Data Science mit Python

Dr. Christoph Giess
MARS – Center for Entrepreneurship

18.10.2023

Inhaltsverzeichnis

Dieser Kurs richtet sich an

- alle, die an „Digital Basics: Einführung in die Programmierung mit Python“ teilgenommen haben
- sowie Personen, die ein bisschen Python programmieren können und jetzt lernen möchten, wie man Daten mit Hilfe von Python auswerten kann.

Kurzvorstellung

- Wer bin ich?
- Wer seid ihr und was wollt ihr am Ende des Kurses machen?

Los gehts

- Jupyter
  - https://jupyter.org/try-jupyter/lab/
  - Notebook - Python (Pyodide)
  - Python im Webbrowser
  - Input-Zeile bedienen
  - Python Programm (Code) und Dokumentation (Markdown)

KAPITEL 1

Wiederholung

Um alle Teilnehmer auf den gleichen Stand zu bringen fangen wir dort an, wo der erste Kurs geendet hat.

1.1 Packages, Funktionen, Arrays und Rechnen

```
import random
random.seed(42)

def rand_numbers(n, start, end):
    """ Erzeugt einen Array mit Zufallszahlen mit 2 Nachkommastellen
    n
      Anzahl der erzeugten Zufallszahlen

    start
      Kleinste mögliche Zahl (inklusiv)

    end
      Größte mögliche Zahl (inklusiv)
    """
    digits = 2
    factor = 10 ** digits
    result = []
    for i in range(n):
        result.append(random.randint(start * factor, end * factor) / factor)

    return result
```

```
rand_numbers(10, -1, 1)
```

```
[0.63, -0.72, -0.94, 0.89, -0.3, -0.38, -0.43, -0.65, 0.88, -0.74]
```

KAPITEL 2

Mehr zu Funktionen

2.1 Default-Werte

An der Funktion ist unschön, dass sie die Zufallszahlen immer mit 2 Nachkommastellen zurück gibt.

In den meisten Fällen ist das OK aber manchmal möchte ich weniger oder auch mehr Nachkommastellen. Dies lässt sich problemlos mit einem weiteren Parameter realisieren. Dem kann man sogar einen Default-Wert geben.

```
def rand_numbers(n, start, end, digits=2):
    """ Erzeugt einen Array mit Zufallszahlen mit 2 Nachkommastellen
    n
      Anzahl der erzeugten Zufallszahlen

    start
      Kleinste mögliche Zahl (inklusiv)

    end
      Größte mögliche Zahl (inklusiv)

    digits
      Anzahl Nachkommastellen, Default: 2
    """
    factor = 10 ** digits
    result = []
    for i in range(n):
        result.append(random.randint(start * factor, end * factor) / factor)

    return result
```

```
rand_numbers(5, 1, 20)
```

```
[17.04, 13.36, 15.18, 17.07, 11.12]
```

```
rand_numbers(5, 1, 20, 1)
```

```
[10.0, 17.5, 5.7, 18.1, 1.4]
```

2.2 Benannte Parameter

Bei `math.sin(math.radians(45))` kann man verstehen, was die Funktionen tun und was `45` bedeutet.

Bei `rand_numbers(5, 1, 20, 1)`, ist das ohne Dokumentation nicht mehr möglich. Um Code verständlicher zu machen können die Parameter von Funktionen beim Aufruf benannt werden.

```
rand_numbers(n=5, start=1, end=20, digits=1)
```

```
[19.9, 9.4, 18.5, 3.1, 4.1]
```

Die Reihenfolge ist beliebig.

```
rand_numbers(end=20, start=1, digits=1, n=5)
```

```
[4.2, 6.9, 3.0, 12.2, 6.4]
```

2.3 Aufgabe

Was passiert, wenn man einzelne Parameter weglässt?

2.4 Lösung

Man kann nur `digits` weglassen, weil dies einen Default-Wert hat. Alle anderen Parameter müssen angegeben werden.

KAPITEL 3

Grafiken

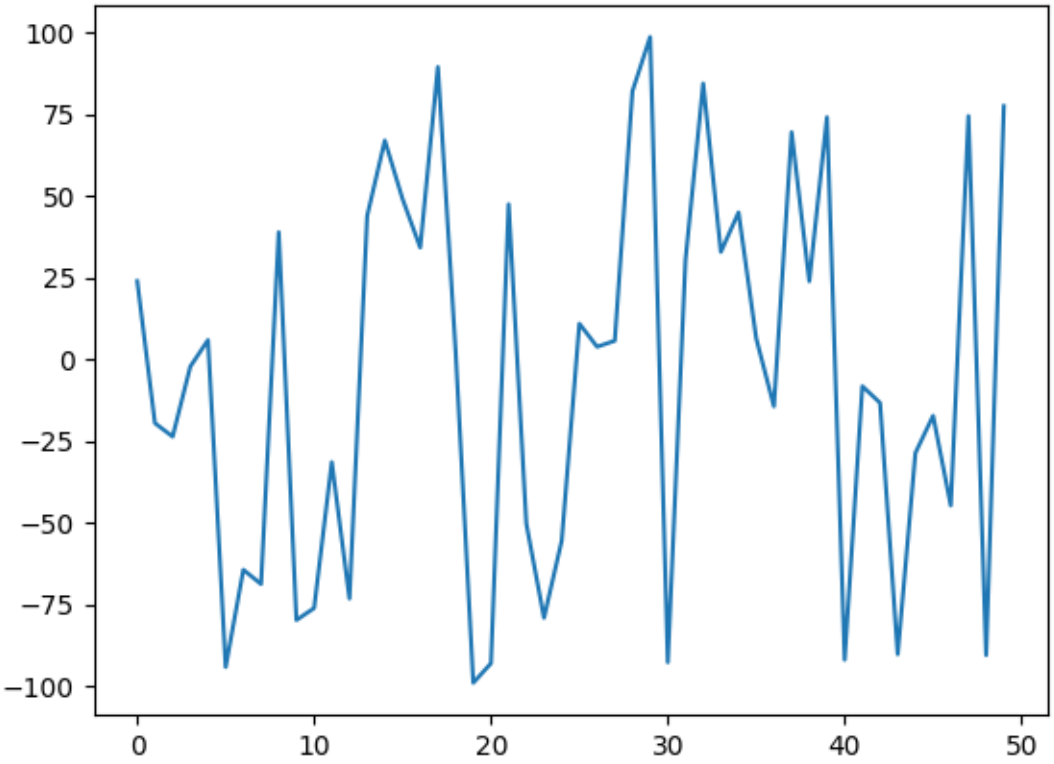
3.1 Matplotlib

Bisher haben wir nur mit Zahlen ein Texten gearbeitet. Dafür haben wir die Packages `math` und `random` verwendet.

Jetzt möchten wir Grafiken erzeugen. Dazu benötigen wir ein weiteres Package: `matplotlib`, genauer gesagt, davon erst einmal nur den Teil `pyplot`.

Um uns Tipparbeit zu sparen sagen wir beim `import`, dass wir im Folgenden dieses Package `plt` nennen möchten.

```
import matplotlib.pyplot as plt
numbers = rand_numbers(50, -100, 100)
plt.plot(numbers);
plt.show() # bei manchen Jupyter-Versionen nicht nötig, einfach mal ohne testen
```



3.2 Aufgabe

Erklärt,

1. was das Programm tut
2. was auf der Grafik zu sehen ist
3. wozu das Semikolon in der vorletzten Zeile dient

3.3 Lösung

1. Was tut Programm?
   - es importiert die Bibliothek `matplotlib.pyplot`. Diese enthält Funktionen zum Zeichnen von Grafiken
   - diese Bibliothek nennen wir `plt` weil das schneller zu schreibenist als `matplotlib.pyplot`
   - 50 Zufallszahlen zwischen -100 und 100 erzeugt und diese im Array mit dem Namen `numbers` speichern
   - die Zufallszahlen in `numbers` zeichnen
2. Was ist auf Grafik zu sehen?
   - Der Wert der Zufallszahlen ist auf der y-Achse
   - Die 0te Zufallszahl ist auf der x-Achse bei x=0, die 1te Zufallszahl bei x=1 usw.
3. Die Funktion plot() zeichnet die Grafik und liefert zusätzlich noch ein Ergebnis zurück. Das Semikolon sorgt dafür, dass das Ergebnis nicht angezeigt wird.

