrame with the previous one, you will see that the index column is different. This is because when joining two DataFrames Pandas does not reset the index by default, doing `df=pd.concat([df1,df2],ignore_index=True)` resets the index. It is also possible to join DataFrames column vise

```
pd.concat([df1,df2],axis=1)
```

Merging DataFrames. In the previous example we had two non overlapping DataFrames (separate countries and times). It could also be the case that some of the data was overlapping e.g. continuing with the Covid-19 data, one could assume that there was one data set from one region and one from another region in the same country

```
my_dict1={'LOCATION':7*['Diamond Princess'],
'TIME':[b+dt.timedelta(days=i) for i in range(7)],
'ELAPSED_TIME_SINCE_OUTBREAK':[0, 1, 2, 3, 4, 5, 6],
'CONFIRMED':7*[1],
'DEATHS':7*[0],
'RECOVERED': 7*[0]}
my_dict2={'LOCATION':2*['Diamond Princess'],
'TIME':[b+dt.timedelta(days=i) for i in range(2)],
'ELAPSED_TIME_SINCE_OUTBREAK':[0, 1],
'CONFIRMED':[60, 60],
'DEATHS':2*[0],
'RECOVERED': 2*[0]}
df1=pd.DataFrame(my_dict1)
df2=pd.DataFrame(my_dict2)
```

If we do `pd.concat([df1,df2])` we will simply add all values after each other. What we want to do is to sum the number of confirmed, recovered and deaths for the same date. This can be done in several ways, but one way is to use `pd.DataFrame.merge()`[12].You can specify the columns to merge on, and choose `outer` which is union (all data from both frames) or `inner` which means the intersect (only data which you merge on that exists in both frames), see figure 2.7 for a visual image.

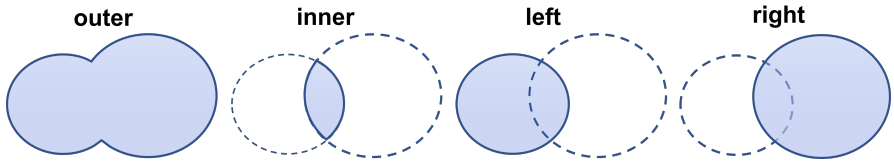

Fig. 2.7 The result of using `how=outer, inner, left`, or `right` in `pd.DataFrame.merge()`.

To be even more specific, after performing the commands

```
df1.merge(df2,on=['LOCATION','TIME'],how='outer')
df1.merge(df2,on=['LOCATION','TIME'],how='inner')
```

we get the results in figure 2.8

	LOCATION	TIME	ELAPSED_TIME_SINCE_OUTBREAK_x	CONFIRMED_x	DEATHS_x	RECOVERED_x	ELAPSED_TIME_SINCE_OUTBREAK_y	CONFIRMED_y	DEATHS_y	RECOVERED_y
0	Diamond Princess	2020-02-07 23:59:00	0	1	0	0	0.0	60.0	0.0	0.0
1	Diamond Princess	2020-02-08 23:59:00	1	1	0	0	1.0	60.0	0.0	0.0
2	Diamond Princess	2020-02-09 23:59:00	2	1	0	0	NaN	NaN	NaN	NaN
3	Diamond Princess	2020-02-10 23:59:00	3	1	0	0	NaN	NaN	NaN	NaN
4	Diamond Princess	2020-02-11 23:59:00	4	1	0	0	NaN	NaN	NaN	NaN
5	Diamond Princess	2020-02-12 23:59:00	5	1	0	0	NaN	NaN	NaN	NaN
6	Diamond Princess	2020-02-13 23:59:00	6	1	0	0	NaN	NaN	NaN	NaN

	LOCATION	TIME	ELAPSED_TIME_SINCE_OUTBREAK_x	CONFIRMED_x	DEATHS_x	RECOVERED_x	ELAPSED_TIME_SINCE_OUTBREAK_y	CONFIRMED_y	DEATHS_y	RECOVERED_y
0	Diamond Princess	2020-02-07 23:59:00	0	1	0	0	0	60	0	0
1	Diamond Princess	2020-02-08 23:59:00	1	1	0	0	1	60	0	0

Fig. 2.8 Merging to DataFrame using `outer` (top) and `inner` (bottom).

