Python from Zero

Modules Part II

Recap

- Much of Python's versatility comes from *modules*
- Custom modules are a good way to organize larger projects

Use modules with the `import` statement (usually at the top of your script)

```
import json
import numpy as np
from my_module import useful_function
```

Recap

Numpy

One of the most important third party libraries for Python

- Great for handling (large) numerical data, fast and efficient

Main class: `numpy.ndarray`

- Can be a single value, a 1-dimensional list, 2-dimensional matrix, and even more dimensions
- Fixed size, single data type (e.g. `np.int`)

Recap

Numpy

```
import numpy as np

np.array([1,2,3])                          # create a 1-dim array from a list
np.zeros(shape=(3,2))                      # create a 2-dim array filled with zeros (3 rows, 2 columns)
np.full(shape=(2,3,2), fill_value=3.14)    # create a 3-dim array filled with the value 3.14
np.ones(shape=(3))                         # create a 1-dim array filled with ones
```

It's Your Turn!

- Open the notebook `Session_3.ipynb` and start reading "A short introduction to Numpy"

Do the exercise!

Exercise 3:

Practice using Numpy! Create a 3-dimensional Numpy array with zeros of the shape `(3,2,4)`.

Replace the first *2 x 4* matrix with an array of the same shape (`(2,4)`), containing only twos.

In the second *2 x 4* matrix, write all numbers from 0 to 7.

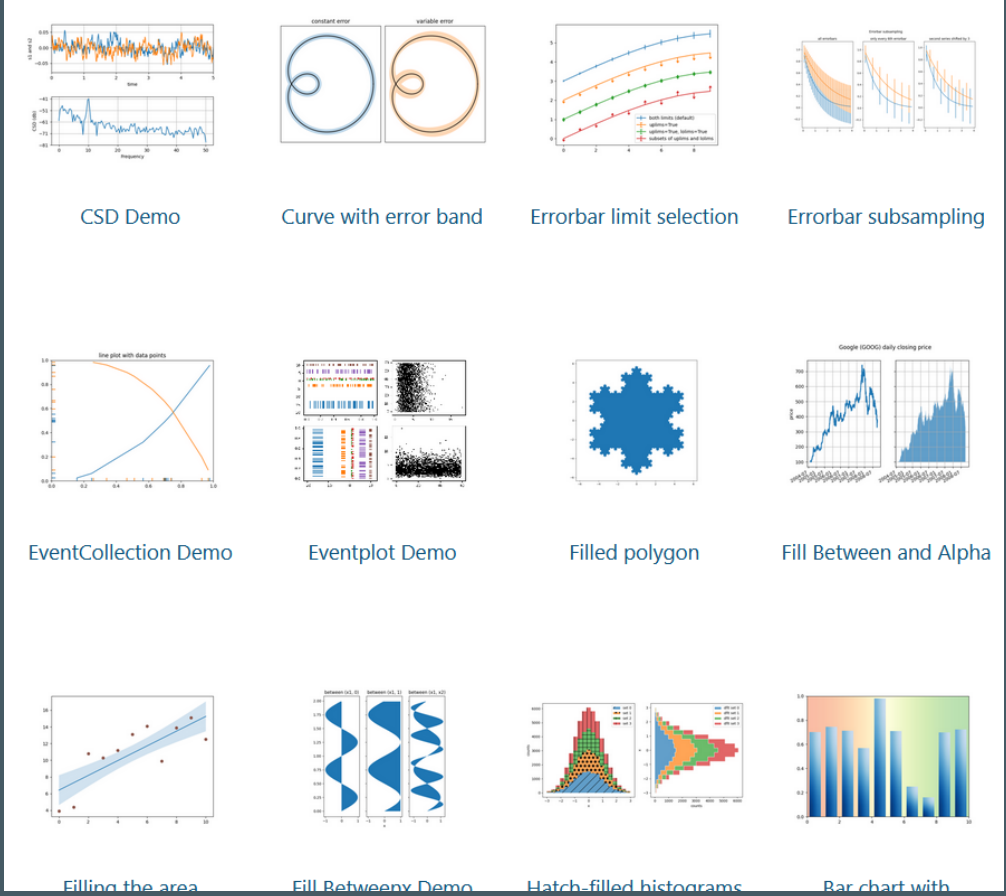
In the third *2 x 4* matrix, store the result of the element-wise multiplication of the first two matrices

Numpy

See it in action on the Notebook...