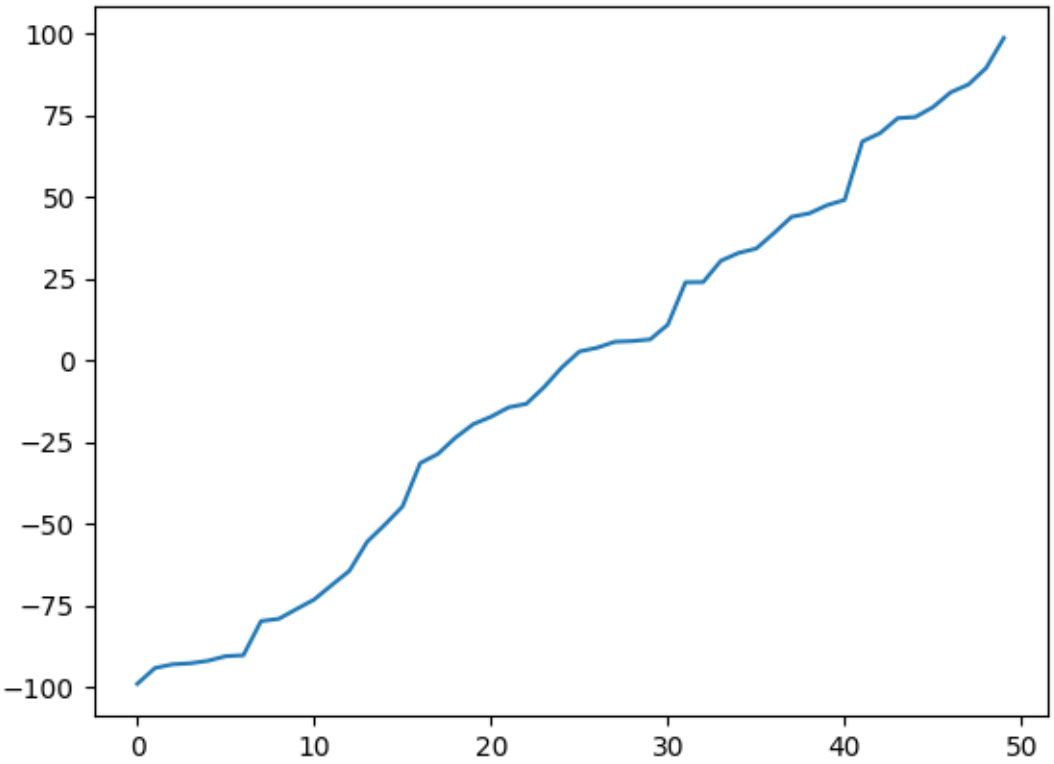
3.4 Aufgabe

Wie kann man erkennen, wie gleichmäßig die Zufallszahlen verteilt sind?

3.5 Lösung

Die Zufallszahlen der Größe nach sortieren. Je gleichmäßiger sie verteilt sind desto gerader ist die Linie im Graphen.

```
numbers.sort()
plt.plot(numbers);
plt.show()
```



Das sieht schon nicht schlecht aus. Abweichungen von der Gerade sind aber deutlich zu sehen.

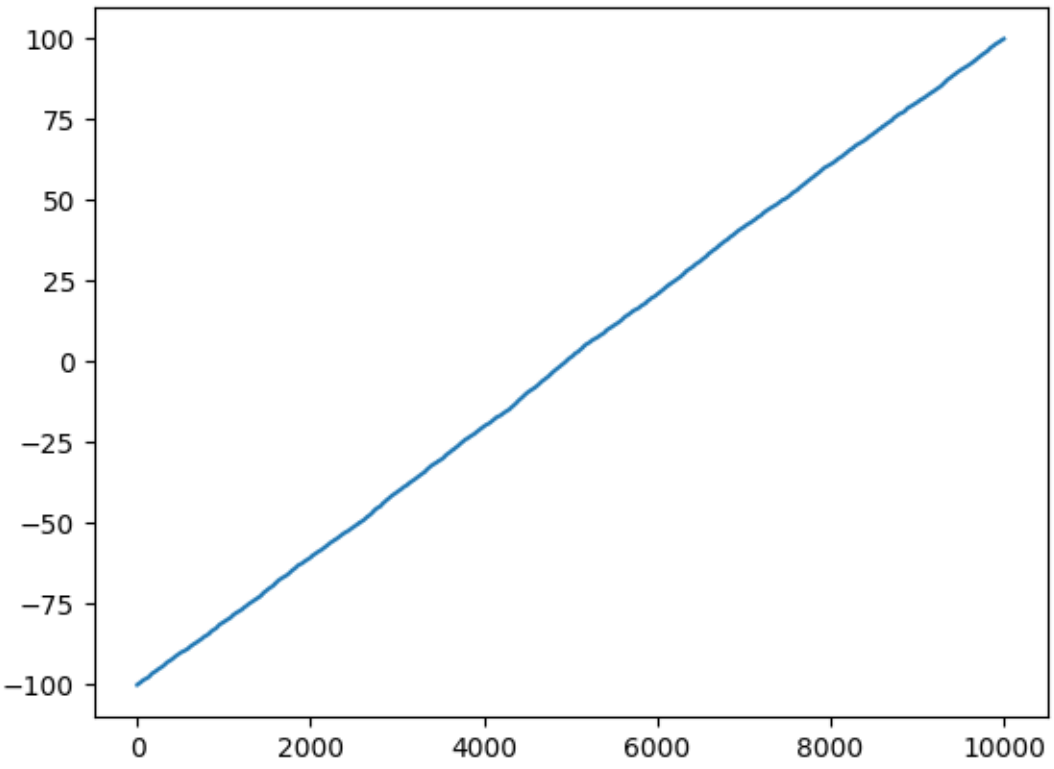
3.6 Aufgabe

Welche Stelle im Programm muss geändert werden damit der erzeugte Graph viel näher an einer Gerade ist?