Clearly in this case we need to choose `outer`. In the merge process pandas adds an extra subscript `_x` and `_y` on columns that contains the same header name. We also need to sum those, which can be done as follows (see figure 2.9 for the final result)

```
df=df1.merge(df2,on=['LOCATION','TIME'],how='outer')
cols=['CONFIRMED','DEATHS', 'RECOVERED']
for col in cols:
    df[col]=df[[col+'_x',col+'_y']].sum(axis=1) # sum row elements
```

[12] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html`

```
    df=df.drop(columns=[col+'_x',col+'_y']) # remove obsolete columns
# final clean up
df['ELAPSED_TIME_SINCE_OUTBREAK']=df['ELAPSED_TIME_SINCE_OUTBREAK_x']
df=df.drop(columns=['ELAPSED_TIME_SINCE_OUTBREAK_y','ELAPSED_TIME_SINCE_OUTBREAK_x'])
```

	LOCATION	TIME	CONFIRMED	DEATHS	RECOVERED	ELAPSED_TIME_SINCE_OUTBREAK
0	Diamond Princess	2020-02-07 23:59:00	61.0	0.0	0.0	0
1	Diamond Princess	2020-02-08 23:59:00	61.0	0.0	0.0	1
2	Diamond Princess	2020-02-09 23:59:00	1.0	0.0	0.0	2
3	Diamond Princess	2020-02-10 23:59:00	1.0	0.0	0.0	3
4	Diamond Princess	2020-02-11 23:59:00	1.0	0.0	0.0	4
5	Diamond Princess	2020-02-12 23:59:00	1.0	0.0	0.0	5
6	Diamond Princess	2020-02-13 23:59:00	1.0	0.0	0.0	6

Fig. 2.9 Result of outer merging and summing.

2.10.10 Working with folders and files

When working with big data sets you might want to split data into smaller sets, and also write them to different folders (or files) to view each individually in excel. Working with files and folders in a way that will work on any kind of platform has always been a challenge, but it is greatly simplified by the Pathlib library[13].

[13] https://docs.python.org/3/library/pathlib.html

Exercises

3

Learning objectives.

- Create an environment using conda and install required packages
- Use Matplotlib for basic visualization
- Loops for repetitive tasks
- File handling
- Use Pandas to combine and manipulate data from different input, Excel and comma separated values (CSV), files.

3.1 Exercise 1: Install Bedmap to visualize Antarctica ice data

3.1.1 Background

There is currently a great deal of concern about global warming. Some critical issues are whether we are more likely to observe extreme local temperatures, increased frequencies of natural disasters like forest fires and droughts, and if there are "tipping points" in the climate system that are, at least on the human timescale, irreversible [2]. One particular question to ask is: How much ice is likely to melt? And, what would be the consequence of ice melting for sea level rise (SLR)?

Since most of the ice on Planet Earth is located in Antarctica, substantial effort has been spent in mapping the ice and the bedrock of this continent. Most of the data is freely available, and we can use Python to

investigate different scenarios. Here we will only show you how to install the packages and visualize the ice thickness.

Part 1.

Create a new environment

The ice data are located in the *bedmap2* dataset [1] The rockhound[a] library can be used to load the data. As an aid to plotting, you might also want to use color maps from the cmocean[b] package [3]. To install bedmap2 data you have to create a new conda environment (we call it ice)

Terminal

```
conda config --add channels conda-forge
conda create -n ice python matplotlib numpy \
scipy xarray==0.19.0 pandas rockhound cmocean pip jupyter
```

Note that we need a specific version of the `xarray` library.

```
conda activate ice
jupyter notebook
```

[a] https://github.com/fatiando/rockhound
[b] https://pypi.org/project/cmocean/

Part 2. The code below is taken from the rockound library documentation[1]:

```
import rockhound as rh
import matplotlib.pyplot as plt
import cmocean
import numpy as np

bedmap = rh.fetch_bedmap2(datasets=["thickness","surface","bed"])
plt.figure(figsize=(8, 7))
ax = plt.subplot(111)
bedmap.surface.plot.pcolormesh(ax=ax,cmap=cmocean.cm.ice,
                        cbar_kwargs=dict(pad=0.01, aspect=30))
plt.title("Bedmap2 Antarctica")
plt.tight_layout()
plt.show()
```

[1] https://www.fatiando.org/rockhound/latest/api/generated/rockhound.fetch_bedmap2.html

- Run the code and reproduce figure 3.1, modify the code to plot the ice surface or the bed rock.

