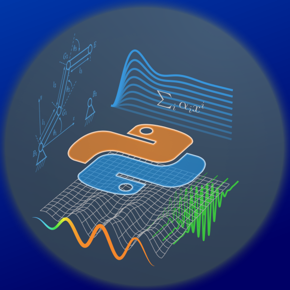
Carsten Knoll
Chair of Fundamentals of Electrical Engineering

Python for Engineers
Pythonkurs für Ingenieur:innen

Data Processing and Analysis
Numerische Datenauswertung

Dresden (Online), 2023-11-21

https://tu-dresden.de/pythonkurs
https://python-fuer-ingenieure.de



Data Analysis: Overview

What do you do when you evaluate (measurement) data?

- load data
- select relevant data (or sections)
- determine new variables from existing ones
- increase information density
- visualize results

Data Analysis: Overview

What do you do when you evaluate (measurement) data?

- load data
- select relevant data (or sections)
- determine new variables from existing ones
- increase information density
- visualize results

Learning objectives:

- load / save data
- productive handling of Numpy arrays
- overview, which algorithms are already implemented
- overview on Pandas

Loading / Saving Data

```python
import numpy as np

# ...  do important calculations then save the result
result_array = np.arange(21).reshape(7, 3)

# same array but with automatic determination of column number
result_array2 = np.arange(21).reshape(7, -1)

np.save("result.npy", result_array) # binary format
np.savetxt("result.dat", result_array) # txt format

# ... in another file ...

array1  = np.load("result.npy") # binary format
array2  = np.loadtxt("result.dat") # txt format
```

$\exists$ other possibilities (e.g. for `*.wav` files or Matlab format: see `scipy.io.*`).

Array Indexing

Basic indexing options of Numpy arrays (see also course03):

- integer numbers (`x[5]`) and “slices” (`x[3:10]`)

Array Indexing

Basic indexing options of Numpy arrays (see also course03):

- integer numbers (`x[5]`) and "slices" (`x[3:10]`)

Advanced indexing option 1: `int` -arrays

- can have arbitrary length; must contain integer values between $-n$ and $+n-1$
- values are interpreted as indices; values can be repeated and omitted

```
x = np.array([10, 11, 12, 13])
idcs = np.array([1, 2, 2, 1, 0 -2])
y = x[idcs]  # -> array([11, 12, 12, 11, 10, 12])
```

Array Indexing

Basic indexing options of Numpy arrays (see also course03):

- integer numbers (`x[5]`) and "slices" (`x[3:10]`)

Advanced indexing option 1: `int` -arrays

- can have arbitrary length; must contain integer values between $-n$ and $+n-1$
- values are interpreted as indices; values can be repeated and omitted

```
x = np.array([10, 11, 12, 13])
idcs = np.array([1, 2, 2, 1, 0 -2])
y = x[idcs]  # -> array([11, 12, 12, 11, 10, 12])
```

Advanced indexing option 2: `boolean` arrays

- length must be the same as indexed array (`x`)
- can only contain values `True` and `False`
- lenght of result equals number of `True`

```
idcs = np.array([True, False, False, True])
x[idcs]  # -> [10, 13] (only first and last value were selected)

# negate all values that are less than 12:
x[x<12]*=-1  # -> [-10, -11, 12, 13]
```

Numerical Differentiation

- reminder: $\frac{df(x)}{dx} \approx \frac{f(x+\Delta x)-f(x)}{\delta x}$ ("difference quotient")
- `numpy.diff` for each array element: calc *successor minus predecessor*.
- for approximation of the first derivative of a function: **you** have divide by Δx.
- `numpy.diff(x)` $\rightarrow$ result is one element shorter than input data `x`

```python
import numpy as np
x = np.arange(20)   # array([0, 1, 2, 3,...])
xd = np.diff(x)   # array([1, 1, 1, ...])
```

- higher derivative orders also possible, see doc
- opposite operation: `np.cumsum(x)` (cumulative sum $\hat{=}$ discrete integral).