3.7 Lösung

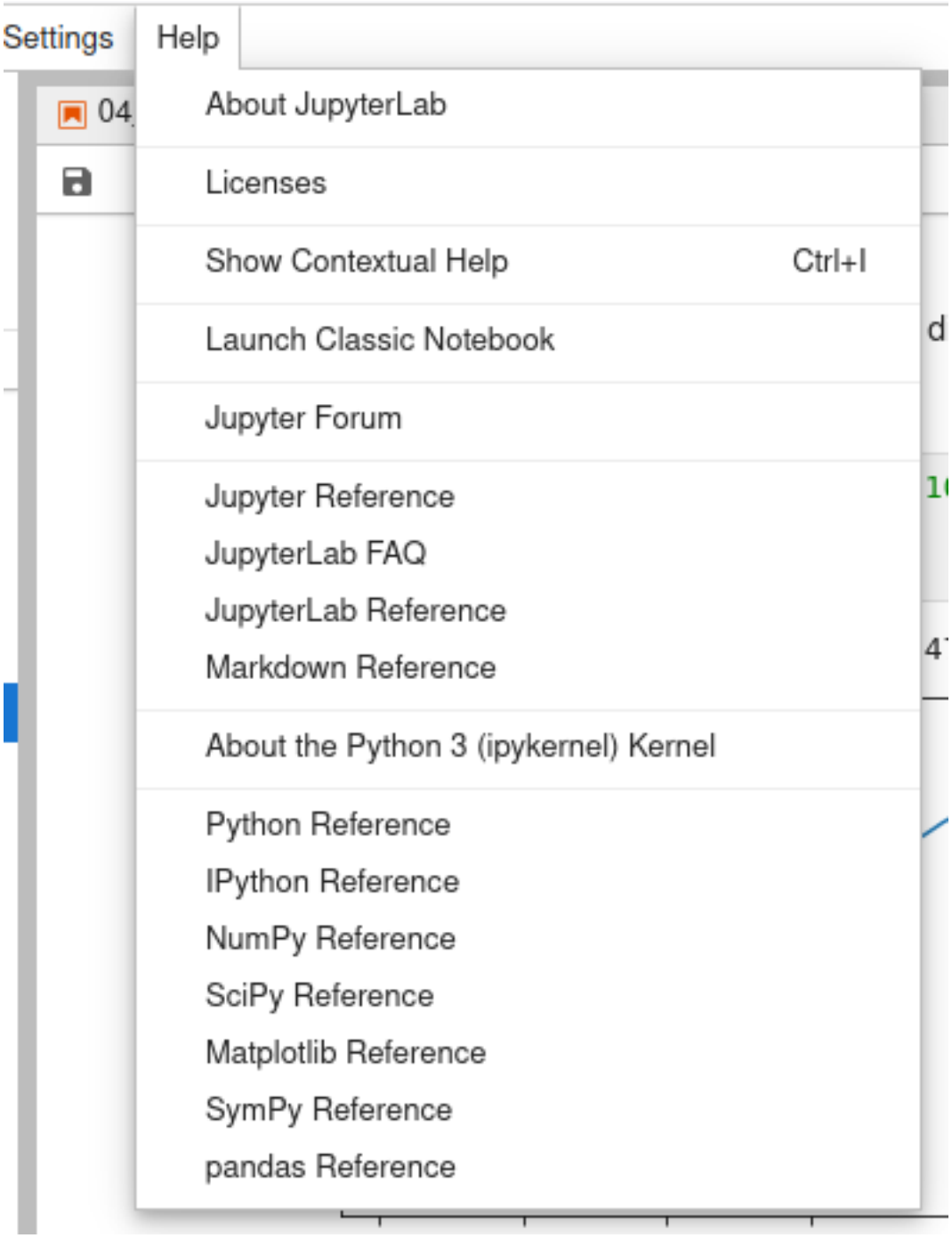
Nach dem Gesetz der großen Zahlen müsste die Abweichung von der Gerade kleiner werden wenn man mehr Zufallszahlen zieht.

```
numbers = rand_numbers(10000, -100, 100)
numbers.sort()
plt.plot(numbers);
plt.show()
```



3.8 Matplotlib Dokumentation

Dokumentation zu Matplotlib und anderen Python Packages findet man im unteren Teil des Hilfe-Menüs.



Falls diese fehlen:

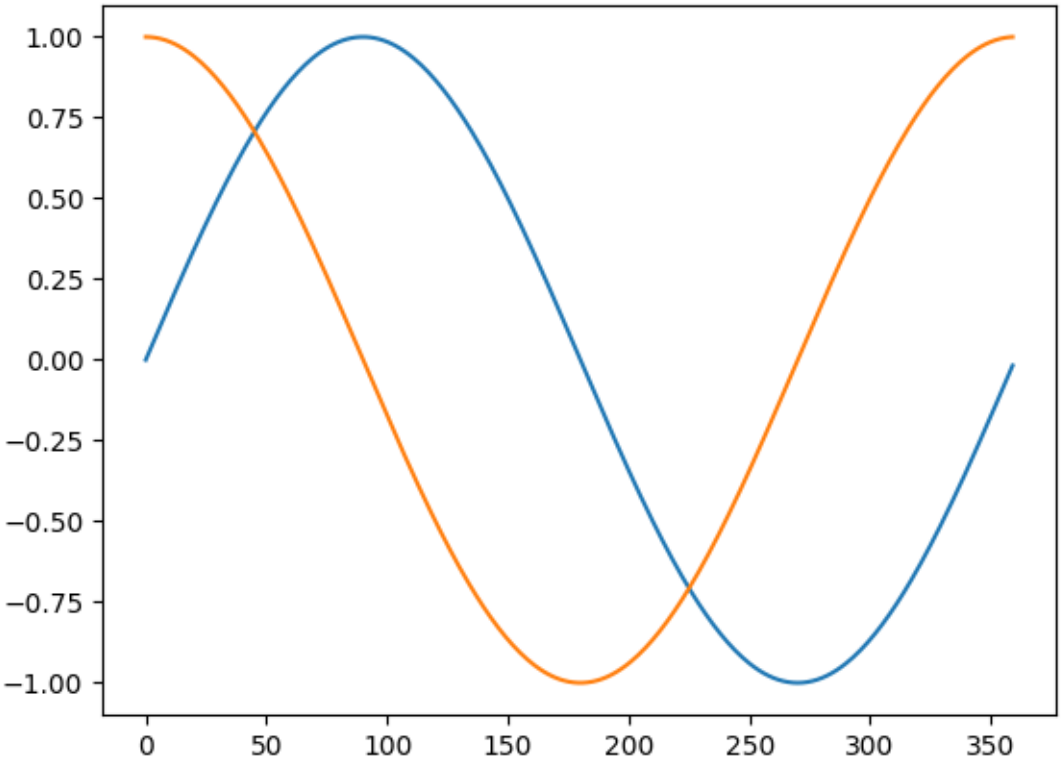
- matplotlib: https://matplotlib.org/stable/api/index.html
- pandas: https://pandas.pydata.org/docs/reference/index.html

3.9 Aufgabe

Erzeuge einen Graph mit Sinus und Cosinus-Funktion.

3.10 Lösung

```
import math

sins = []
coss = []

for i in range(360):
    sins.append(math.sin(math.radians(i)))
    coss.append(math.cos(math.radians(i)))

plt.plot(sins)
plt.plot(coss);
plt.show()
```

