Libraries in Python: os, numpy, csv, re

Tino Cestonaro

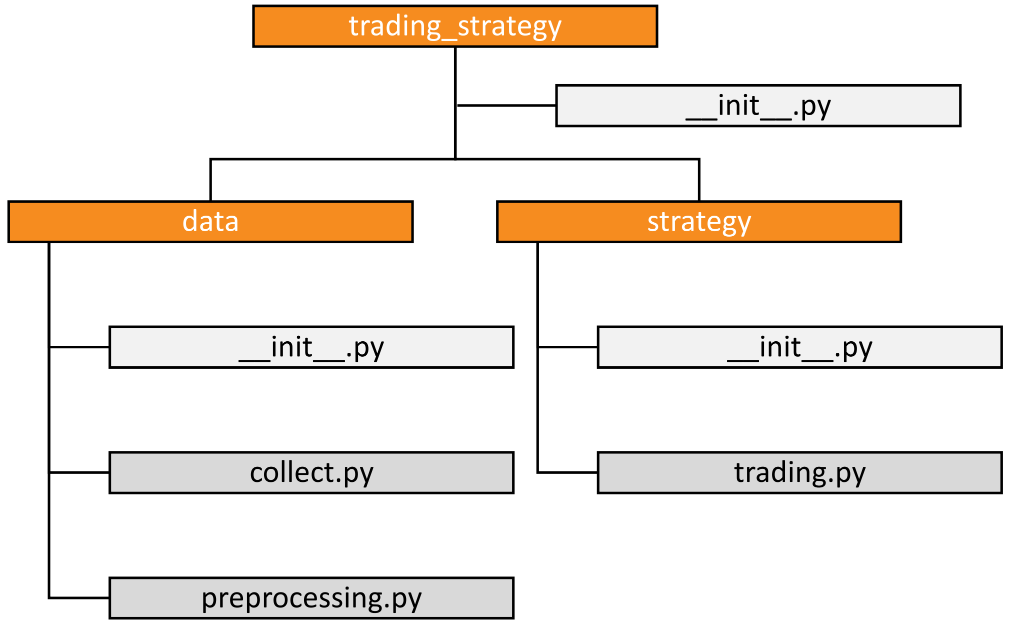
Table of contents

Recap – day 1, outlook – day 2

- Introduction into basics of Python, i.e., data types and structures, control structures, and functions
- Python is also (particularly) very powerful in light of its library coverage
- Today, we will dive into libraries that we consider to be useful when analzying data
    1. Basic libraries that make life in Python easier
    2. Use Python for mathematical, e.g., mean, and database-related operations, e.g., join data tables, when working with data
    3. Import and visualize data with Python in an automated way
    4. Sum up the gain knowledge on Python libraries and give you some ideas how to use these tools in a Data Science project

Some basics on libraries in Python

- For instance, previously we defined our **own functions** for computing the mean and variance of a list of integers → **error prone** and **slow**.
- Instead: use code **from other (professional) developers** that is heavily **tested** and (usually) runs much **faster** and **more stable** than user-written code
- Python community provides:
  - **Modules** – a python file (.py) that contains related code, here: *collect.py*
  - **Packages** – collection of modules, here: *trading_strategy*
  - **Libraries** – collection of packages



Manage libraries using pip

- Python libraries can be managed using the **package manager pip**
- You can access pip using the terminal in your editor and the command pip or (when using anaconda Spyder) via **conda**
- Some important **pip commands:**

```
# This is your terminal
# List all installed packages
>> pip list
Package                 Version
------------            ----------
docopt                  0.6.2
idlex                   1.13
jedi                    0.9.0
```

```
# Install the package numpy
>> pip install numpy
```

```
# Uninstall the package numpy
>> pip uninstall numpy
```

```
# Update the package numpy
>> pip install --upgrade numpy
```

Import libraries (1)