Useful Engineering Tools: Filter and FFT

- filtering (low pass, moving average, ...).

$\rightarrow$ package: scipy.signal

- Representation of the frequency spectrum

$\rightarrow$ package: numpy.fft (Fast Fourier Transform)
"most important algorithm" of the information age

both require quite some background knowledge (therefore not covered here)

Interpolation (1)

- Interpolation (generate intermediate values, change sampling rate, ...)

→ Paket: scipy.interpolate

Listing: example-code/01_interpolation.py

```
import numpy as np
import scipy.interpolate as ip
import matplotlib.pyplot as plt

# original data
x = [1, 2, 3, 4]
y = [2, 0, 1, 3]

plt.plot(x, y, "bx")  # blue crosses
plt.savefig("interpolation0.pdf")

# create linear interpolator function
fnc1 = ip.interp1d(x, y)

# achieve higher x-resolution by evaluation of fnc1
xx = np.linspace(1, 4, 20)
plt.plot(xx, fnc1(xx), "r.-")  # red solid line and dots
plt.savefig("interpolation1.pdf")
plt.show()
```

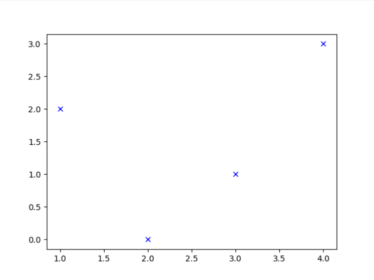
Interpolation (1)

- Interpolation (generate intermediate values, change sampling rate, ...)

→ Paket: scipy.interpolate

Listing: example-code/01_interpolation.py

```
import numpy as np
import scipy.interpolate as ip
import matplotlib.pyplot as plt

# original data
x = [1, 2, 3, 4]
y = [2, 0, 1, 3]

plt.plot(x, y, "bx")  # blue crosses
plt.savefig("interpolation0.pdf")

# create linear interpolator function
fnc1 = ip.interp1d(x, y)

# achieve higher x-resolution by evaluation of fnc1
xx = np.linspace(1, 4, 20)
plt.plot(xx, fnc1(xx), "r.-")  # red solid line and dots
plt.savefig("interpolation1.pdf")
plt.show()
```



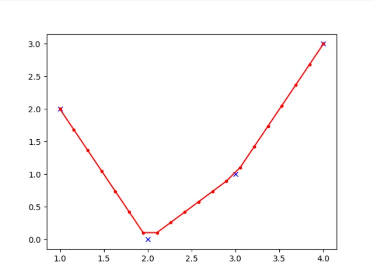
Interpolation (1)

- Interpolation (generate intermediate values, change sampling rate, ...)

→ Paket: scipy.interpolate

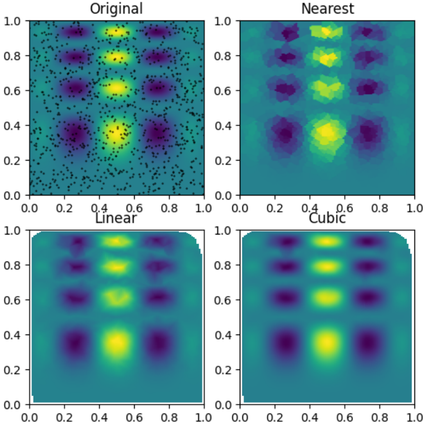
Listing: example-code/01_interpolation.py

```python
import numpy as np
import scipy.interpolate as ip
import matplotlib.pyplot as plt

# original data
x = [1, 2, 3, 4]
y = [2, 0, 1, 3]

plt.plot(x, y, "bx")  # blue crosses
plt.savefig("interpolation0.pdf")

# create linear interpolator function
fnc1 = ip.interp1d(x, y)

# achieve higher x-resolution by evaluation of fnc1
xx = np.linspace(1, 4, 20)
plt.plot(xx, fnc1(xx), "r.-")  # red solid line and dots
plt.savefig("interpolation1.pdf")
plt.show()
```



Interpolation (2)

- Works also in higher dimensions with irregularly distributed input data (see `example_code/01b_griddata.py`, taken from docs.scipy.org/...)



