E-MOD321 Basic Python coding for subsurface applications

Aksel Hiorth, University of Stavanger

Jan 15, 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

1 Preliminaries

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.

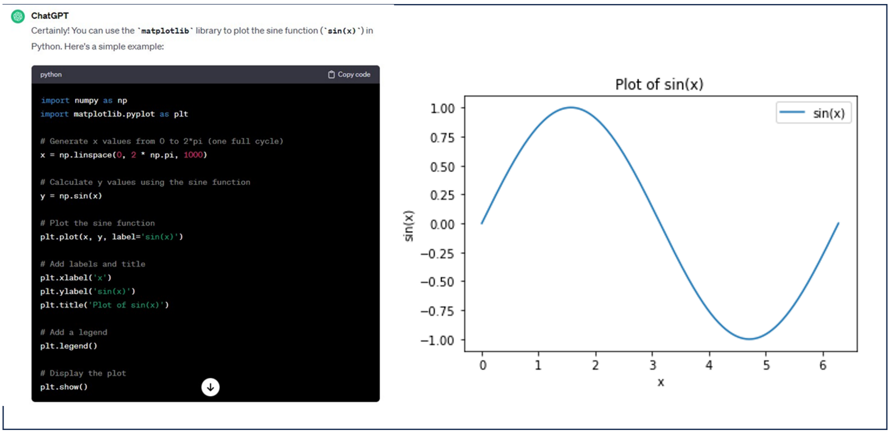


Fig. 1.1 Output from ChatGPT and the result after running the code.

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.

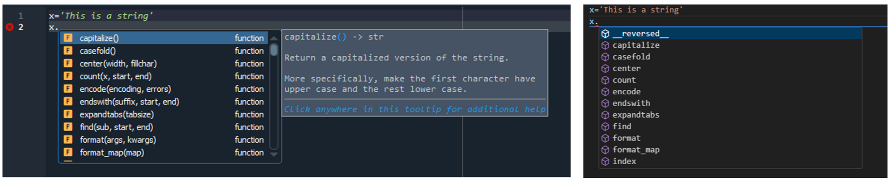


Fig. 1.2 Two IDEs to write Python code (left) Spyder (right) VS Code.

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.

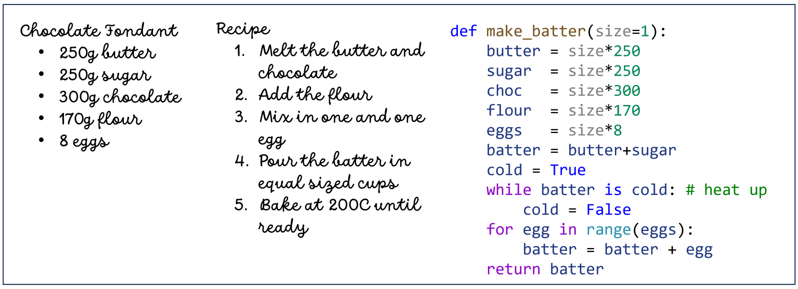
Fig. 1.3 A visualization of a variable.

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.


Chocolate Fondant

- 250g butter
- 250g sugar
- 300g chocolate
- 170g flour
- 8 eggs

Recipe

1. Melt the butter and chocolate
2. Add the flour
3. Mix in one and one egg
4. Pour the batter in equal sized cups
5. Bake at 200C until ready

```python
def make_batter(size=1):
    butter = size*250
    sugar  = size*250
    choc   = size*300
    flour  = size*170
    eggs   = size*8
    batter = butter+sugar
    cold = True
    while batter is cold: # heat up
        cold = False
    for egg in range(eggs):
        batter = batter + egg
    return batter
```

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.

2 Introduction to Common Libraries: Matplotlib, Numpy, Pathlib, Pandas

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

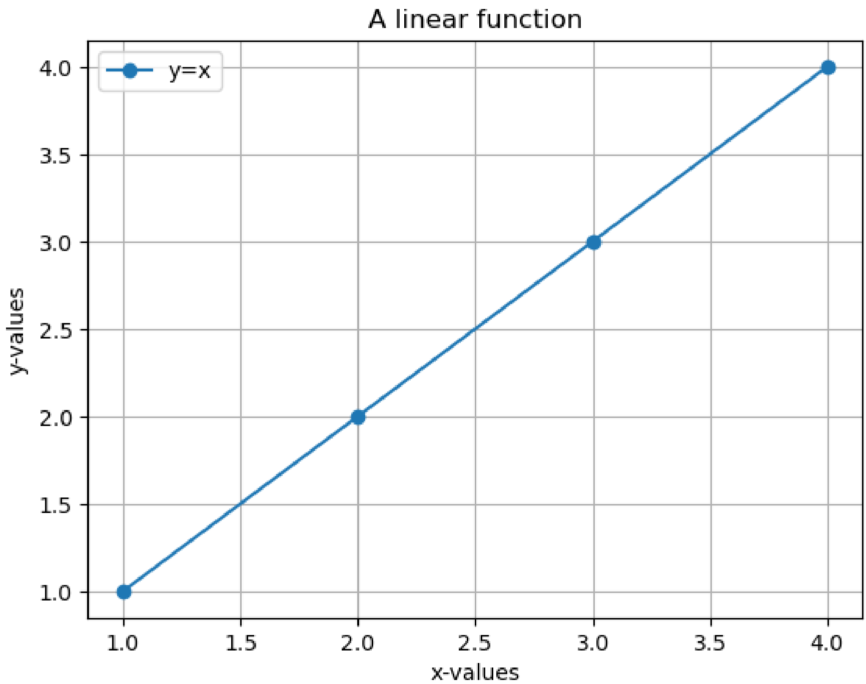


Fig. 2.1 A plot of a linear function.