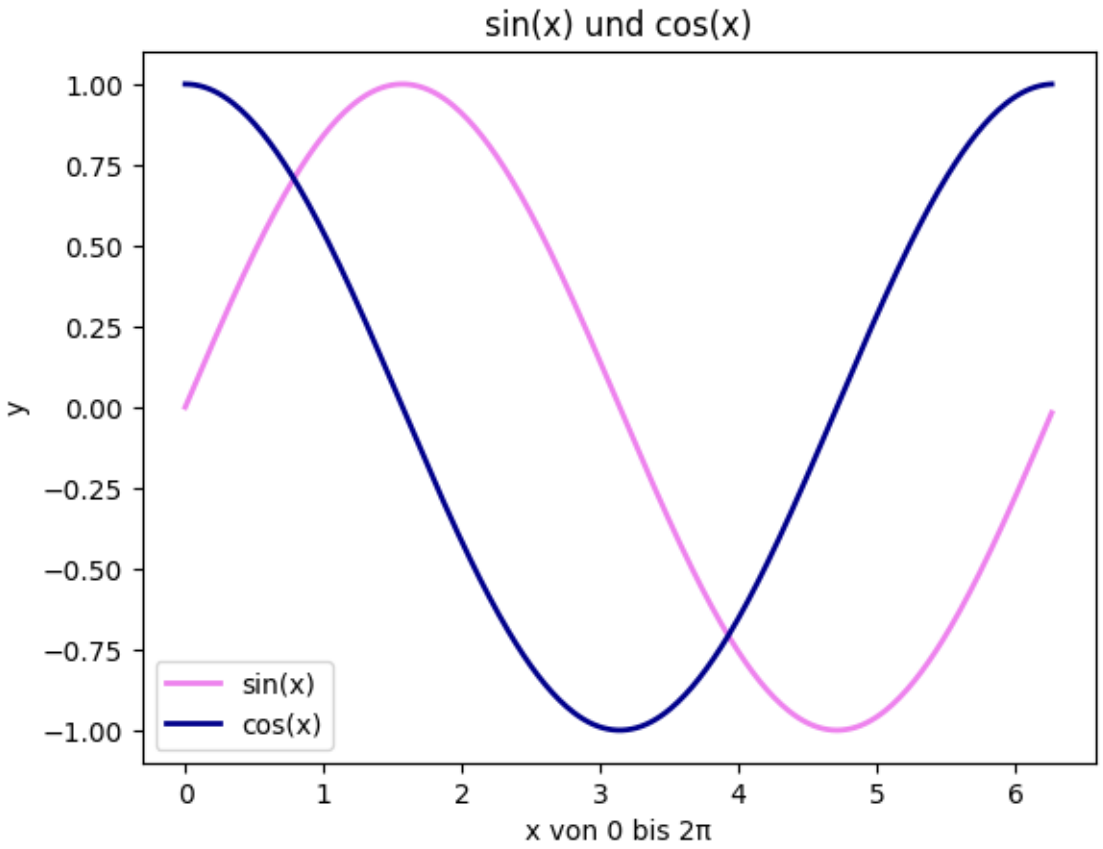


3.11 Aufgabe

- Lest die Dokumentation zu `plt.plot` und verändert
  - Farbe der Linie
  - Dicke der Linie
  - Beschriftung der x-Achse (Werte in Bogen- statt Gradmaß)
- Lest die Dokumentation zu `plot` und versucht den Graphen zu beschriften

3.12 Lösung

```
import math

sins = []
coss = []
rads = []

for i in range(360):
    r = math.radians(i)
    rads.append(r)
    sins.append(math.sin(r))
    coss.append(math.cos(r))

plt.plot(rads, sins, color='violet', linewidth=2)
plt.plot(rads, coss, color='darkblue', linewidth=2)

plt.title('sin(x) und cos(x)')
plt.xlabel('x von 0 bis 2π')
plt.ylabel('y')
plt.legend(['sin(x)', 'cos(x)']);
plt.show()
```



3.13 Grafiken speichern/exportieren

Bilder können nicht nur angezeigt sondern auch gespeichert werden. Dies muss *vor* dem Aufruf von `plt.show()` geschehen:

```
plt.savefig('sin_cos.png');
plt.savefig('sin_cos.pdf');
plt.savefig('sin_cos.svg');
plt.savefig('sin_cos.png', dpi=300, transparent=True, bbox_inches='tight')

plt.show()
```

KAPITEL 4

Daten analysieren

Zufallszahlen und Sinuswerte zu visualisieren ist ganz nett, in der Realtität müssen aber Ergebnisse aus Umfragen oder Messwerte von Experimenten ausgewertet werden.

Standard für die Datenanlyse mit Python ist das Package `pandas`.

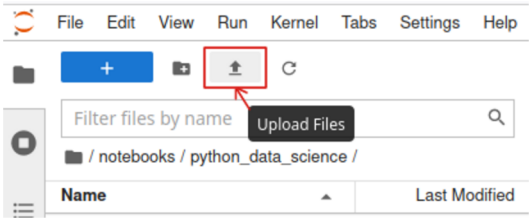
```
import pandas
```

Um zu demonstrieren, wie `pandas` funktioniert, brauchen wir ein paar Daten. Die finden wir in der Datei `car.csv`, in der die Kosten eines Autos über mehrere Jahre hinweg erfasst wurden.

CSV (Comma Separated Values) gibt es in verschiedenen Ausprägungen. In unserer Datei sind die Daten nicht mit Komma sondern einem Tabulator voneinander getrennt. Das muss man beim Einlesen mit `sep="\t"` angeben.

4.1 Hinweis

Bei den folgenden Beispielen wird die Datei `car.csv` benötigt. Diese findet sich in Moodle. Diese Datei muss auf den Jupyter-Rechner liegen. Dazu muss die Datei mittels `Upload Files` hochgeladen werden.



4.2 CSV Dateien mit Pandas lesen

```
d = pandas.read_csv("car.csv", sep="\t")
d
```

```
          Datum      Typ Beschreibung     Preis      km  Liter
0    2012-07-07     Kauf     Autohaus  13800.00      30    NaN
1    2012-07-10   Benzin         ESSO     57.01     199  34.89
2    2012-07-11  Versich     Haftpfl.    104.30     400    NaN
3    2012-07-23   Benzin     Kaufland     55.03     828  34.20
4    2012-08-10   Benzin     Kaufland     56.72    1444  35.47
..          ...      ...          ...       ...     ...    ...
227  2021-08-28   Benzin         AVIA     47.10  104552  29.27
228  2021-10-09   Benzin     Kaufland     55.10  105147  33.97
229  2021-10-19   Benzin          JET     40.10  105623  24.32
230  2021-12-04   Benzin          JET     53.30  106186  34.19
231  2021-12-25   Benzin         AVIA     51.42  106727  31.96

[232 rows x 6 columns]
```

Das scheint funktioniert zu haben. Eine kleine Änderung werden wir noch vornehmen. Die erste Spalte enthält ein Datum. Da dies verschieden geschrieben werden kann (31.12.2020, 12/31/2020, …) müssen wir bei der Erkennung ein bisschen nachhelfen. In unserem Fall ist es ausreichend zu sagen, welche Spalten ein Datum enthalten.

```
d = pandas.read_csv('car.csv', sep='\t', parse_dates=['Datum'])
```

4.3 Auf einzelne Daten zugreifen

Der Rückgabewert von `read_csv` ist ein **Data Frame**. Das sieht aus wie eine Tabelle. Was kann man damit nun machen?

```
d["Preis"]
```

```
0      13800.00
1         57.01
2        104.30
3         55.03
4         56.72
         ...
227       47.10
228       55.10
229       40.10
230       53.30
231       51.42
Name: Preis, Length: 232, dtype: float64
```

Einzelne Spalten kann man über den Spaltentitel adressieren. So eine Spalte nennt Pandas **Series**. Möchte man einen Wert aus so einer Series haben, so muss man zusätzlich dessen Zeilennummer angeben.

```
d["Preis"][0]
```

```
13800.0
```

Das Ganze geht auch umgekehrt. Eine Zeile bekommt man über den Array `iloc`:

```
d.iloc[0]
```

```
Datum           2012-07-07 00:00:00
Typ                           Kauf
Beschreibung              Autohaus
Preis                      13800.0
km                              30
Liter                          NaN
Name: 0, dtype: object
```

… und einen einzelnen Wert daraus über den Spaltennamen.

```
d.iloc[0]["km"]
```

```
30
```

Mit `head()` und `tail()` kann man zudem die ersten bzw. letzten N Spalten auswählen.

```
d.head(2).tail(1)
```

```
       Datum     Typ Beschreibung  Preis   km  Liter
1 2012-07-10  Benzin         ESSO  57.01  199  34.89
```

4.4 Aufgabe

Wie bekommt man die 4. bis 6. Zeile?

4.5 Lösung

Achtung: die 4. Zeile hat die Nummer 3 weil die Nummerierung mit 0 beginnt!

```
d.head(6).tail(3)
```

```
       Datum      Typ Beschreibung  Preis    km  Liter
3 2012-07-23   Benzin     Kaufland  55.03   828  34.20
4 2012-08-10   Benzin     Kaufland  56.72  1444  35.47
5 2012-08-23  Steuern   Kfz-Steuer  50.00  1500    NaN
```

4.6 Was wissen die Daten über sich selbst?

Der Data Frame kann über sich selbst etwas sagen:

```
d.dtypes
```

```
Datum           datetime64[ns]
Typ                     object
Beschreibung            object
Preis                  float64
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
km                      int64
Liter                 float64
dtype: object
```

Was bedeutet dies? Für jede Spalte wird angegeben, welchen Typ die darin enthaltenen Daten besitzen.