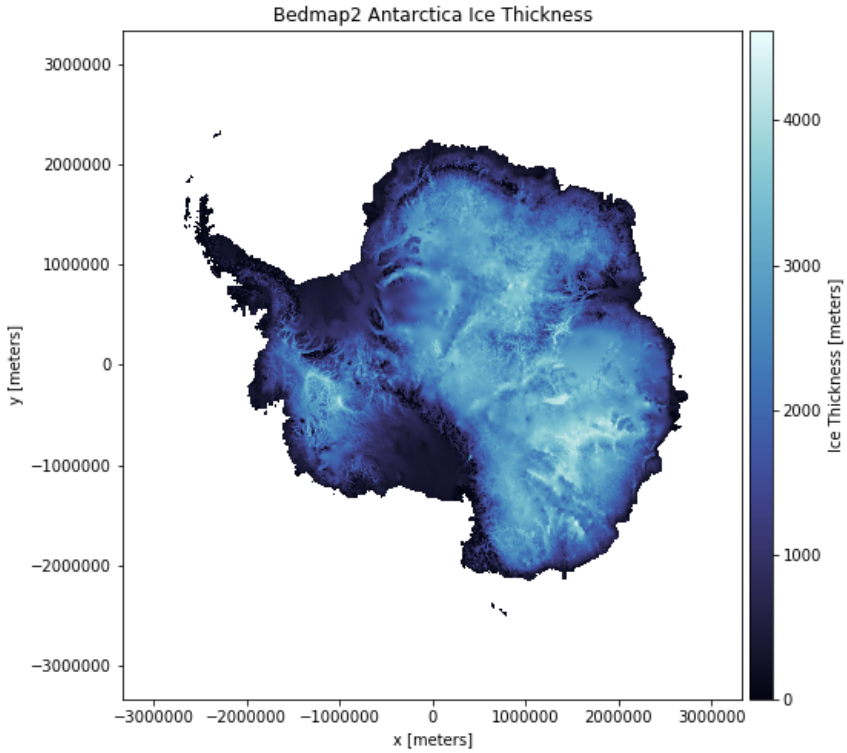

Fig. 3.1 Visualization of the ice thickness in Antarctica.

- Check out the gallery[2]. Choose one of the datasets, copy and paste the code and reproduce one of the figures in the gallery.

3.2 Exercise 2: Matplotlib visualization

To access the data a bit easier I have added a file `draugen_pandas.py` in the `data` folder containing our data as lists. We can access them by

[2] `https://www.fatiando.org/rockhound/latest/gallery/index.html`

importing them directly into our script. However, as we are in the `src` folder it causes a practical problem to access a Python file in a different folder and the only way to do this is to add the `data` folder to our path

```
import sys
sys.path.append('../data/')
from draugen_data import years,months,oil,gas,wat,cond,oe
```

Question1:

- Use matplotlib to plot oil equivalents, `oe`, vs the `years` data, compare with the official data[3].

Question2:

- Our plot looks a bit strange, because we do not take into account the month column. Instead of only plotting the year array, plot `years+months/12` on the x-axis.

```
import matplotlib.pyplot as plt
plt.plot(years,oe)
```

Solution2:

Use vanilla Python (hard). Here we need to loop over all elements and for each year add the month, divided by 12 to convert the month to year. What we want is

1. To start with an empty list `year_month=[]`
2. Loop over all elements and add first year and first month divided by 12

How do we do loop over elements in Python?

We loop over elements using a `for` loop. The keyword `for` is always accompanied by the `in` keyword.