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

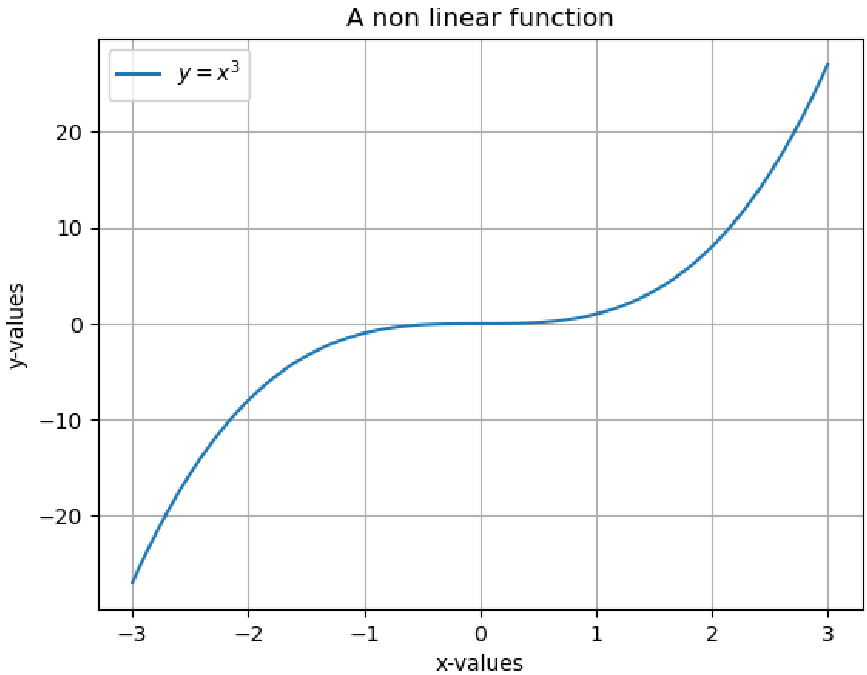


Fig. 2.2 A plot of a non linear function.

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

If you want to print the full path do `print(x.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.

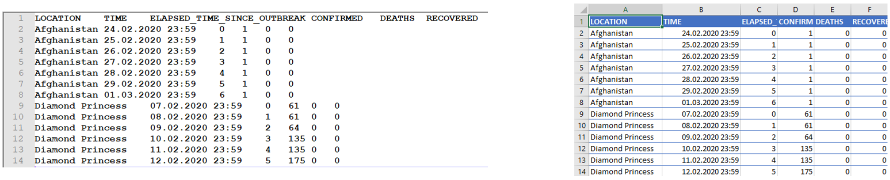
2.10.1 DataFrame: The basic object in Pandas

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

	A	B	C	D	E	F
1	LOCATION	TIME	ELAPSED_	CONFIRM	DEATHS	RECOVERE
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

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":
df2=pd.read_csv('../data/corona_data.dat',sep='\t',dtype=types_dict,parse_dates=['TIME']) # set d
```

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).

	ELAPSED_TIME_SINCE_OUTBREAK	CONFIRMED	DEATHS	RECOVERED
LOCATION				
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.

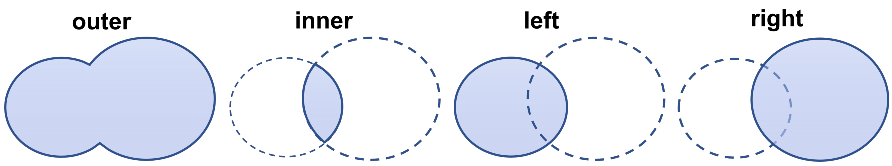


Fig. 2.7 The result of using `how=outer`, `inner`, `left`, or `right` in `pd.DataFrame.merge()`.

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.

3 Functions in Python

3.1 What is a function?

As explained in the introduction, a function is several lines of code that perform a specific task. In many ways you 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).

You have already used built in functions in Python, such as `print()`, `pandas.DataFrame()`, etc.

3.1.1 How to create a function in Python

We can define our own functions using the `def` keyword, it best illustrated with some examples

```
def my_func():
    print('My first function')
my_func() #My first function
```

You can pass arguments to the function, to make them more general

```
def greeting(name):
    print('My name is :', name)
greeting('Bob') # My name is Bob
```

It is also common practice to add a docstring to help people better understand what the function does.

```
def greeting(name):
    '''
    prints out a greeting
    '''
    print('My name is :', name)
greeting('Bob') # My name is Bob
```

If we do `help(greeting)`, Python will print out our docstring. A function can also return something

```
def add(x,y):
    '''
    adds two numbers
    '''
    return x+y
add(5,6) # 11
```

3.2 When to define a function?

When to use functions? There is no particular rule, *but whenever you start to copy and paste code from one place to another, you should consider to use a function*. Functions makes the code easier to read. It is not easy to identify which part of a program is a good candidate for a function, it requires skill and experience. Most likely you will end up changing the function definitions as your program develops.

3.2.1 What is a good function?

Even if you only write code for yourself, you will quickly forget what the code does. That is why it is so important to write functions that are good, below are some suggestions

A function:

- should have a descriptive name.
- should only do one thing.
- should only be dependent on its input argument and give the same answer independent on how many times it is being called.
- should contain a docstring (see below for examples).
- should have a descriptive name, use small letters with an underscore to separate words.

DRY - Do not Repeat Yourself [3].

If you need to change the code in more than one place to extend it, you may forget to change everywhere and introduce bugs. The DRY principle also applies to *knowledge sharing*, it is not only about copy and paste code, but knowledge should only be represented in one place.

3.3 Special use of functions

Before proceeding, and discuss how you best can build your own functions. It is worth noting that to use some libraries effectively, you sometimes need to interact with the library via functions. Let us consider a simple example, assume that you have a DataFrame that contains white spaces or some other symbols you would like to remove. Then you can define your own function that does what you want on a single element and use the `apply()` function in Pandas

```
# first create a simple DataFrame
dict={'col1':['aa', ' aa ', 'a b c '],'col2':[1,2,3]}
df=pd.DataFrame(dict)
print(df)
def remove_space(x):# only works for single elements
    return x.strip()
df['col1']=df['col1'].apply(remove_space)
print(df)
```

Note: There already exists a function in Pandas for removing spaces (`df['col1'].str.strip()`), but with the above example you can easily extend the function to do other things.

3.4 Pythons lambda function

The function defined in the example above is relatively small, and Python has a special syntax, that makes it possible to define small functions in one line. The functions can only have one expression. The syntax is `lambda <arguments>: <expression>`. Below are some examples

```
#remove white space
```

```
remove_space=lambda x : x.strip()
remove_space('  a a ')
```

The neat thing is that you can pass a lambda function directly to Pandas `apply()` function, without naming it

```
df['col1'].apply(lambda x: x.strip())
```

Another example from here[1].

```
# check if number is even or odd
result = lambda x : f"{x} is even" if x %2==0 else f"{x} is odd"
print(result(12)) # even
print(result(13)) # odd
```

3.5 Exercise: Create a function from the following code

It is a good exercise to take code you have already written and create one or several functions based on that code.

Question: Create a function from the following code

```
# this code replace space and slash in names
name='16/1-12 Troldhaugen'
chars=[" ", "/"]
new_chars=["","_"]
new_name=name
for ch,nch in zip(chars,new_chars):
    new_name = new_name.replace(ch, nch)
```

```
def replace_chars(name):
    '''
    name: A string
    returns input strings where space is removed and slash is
    replaced with underscore
    '''
    chars=[" ", "/"]
    new_chars=["","_"]
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
name='16/1-12 Troldhaugen'
replace_chars(name) #prints 16_1-12Troldhaugen
```

[1] https://www.geeksforgeeks.org/how-to-use-if-else-elif-in-python-lambda-functions/

Docstring

In the example above we added some text just after `def` statement. This is a *docstring*. A docstring is to tell people what the code does without them having to read the code. If you type `help(replace_chars)`, Python will print out the docstring.

3.6 Making functions more general

In the above example, the function is already quite useful, but if you later decide that you e.g. do not want to remove spaces from names, you would have to write a new function. However we can achieve a more general function, by using *default arguments*.