- datetime64[ns] - ist ein Zeitstempel bestehend aus Datum und Uhrzeit wobei letztere eine Genauigkeit von Nanosekunden hat
- object - das sind Texte
- float64 - Zahl mit Nachkommastellen.
- int64 - Zahl ohne Nachkommastellen

Über die Verteilung der Zahlen gibt die Funktion `describe()` Auskunft.

```
d.describe()
```

```
                               Datum         Preis             km       Liter
count                            232    232.000000     232.000000  201.000000
mean   2016-12-22 16:51:43.448275968    117.583578   53910.508621   31.144726
min              2012-07-07 00:00:00      0.000000      30.000000   10.140000
25%              2014-09-30 06:00:00     39.840000   27386.750000   29.890000
50%              2017-01-11 00:00:00     45.160000   53140.000000   32.640000
75%              2019-03-17 18:00:00     50.000000   81382.000000   33.650000
max              2021-12-25 00:00:00  13800.000000  106727.000000   37.930000
std                              NaN    904.538671   31374.136420    4.261039
```

4.7 Aufgabe

1. Welche Bedeutung haben diese Zahlen?
2. Welche davon helfen beim Verständnis der Daten?
3. Gibt es Datensätze, wo diese Funktion noch viel hilfreicher ist?

4.8 Lösung

Die besprechen wir im Kurs

4.9 Zeilen zählen

```
d.count()
```

```
Datum          232
Typ            232
Beschreibung   232
Preis          232
km             232
Liter          201
dtype: int64
```

In der Spalte **Liter** fehlen einige Einträge. Darum liefert `count()` für diese einen kleineren Wert. Nicht vorhandene Werte werden als **NaN** (Not a Number) angezeigt.

4.10 Werte zählen

```
d["Typ"].value_counts()
```

```
Typ
Benzin     201
Versich     10
Steuern     10
Werkst       9
Kauf         2
Name: count, dtype: int64
```

4.11 Rechnen

Welche Kosten sind insgesamt angefallen? Dazu muss man alle Einträge der Spalte `Preis` aufsummieren.

```
d["Preis"].sum()
```

```
27279.390000000003
```

4.12 Aufgabe

Was ist der Durchschnittsverbrauch des Autos über die gesamte erfasste Zeit?

Hinweise

- Gesamtverbrauch, d.h. wieviele Liter wurden insgesamt verbraucht
- Fahrstrecke, d.h. km-Stand am Ende - km-Stand am Anfang
- Verbrauch wird i.d.R. in l/100km angegeben

4.13 Lösung

```
l_total  = d["Liter"].sum()

km_start = d["km"].min() # oder d["km"][0] -> km-Stand aus der 0ten Zeile
km_end   = d["km"].max()   # oder d["km"].iloc[-1] -> km-Stand aus der letzten␣
↪Zeile,
                         # geht man von 0 eins zurück fängt man am Ende wieder an

km_total = km_end - km_start

fuel_avg = 100 * l_total / km_total
fuel_avg
```

```
5.867165899697274
```

4.14 Daten filtern

Wenn man nur an einem Teil der Daten interessiert ist kann man sich diesen selektieren.

```
fuel_only = d[d["Typ"] == "Benzin"]
fuel_only.head(2)
```

```
       Datum     Typ Beschreibung  Preis   km  Liter
1 2012-07-10  Benzin         ESSO  57.01  199  34.89
3 2012-07-23  Benzin     Kaufland  55.03  828  34.20
```

4.15 Aufgabe

1. Selektiere die Zeilen, bei denen der Preis kleiner als 100 ist.
2. Selektiere die Zeilen, bei denen der Preis gleich 50 ist.
3. Selektiere die Zeilen, bei denen Liter größer 36 und kleiner 38 sind.

4.16 Lösung

```
d[d["Preis"] < 100]
d[d["Preis"] == 50]
# d.described() sagte, dass der größte Wert 37.93 ist
d[d["Liter"] > 36]
d[d["Liter"].between(36,38)]
```

```
         Datum     Typ Beschreibung  Preis     km  Liter
6   2012-09-10  Benzin         AVIA  60.80   2061  36.87
7   2012-09-14  Benzin          OMV  61.10   2710  36.83
106 2016-09-15  Benzin     Kaufland  48.51  48872  37.93
124 2017-05-30  Benzin         Real  46.78  57868  36.01
```

4.17 Daten Gruppieren

Die Kosten sollen nach Spalte `Typ` bzw `Beschreibung` aufsummiert werden.

```
d.groupby("Typ")["Preis"].sum()
```

```
Typ
Benzin      8704.15
Kauf       14296.68
Steuern      500.00
Versich     2596.02
Werkst      1182.54
Name: Preis, dtype: float64
```

```
fuel_only.groupby("Beschreibung")["Preis"].sum()
```

```
Beschreibung
AGIP            32.80
ARAL           795.33
AVIA           671.59
Autohof         45.30
Avanti          64.10
BFT             88.66
BP              46.05
Bavaria         44.00
ESSO          1320.52
Elf             45.00
Globus         798.94
HEM             27.70
JET           1074.80
KK             183.34
Kaufland      1080.48
OMV             95.38
Oil            261.71
Real           222.15
SHELL          539.30
Star            99.80
Tango           56.85
Tankcenter     413.43
Total          481.25
UNO-X           43.36
Unbekannt      172.31
Name: Preis, dtype: float64
```

Das ist doch nett. Noch schöner wäre aber, wenn die Daten nach der zweiten Spalte absteigend sortiert wären. Zudem ist man oft nur an den größten Ergebnissen interessiert.

```
fuel_only.groupby("Beschreibung")["Preis"].sum().sort_values(ascending=False).
↪head(10)
```

```
Beschreibung
ESSO          1320.52
Kaufland      1080.48
JET           1074.80
Globus         798.94
ARAL           795.33
AVIA           671.59
SHELL          539.30
Total          481.25
Tankcenter     413.43
Oil            261.71
Name: Preis, dtype: float64
```

KAPITEL 5

Daten visualiseren

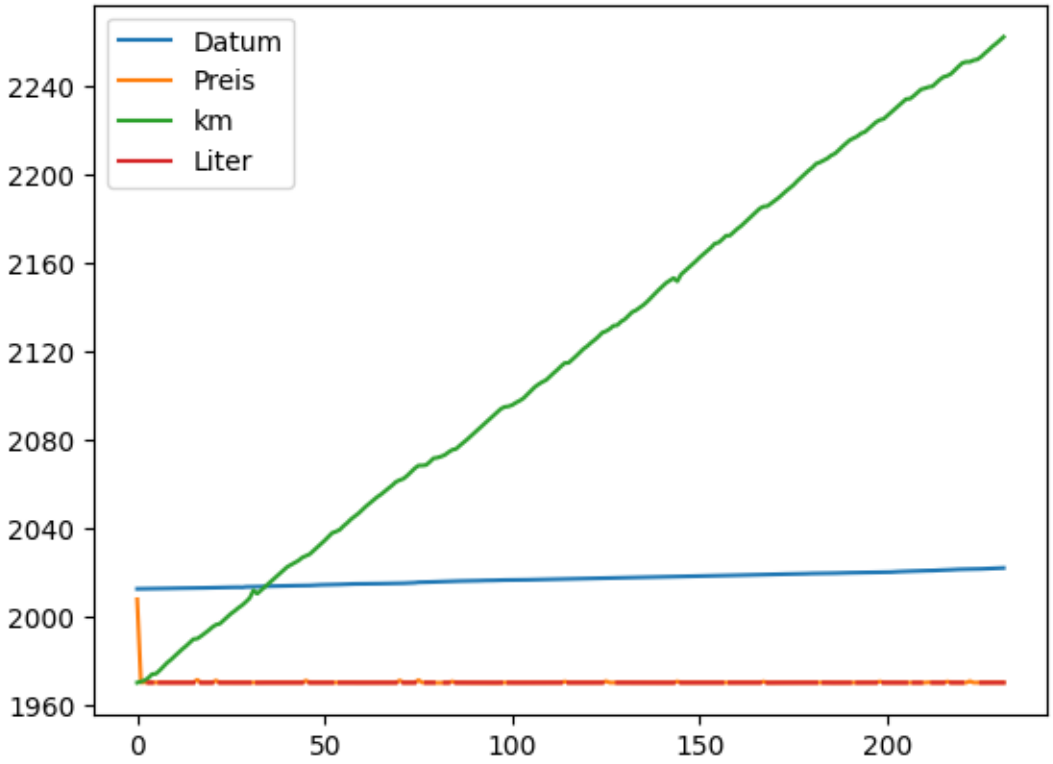
```
import pandas
d = pandas.read_csv('car.csv', sep='\t', parse_dates=['Datum'])
fuel_only = d[d["Typ"] == "Benzin"]
```

