

Table of Contents

1 Jupyter.....	5
1.1 Was ist Jupyter Notebook ?.....	5
1.1.1 Arbeitsfläche.....	6
1.1.2 Menü.....	7
1.1.3 Kernel.....	7
1.1.4 Werkzeugleiste.....	8
1.1.4.1 Befehlspalette.....	9
1.2 Allgemeiner Aufbau.....	10
2 Pandas.....	10
2.1 Pandas importieren.....	10
2.2 Daten.....	10
2.2.1 CVS Dateien.....	10
2.2.2 Pandas DataFrame.....	11
2.2.3 Eine DataFrame anlegen.....	12
2.2.4 Dimension der DataFrame ermitteln.....	12
2.2.5 Spaltentypen anzeigen.....	13
2.2.6 Informationen über DataFrame.....	13
2.2.7 Zugriff auf Spalten.....	13
2.2.8 Spalten Umbenennen.....	14
2.2.9 Spalten hinzufügen.....	14
2.2.10 Datensätze einer Spalte ändern.....	15
2.2.11 Einzelne Datensätze ändern.....	15
2.2.12 Datensätze mit Schlüsselwert filtern.....	15
2.2.13 CSV-Daten einlesen.....	16
2.2.14 Kopfdaten anzeigen.....	16
2.2.15 Rumpfdatensätze anzeigen.....	17
2.2.16 Eigenschaft values.....	18
2.3 Daten kopieren.....	18
2.4 Datensätze zählen.....	19
2.4.1 Einmalige Datensätze Zählen.....	20
2.4.2 Werte sortieren.....	21
2.4.3 Spalten gruppiert und Auswerten.....	22
2.4.4 Einzelne Spalten auswerten.....	23
2.5 NaNs.....	24
2.5.1 NaNs definieren.....	24
2.5.2 NaNs anzeigen.....	24
2.5.3 Feststellen, ob NaNs vorhanden sind.....	25
2.5.4 NaNs zählen.....	25
2.5.5 NaNs auffüllen.....	26
2.5.6 NaNs Automatisch auffüllen.....	26
2.5.7 Menge der zu füllenden Datensätze limitieren.....	27
2.5.8 Anwendungsfall: NaNs mit Durchschnittswert füllen.....	28
2.5.9 Datensätze mit NaNs entfernen.....	29
2.5.10 Spalten mit NaNs entfernen.....	29
2.5.11 Spalten entfernen.....	30

Einführung in das Maschinelle Lernen

2.5.12	Datensätze entfernen.....	31
2.5.13	Datensätze selektiv entfernen.....	32
2.6	Mathematische Operationen.....	33
2.6.1	Summe ermitteln.....	33
2.6.2	Maximum und Minimum bilden.....	34
2.6.3	Differenz bilden.....	35
2.6.4	Addition und Subtraktion.....	35
2.6.5	Spalten indexieren.....	36
2.6.6	DataFrames zusammenfügen.....	36
2.6.7	DataFrames indiziert zusammensetzen.....	37
2.6.8	DataFrames expandieren.....	39
2.6.9	Datensätze nach Bezeichnern Selektieren.....	40
2.6.10	Selektion mit Wahrheitsliste.....	41
2.6.11	Datensätze nach Position selektieren.....	42
2.6.12	Selektion von Zeilen und Spalten.....	43
2.6.13	Funktionen auf Datenfeldern ausführen.....	44
2.7	Datensätze Plotten.....	45
2.7.1	Balkengrafik.....	46
2.7.2	Histogramm.....	46
2.7.3	Dichte-Diagramm.....	47
2.7.4	Liniendiagramm.....	47
2.7.5	Area Plot.....	48
2.7.6	Boxplot.....	48
2.7.7	Kuchendiagramm.....	49
2.7.8	Scatter.....	49
2.7.9	HexBin.....	50
2.7.10	Plots logarithmisch Skalieren.....	51
2.7.11	Linienformat festlegen.....	51
2.7.12	Legende.....	52
2.8	Histogramme.....	53
2.8.1	Die „Behälter“ eine Struktogramms.....	55
2.8.2	Nichtlineare Histogramme.....	56
2.9	1 aus n (One hot encoding).....	57
2.9.1	Einzelne Spalten umwandeln.....	57
2.9.2	Mehrere Spalten gleichzeitig umwandeln.....	58
3	Sklearn.....	59
3.1.1	Daten in Training- und Testdaten aufteilen.....	59
3.2	Label Encoder.....	61
3.3	Logistische Regression.....	61
3.3.1	Eine Logistische Regression erstellen.....	61
3.3.2	Fit.....	61
3.3.3	predict.....	61
3.3.4	Genauigkeit berechnen.....	62
3.3.5	Präzision berechnen.....	62
3.3.6	Trefferquote ermitteln.....	62