```
def replace_chars(name,chars=[" ","/"],new_chars=["","_"]):
    '''
    name: A string
    returns input strings where space is removed and slash is
    replaced with underscore
    '''
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
```

Now you can use the same call signature as before `replace_chars(name)`, if you want to only replace `/`, you write

```
name='16/1-12 Troldhaugen'
replace_chars(name,["/"],["_"]) # prints 16_1-12 Troldhaugen
```

Positional arguments

The variable `name` in the function definition of `replace_chars` is called a *positional argument*. On the other hand `chars=[" ","/"]` is called a *default argument*. In Python default argument, must always come *after* positional arguments. Hence, it is not allowed to write

```
def replace_chars(chars=[" ","/"],new_chars=["","_"], name).
```

3.7 Improving robustness of functions

As time goes, you start to forget what a function does, and you can start using it wrong. A typical situation in the above example is that we could mix up the order of `new_chars` and `chars`, i.e. we do `replace_chars(name,["_"],["/"])` instead of `replace_chars(name,["/"],["_"])`, which would give the opposite effect. Python has a very neat syntax to avoid this behavior, by adding a `*` in the argument list

```
def replace_chars(name,*, chars=[" ","/"],new_chars=["","_"]):
    '''
    name: A string
    returns input strings where space is removed and slash is
    replaced with underscore
    '''
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
```

You can still call the function as before `replace_chars(name)`, but if you try to do

```
name='16/1-12 Troldhaugen'
replace_chars(name,["/"],["_"])
```

You will get an error

Warning

```
TypeError: replace_chars() takes 1 positional argument
but 3 were given
```

This error might be a bit hard to interpret, but basically `name` is a positional argument according to the definition and the two next arguments are default arguments. When we use the `*` in the function definition we have to explicitly enter the variable name of the default arguments

```
name='16/1-12 Troldhaugen'
replace_chars(name,chars=["/"],new_chars=["_"])
```

By forcing the user to use `chars` and `new_chars` it becomes less probable to mix them up. This is also why we should try and use variable names like `chars` and `new_chars` that are descriptive.

3.8 Assert, raise and try statements

Still there are plenty of things that can go wrong with our function. In many cases you would like to catch errors as quickly as possible, to help users to discover where the errors occurs. Lets look at an example, lets say we wrongly use our function

```
replace_chars(2)
```

We get

Warning

```
new_name = name
           for ch,nch in zip(chars,new_chars):
---->          new_name = new_name.replace(ch, nch)
           return new_name

AttributeError: 'int' object has no attribute 'replace'
```

Note that the errors only occurs when we call `new_name.replace`, because we used an integer as an argument. An integer does not have a function named `replace()`, but the errors happens earlier because we used the function `replace_chars` wrongly. Thus, it would be much better if we could catch the error at a very early stage and give the user an error message.

3.8.1 Try and Except

Try and except, is actually as simple as it sounds. It is a way to tell Python to try a piece of code, if the piece of code fails, we move to the except statement. Following this a lazy, and crude way of improving our function is to declare our function as

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    try:
        new_name = name
        for ch,nch in zip(chars,new_chars):
            new_name = new_name.replace(ch, nch)
        return new_name
    except:
        print('Something went wrong in replace_chars')

replace_chars(2)# prints Something went wrong in replace_chars
```

Now the code prints out a message if something is not working. There are several drawbacks

1. The code is still running, even if something went wrong, the program should stop with an error message
2. We do not know exactly where something went wrong, it could be that `name` is not a string, but it could also be e.g. that the user enters a different length for `chars` and `new_chars`

3.8.2 Raise

We can raise errors by using the keyword `raise`. The `raise` keyword needs to be followed by a function from the `BaseExceptionclass`[2], typically you might use `raise Exception('Something went wrong in replace_chars')`

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    try:
        new_name = name
        for ch,nch in zip(chars,new_chars):
            new_name = new_name.replace(ch, nch)
        return new_name
    except:
        raise Exception('Something went wrong in replace_chars')

replace_chars(2)# prints Something went wrong in replace_chars
```

Running the following code

```
replace_chars(2)
```

We get

[2] https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Warning

```
 except:
--->    raise Exception('Something went wrong in replace_chars')
     replace_chars(2)

Exception: Something went wrong in replace_chars
```

As compared to before we raise the exception when our function is called and not when the `string.replace()` function is called.

Exceptions should be specific

When raising errors, we should try to be as specific as possible to help the user as much as possible.

Let us extend our function using more specific raises

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    if type(name)!=str:
        raise ValueError('replace_chars: name must be a string')
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
replace_chars(2)# ValueError: replace_chars: name must be a string
```

Exercise: Improve `replace_chars` using raise.

Question: How can we improve `replace_chars` to make sure that the length of `chars` and `new_chars` are of equal length?

```
def replace_chars(name,*,chars=[" ", "/"],new_chars=["","_"]):
    ''' replace Norwegian characters and space in names'''
    if type(name)!=str:
        raise ValueError('replace_chars: name must be a string')
    if len(chars) != len(new_chars):
        raise ValueError('replace_chars: chars and new_chars must same size')
    new_name = name
    for ch,nch in zip(chars,new_chars):
        new_name = new_name.replace(ch, nch)
    return new_name
replace_chars('2',chars=["/"])# ValueError: replace_chars: chars and new_chars must same size
```

3.9 Using assert to test our code

When developing code it is extremely useful to design tests that checks that our code does what it is supposed to do. This is mainly done to make sure that the expected behavior of a function does not changes over time. For our small function we can use the specific keyword assert. The syntax is assert <condition>,<error message>

assert only works in debug mode

If you for some reason compile your code or in other ways turn off the debug option in Python, assert will not work.

```
def test_replace_chars():
    assert replace_chars('  ') == ''
    assert replace_chars('//') == '__'
    assert replace_chars('G O//O D') =='GO__OD'
test_replace_chars()
```

Each time we start working and stop working on our code , we run all tests that we have defined. If nothing fails we know that no one has introduced errors before we start coding, and that we have not made any changes.