- After you installed the library, it is available in your virtual environment and **you can apply the classes and functions of the library** in your own script
- To use the components of the package, you have to **import the libraries** you want to use in a script, usually this is done on top of the script:

```
# This is your .py script or .ipynb file
import numpy
lst = [2, 7, 13, 99]
result = numpy.std(lst) # calc. std. deviation
print(result)
>> 39.8834
```

- If the name of the package is followed by **as**, then the **name following `as` is bound directly to the imported package** (here: name np is bound to the package numpy)

```
import numpy as np
result = np.std([2, 7, 13, 99])
print(result)
>> 39.8834
```

Import libraries (2)

- It is also possible to import individual names that a library defines (i.e. the functions, variables and classes)
- To import only selected names of the library, you start with `from` then **the name of the package** and then the individual name you want to import:

```
from numpy import std
result = std([2, 7, 13, 99])
print(result)
>> 39.8834
```

- Depending on the library structure it may be required to get one level deeper

```
from numpy.random import shuffle
```

- Note that you can also combine the `from` and `as` statement

```
from numpy import std as bestimme_die_std
result = bestimme_die_std([2, 7, 13, 99])
print(result)
>> 39.8834
```

numpy package structure

```
numpy/                # top-level package
    __init__.py       # def. of fct. std
    random/           # subdirectory
        __init__.py   # def. of shuffle
        …
    …
```

Import user-generated modules

- It is highly recommended to structure a Python project in a meaningful way using submodules and subfolders
- To import **user-generated (sub)modules** we proceed in the same way as before. Assume that **we are in the *main.py*** file and want to **import variables** (x1, x3) **and functions** (fct1) from ***analysis.py***

```
from analysis import x1, x3, fct1
result = fct1(x1, x3)
print(result)
>> 4
```

- Given the project structure, it is required to get one level deeper to get functionality from ***clean_data.py*** into ***main.py***

```
from data.clean_data import clean_list
print(clean_list)
>> ['DAX30', 'CAC40', 'S&P500']
```

myproject structure

```
myproject/            # top-level package
  __init__.py
  main.py
  analysis.py         # contains x1,x3,fct1
  data/               # subfolder
    __init__.py
    get_data.py
    clean_data.py     # contains clean_list
```

Table of contents

Interact with your OS using Python: the os library

- The package **os** provides dozens of functions for interacting with the operating system
- For example, **os** allows you to:
  - **Navigate** through your system
  - **Create/delete/move files** and **directories**
  - Get **system wide information** such as RAM, CPU usage, etc.
- Some basics:

```
import os

print(os.getcwd())              # displays the current working directory
>> "/home/example/path"

path = "/home/tino/data"        # or r"C:\Users\tino\data" under Windows
os.chdir(path)                  # changes the current working directory
print(os.getcwd())
>> "/home/tino/data"
```

os - Automatic generation of multiple folders

- **E.g., instead of** using your **GUI**, e.g., Windows Explorer, to create a folder by using your mouse and/or keyboard, you can use Python for this task.
- Instead of creating a new folder yourself, you can use Python to to **create 10 folders in your current working directory** that are named folder_0, folder_1, …, folder_9:

```
for i in range(10):
    os.mkdir("folder_"+str(i))
```

- You can remove the created folders using

```
for i in range(10):
    os.removedirs("folder_"+str(i))
```

Automatically navigate through your OS (1)

- Given a data science problem, the os package is very helpful when working with multiple (local) data sources and files
- With the os package, you can iterate **through all files and folders in a given directory** and, e.g. automatically open the files (*later*)

```
path = "/home/tino/data"            # or r"C:\Users\tino\data" under Windows
path_content = os.listdir(path)     # list of all dirs and files in path

for f in path_content:              # iterate trough all files and folders in path
   joined = os.path.join(path, f)   # join various path components
   print(joined)                    # print all file and folder names in path

>> "/home/tino/data/customer1.csv"
>> "/home/tino/data/customer2.csv"
>> "/home/tino/data/deeper_dir"
>> "/home/tino/data/deeper_dir2"
```