Method 1: If you have coded before, this might be very familiar.

```
N=len(years) # N is the length of the years list
year_month=[]
for i in range(N):
    year_month.append(years[i]+months[i]/12)
```

[3] `https://factpages.sodir.no/en/field/PageView/All/43758`

Indentation matters!

The for-loop above ends with `:`, and then Python uses indentation (a tab) 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*.

```
for i in range(N):
year_month.append(years[i]+months[i]/12)#Error!! because no indentation
```

The statement `range(N)` is a *generator* and it generates a sequence of integers with length N, starting from zero to $N-1$.

Method2: This method is slightly more pythonic, than the previous. Instead of accessing the different elements in `years` by `years[i]`, we can loop directly over them

```
i=0
for year in years:
    year_month.append(year+months[i]/12)
    i = i+1
```

Meaningful variable names

The specific name we give the counter, `year`, is not important for the computer. But if you choose a descriptive name it makes the code easier to read for humans.

The code above is ok, but it seems unnecessary to introduce the extra counter `i`, the way that we have done it. To access the index of each element in addition to the value, we can use the `enumerate()` function

```
for i,year in enumerate(years):
    year_month.append(year+months[i]/12)
```

Method3: If you have lists of the same length we can access the elements in a loop using the `zip` function

```
for month,year in zip(months,years):
    year_month.append(year+month/12)
```

The `zip` function uses a nice feature in Python, which is called *variable unpacking*. This is a special assignment operation, where we can assign all variables in a an iterable object in one go e.g.

```
my_list=[2024,1,9]
year,month,day=my_list #year=2024,month=1,day=9
```

Method4: *List comprehension*[4] is a very pythonic way of creating new lists. It allows us to write a for loop while creating a python list.

```
year_month = [year+month/12 for month,year in zip(months,years)]
```

- Use one of the methods above `year_month=np.array(year)+np.array(month)/12` and create a new plot, with `year_month` on the x-axis.
- Try to make the plot as similar as possible to figure 3.2

Convert list to Numpy arrays and plot (easy). As we have seen `year_month=year+month/12` will not work for lists as Python does not understand what `month/12` is, and even if it did the `+` operation would not give the expected results. But, if we convert our list to a Numpy array, life becomes easy. To convert a list to a Numpy array, we use what programmers calls *casting*[5]. Casting is when we tell the computer to convert a variable from one type to another, e.g.

```
my_list=[1,2,3]
my_np_array=np.array(my_list) # cast to array
# more examples
a='1' # a is a string
b='2' # b is a string
a+b # gives a new string '12'
int(a)+int(b) # gives integer 3
float(a)+float(b) # gives float 3.0
```

Vectorized operations in Numpy

One of the major strengths of Numpy is that it is vectorized. This means that mathematical operations you do with numbers such as `+`, `-`, `/`, `*`, you can also do with Numpy arrays with the effect that the operation is done on each element. *This only works if the arrays have the same length.* The only exception is if one of the elements is a single number, e.g.

[4] https://www.w3schools.com/python/python_lists_comprehension.asp
[5] https://www.w3schools.com/python/python_casting.asp

```
# will divide all elements in month by 12
np.array(months)/12
# add first element in year with first element in month/12 and so on
np.array(years)+np.array(months)/12
```

- Use `year_month=np.array(years)+np.array(months)/12` and create a new plot, with `year_month` on the x-axis.
- Try to make the plot as similar as possible to figure 3.2