Matplotlib

Most important module when it comes to data vizualization

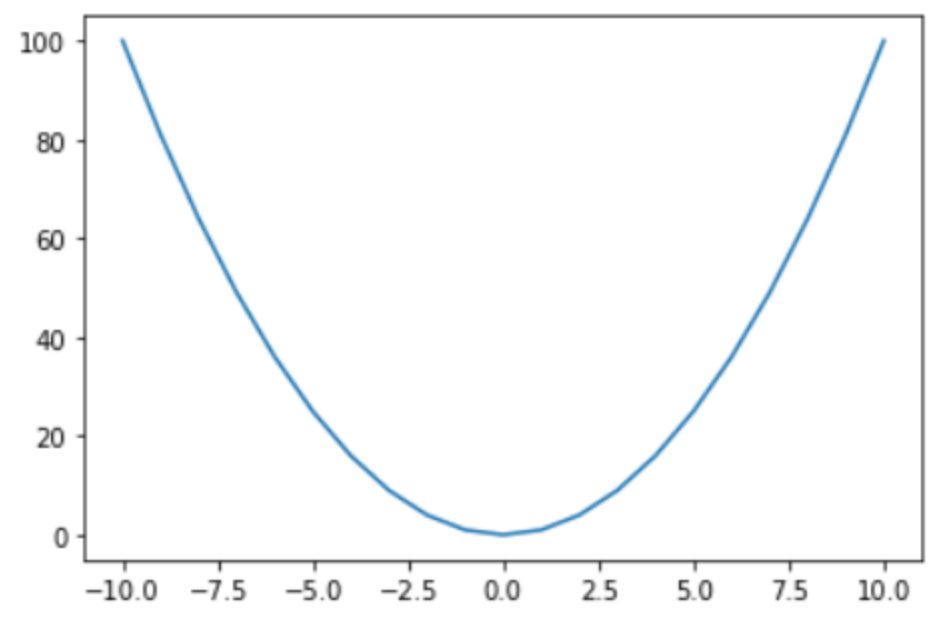


Screenshot https://matplotlib.org/stable/gallery/index.html#

Matplotlib

```
import matplotlib.pyplot as plt

x = range(-10,11)
y = [i**2 for i in x] # short-hand notation of a for-loop
plt.plot(x, y)
```



Figure, Axes, Axis

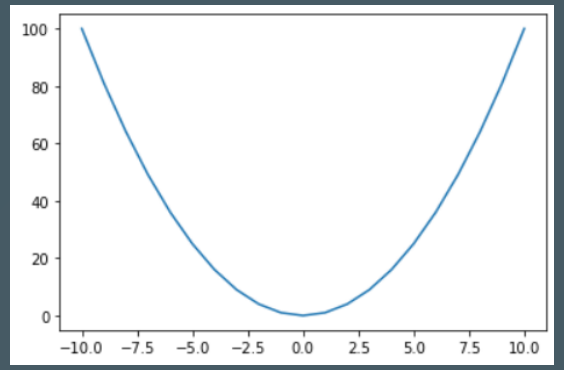
Matplotlib lingo

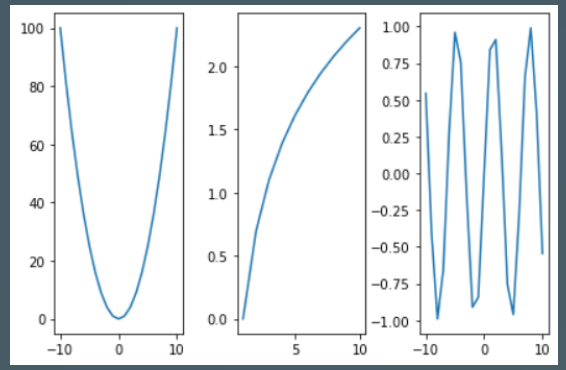
Figure

- The whole thing, containing all the other elements

Axes (with an "e")

- A "plot" inside the figure
  - A figure can contain a single axes or multiple axes



