Introduction to Practical Applications

Aksel Hiorth

University of Stavanger

Dec 20, 2023

Contents

1 Why Python is so popular: The import statement

It is always hard decide which coding language to learn, it usually depends on what you want to do. If you want to do very fast numerical calculations Fortan or C used to be the most popular languages, and for web or application programming Java. However, in recent years Python has become more and more

popular, one of the reasons is its large amount of libraries and communities. If you have an idea of what you want to do, you can almost be certain that there exist a Python library written for that purpose. In the next chapters we will cover some advanced operations in Python and use them to motivate to learn more about the basic operations.

Which library to use?

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

2 Matplotlib: Basic plotting in Python

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

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

Let us go through each line

1. `import matplotlib.pyplot as plt` this line tells Python to import the matplotlib.pyplot[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.

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

3 Data structures (Basic)

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

3.1 Lists

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

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

Notice.

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

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

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

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

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

print statement.

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

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

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

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

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

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

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

Solution.

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

3.2 Dictionaries

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

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

This could be represented as a dictionary as

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

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

Example: Add numbers to dictionary and plot.

- Add two numbers to `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'],'*')
```

3.3 Tuples

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

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

Tuples vs Lists.

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

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

4 Numpy: Working with numerical arrays in Python

In the above example we used lists to store values that we wanted to plot. Lists are one of the basic data structures in Python, but since they are so flexible they are not well suited for mathematical operations. If you are only working with arrays that contain numbers you should use the Numpy[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
x=np.array([1,4,7]) # now x is a Numpy array
x+x # would give [2,8,14]
x*x # would give [1,16,49]
x/x # would give [1.,1.,1.]
```

5 Boolean masking

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

```
x=np.array([4,5,7,8,9]) #create numpy array from a list
```

[3] `https://numpy.org/`

```
print(x>5) # [False, False, True, True, True]
print(np.sum(x>5)) # 3
x[x>5] # [7,8,9]
```

6 Pathlib: Working with files and folders in Python

When you want to open a file in a Python script, you first have to locate it. If you have downloaded all the files, there will be a `data` folder in which there are several data sets, we look at some well data from GeoProvider[4]. Usually we have files and folder located at different places on the computer. This could be Excel files containing different types of information, and you might want to combine data from them. You could of course open them separately, copy and paste data manually. However, it is so much easier to use Python. Let us start with the following code that will list all sub directories and files in a folder

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

Let us go through each line

1. `import pathlib as pt` as before, imports a useful library, in this case Pathlib which we name `pt`. Functions in Pathlib is accessed with the `.` syntax.

2. `p=pt.Path('.')` we create a Path object and name it `p`. We will return to objects and classes later, for now you can think of it as a variable that may contain data and functions that can be used to manipulate

References

[4] `http://geoprovider.no/`

Index