Regression (or “Fitting”)

→ `numpy.polyfit` and `numpy.polyval`

Listing: example-code/02_regression.py

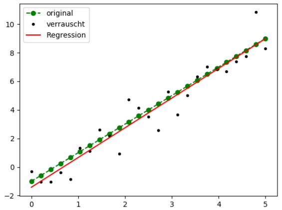
```
import numpy as np
import matplotlib.pyplot as plt

N = 25
xx = np.linspace(0, 5, N)

# cool trick: two assignments in one line
m, n = 2 , -1

# evaluate equation of straigt line: y = m*x + n
yy = np.polyval([m, n], xx)
yy_noisy = yy + np.random.randn(N)  # add some random noise

# create linear fit (regression 1st order polynomial)
mr, nr = np.polyfit(xx, yy_noisy, 1)  # calculate fit
yyr = np.polyval([mr, nr], xx)  # evaluate the function

plt.plot(xx, yy, 'go--', label="original")
plt.plot(xx, yy_noisy,'k.', label="noisy data")
plt.plot(xx, yyr,'r-', label="regression")
plt.legend()
plt.savefig("regression.png")

plt.show()
```

Regression (or “Fitting”)

→ `numpy.polyfit` and `numpy.polyval`

Listing: example-code/02_regression.py

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

N = 25
xx = np.linspace(0, 5, N)

# cool trick: two assignments in one line
m, n = 2 , -1

# evaluate equation of straigt line: y = m*x + n
yy = np.polyval([m, n], xx)
yy_noisy = yy + np.random.randn(N)  # add some random noise

# create linear fit (regression 1st order polynomial)
mr, nr = np.polyfit(xx, yy_noisy, 1)  # calculate fit
yyr = np.polyval([mr, nr], xx)  # evaluate the function

plt.plot(xx, yy, 'go--', label="original")
plt.plot(xx, yy_noisy,'k.', label="noisy data")
plt.plot(xx, yyr,'r-', label="regression")
plt.legend()
plt.savefig("regression.png")

plt.show()
```

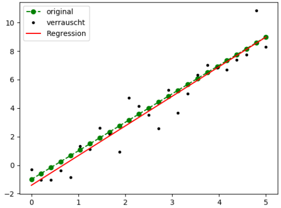


Regression (or “Fitting”)

→ `numpy.polyfit` and `numpy.polyval`

Listing: example-code/02_regression.py

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

N = 25
xx = np.linspace(0, 5, N)

# cool trick: two assignments in one line
m, n = 2 , -1

# evaluate equation of straigt line: y = m*x + n
yy = np.polyval([m, n], xx)
yy_noisy = yy + np.random.randn(N)  # add some random noise

# create linear fit (regression 1st order polynomial)
mr, nr = np.polyfit(xx, yy_noisy, 1)  # calculate fit
yyr = np.polyval([mr, nr], xx)  # evaluate the function

plt.plot(xx, yy, 'go--', label="original")
plt.plot(xx, yy_noisy,'k.', label="noisy data")
plt.plot(xx, yyr,'r-', label="regression")
plt.legend()
plt.savefig("regression.png")

plt.show()
```



- higher polynomial orders also possible

“Machine Learning” (ML)

- widely used since 2010’s; very successful on some tasks
- can also be understood as **function approximation**
- three important subareas
  - **supervised learning**
    - classification (dog or cat? Mozart or Helene Fischer?)
    - regression (How well will movie X please person Y?)
  - **unsupervised learning** (= automatic cluster detection)
  - **Reinforcing learning** (agent adapts responses to environment to maximize reward)