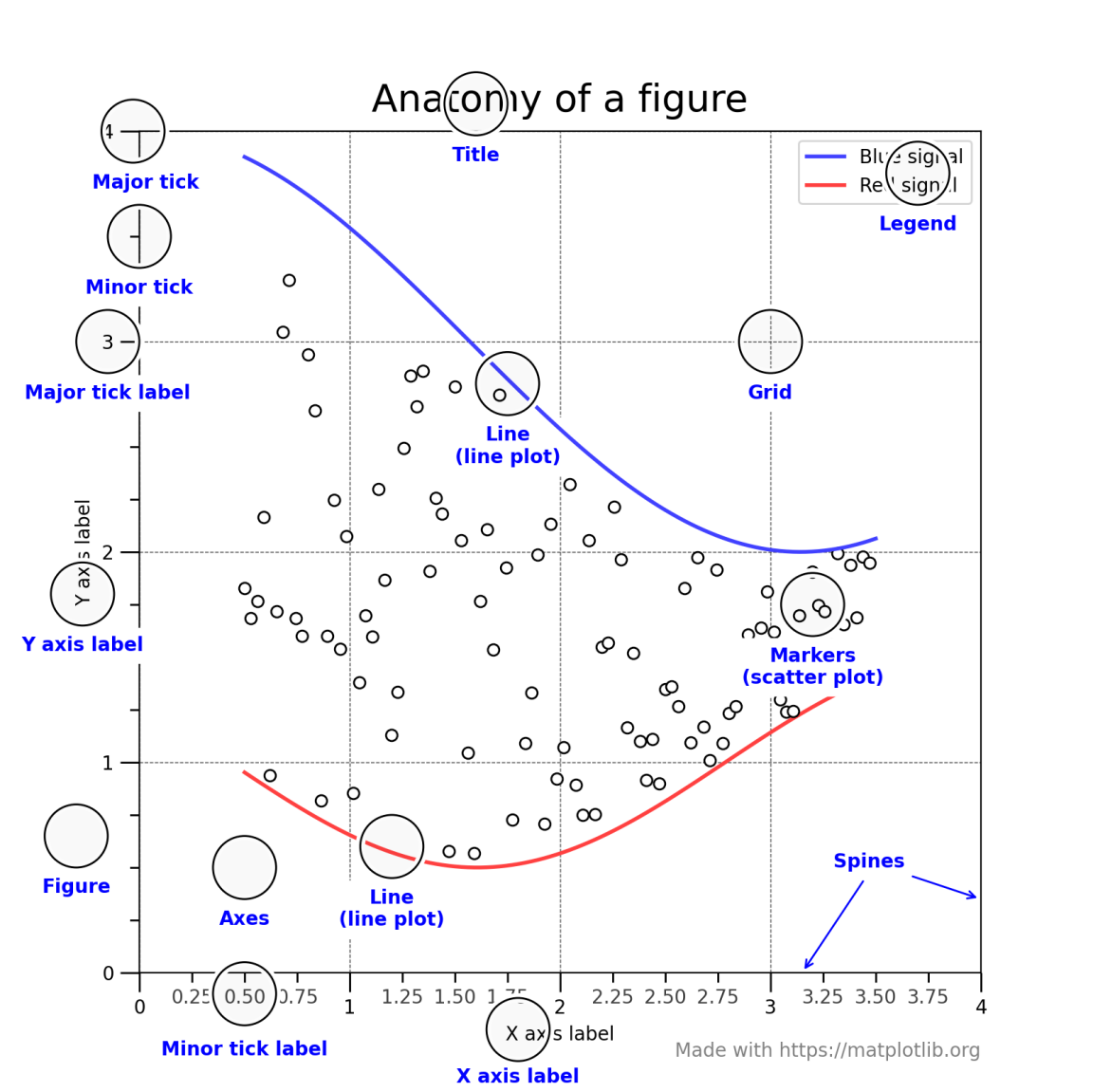


Figure, Axes, Axis

Axis (with an "i")

- The actual axis of a plot, e.g. the x and y-axis in a 2 dimensional graph

Image from https://matplotlib.org/stable/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py