5.1 Alles auf einmal

Mittels `plot()` werden alle Zahlenspalten in einer Grafik dargestellt.

Hinweis: auf https://jupyter.org/try-jupyter/lab/ wird nur dann eine Grafik angezeigt wenn man explizit `show()` aus `matplotlib.pyplot` aufruft.

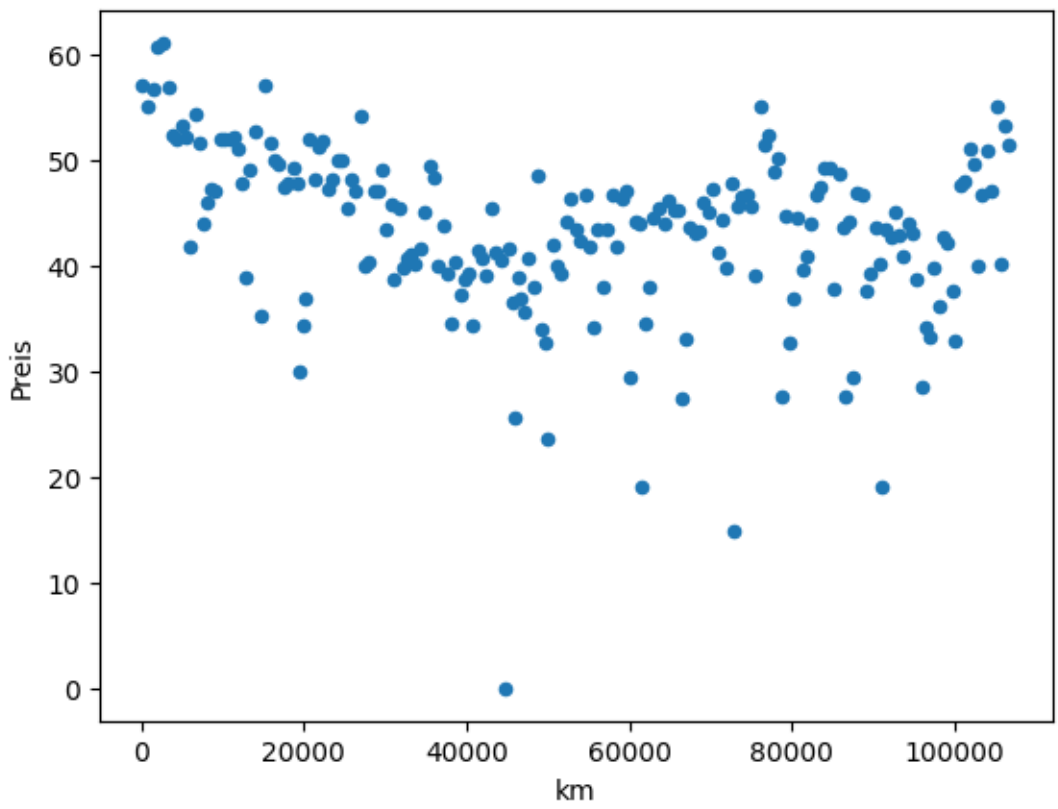
```
import matplotlib.pyplot as plt

d.plot();
plt.show()
```



Das ist in diesem Fall nicht besonders hilfreich weil jede Spalte eine andere Bedeutung hat. Bei Messwerten, bei dem jede Spalte ein anderes Experiment ist könnte dies aber nützlich sein.

5.2 Zwei Spalten als X-Y-Koordinaten darstellen

```
fuel_only.plot(kind="scatter", x="km", y="Preis");
plt.show()
```



Das sieht schon besser aus. Allerdings sind die Zahlen nicht wirklich vergleichbar. Besser wäre, den Preis / l darzustellen. Den müssen wir aber erst berechnen.

```
ppl = fuel_only["Preis"] / fuel_only["Liter"]
```

Jetzt haben wir eine Spalte `ppl` (Preis pro l) und die Tabelle `fuel_with_ppl`. Die Spalte `ppl` soll aber in die Tabelle eingefügt werden. Dazu benötigt sie zuerst einmal einen Namen.

```
ppl_with_name = ppl.rename("Preis_pro_l")
```

`rename()` benennt aber `ppl` nicht einfach um sondern erzeugt eine neue Spalte die den Namen enthält. Diese kann kann mit der Tabelle vereinigt werden wobei wiederum eine neue Tabelle entsteht.

```
fuel_with_ppl = fuel_only.join(ppl_with_name)
```

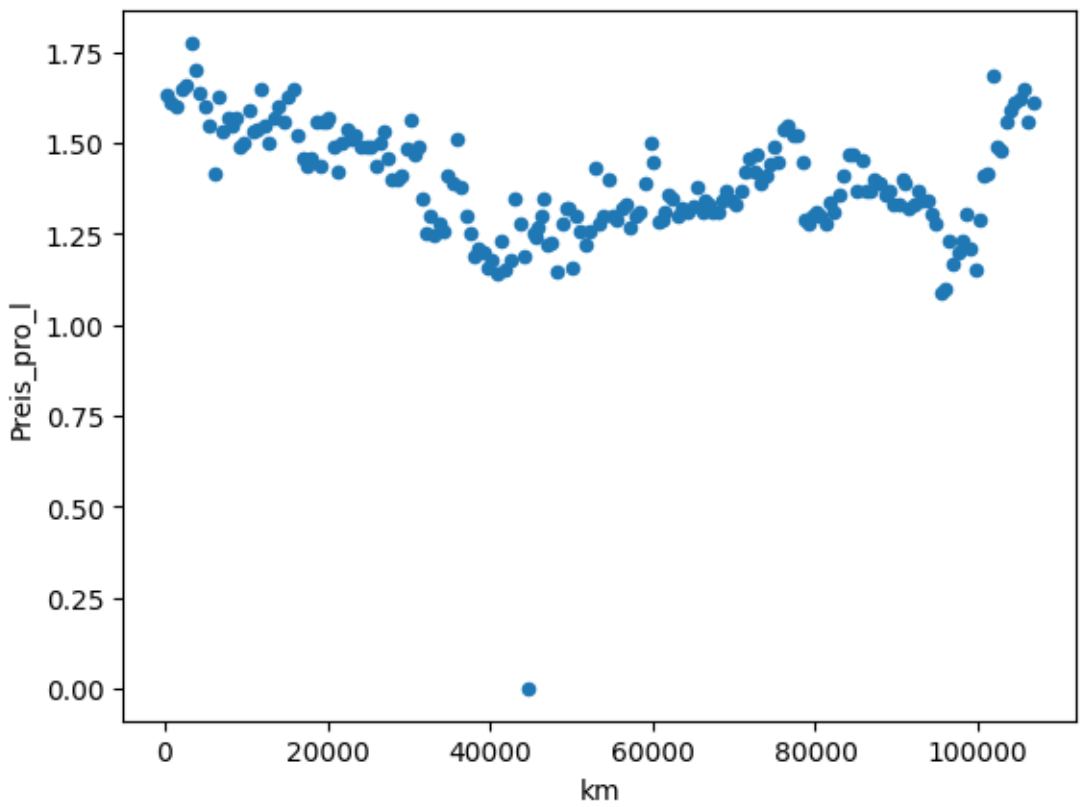
Diese drei Befehle kann man auch in einem Zusammenfassen. Damit spart man sich neue Variablennamen.

```
fuel_with_ppl = fuel_only.join((fuel_only["Preis"] / fuel_only["Liter"]).rename(
↪"Preis_pro_l"))
fuel_with_ppl.head(2)
```

```
        Datum     Typ Beschreibung  Preis   km  Liter  Preis_pro_l
1 2012-07-10  Benzin         ESSO  57.01  199  34.89     1.633993
3 2012-07-23  Benzin     Kaufland  55.03  828  34.20     1.609064
```