/home/tino/data

```
data/
    customer1.csv
    customer2.csv

    deeper_dir/
       customer3.csv
       customer4.csv

    deeper_dir2/
       customer5.csv
```

Automatically navigate through your OS (2) - Outlook

- Similarly, you can use `os` to iterate **through <u>all files</u> of the path and its subdirectories**

```
path = "/home/tino/data"            # or r"C:\Users\tino\data" under Windows
path_content = os.walk(path)        # list of all dirs and files in path

for root, dirs, files in path_content:      # walk the tree
   for name in files:
       joined = os.path.join(root, name)    # join various path components
       print(joined)                        # print all filenames in path+depper

>> "/home/tino/data/customer1.csv"
>> "/home/tino/data/customer2.csv"
>> "/home/tino/data/deeper_dir/customer3.csv"
>> "/home/tino/data/deeper_dir/customer4.csv"
>> "/home/tino/data/deeper_dir2/customer5.csv"
```

/home/tino/data

```
data/
    customer1.csv
    customer2.csv

    deeper_dir/
       customer3.csv
       customer4.csv

    deeper_dir2/
       customer5.csv
```

Table of Contents

NumPy: The math library

- A main task in data analysis is **calculus** and **statistics**.
- Python itself is already quite powerful, since you can turn mathematical algorithms into code.
- More sophisticated mathematical operations require a lot of code and are often highly standardizable → **Numerical Python** (**NumPy**) Library, a package for linear algebra and statistics.
- Some of the routines are computationally intensive and thus, optimized code from a package is favorable to user-written code (in terms of reliability, very well tested, very fast, etc.).

NumPy: The ndArray for data representation

- Similar to python lists
- Stores scalars of multiple dimensions:
  - Scalar: 2.25
  - Vector:[1,2,3]
  - Matrix: [[1,2,3], [4,5,6]]
- When creating a NumPy Array, **every element must be of the same type** (different to Python list), e.g., do not mix strings and integers.
- Fast mathematical computations are possible with the array, between arrays and on the elements of the arrays.
- Callable functions from the package at your hand!

NumPy: Python list vs. ndArray

- Initialize a python list and a np.array to create a vector in numpy. The content should be the integers 1,2 and 3.

```
import numpy as np
lst = [1,2,3]
myvector = np.array(lst)  # alternatively: np.array([1,2,3])
print(myvector.shape)
>> (3,)
```

- Initialize a multidimensional array (matrix) with 3x3 elements.

```
mymatrix = np.array([[1,2,3],
                     [4,5,6],
                     [7,8,9]])
print(mymatrix.shape)
>> (3,3)
```

- Default datatype of numpy arrays is int32, a very basic datatype. This is made so computational efforts are reduced to a minimum.

Navigating elements in numpy arrays

- We can access an array by **using square brackets**
- When you're accessing elements, remember that **indexing** in Python **starts at 0**

```
mymatrix = np.array([[1,2,3],
                     [4,5,6],
                     [7,8,9]])

print(mymatrix[0])     # print first row
>> [1 2 3]

print(mymatrix[:,1])   # print second column
>> [2 5 8]

print(mymatrix[2,1])   # value in last row and second column
>> 8
```

Vector and matrix operations

- Numpy offers a huge variety of useful functions for mathematical operations that you can use not only on numpy arrays, but also on other data classes, such as lists.
- The most basic operations are addition, subtraction, multiplication, and division

```
v1 = np.array([3,5,8])
v2 = np.array([4,1,1])

print(v1*v2)
>> [12,5,8]
print(np.dot(v1,v2))
>> 25
```

```
m1 = np.array([[1,1,1], [2,5,9],
               [3,5,8]])
m2 = np.array([[4,1,1], [0,2,3],
               [3,0,2]])

print(m1*m2)
>> [[4,1,1],[0,10,27],[9,0,16]]
print(np.dot(m1,m2))
>> [[7,3,6],[35,12,35],[36,13,34]]
```