4 Classes in Python

4.1 Why classes?

A class can be a way for you to create a clean interface to your code. You have already used a lot of classes. The `DataFrame()` in Pandas is a class, and we access functions inside this class by using a dot (`.`). Classes also provides encapsulation: By wrapping parts of your code into classes, and particular realizations of classes (objects), you facilitate code re-use, and it can make your code easier to understand and work with, thus reducing the probability of introducing bugs which may be hard to track down.

To get started, there are really only a couple of things you need to know. First, all of your classes should include a special function called `__init__`, in which you declare the variables (attributes) you wish an instance / object of the class to keep track of.

Second, when setting, updating, or fetching attributes stored within the class, you should always use the prefix `self`, followed by a dot. Furthermore, the functions you define inside the class should have `self` as the first function argument (there are exceptions[1]), but we will not consider those here.

[1] `https://realpython.com/python3-object-oriented-programming/`

4.2 Example: A class for production data

Earlier in this course we have made functions to read data from an Excel file, and plot the data. This can also be done within a class, below is an example

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class ProdData:
    """
    A class to extract production data from FactPages
    """
    def __init__(self):
        self.df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')

    def get_data(self,field):
        """
        Extracts data for a specific field
        """
        df= self.df_prod[(self.df_prod['Field (Discovery)'] == field)]
        return df

    def plot(self,field,cols=[3,4,5,6,7]):
        """
        Plots the different columns in the DataFrame
        """
        df=self.get_data(field)
        xcol=df['Year']+df['Month']/12
        for col in cols:
            plt.plot(xcol,df.iloc[:,col],label=df.columns[col])
        plt.legend(loc='center', bbox_to_anchor=(0.5,-.3),
          ncol=3, fancybox=True, shadow=True)
        plt.title(field)
        plt.xlabel('Years')
        plt.ylabel('mill Sm$^3$')
        plt.grid()

ff=ProdData()
ff.plot('DRAUGEN')
```

The nice thing about the class is that it has a very nice interface, if the user wants to plot data from another field, it is just give the name of that field

```
ff.plot('EKOFISK')
```

4.3 Example: A class for a mathematical function

A mathematical function should be the perfect example of *when to use a function*, but it turns out that it can be quite convenient to use a class. Here we will consider the mathematical function in the equation below

$$f(x) = \sin(bx) \cdot e^{-ax^2}. \tag{4.1}$$

If we implement this function in a class, we can also add other functionalities to our function such as an ability to plot itself

```
class WavePacket:
    """
    A class representation of a wave packet-function.
    """
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def f(self, x):
        return np.sin(self.b*x)*np.exp(-self.a*x*x)

    def plot(self, x_min=-10, x_max=10, dx=0.01):
        """
        A simple plotting routine for plotting f(x) in some range.
        """
        x = np.arange(x_min, x_max, dx)
        y = self.f(x)
        fig = plt.figure()
        plt.plot(x, y)
        plt.grid()
```

Besides the initialization method and a function that calculates $f(x)$ from equation (4.1), the class includes a simple plotting routine. A major difference from before is the following: when our function $f(x)$ is defined inside a class, we do not have to pass around a and b as arguments to the function `f`. Instead, we simply access a and b from inside the class itself (using the `self`-prefix).

Below is an example of how to use the class:

```
# Create two WavePacket objects, having their own parameter values
WP1 = WavePacket(0.1, 2)  # a=0.1, b=2
WP2 = WavePacket(0.1, 10)  # a = 0.1, b=10

# Evaluate the two functions at a specific point
x = 1
print(WP1.f(x))
print(WP2.f(x))
```

```
# Plot the two functions
WP1.plot()
WP2.plot()
```

Although we had to write slightly more code, we hope you appreciate how easy this makes running parallel simulations with different parameters. Actually, Python provides a way for us to simplify even further, by defining the special `__call__`[2] method for the class:

```
class FancyWavePacket:
    """
    A slightly more fancy class representation of a wave packet-function.

    In this version, we define the dunder (double-underscore) method __call__,
    which lets us treat objects of the class as if they were real functions!
    """
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self, x):
        return np.sin(self.b*x)*np.exp(-self.a*x*x)
```

Compared to the first example of the class, observe that we have replaced the function `f` by `__call__` (with two underscores on both sides of "call"). This way, we we can write our code as if `FancyWavePacket` was a function:

```
WP1 = FancyWavePacket(0.1, 2)  # a=0.1, b=2
WP2 = FancyWavePacket(0.1, 10)  # a = 0.1, b=10

# Evaluate the two functions at a specific point
x = 1
print(WP1(x))  # If WP1 had been a function, the syntax would be the same here!
print(WP2(x))  # Again, we no longer have to type "WP2.f(x)", we can do "WP2(x)".
```

[2] https://www.realpythonproject.com/python-magic-oop-dunder/

5 Some advanced topics

In this chapter we have collected some topics that are slightly more advanced.

5.1 Running Python files from command line

So far we have used notebooks to write Python code. Jupyter notebooks ends with an extension `.ipynb`, and if you open them in a text editor like notepad or emacs it does not look like it does in your web browser. On the other hand you can also write Python code in files that have an extension `.py`, then you would typically use an IDE such as Visual Studio Code. In the `.py` files you cannot write markdown text. Sometimes it can be advantageous to use Python files and not Jupyter notebooks, the notebooks can take time to open and then you have to manually execute the cells you want to run.

It can be very convenient to have some small programs that could do some simple tasks for you, e.g.

1. search for files of a specific type
2. remove files of a specific type
3. remove or print out files that are large
4. download data from the web or a server

The only thing you have to do is to write your code in a suitable editor and save it, if you use Visual Studio Code or Spyder you can of course run the Python file in the editor. Next, you open the anaconda power shell, use `cd` to move where your Python file is located and write

```
Terminal
(base) PS C:\Users\Aksel Hiorth> python <name_of_file>
```

remember to include the `.py` extension. Below are an example

5.1.1 Search for files of a specific type

Put the following content in a file called `find_files.py`

```
#%%
#content of find_files1.py
import pathlib as pt
def find_files_of_type(dir='.',extensions=['.xlsx','.txt','.INC']):
    """
    return a list files of type defined in extensions
    """
    p=pt.Path(dir)
    files=[]
    for ext in extensions:
        for x in p.rglob("*"+ext):
            if x.is_file():
                files.append(x.absolute())
    return files
if __name__ == '__main__':
    files=find_files_of_type()
    for f in files:
        print(f)
# %%
```

Then open the anaconda power shell where this file is and run

```
Terminal
(base) PS C:\Users\Aksel Hiorth> python find_files.py
```

The reason we use the statement `if __name__ == '__main__':` is to only run the code if this is the main file. This means that we can import the file `find_files.py` in another program without running the last part, because then `find_files.py` is not the main program.

When running files from command line it would be useful to give some command line arguments. The `sys` module and the `sys.argv` can be used to access command line argument, the `sys.argv` command returns a list of strings, where the first element is always the name of the program. In the code below we have made it possible to also to give in the directory name of where to start the search.