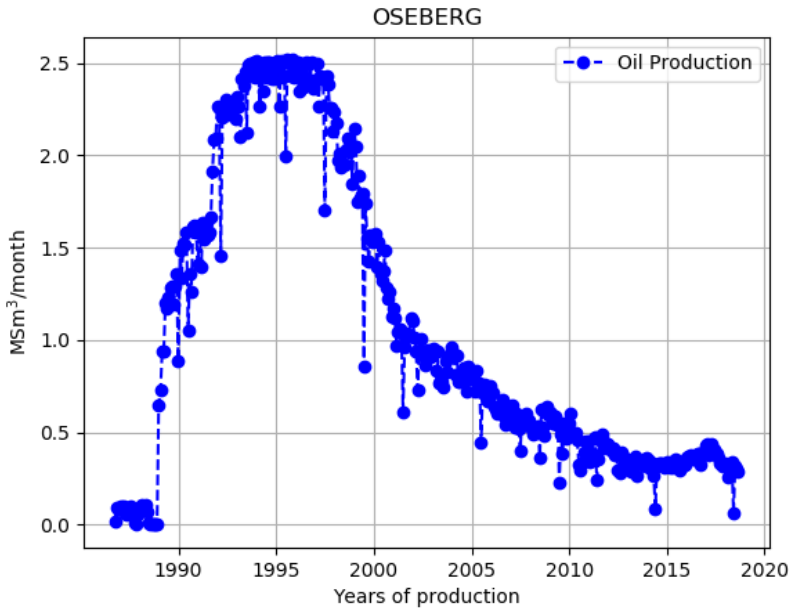

Fig. 3.2 Production of oil equivalents on the Draugen vs time.

3.3 Exercise 3: Group data

If you compare the plot in figure 3.2 with the official plot[6], and for convenience illustrated below in figure 3.3, you will see that they are

[6] `https://factpages.sodir.no/en/field/PageView/All/43758`

different. The difference is that the production in figure 3.2 is per month, whereas in figure 3.3 the production is per year.

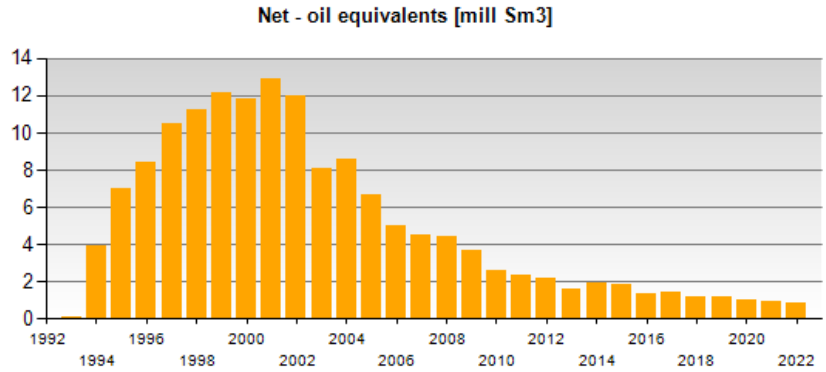

Fig. 3.3 Production of oil equivalents on the Draugen field.

Question: Use the imported data for Draugen, and plot the production per year vs time. That means we want to sum up all oil equivalents when the years have the same value.

Solution1, Vanilla Python (hard): There are probably many ways of solving this problem, but here we will use a dictionary. The reason we use a dictionary is that there will be an unique key (the year) for each entry. The tricky part is the initialization

- if the key exists in the dictionary, we want to add the oil equivalents to the previous values,
- if the key does not exists (we have not summed the oil equivalents for that year) we need for that key to set the first value.

Thus we need to check if the key already exists

```
data={} #just an empty dictionary
for year,o in zip(years,oe):
    key=year
    if key in data: #key exists
       data[key] += o #add oil equivalents
    else: # new key
       data[key] = o  #set equal to first month
```

Next, we need to extract all oil equivalents for each year, and make the plot

```
oe_per_year=[data[key] for key in data] #list comprehension
```

```
year=[key for key in data]
plt.plot(year,oe_per_year)
plt.bar(year,oe_per_year,color='orange')
```

Solution2, Numpy and Boolean masking (hard): We can also Boolean masking, if we convert the lists to Numpy arrays. If we want to pick out all data for a specific year, e.g. 2008 and sum the oil equivalents we can do as follows

- `np.array(years) == 2008` will give `True` for all entries containing 2008 and `False` otherwise
- If we pass this to our oil equivalent array, only the values corresponding to `True` will be picked out and then we can quickly sum them by `np.sum`

```
years=np.array(years) # cast to Numpy array
oe=np.array(oe) # ditto
np.sum(oe[years==2008]) # 4.265517 mill Sm^3
```