Matplotlib

Getting a figure and an axis

```
import matplotlib.pyplot as plt

fig = plt.figure()
fig = plt.figure(figsize=(19.2, 10.8), dpi=100) # set the desired figure size in inches
                                                # and the image resolution (dpi, dots per inch)
                                                # -> this image has HD resolution (1920x1080 px)

ax = fig.add_subplot() # add a single axes
```

or all in one call:

```
fig, axs = plt.subplots(nrows=2, ncols=3) # imagine fig to be a 2x3 matrix, and each cell
                                          #   holds a separate axes to plot with

print(type(axs)) # <class 'numpy.ndarray'> - a numpy array of axes objects!
print(axs.shape) # (2, 3)
```

Matplotlib

Figure methods

- `fig.add_suplot()` - add an axes to the figure
- `fig.savefig(filename)` - create an image file `filename` of the figure
- ... and many more

Axes methods

- `ax.plot()` - "Plot y versus x as lines and/or markers"
- `ax.scatter()` - "A scatter plot of y vs. x with varying marker size and/or color."
- `ax.bar()` - make a bar plot
- `ax.boxplot` - make a boxplot

Matplotlib

More axes methods

- `ax.set_ylim()` - set a value range for the y-ax**i**s (also `set_xlim()`)
- `ax.set_title()` - set a title for the ax**e**s
- `ax.legend()` - add a legend
- ... and many more

Matplotlib:

- `plt.show()` display the figure

Matplotlib

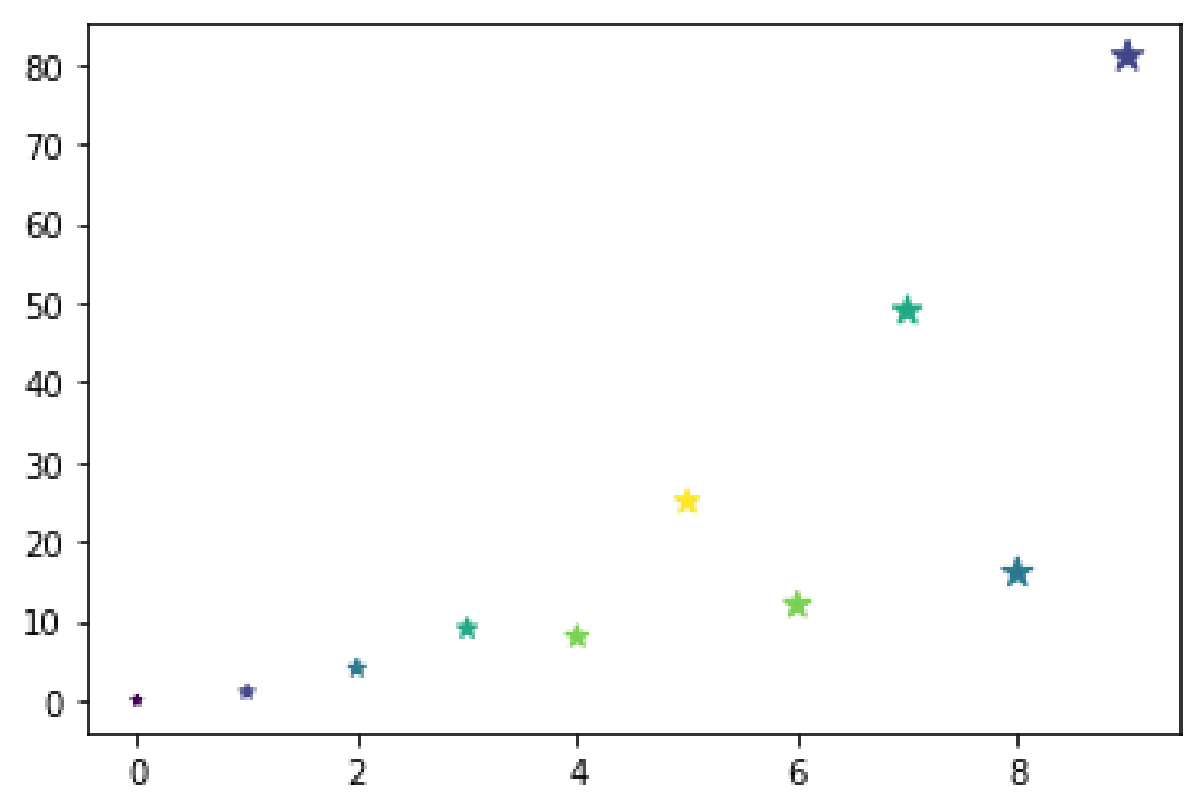
```
import matplotlib.pyplot as plt

# define some data to plot
x = [0,1,2,3,4, 5, 6, 7, 8, 9]
y = [0,1,4,9,8,25,12,49,16,81]

fig, ax = plt.subplots(1,1) # figure with a single axes
ax.scatter(x, y,
           s = [10,20,30,40,50,60,70,80,90,100],    # size of each point
           c = [ 0, 1, 2, 3, 4, 5, 4, 3, 2,  1],    # color of each point
           marker="*")                              # style of the marker points
plt.show()
```