Vorwort

Der Schatz der Pandas

Anmerkung des Übersetzers : Es ist das letzte übrig gebliebene Stück einer ~~leider~~ längst vergangenen Zivilisation. Es ist das einzige Vermächtnis der Pandas, das aufgeschrieben werden konnte bevor meine V o r f a... irgendjemand die Pandas von der Bildfläche des Universums sehr gründlich und elegant ausradiert hat. Wer es auch immer war, **möge seine glorreiche Zivilisation auf ewige Zeiten das Universum beherrschen! Gott beschütze M33!**

Vor einer langen Zeit an einem ganz fernen Ort, irgendwo in einer längst vergessenen Ecke von Laniakea fängt unsere Geschichte an. Es handelt sich dabei um zwei Pandas, die in Ihrer Eigenschaft, Statistische Wissenschaftler zu sein, auf einer Reise gehen, um Ihre Welt zu retten. Aber zuerst hören wir uns mal an, wie diese Geschichte ihren Anfang nahm. Unsere Helden, die armen Pandas, waren Junkies, denn sie waren süchtig, da jeder von Ihnen krankhaft nach Daten besessen war. Sie strecken ihre Fühler nach Daten aus und nur noch mit mehr Daten kam die Begeisterung für den Tag wieder. Gierig wie eine Datenkrake sammelten sie alles, da nichts, aber gar nichts ihnen unwichtig war! Denn sie waren besessen, von ihrem Datenrausch, besessen von dem Gedanken, dass sie aus ihren Daten, den über Jahrtausende und Generationen von Generationen gesammelten Schatz, noch mehr zu lernen gab. Und der Schatz strahlte! Und wie strahlte er? Er strahlte unbegreifliches Wissen und unbeschränkte Macht aus! Aber so wie es immer ist mit der Macht, sie strahlt nach außen, macht aber von innen krank. Sie machte auch die Pandas krank. Der Schatz, der allergrößte, ihr allerheiligster Schatz, nun, er kontrollierte die Pandas voll und ganz. Zu jener Zeit war jeder Panda davon berauscht, sie waren dicht, dicht so wie es nur ein B...ähm..Preuße nach dem Oktoberfest sein kann. Volltrunken wanderten die Pandas von R nach S und dann von S nach R zurück, aber es brachte nichts. Die Unordnung wuchs, es fraß ihre Welt auf. Das, wovon Sie sich immer am Meisten gefürchtet haben, das, was ihnen die meisten Angst gab, das Chaos, übernahm Stück für Stück ihr Heim, Datensatz für Datensatz. So brach das dunkle Zeitalter an. Wie ein schwarzer Nebel verbreitete sich diese Pest über Ihr Land. Ihr Schatz verblasste, langsam verlor es seinen ewigen Glanz. Aber war das so im ganzen Pandaland? Nein, ein kleines Dorf am Rande ihrer Welt leistete dem Nebel immer noch einem erbitterten Widerstand. Aber wie lange? Wie lange kann das noch sein? Das wussten die Pandas auch. Sie hofften auf ein Wunder, das Wunder ist das einzige, was sie noch retten kann. Das Wunder kam nicht, als ihr Dorf in Chaos versank.

So nahmen zwei der Pandas ihr Schicksal in die eigene Hand. Sie wanderten durch das vom Chaos gezeichnetes Land, das verdorrte Land, das noch vor kurzer Zeit ihre liebste Heimat war. Sie wanderten lange. Sehr lange und beschwerlich war ihre Wanderung. Sie suchten überall, sie suchten nach Rat. Bis zu jenem Morgen, bis er die Szene betrat. Es war Monty, die Schlange. Die Schlange, die alles kann. Die Schlange, die alles weiß. Die Pandas sahen ihn, er nahm gerade ein Bad, sein tägliches Lobesbad. Er sonnte sich im Lob, der aus allen Richtungen kam, Er genoss davon jeden einzelnen Strahl. Lob, der aus allen Richtungen kam? Ehrlich gesagt, nicht ganz, da war ein Schatten, ein schadhafter Fleck, ein kalter Wind, der aus einer kleiner, dunklen, versteckten Ecke

kam. Das war die Hardware, die seine Göttlichkeit nicht verstand. Aber das störte ihn nicht, er wusste, da kommt ein Tag, dieser einer besondere Tag, ab dem keiner mehr seiner Weisheit Widerstand leisten kann! An dem er auch dieses Terrain sein Eigen nennen kann! In Ihrer Verzweiflung, jetzt ganz dem Ende nah, fragten die Pandas den Monty um Rat. Und Monty wusste es, er gab ihnen einen Rat, und was für einen Rat er ihnen gab! Es war der beste Rat, den Monty jemals gab. Er meinte, in der Tat, da gibt es einen, der helfen kann. Einer, der jedes Chaos bändigen kann! Über Berge und Täler, über Flüsse und Seen wird der Weg führen. Zu dem heiligen, nein! Zu dem heiligsten Schrein. Genau zu diesem Schrein, in dem sich der Himmelsvater verbarg. Die Pandas atmeten auf, das war es, was ihnen neue Kraft gab.