```
#%%
#content of find_files1.py
import pathlib as pt
def find_files_of_type(dir='.',extensions=['.xlsx','.txt','.INC']):
    """
    return a list files of type defined in extensions
    """
    p=pt.Path(dir)
    files=[]
    for ext in extensions:
        for x in p.rglob("*"+ext):
            if x.is_file():
                files.append(x.absolute())
    return files
if __name__ == '__main__':
    files=find_files_of_type()
    for f in files:
        print(f)
# %%
```

To start the search two directories above

Terminal

```
(base) PS C:\Users\Aksel Hiorth> python find_files2.py ../../
```

5.2 Creating executable programs from Python file

Even if someone has not Python installed on their computer, they could still be interested in your programs. It is actually possible to create executable files from your Python scripts, where all libraries are included in the executable file. Here we show the bare minimum and only the `pyinstaller` library, for more information check out this blog[1].

Install packages: Open anaconda power shell, activate your environment and install `pyinstaler`

Terminal

[1] https://www.blog.pythonlibrary.org/2021/05/27/pyinstaller-how-to-turn-your-python-code-into-an-ex

```
(mod322) PS C:\Users\Aksel Hiorth> conda install pyinstaller
```

Create executable: Next, we must create an executable, run the following command (where your python file is located)

Terminal

```
(mod322) PS C:\Users\Aksel Hiorth> pyinstaller find_files2.py --onefile
```

`pyinstaller` now have created two folder, a `build` folder and a `dist` folder. Inside the `dist` folder there should be a single file called `find_files2.exe`. This file can be shared with other people. You can run this file from command line as before, and it should behave as your python script.

5.3 Passing functions to functions

In many cases, you would also pass a function to another function to make your code more modular. Lets say we want to calculate the derivative of $\sin x$, using the most basic definition of a derivative $f'(x) = f(x + \Delta x) - f(x)/\Delta x$, we could do it as

```
def derivative_of_sine(x,delta_x):
    ''' returns the derivative of sin x '''
    return (np.sin(x+delta_x)-np.sin(x))/delta_x

print('The derivative of sinx at x=0 is :', derivative_of_sine(0,1e-3))
```

If we would like to calculate the derivative at multiple points, that is straightforward since we have used the Numpy version of $\sin x$.

```
x=np.array([0,.5,1])
print('Derivative of sinx at x=0,0.5,1 is :', derivative_of_sine(x,1e-3))
```

The challenge with our implementation is that if we want to calculate the derivative of another function we have to implement the derivative rule again for that function. It is better to have a separate function that calculates the derivative

```
def f(x):
    return np.sin(x)
```

```python
def df(x,f,delta_x=1e-3):
    ''' returns the derivative of f '''
    return (f(x+delta_x)-f(x))/delta_x
print('Derivative of sinx at x=0 is :', df(0,f))
```

Note also that we have put `delta_x=1e-3` as a *default argument*. Default arguments have to come at the end of the argument lists, `df(x,delta_x=1e-3,f)` is not allowed. All of this looks well, but what you would experience is that your functions would not be as simple as $\sin x$. In many cases your functions need additional arguments to be evaluated e.g.:

```python
def s(t,s0,v0,a):
    '''
    t  : time
    s0 : initial starting point
    v0 : initial velocity
    a  : acceleration
    returns the distance traveled
    '''
    return s0+v0*t+a*t*t*0.5 #multiplication (0.5)is general faster
                            #than division (2)
```

How can we calculate the derivative of this function? If we try to do `df(1,s)` we will get the following message

```
TypeError: s() missing 3 required positional
       arguments: 's0', 'v0', and 'a'
```

This happens because the `df` function expect that the function we send into the argument list has a call signature `f(x)`. What many people do to avoid this error is to use global variable, that is to define `s0, v0`, and `a` at the top of the code. This is not always the best solution. Python has a special variable `*args` which can be used to pass multiple arguments to your function, thus if we rewrite `df` like this

```python
def df(x,f,*args,delta_x=1e-3):
    ''' returns the derivative of f '''
    return (f(x+delta_x,*args)-f(x,*args))/delta_x
```

we can do (assuming `s0=0`, `v0=1`, and `a=9.8`)

```python
print('The derivative of sinx at x=0 is :', df(0,f))
print('The derivative of s(t) at t=1 is :', df(0,s,0,1,9.8))
```

5.4 Scope of variables

In small programs you would not care about scope, but once you have several functions, you will easily get into trouble if you do not consider the scope of a variable. By scope of a variable we mean where the variable is available, first some simple examples

A variable created inside a function is only available within the function: “

```
def f(x):
    a=10
    b=20
    return a*x+b
```

Doing `print(a)` outside the function will throw an error: `name ’a’ is not defined`. What happens if we define variable `a` outside the function?

```
a=2
def f(x):
    a=10
    b=20
    return a*x+b
```

If we first call the function `f(0)`, and then do `print(a)` Python would give the answer `2`, *not* `10`. A *local* variable `a` is created inside `f(x)`, that does not interfere with the variable `a` defined outside the function.

The `global` keyword can be used to pass and access variables in functions: “

```
global a
a=2
def f(x):
    global a
    a=10
    b=20
    return a*x+b
```

In this case `print(a)` *before* calling `f(x)` will give the answer `2` and *after* calling `f(x)` will give `10`.

Use of global variables

Sometimes global variables can be very useful, and help you to make the code simpler. But make sure to use a *naming convention* for them, e.g. end all the global variables with an underscore. In the example above we would write `global a_`. A person reading the code would then know that all variables ending with an underscore are global, and can potentially be modified by several functions.

5.5 Passing arrays and lists to functions

In the previous section, we looked at some simple examples regarding the scope of variables, and what happened with that variable inside and outside a function. The examples used integer or floats. However in most applications you will pass an array or a list to a function, and then you need to be aware that the behavior is not always would you might expect.

Unexpected behavior

Sometimes functions do not do what you expect, this might be because the function does not treat the arguments as you might think. The best advice is to make a very simple version of your function and test it for yourself. Is the behavior what you expect? Try to understand why or why not.

Let us look at some examples, and try to understand what is going on and why.

