5 – Basic libraries in Python

Introduction to Python
efl Data Science Courses

Timo Schäfer

Table of Contents

Recap from day 1 – where are we and where will we go to?

- Introduction into basics of Python, i.e., data structures/types, 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 **functions from libraries** that are heavily **tested** and (usually) run much **faster** and **more stable** than user-written functions.
- Libraries are usually downloaded via **pip** or (when using anaconda Spyder) via **conda.**
- You have to tell Python which libraries you want to use in a script, i.e.:

```
import exampleLibrary
import exampleLibrary as eL
```

Table of Contents

Interact with your OS using Python: **os**

- **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.

```
import os
print(os.getcwd())
```

- `import os` shows Python that your code **includes** the **library** `os`
- `os.getcwd()` shows the current working directory where you Python code runs

Automatic generation of multiple folders

- Instead of creating a new folder yourself, you can use Python:

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

- This script uses the syntax you know from day 1, but additionally uses the os library to create 10 folders that are named folder0, folder1, …, folder9.

Automatically navigate through your OS

- Similarly, you can use `os` to navigate through many folders and automatically open a file (*later*) an navigate to the next folder.

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

- `os.chdir`: change current directory
- `os.chdir("..")`: go one step back in folder hierarchy

Outlook **os**

- I often use os to iterate through files in a given directory:

```
os.listdir(os.getcwd())
```

- There are many more useful applications where the usage of `os.system` is very powerful, which allows you to operate (in most cases seamlessly) with your OS (at least in Unix os):

```
os.system("mv *.txt newDirectory/")
```

- In Windows:

```
os.move("inputData.txt", "folder_0/inputData.txt")
```

Remove folders again

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

Explore **os** and other libraries yourself, e.g.,

- Show all available methods for os:

```
dir(os)
```

- For a given method, call the help description:

```
help(os.remove)
```

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:

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

Write to file

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

```
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)
```

The output is as follows

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

- 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))
```

- Output yields : [('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 ;)

Exercises

1. Create folders test1,…,test100 and create files named test1.txt,...,test100.txt in the respective folder
2. Open each of the files and write the *i*-th element from the list `inputData.txt` to it, see code skeleton.
3. On the previous slide, we extracted the date of each file name. Do the same now for the content of each file test`i'.txt and – if time permits – convert string month to int month, e.g., May -> 5. Store the result in a dictionary named `output`, where the key is the *i*-th row (file name) and the value the numeric date.
4. Delete all the files (and folders) you created.

References

- 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/