Damit können wir jetzt die Treibstoffkosten in Abhängigkeit vom km-Stand darstellen.

```
fuel_with_ppl.plot(kind="scatter", x="km", y="Preis_pro_l");
plt.show()
```



Interessant ist jetz noch der Verbrauch des Fahrzeugs, d.h. wieviele Liter pro 100 km es verbraucht hat. Dazu muss man die Differenz zwischen jeweils zwei km-Ständen berechnen. Dies geschieht mit der Funktion `diff()` die man auf eine Spalte der Daten anwendet.

5.3 Aufgabe

1. Was macht die Funktion `diff()` genau?
2. Bei welchen Daten kann man die noch verwenden?

5.4 Lösung

Schauen wir uns das mal genauer an. Zuerst die Daten

```
fuel_with_ppl["km"].head(5)
```

```
1     199
3     828
4    1444
6    2061
7    2710
Name: km, dtype: int64
```

```
fuel_with_ppl["km"].diff().head(5)
```

```
1      NaN
3    629.0
4    616.0
6    617.0
7    649.0
Name: km, dtype: float64
```

`diff()` berechnet die Differnz von zwei aufeinanderfolgenden Zeilen. Den Abstand kann man aber auch selbst festlegen. Möchte man die Differenz von jeweils den übernächsten Zeilen muss man den Parameter `periods` verwenden.

```
fuel_with_ppl["km"].diff(periods=2).head(5)
```

```
1       NaN
3       NaN
4    1245.0
6    1233.0
7    1266.0
Name: km, dtype: float64
```

5.5 Weiter mit der Verbrauchsberechung

Wir erzeugen jetzt eine neue **Series** in unserem **Data Frame** die gefahrenen km zwischen zwei Tankstopps enthält. Mit der Funktion `diff()` geht das ohne Umwege:

```
fuel_with_ppl["km_driven"] = fuel_with_ppl["km"].diff()
fuel_with_ppl.head(2)
```

```
       Datum     Typ Beschreibung  Preis   km  Liter  Preis_pro_l  km_driven
1 2012-07-10  Benzin         ESSO  57.01  199  34.89     1.633993        NaN
3 2012-07-23  Benzin     Kaufland  55.03  828  34.20     1.609064      629.0
```

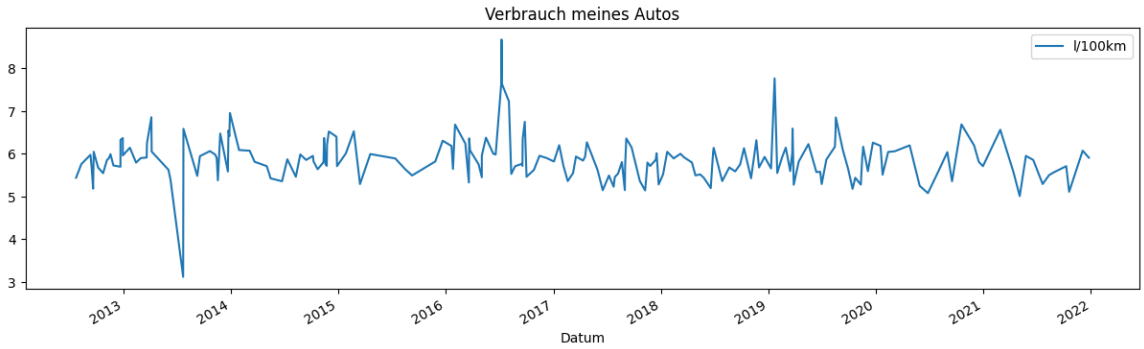
Nun können wir den Verbrauch berechnen …

```
fuel_all = fuel_with_ppl.join((fuel_with_ppl["Liter"] * 100 / fuel_with_ppl["km_
↪driven"]).rename("l/100km"))
fuel_all.head(2)
```

```
       Datum     Typ Beschreibung  Preis   km  Liter  Preis_pro_l  km_driven  \
1 2012-07-10  Benzin         ESSO  57.01  199  34.89     1.633993        NaN
3 2012-07-23  Benzin     Kaufland  55.03  828  34.20     1.609064      629.0

    l/100km
1       NaN
3  5.437202
```

… und den Verbrauch in der Grafik darstellen. Dabei ändern wir gleich noch ein paar Parameter um zu zeigen, was es alles so für Möglichkeiten gibt.

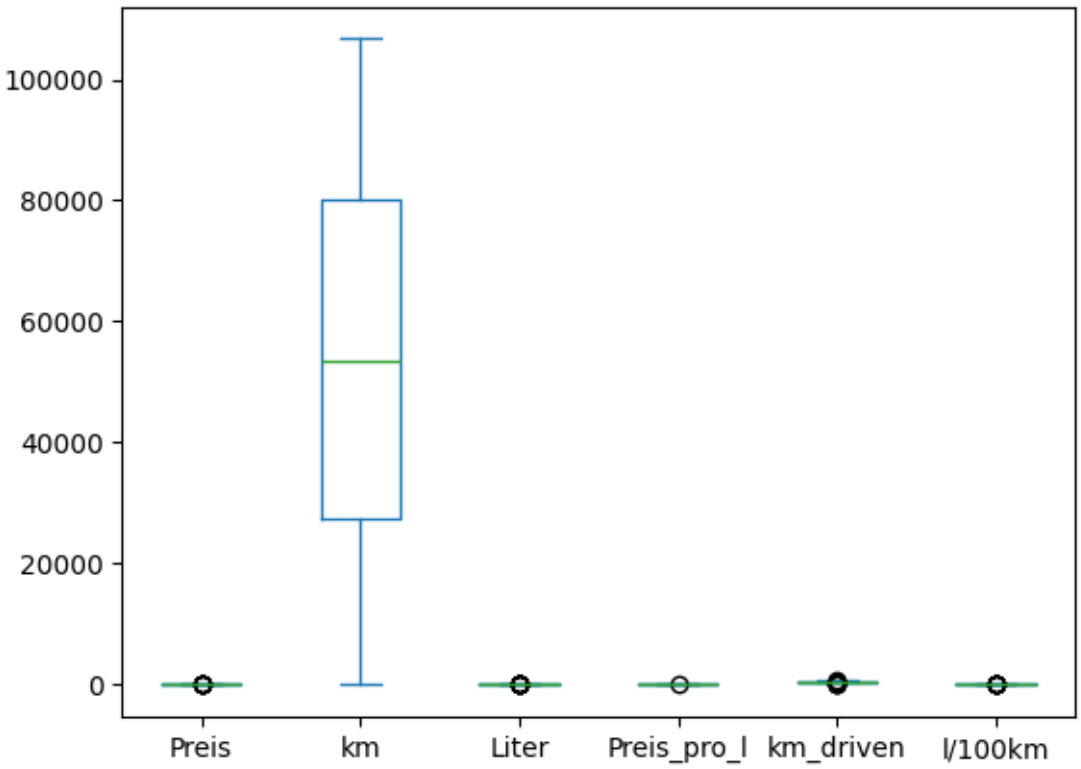
```
fuel_all.plot(x="Datum", y="l/100km", figsize=(15,4), title="Verbrauch meines␣
↪Autos");
plt.show()
```