Calculating the mean

- In the chapter on functions, we had a look at the mean and variance of lists. We can (and should) use **numpy functions** for this!
- See how we can use these two functions with numpy arrays:

```
mymatrix = np.array([[1,1,1],[2,5,9],[3,5,8]])
```

```
aver = np.mean(mymatrix)
aver_c = np.mean(mymatrix, axis=0)
aver_r = np.mean(mymatrix, axis=1)

print(aver)
>> 3.89
print(aver_c)
>> [2.00, 3.67, 6.00]
print(aver_r)
>> [1.00, 5.33, 5.33]
```

```
stdabw = np.std(mymatrix)
stdabw_c = np.std(mymatrix, axis=0)
stdabw_r = np.std(mymatrix, axis=1)

print(stdabw)
>> 2.88
print(stdabw_c)
>> [0.82, 1.89, 3.56]
print(stdabw_r)
>> [0.00, 2.87, ,2.05]
```

Exercises

1. Download the file numpy_data.zip and extract the data such that you have a directory: **/your/path/to/numpy_data/** that only contains
   - *customer1.csv, customer2.csv, customer3.csv*
2. Use a basic (text) editor to open the csv-files to **get an understanding about the data** you're dealing with
3. Iterate through the directory and read all 3 files using ***np.genfromtxt(filepath, delimiter=',')***
   - Hint: **Skip all non-numeric entries** (first row and first column) **using indexing**
4. Calculate the **mean and standard deviation** of fruit consumption **per customer across the days** and print it
5. [Bonus] Calculate the **mean and standard deviation** of the **aggregated fruit consumption of all customers** and print it
   - Hint: Initialize a matrix with zeros and of shape (5,3) and add up each customer consumption

Table of Contents

Reading and writing data to external files

- The previous section on os mentioned “opening a file”.
- This section will look at basic methods, i.e., the csv library, that can accomplish the task of reading from and writing to files.
- We want to write the following test data to a csv file:

```
s_test_string = "this is a test"
data = [(i,j) for i,j in enumerate(s_test_string.split())]
print(data)

>> [(0, 'this'), (1, 'is'), (2, 'a'), (3, 'test')]
```

Write to file

- Import the built-in package **csv**
- **Open a file** with the name “test.csv” and set mode to **writing** (“w”)

```
import csv
with open ("test.csv", "w", newline="") as testFile:
    writer = csv.writer(testFile, delimiter=",")
    for row in data:
        writer.writerow(row)
```

Write to file

- Define a csv writer that writes data to “test.csv” using a comma as **delimiter** between entries in a row

```
with open ("test.csv", "w", newline="") as testFile:
    writer = csv.writer(testFile, delimiter=",")
    for row in data:
        writer.writerow(row)
```

Write to file

- For each element in our artificially created `data` object, we write a row to the test.csv file.

```
with open ("test.csv", "w", newline="") as testFile:
    writer = csv.writer(testFile, delimiter=",")
    for row in data:
        writer.writerow(row)
```

- **Open** the created csv-file using an editor of your choice, the output looks like this:

```
0,this
1,is
2,a
3,test
```

Read csv files

- Similar to writing data to a file, you can use Python to **read** (“r”) from a file, i.e., storing a tuple of each row to the list `output`

```
with open("test.csv", "r") as testFile:
    reader = csv.reader(testFile, delimiter=",")
    output = []
    for row in reader:
        output.append(tuple(row))

print(output)
>> [('0', 'this'), ('1', 'is'), ('2', 'a'), ('3', 'test')]
```

Table of Contents

Working with strings: Regular Expressions

- RegEx useful to **find pre-defined patterns in strings**
- For instance, find all single integers in a string:

```
import re
re.findall(pattern="[0-9]", string="1 plus 1 yields 2")
```

- Returns ['1', '1', '2']
- `pattern` is defines what to look for in a string
- This uses **RegEx specific language**, e.g., [0-9] means all integers 0,1,…,9.
- Starting point – RegEx editor: https://regex101.com/ → see “Quick Reference” for a *very* extensive list of RegEx definitions