Matplotlib

```
ax.scatter(x, y,
           s = [10,20,30,40,50,60,70,80,90,100],    # size of each point
           c = [0,1,2,3,4,5,4,3,2,1],               # color of each point
           marker="*")                              # style of the marker points
```



Matplotlib

Some more expamples in the notebook...

Pandas

Another popular library to handle (scientific) data

- E.g. scientific experiments might yield datapoints for a number of samples, and the data points for each sample may be of different data type (number, string, time, ...)
- Numpy arrays cannot (easily) handle mixed data types,
  and basic Python types like `list` or `dict` lack some desireable functionality

Pandas offers a solution with its `DataFrame`

Pandas

Main class from Pandas: `DataFrame`

- Imagine a table with *rows* (usually numbered) and *columns* (usually named)

	column A	column B	column C	...
0				
1				
2				
...				

- Each *column* can have a different data type

Pandas

```
import pandas as pd
```

Creating a DataFrame, usually from a `dict`

```
df = pd.DataFrame(
    {
        "Name": [
            "Braund, Mr. Owen Harris",
            "Allen, Mr. William Henry",
            "Bonnell, Miss. Elizabeth",
        ],
        "Age": [22, 35, 58],
        "Sex": ["male", "male", "female"],
    }
)

df # output the DataFrame
```

Example from https://pandas.pydata.org/docs/getting_started/intro_tutorials/01_table_oriented.html

Pandas

```
df = pd.DataFrame( # dict keys are the column names, values are lists of uniform size
    {
        "Name": [
            "Braund, Mr. Owen Harris",
            "Allen, Mr. William Henry",
            "Bonnell, Miss. Elizabeth",
        ],
        "Age": [22, 35, 58],
        "Sex": ["male", "male", "female"],
    }
)

df
```

```
                      Name  Age     Sex
0  Braund, Mr. Owen Harris   22    male
1 Allen, Mr. William Henry   35    male
2 Bonnell, Miss. Elizabeth   58  female
```

Pandas

Also very convenient to load data from files (if it is in a table format)

- Popular format: CSV (comma separated values)

```
Name,Age,Sex
"Braund, Mr. Owen Harris",22,male
"Allen, Mr. William Henry",35,male
"Bonnell, Miss. Elizabeth",58,female
```

```
df = pd.read_csv("tab.csv")  # as simple as that (assume above text is the content of tab.csv)
df
```

```
                      Name  Age     Sex
0  Braund, Mr. Owen Harris   22    male
1 Allen, Mr. William Henry   35    male
2 Bonnell, Miss. Elizabeth   58  female
```

Pandas

DataFrame properties

- `index` - the row labels
- `columns` - the column names
- ...

```
print(df.columns)
print(df.index)
```

```
Index(['Name', 'Age', 'Sex'], dtype='object')
RangeIndex(start=0, stop=3, step=1)
```

Pandas

DataFrame methods

- `describe()` - Summary of the numeric data in the DataFrame
- ...

```
df.describe()
```

```
             Age
count   3.000000
mean   38.333333
std    18.230012
min    22.000000
25%    28.500000
50%    35.000000
75%    46.500000
max    58.000000
```

Pandas

Columns or *Series*

Each column in a DataFrame is a `Series` object

Extract a column from a DataFrame with the `[ ]` operator:

```
df['Age']
```

```
0    22
1    35
2    58
Name: Age, dtype: int64
```

```
list(df['Age']) # yields the list [22, 35, 58]
```

Pandas

Series methods

- `max()` - return the max value of the column (also `min()`)
- `sum()` - return the sum of the column values
- `mean()`, `median()`, `var()`, ...

```
df['Age'].max()
```

```
58
```