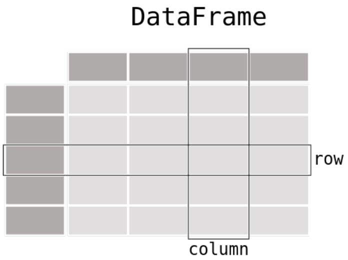
Python is (arguably) the most important languages in ML:
Subjective Link Selection:

- `https://scikit-learn.org` (neural networks, Gaussian processes, and many more).
- `https://pytorch.org` (neural networks)

The Pandas Package

→ **"“spreadsheet processing with Python”"**

- most important package for “Data Science”
- based on Numpy
- most important data type: `pandas.DataFrame`
  - models a table
  - columns can have names
  - columns can have different data types
- second most important data type: `pandas.Series`
  - models row or column



extensive documentation: `https://pandas.pydata.org/docs/`

Pandas: Create Data Frames

Listing: 03_pandas.py

```
import pandas as pd
import numpy as np

# create df from a numpy array:
arr = np.random.randn(6, 4)
df1 = pd.DataFrame(arr, columns=list("ABCD"))

# create df from a dict
shop_articles = {
    "weight": [10.1, 5.0, 8.3, 7.2],
    "color": ["red", "green", "blue", "transparent"],
    "availability": [False, True, True, False],
    "price": 8.99  # all have the same price
}
article_numbers = ["A107", "A108", "A109", "A110"]
df2 = pd.DataFrame(shop_articles, index=article_numbers)

# each has its own data type
print("column types:\n", df2.dtypes)showspaces
```

Pandas: Save and Load Data

Listing: 03_pandas.py

```
fname = "things.csv"

# save the data frame as CSV file (Comma Separated Values )
# csv file will also contain header information (column labels)
df2.to_csv(fname)



# Pandas function to load csv-data into DataFrame
# (Detects column names automatically)
df2_new = pd.read_csv(fname)

# display(df2_new)  # Jupyter-Notebook-specific
```

Pandas: Read/Write Access to Cells

Listing: 03_pandas.py

```
# access individual values (by verbose index and column):
print(df2.loc["A108", "weight"])
df2.loc["A108", "weight"] = 3.4  # set new value
df2.loc["A108", "weight"] += 2  # increase by two

# access by numerical indices
print(df2.iloc[1, 0])  # row index: 1, column index: 0

# use slices to change multiple values
df2.loc["A108":"A109", "price"] *= 0.30  # 30% discount

# access multiple columns (-> new df object)
print(df2[["price", "weight"]])

# new column (-> provide a height value for every article)
df2["height"] = [10, 20, 30, 40]

# new row (-> provide a value for every column (weight, color, ...) )
df2.loc["X400"] = [15 , "purple" , True , 25.00 , 50]
```

Pandas: Select Data by Boolean Indexing

Listing: 03_pandas.py

```python
# create Series-object with bool-entries
idcs = df2["weight"] > 8

# use this Series-object for indexing
print(df2[idcs])

# similar statement without intermediate variable:
print(df2[df2["weight"] < 8])
```

Pandas: Apply Funtions

Listing: 03_pandas.py

```
df2.describe()
df2["price"].mean()
df2["weight"].median()
df2["weight"].max()

print("shorthand notation (if column label is valid python name)")
print(df2.weight == df2["weight"])
print(all(df2.weight == df2["weight"]))

# combine function application with boolean indexing
df2[df2.weight>8].weight.mean()

# apply an arbitrary function (here np.diff) to each (selected) column
print(df2[["price", "weight"]].apply(np.diff))
```

Summary

- access to data (`np.load`, `pd.read_csv`)
- selection (integer indexing, boolean indexing of both arrays and data frames)
- interpolation
- regression
- see docs for more details