5.6 Box-and-Whisker-Plots

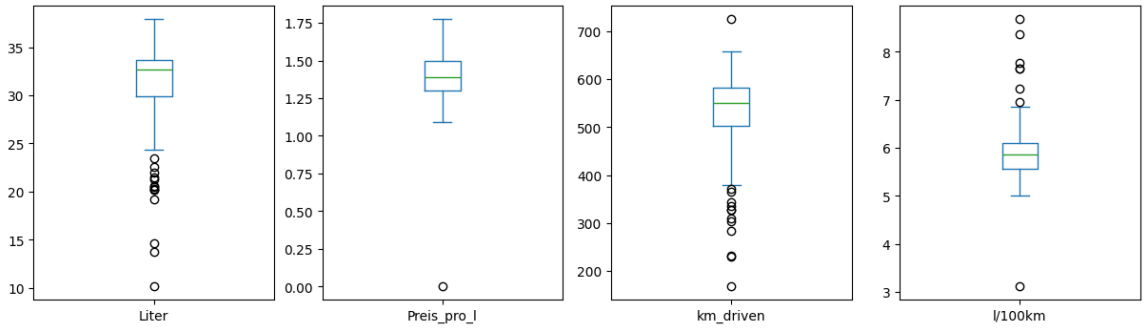
Auf einen Blick sehen, wie die Daten verteilt sind. Wer erinnert sich noch an `describe()`?

```
fuel_all.plot(kind="box");
plt.show()
```



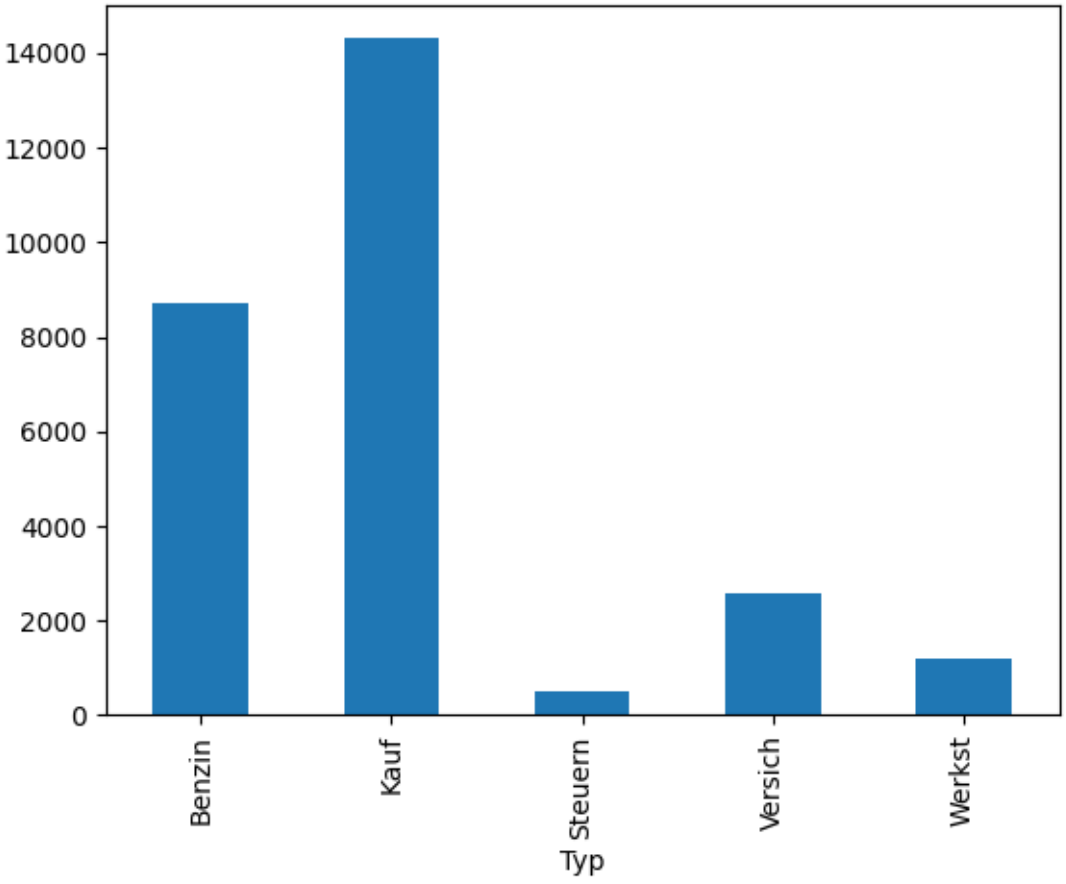
Wenn sich die Werte in den Spalten voneinander unterscheiden ist es besser, wenn man die Daten einzeln visualisert:

```
fuel_all[["Liter", "Preis_pro_l", "km_driven", "l/100km"]].plot(kind="box",␣
↪subplots=True, figsize=(15,4));
plt.show()
```



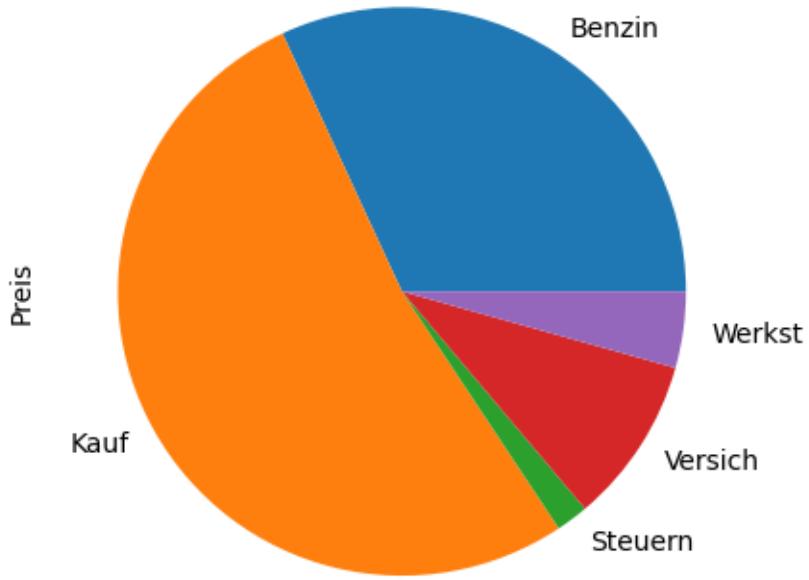
5.7 Bar-Charts

```
d.groupby("Typ")["Preis"].sum().plot(kind="bar");
plt.show()
```



5.8 Pie-Charts

```
d.groupby("Typ")["Preis"].sum().plot(kind="pie");
plt.show()
```



5.9 Grafiken speichern/exportieren

Zum speichern der Grafiken muss man sich zuerst mittels `get_figure()` einen Verweis auf die Grafik holen Danach kann man über diese Referenz die Grafik in verschiedenen Formaten speichern:

```
fig = d.groupby("Typ")["Preis"].sum().plot(kind="pie").get_figure();
fig.savefig('pie.png');
fig.savefig('pie.pdf');
```