Indexing a DataFrame

Multiple columns

```
df[["Age", "Sex"]]  # pass the list ["Age", "Sex] to the [ ]-operator
```

```
   Age     Sex
0   22    male
1   35    male
2   58  female
```

Rows (based on the row indices) with `iloc` and `[ ]`

```
df.iloc[0:2] # note that slicing works!
```

```
                     Name  Age   Sex
0   Braund, Mr. Owen Harris   22  male
1  Allen, Mr. William Henry   35  male
```

Indexing a DataFrame

One strength of a DataFrame is indexing by condition

- For example, select all people that are younger than 50

```
df['Age']       # get the age column:        [ 22,  35,  58]
df['Age'] < 50  # creates a Series of bools: [True, True, False]

df[ df['Age'] < 50 ] # only select "True" rows!
```

```
                     Name  Age   Sex
0   Braund, Mr. Owen Harris   22  male
1  Allen, Mr. William Henry   35  male
```

Pandas

Other example in the notebook...

It's Your Turn!

Go to our Moodle page (https://moodle.uni-greifswald.de/course/view.php?id=9565) and take the sixth quiz! ("Quiz 6 - Pandas")

Use the notebook from Session 3 to answer the questions, i.e.

- Create a new cell
- Import the Pandas module
- Load the file: `pd.read_csv('data/glass.csv')`
- Use DataFrame methods to get the information you need

Python From the Command Line and Argparse

Create a text file with the file ending `.py`, e.g. `my_script.py`, and write your program in this file

In a terminal (e.g. Bash in Linux), run

```
python3 path/to/my_script.py
```

That's it!

(Note: Sometimes, there might be no command `python3` but just `python`. In this case, check with `python --version` that you are using the correct Python version, e.g. 3.9)

Python From the Command Line and Argparse

Often, you want to tweak the behaviour of your program with some parameters, but re-writing the script everytime is way too cumbersome!

```
import sys

print(sys.argv)
```

Run from command line and observe output:

```
$ python3 my_script.py 1 2 foo

['my_script.py', '1', '2', 'foo']
```

Python From the Command Line and Argparse

```
import sys

print(sys.argv)
```

`sys.argv` is a list of strings, containing everything you wrote in the command line when starting the program

- The first element is always the script name (or path)

You could now write a program that can parse arguments via the command line:

```
python3 my_script.py -n 10 --verbose --out output.txt --in my_data.csv
```

But there is also a module that can help you with that!

Argparse

Very briefly, the Argparse module provides a class `ArgumentParser` that has methods to help you define and read command line arguments

```
import argparse

parser = argparse.ArgumentParser(description="You can describe your program if you want")

# add a command line argument you are expecting
parser.add_argument('-n',            # argument name
                    dest='cli_n',    # key to get value
                    type=int,        # expected value type
                    required=True,   # don't start unless the argument is given
                    help="Helpful description of what the argument does")

# args contains the argument keys and their values, if given
args = parser.parse_args()

print("The value of the argument `-n` was", args.cli_n)
```

Argparse

```
$ python my_script.py -n 42

The value of the argument `-n` was 42
```

Other example in the notebook...

Getting Help

Nobody expects you to remember everything, especially not details on how to use third-party modules!

Your favourite search engine is your friend (Google, DuckDuckGo, ...), and most modules have a detailed documentation online

- Often also a tutorial or a "how to get started" from the module authors

Important Learnings from this Course

- Understanding the core concepts of programming:
  - Basic Python syntax
  - variables and lists (or other containers)
  - boolean logic (`and`, `or`, `not`)
  - conditional programming (`if-elif-else`)
  - loops
  - functions

Important Learnings from this Course

Bonus Learnings

- Knowing that things like scope exist
- Having an idea that classes (types) can do much more than just store a value
- There are many modules out there that make accomplishing tasks in Python easy

...and if you forgot something, just look it up!

Continue Learning

Learning any (programming) language is learning by doing!

Just start writing Python scripts to accomplish some tasks, e.g. from work.
That's the best way to learn.

Also, do more tutorials if you don't feel ready yet, e.g.
https://www.w3schools.com/python/default.asp

Thank you for participating!