Now, we just need to loop over all the years and collect the data

```
unique_list=np.unique(years) #remove duplicates
oe_tot=[] # empty list
for year in unique_list:
    oe_tot.append(np.sum(oe[years==year]))
#make the plot
plt.plot(unique_list,oe_tot)
plt.bar(unique_list,oe_tot,color='orange')
```

Solution3, Pandas (easy): To use Pandas we need to make a DataFrame of the data and then we use the `groupby`[7] function in Pandas, which is extremely powerful.

```
import pandas as pd
df=pd.DataFrame() # empty DataFrame
df['Year']=years
df['oe']=oe
print(df) # inspect to see what happens
df2=df.groupby(by='Year').sum()
print(df2) # inspect to see what happens
df2.plot.bar() #make the plot
```

Note that most of the code above was to create the DataFrame. Pandas also has built-in plotting.

[7] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html`

3.4 Exercise 4: Read tabulated data from file

As part of this project we will look at some of the datasets that are available at the Norwegian Offshore Directorate website[8].

Data directory

In the following we will assume that you have a `src` directory containing all your Python code and a `data` folder inside your course folder. Thus, if you are in your `src` folder you can access files in the `data` folder by moving one folder up an down inside your `data` folder.

Question: In the `data` folder, there is a file `draugen.txt` containing production data from Draugen. Open it in a text editor. Read the data from the file using Python, and store the `Year`, `Month` and `oil equivalents [mill Sm3]` in separate variables.

Solution :

3.4.1 Solution 1 Pandas (easy):

Pandas will most likely be your first choice, because it has so much built in functionalities

- Run the following code and explain why the first `pd.read_csv` fails or does not produce the output we want

```
import pandas as pd
df=pd.read_csv('../data/draugen.txt')
print(df)
df=pd.read_csv('../data/draugen.txt',sep='\t')
print(df)
```

- Print only the year column from `df` i) using the label `Year` and ii) `df.columns[0]`
- Print the column with oil equivalents

[8] `https://factpages.sodir.no/`

3.4.2 Solution 2 `numpy.loadtxt` (medium):

`Numpy.loadtxt` is a build in function in Numpy that reads tabulated data. It does not know how to process header lines, and we need to skip them.

```
import numpy as np
data=np.loadtxt('../data/draugen.txt',skiprows=1)
```

- How can you print out only the year, month, etc. columns from the `data` variable?
- Use `data=np.loadtxt('../data/draugen.txt',skiprows=1,unpack=True)`, what changes now?

3.4.3 Solution 3 Vanilla Python (hard):

When accessing files, it can be easy to forget to *close* the file after opening it. This can lead to problems as an open file, in many cases, cannot be accessed by other programs. To avoid this we use the `with` statement. The following code can be used to print out all the lines of a file, and Python will open and close the file for you

```
with open("../data/draugen.txt") as my_file:
    for line in my_file:
        print(line)
```

After the code is run, the variable `line` will still hold the last line of the file, which should be `'2023\t10\t0.072419\t0.019985\t0\t0.092404\t1.005672\n'`. The first two numbers are year (2023) and the month (10), for the rest check the header in the file. The `\t` means tabular and `\n` means a newline. In order to parse this line we need to pick out the numbers, the easiest way in Python is to split the line on `\t`, if you do

```
data_list=line.split('\t')
print(data_list)
```

we conveniently get all the elements separated by `\t` into a list (of strings). To convert a string to a number we can do e.g. `float(data_list[0])`.

- Modify the following code to extract year, month, and oil equivalents. Remember we need to skip the first header line, which contains only text.

```
years=[] # empty list
months=[] # empty list
oe=[] # empty list
read_first_line=False
with open("../data/draugen.txt") as my_file:
     for line in my_file:
          if read_first_line:
        data_list=line.split('\t')
        years.append(float(data_list[0]))
        #same for month
        #same for oil equivalents
     read_first_line=True
```

Store the data in a dictionary (optional). Tabulated data with a header is perfect for a dictionary, here we create a dictionary based on the header in the file.

```
data_dict={'year':[], 'month':[],'oil':[],'gas':[],'cond':[],'oe':[],'wat':[]}
```

- modify the code above to store all data in each line in the dictionary. To loop over all entries in the `data_dict`, and the list elements in `data_list`, you can use the `zip` function

```
for key,data in zip(data_dict,data_list):
    data_dict[key].append(float(data))
```

3.5 Exercise 5: Splitting data into files using Pandas

If you open the file `field_production_gross_monthly.xlsx` in the `data` folder in Excel, you will see that the field names are listed in the leftmost column.