```
x=3
def f(x):
    x = x*2
    return x
print('x =',x)
print('f(x) returns ', f(x))
print('x is now ', x)
```

In the example above we can use `x=3`, `x=[3]`, `x=np.array([3])`, and after execution `x` is unchanged (i.e. same value as before `f(x)`) was called. Based on what we have discussed before, this is maybe what you would expect, but if we now do

```
x=[3]
def append_to_list(x):
    return x.append(1)
print('x = ',x)
print('append_to_list(x) returns ', append_to_list(x))
print('x is now ', x)
```

(Clearly this function will only work for lists, due to the append command.) After execution, we get the result

```
x = [3]
append_to_list(x) #returns  [3 1], x is now  [3, 1]
```

Even if this might be exactly what you wanted your function to do, why does `x` change here when it is a list and not in the previous case when it is a float? Before we explain this behavior let us rewrite the function to work with Numpy arrays

```
x=np.array([3])
def append_to_np(x):
    return np.append(x,1)
print('x = ',x)
print('append_to_np(x) returns ', append_to_np(x))
print('x is now ', x)
```

The output of this code is

```
x =  np.array([3])
append_to_np(x) #returns  [3 1], x is now  [3]
```

This time `x` was not changed, what is happening here? It is important to understand what is going on because it deals with how Python handles variables in the memory. If `x` contains million of values, it can slow down your program, if we do

```
N=1000000
x=[3]*N
%timeit append_to_list(x)
x=np.array([3]*N)
%timeit append_to_np(x)
```

On my computer I found that `append_to_list` used 76 nano seconds, and `append_to_np` used 512 micro seconds, the Numpy function was about 6000 times slower! To add to the confusion consider the following functions

```
x=np.array([3])
def add_to_np(x):
    x=x+3
    return x

def add_to_np2(x):
    x+=3
    return x
print('x = ',x)
print('add_to_np(x) returns ', add_to_np(x))
```

```
print('x is now ', x)

print('x = ',x)
print('add_to_np2(x) returns ', add_to_np2(x))
print('x is now ', x)
```

The output is

```
x =  np.array([3])
add_to_np(x) #returns [6], x is now [3]
x =  np.array([3])
add_to_np2(x) #returns  [6], x is now  [6]
```

In both cases the function returns what you expect, but it has an unexpected (or at least a different) behavior regarding the variable `x`. What about speed?

```
N=10000000
x=np.array([3]*N)
%timeit add_to_np(x)
x=np.array([3]*N)
%timeit add_to_np2(x)
```

`add_to_np` is about twice as slow as `add_to_np2`. In the next section we will try to explain the difference in behavior.

Avoiding unwanted behavior of functions

The examples in this section are meant to show you that if you pass an array to a function, the array can be altered outside the scope of the function. If this is not what you want, it could lead to bugs that are hard to detect. Thus, if you experience unwanted behavior pick out the part of function involving list or array operations and test one by one in the editor.

5.6 Call by value or call by reference

For anyone that has programmed in C or C++ call by reference or value is something one need to think about constantly. When we pass a variable to a function there are two choices, should we pass a copy of the variable or should we pass information about where the variable is stored in memory?

Value and reference

In C and C++ pass by value means that we are making a copy in the memory of the variable we are sending to the function, and pass by reference means that we are sending the actual parameter or more specific the address to the memory location of the parameter. In Python all variables are passed by object reference.

In C and C++ you always tell in the function definition if the variables are passed by value or reference. Thus if you would like a change in a variable outside the function definition, you pass the variable by reference, otherwise by value. In Python we always pass by (object) reference.

5.6.1 Floats and integers

To gain a deeper understanding, we can use the `id` function, the `id` function gives the unique id to a variable. In C this would be the actual memory address, lets look at a couple of examples

```
a=10.0
print(id(a)) #gives on my computer 140587667748656
a += 1
print(id(a)) #gives on my computer 140587667748400
```

Thus, after adding 1 to `a`, `a` is assigned *a new place in memory.* This is very different from C or C++, in C or C++ the variable, once it is created, *always has the same memory address.* In Python this is not the case, it works in the opposite way. The statement `a=10.0`, is executed so that *first* 10.0 is created in memory, secondly `x` is assigned the reference to 10.0. The assignment operator `=` indicates that `a` should point to whatever is on the right hand side. Another example is

```
a=10.0
b=10.0
print(a is b) # prints False
b=a
print(a is b ) # prints True
```

In this case 10.0 is created in two different places in the memory and a different reference is assigned to `a` and `b`. However if we put `b=a`, `b` points to the same object as `a` is pointing on. More examples

```
a=10
```

```
b=a
print(a is b) # True
a+=2
print(a is b) # False
```

When we add 2 to `a`, we actually add 2 to the value of 10, the number 12 is assigned a new place in memory and `a` will be assigned that object, whereas `b` would still points the old object 10.

5.6.2 Lists and arrays

Yous should think of lists and arrays as containers (or a box). If we do

```
x=[0,1,2,3,4]
print(id(x))
x[0]=10
print(id(x)) # same id value as before and x=[10,1,2,3,4]
```

First, we create a list, which is basically a box with the numbers 0, 1, 2, 3, 4. The variable `x` points to *the box*, and `x[0]` points to 0, and `x[1]` to 1 etc. Thus if we do `x[0]=10`, that would be the same as picking 0 out of the box and replacing it with 10, but *the box stays the same*. Thus when we do `print(x)`, we print the content of the box. If we do

```
x=[0,1,2,3,4]
y=x
print(x is y) # True
x.append(10)  # x is now [0,1,2,3,4,10]
print(y)      # y=[0,1,2,3,4,10]
print(x is y) # True
```

What happens here is that we create a box with the numbers 0, 1, 2, 3, 4, `x` is referenced that box. Next, we do `y=x` so that `y` is referenced the *same box* as `x`. Then, we add the number 10 to that box, and `x` and `y` still points to the same box.

Numpy arrays behave differently, and that is basically because if we want to add a number to a Numpy array we have to do `x=np.array(x,10)`. Because of the assignment operator `=` , we take the content of the original box add 10 and put it into a *new* box