Cont’d

- Compare these two expressions using the + operator: **one or more**

```
re.findall(pattern="[0-9]", string="4 plus 8 yields 12")
re.findall(pattern="[0-9]+", string="4 plus 8 yields 12")
```

- The first gives you ['4', '8', '1', '2'], while the second ['4', '8', '12']
- Further real-word examples (**pattern extraction**):
  - Find all **URLs** in a string:

    ```
    re.findall("(www[^ ]+)", "my webpage is: www.example.com")
    ```

    `[^ ]` means: every character except for white space
  - Find all **dates** in a string that has a specific, unified format as given, e.g., in file names:

    ```
    re.findall("([0-9]{4,4})-([A-Za-z]+)-([0-9]{2,2})", "2018-Oct-
    22-HAL.N-140579834485-Transcript")
    ```

    year: matched length has to be four

Cont’d

- Compare these two expressions using the + operator: **one or more**

```
re.findall(pattern="[0-9]", string="4 plus 8 yields 12")
re.findall(pattern="[0-9]+", string="4 plus 8 yields 12")
```

- The first gives you ['4', '8', '1', '2'], while the second ['4', '8', '12']
- Further real-word examples (**pattern extraction**):
  - Find all **URLs** in a string:

```
re.findall("(www[^ ]+)", "my webpage is: www.example.com")
```

`[^ ]` means: every character except for white space

  - Find all **dates** in a string that has a specific, unified format as given, e.g., in file names:

```
re.findall("([0-9]{4,4})-([A-Za-z]+)-([0-9]{2,2})", "2018-Oct-
22-HAL.N-140579834485-Transcript")
```

year: only match letters from A to Z – either capitalized (**A**) or uncapitalized (**a**)
minimum length of matched string is 1 (**+**)

Cont’d

- Compare these two expressions using the + operator: **one or more**

```
re.findall(pattern="[0-9]", string="4 plus 8 yields 12")
re.findall(pattern="[0-9]+", string="4 plus 8 yields 12")
```

- The first gives you ['4', '8', '1', '2'], while the second ['4', '8', '12']
- Further real-word examples (**pattern extraction**):
  - Find all **URLs** in a string:

```
re.findall("(www[^ ]+)", "my webpage is: www.example.com")
```

`[^ ]` means: every character except for white space

  - Find all **dates** in a string that has a specific, unified format as given, e.g., in file names:

```
re.findall("([0-9]{4,4})-([A-Za-z]+)-([0-9]{2,2})", "2018-Oct-
22-HAL.N-140579834485-Transcript")
```

day: matched length has to be two

Cont'd

- Compare these two expressions using the + operator: **one or more**

```
re.findall(pattern="[0-9]", string="4 plus 8 yields 12")
re.findall(pattern="[0-9]+", string="4 plus 8 yields 12")
```

- The first gives you ['4', '8', '1', '2'], while the second ['4', '8', '12']
- Further real-word examples (**pattern extraction**):
  - Find all **URLs** in a string:

```
re.findall("(www[^ ]+)", "my webpage is: www.example.com")
```

    `[^ ]` means: every character except for white space

  - Find all **dates** in a string that has a specific, unified format as given, e.g., in file names:

```
re.findall("([0-9]{4,4})-([A-Za-z]+)-([0-9]{2,2})", "2018-Oct-
22-HAL.N-140579834485-Transcript")
```

    → finally gives you the following result: [('2018', 'Oct', '22')]

Outlook re

- Purpose: introduce idea of RegEx and how it can be used
- For specific case and without much experience:
  - Look at online regex tables
  - Trial and error
  - Google for it
- We cannot teach all variants of RegEx ;)

References

- Importing Libraries
  - https://docs.python.org/3/tutorial/modules.html

- Built-in libraries
  - https://docs.python.org/3/library/os.html
  - https://docs.python.org/3/library/csv.html
  - https://docs.python.org/3/library/re.html
  - https://regex101.com/

- NumPy
  - https://numpy.org/doc/stable/index.html