Question: Open the file `field_production_gross_monthly.xlsx` in Pandas and write a new file in the same directory containing only data for a given field.

Solution:

```
import pandas as pd
df=pd.read_excel('../data/field_production_gross_monthly.xlsx')
field='DRAUGEN'
file_out='../data/'+field+'.xlsx'
df2=df[df[df.columns[0]]==field]
df2.to_excel(file_out,index=False)
```

The tricky part is perhaps the syntax `df[df[df.columns[0]]==field]`. To understand it in more detail, start by printing the innermost statements and work from there

```
print(df.columns[0]) # gives the header of the first column
# gives True for all entries in the first column that contains
# the specific field name (in this case DRAGUEN)
df[df.columns[0]]==field
# Following code is equivalent, we use iloc to specify the first column
df.iloc[:,0]==field
```

3.6 Exercise 6: Splitting all field data into separate files

Question: Split all production in `field_production_gross_monthly.xlsx` into different Excel files containing only data from one specific field.

To help you, here are the different steps

1. First we need to find a unique list of all field names, this can be done by `fields=df[df.columns[0]].unique()`
2. Then we need to loop over all these fields and perform the operations as in the previous exercise

Solution: There is one problem, and that is that some of the field names contains a slash `/`. A `/` indicates a new directory which does not exists, hence we need to replace the slash with something else

```
df=pd.read_excel('../data/field_production_gross_monthly.xlsx')
fields=df[df.columns[0]].unique()
for field in fields:
    new_name=str.replace(field,'/','')
    file_out='../data/'+new_name+'.xlsx'
    df2=df[df[df.columns[0]]==field]
    df2.to_excel(file_out,index=False)
```

3.7 Exercise 7: Splitting field data into separate files and folder

Splitting the data into different Excel files generates a lot of files in the same directory.

Question: Use Pathlib to split the data into a different folder. All data should be stored in a folder named `tmp_data`, the `tmp_data` should contain a directory named after the field and this folder should contain a file named `production_data.xlsx` containing only data for that field.

Solution:

```
df=pd.read_excel('../data/field_production_gross_monthly.xlsx')
fields=df[df.columns[0]].unique() #skip duplicates
data_folder=pt.Path('../tmp_data')
data_folder.mkdir(exist_ok=True)
for field in fields:
    new_name=str.replace(field,'/','')
    new_path=data_folder / new_name
    new_path.mkdir(exist_ok=True)
    df2.to_excel(new_path/'production_data.xlsx',index=False)
```

References

[1] Peter Fretwell, Hamish D. Pritchard, David G. Vaughan, Jonathan L. Bamber, Nicholas E. Barrand, R. Bell, C. Bianchi, RG Bingham, Donald D. Blankenship, and G. Casassa. Bedmap2: Improved ice bed, surface and thickness datasets for antarctica. *The Cryosphere*, 7(1):375–393, 2013.

[2] V. Masson-Delmotte, P. Zhai, A. Pirani, S. L. Connors, C. Pean, S. Berger, N. Caud, Y. Chen, L. Goldfarb, M. I. Gomis, M. Huang, K. Leitzell, E. Lonnoy, J. B. R. Matthews, T. K. Maycock, T. Waterfield, O. Yelekci, R. Yu, and B. Zhou (eds.). Climate Change 2021: the Physical Science Basis. Contribution of Working Group I to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change. Technical report, Cambridge University Press. In Press., August 2021.

[3] Kristen M. Thyng, Chad A. Greene, Robert D. Hetland, Heather M. Zimmerle, and Steven F. DiMarco. True colors of oceanography: Guidelines for effective and accurate colormap selection. *Oceanography*, 29(3):9–13, 2016.

Index