```
x=np.array([0,1,2,3,4])
y=x
print(x is y)      # True
x=np.append(x,10) # x is now [0,1,2,3,4,10]
print(y)           # y=[0,1,2,3,4]
print(x is y)      # False
```

The reason for this behavior is that the elements in Numpy arrays (contrary to lists) have to be continuous in the memory, and the only way to achieve this is to create a new box that is large enough to also contain the new number. This also explains that if you use the `np.append(x,some_value)` inside a function where `x` is large it could slow down your code, because the program has to delete `x` and create a new very large box each time it would want to add a new element. A better way to do it is to create `x` *large enough* in the beginning, and then just assign values `x[i]=a`.

5.7 Mutable and immutable objects

What we have explained in the previous section is related to what is known as mutable and immutable objects. These terms are used to describe objects that have an internal state that can be changed (mutable), and objects that have an internal state that cannot be changed after they have been created. Example of mutable objects are lists, dictionaries, and arrays. Examples of immutable objects are floats, ints, tuples, and strings. Thus if we create the number 10 its value cannot be changed (and why would we do that?). Note that this is *not the same as saying that* `x=10` and that the internal state of `x` cannot change, this is *not* true. We are allowed to make `x` reference another object. If we do `x=10`, then `x is 10` will give true and they will have the same value if we use the `id` operator on `x` and `10`. If we later say that `a=11` then `a is 10` will give false and `id(a)` and `id(10)` give different values, but * `id(10)` will have the same value as before*.

Lists are mutable objects, and once a list is created, we can change the content without changing the reference to that object. That is why the operations `x=[]` and `x.append(1)`, does not change the id of x, and also explain that if we put `y=x`, `y` would change if `x` is changed. Contrary to immutable objects if `x=[]`, and `y=[]` then `x is y` will give false. Thus, whenever you create a list it will be an unique object.

A final tip

You are bound to get into strange, unwanted behavior when working with lists, arrays and dictionaries (mutable) objects in Python. Whenever, you are unsure, just make a simple version of your lists and perform some of the operations on them to investigate if the behavior is what you want.

Finally, we show some "unexpected" behavior, just to demonstrate that it is easy to do mistakes and one should always test code on simple examples.

```
x_old=[]
x  = [1, 2, 3]
x_old[:] = x[:] # x_old = [1, 2, 3]
x[0] = 10
print(x_old) # "expected" x_old = [10, 2, 3], actual [1, 2, 3]
```

Comment: We put the *content* of the `x` container into `x_old`, but `x` and `x_old` reference different containers.

```
def add_to_list(x,add_to=[])
    add_to.append(x)
    return add_to

print(add_to_list(1)) # "expected" [1] actual [1]
print(add_to_list(2)) # "expected" [2] actual [1, 2]
print(add_to_list(3)) # "expected" [3] actual [1, 2, 3]
```

Comment: `add_to=[]` is a default argument and it is created once when the program starts and not each time the function is called.

```
x = [10]
y = x
y = y + [1]
print(x, y) # prints [10] [10, 1]

x = [10]
y = x
y += [1]
print(x, y) # prints [10, 1] [10, 1]
```

Comment: In the first case `y + [1]` creates a new object and the assignment operator `=` assign `y` to that object, thus `x` stays the same. In the second case the `+=` adds `[1]` to the `y` container without changing the container, and thus `x` also changes.

6 Exercises

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.
- Define your own functions
- Catch errors
- Create your own classes

6.1 Before you start

In some of these exercises we are going to work with files and folders. It is very important to not spread files and folders everywhere on the computer. My suggestion would be to create a new folder inside the `exercise` folder called `src`, `src` is used to indicate source code, as illustrated in figure 6.1.

If you have a Python file inside the `src` folder and wants to access e.g. an Excel file in the `data` folder, we do that by using *relative paths*. A double dot `..` is to step up relative to where you are standing, a slash `/` followed by a name indicates a directory or file, e.g. the `draugen.txt` file can be accessed from the `src` directory by `../data/draugen.txt`. You can also use the Pathlib library to check if everything went ok

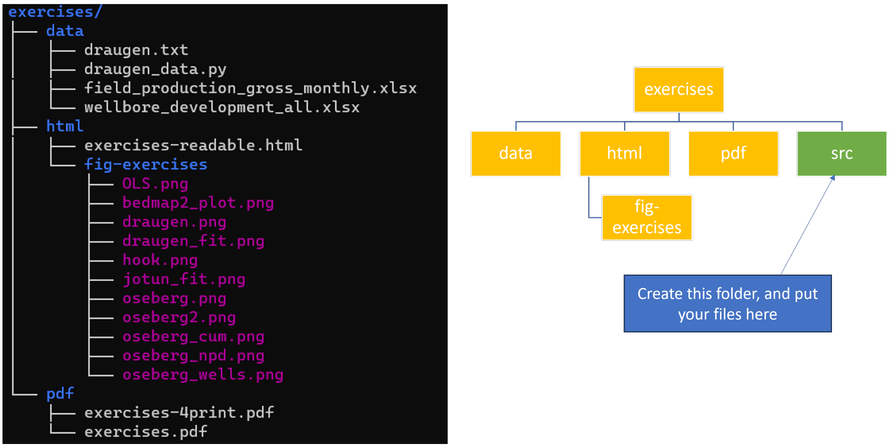


Fig. 6.1 Folder tree structure.

```
import pathlib as pt
p=pt.Path('../data/draugen.txt')
print(p.exists()) # will print True if yes
```

6.2 Exercise 1: Install Bedmap to visualize Antarctica ice data

6.2.1 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 [4]. 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()
```

- Run the code and reproduce figure 6.2, modify the code to plot the ice surface or the bed rock.

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

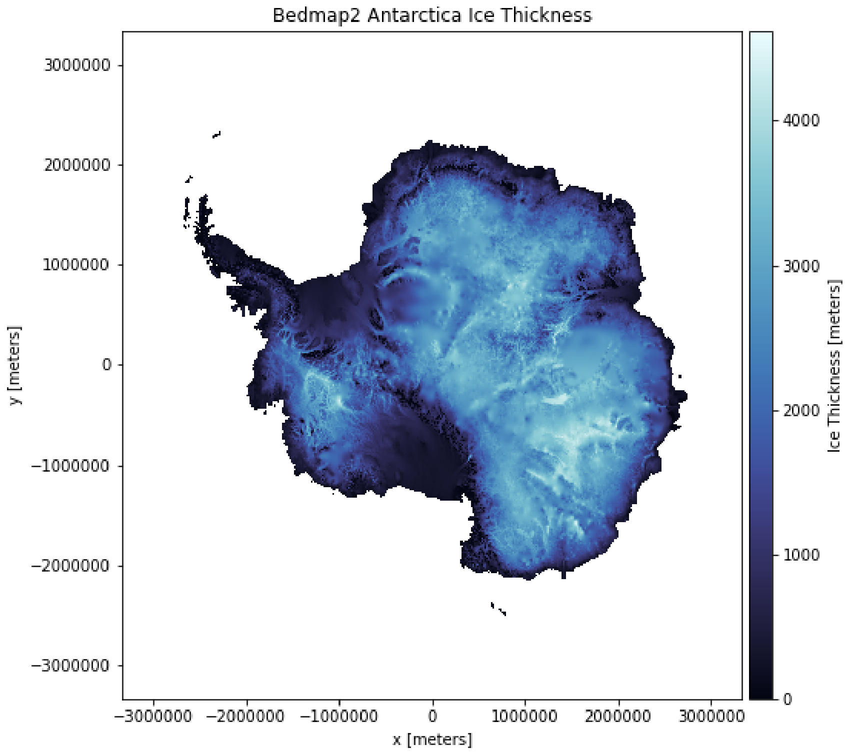


Fig. 6.2 Visualization of the ice thickness in Antarctica.

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

6.3 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 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
```

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

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)
```

> **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*

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

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 6.3

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.

```
# 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
```

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

- 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 6.3

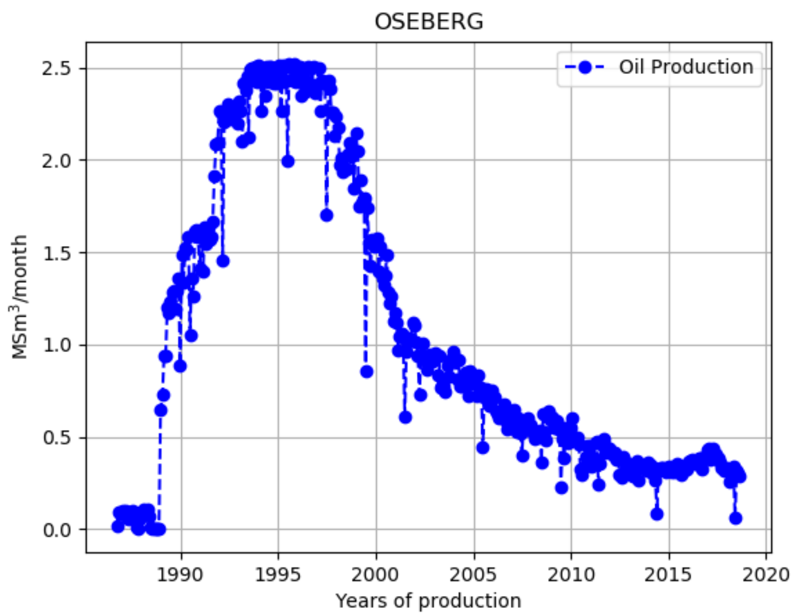


Fig. 6.3 Production of oil equivalents on the Draugen vs time.

6.4 Exercise 3: Group data

If you compare the plot in figure 6.3 with the official plot[6], and for convenience illustrated below in figure 6.4, you will see that they are different. The difference is that the production in figure 6.3 is per month, whereas in figure 6.4 the production is per year.

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.

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

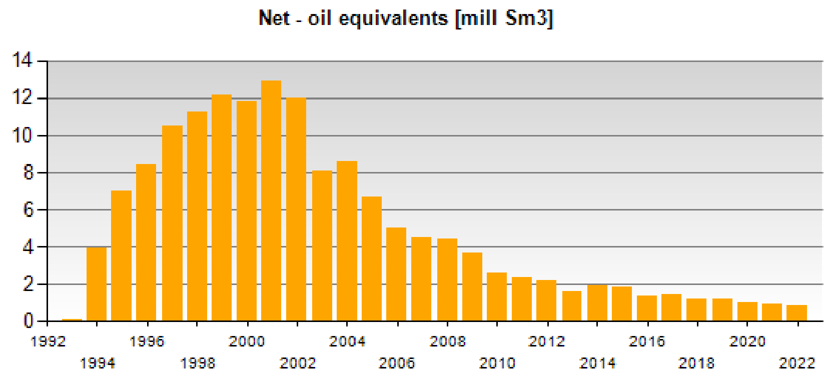


Fig. 6.4 Production of oil equivalents on the Draugen field.

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.

6.5 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].

[7] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html
[8] https://factpages.sodir.no/

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 :

6.5.1 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

6.5.2 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?

6.5.3 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))
```

6.6 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
```

6.7 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)
```

6.8 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)
```

6.9 Exercise 8: Create a function for extracting data

Question: Create a function from the following code. The function should take as argument the field name and return a DataFrame with field data. Include a docstring in the function.

```
df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')
df=df_prod[df_prod['Field (Discovery)'] == 'DRAUGEN']
```

```
def get_data(field):
        """
        Extracts data for a specific field
        """
    df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')
        df= df_prod[df_prod['Field (Discovery)'] == field]
        return df

# example of use
df=get_data('DRAUGEN')
```

6.10 Exercise 9: Improve the previous function

To check if a DataFrame, `df`, contains any data, we can use `df.empty`. If `df.empty` is `True`, there are no data in the DataFrame.

Question: Extend the previous function, by using `df.empty` to give a message if no data are available for a field.

```
def get_data(field):
        """
        Extracts data for a specific field
        """
    df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')
        df= df_prod[df_prod['Field (Discovery)'] == field]
    if df.empty:
       print('No data for ', field)
        return df

# example of use
df=get_data('draugen')
```

6.11 Exercise 10: More improvements

Question: Extend the previous function such that it is case insensitive, i.e. that `get_data('draugen')` would actually return data.

```
def get_data(field):
        """
        Extracts data for a specific field
        """
    df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')
    field = field.upper()
        df= df_prod[df_prod['Field (Discovery)'] == field]
    if df.empty:
       print('No data for ', field)
        return df

# example of use
df=get_data('draugen')
```

6.12 Exercise 11: Increase speed

The previous function is a bit slow, because for each field we read the big Excel file `field_production_gross_monthly.xlsx` each time. It is much better to read it only once.

Question: Read the data outside the function and pass the production DataFrame, using a default argument.

Solution: Variables defined outside a function in Python is considered as global, thus you do not send them as argument, but it is generally considered bad coding practice to use global variables, hence

```
df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')

def get_data(field,df_prod=df_prod):
        """
        Extracts data for a specific field
        """
    field = field.upper()
        df= df_prod[df_prod['Field (Discovery)'] == field]
    if df.empty:
       print('No data for ', field)
        return df
```

```
# example of use
df=get_data('draugen')
df2=get_data('ekofisk')
```

6.13 Exercise 12: Encapsulate in a class

In the previous exercise we had to use a global variable `df_prod` and then pass it to our function. In many cases it might be much easier to use a class.

Question: Create a class:

1. where the `__init__` function reads in the data
2. and add a class function that returns data for a given field

```
class ProdData:
    """
    A class to extract production data from FactPages
    """
    def __init__(self):
        self.df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')

       def get_data(self,field):
        """
        Extracts data for a specific field
        """
        field=field.upper()
        df= self.df_prod[(self.df_prod['Field (Discovery)'] == field)]
        if df.empty:
             print('No data for ', field)
        return df
ff=ProdData()
```

Note that the class contains all the data, we do not have to use a global variable

```
#example of use
df=ff.get_data('draugen')
df2=ff.get_data('ekofisk')
```

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] David Thomas and Andrew Hunt. *The Pragmatic Programmer: Your Journey to Mastery.* Addison-Wesley Professional, 2019.

[4] 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