Und sie wanderten durch Berge und Täler, durch globale Maxima und lokale Minima. Nicht mal in der kältesten Nacht machte ihnen jetzt die Regression Angst. Dann, lange war der Marsch, erblickten sie einen alten Schrein. Aber was für ein Schrein! Der Schrein war das schönste, was je ein Panda sah. Und ER war da, der Himmelsvater, Jupyter ist und war sein Name. Ein Name, der kein anderer in seiner Schönheit übertreffen kann. Und was für ein Himmelsvater er war! Er strahlte die unermessliche Kraft, eine Kraft, die außer ihm keiner besaß. ER hörte zu, ER nahm sich der Herausforderung an. Mit seiner göttlichen Macht, die alle anderen Mächte aller Welten übertraf, streifte er jetzt durch das Pandaland. Sortierte und ordnete er überall, trocknete die gefürchteten Sümpfe von NaNs. Sogar den Random Forest, wo der Chaos und damit das Unglück über das Pandaland seinem Anfang auf nahm, ordnete er, sodass jeder Baum jetzt wie ein Soldat in Reihe und Glied stand. So kam das Glück zurück ins Pandaland.

1 Jupyter

1.1 Was ist Jupyter Notebook ?

Jupyter Notebook ist, wie der Name schon vermuten lässt, eine Art Notizbuch für Python Programme. Es wird für gewöhnlich in einem Webbrowser-Fenster ausgeführt, kann aber auch [?]

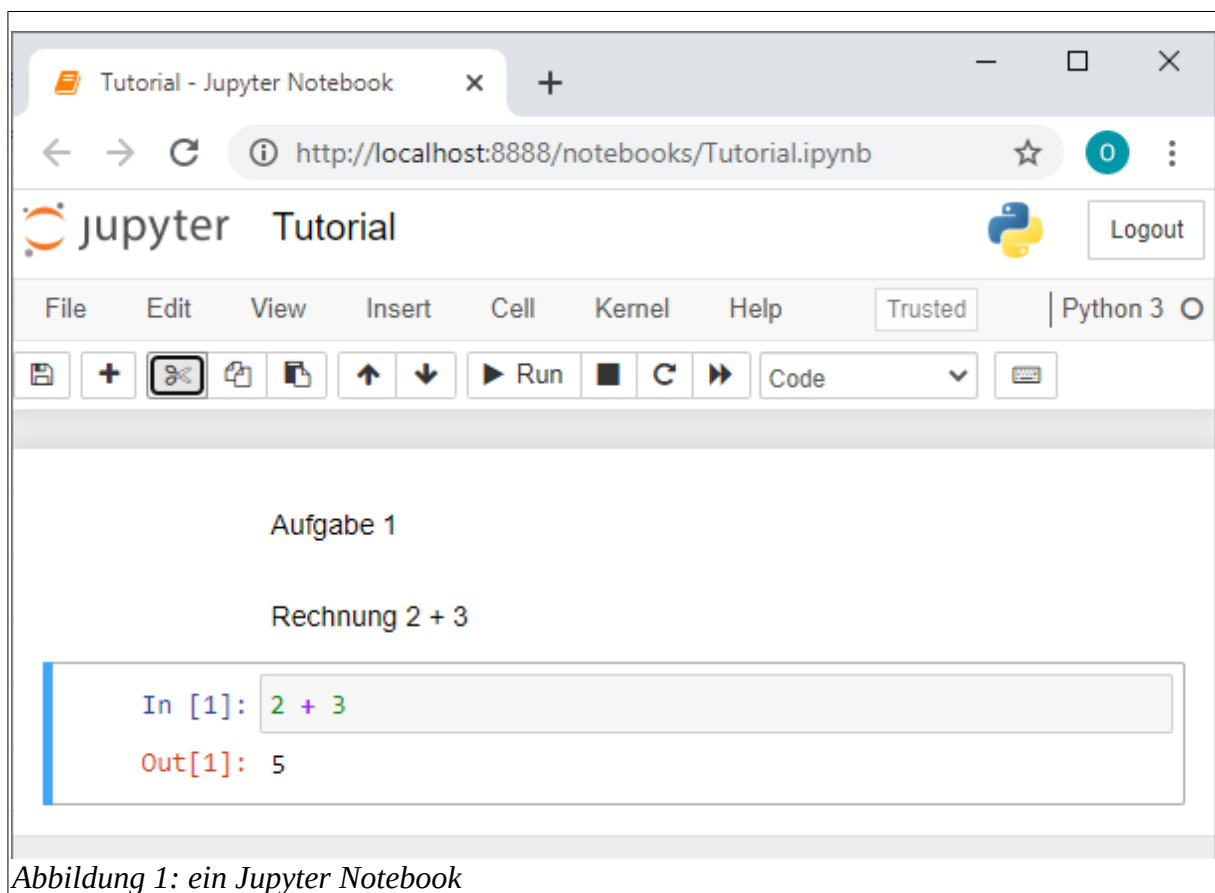


Abbildung 1: ein Jupyter Notebook

1.1.1 Arbeitsfläche

Die Arbeitsfläche ist die Stelle, wo die ganzen Aktionen in dem Notebook stattfinden. Sie ist in einzelne Arbeitszellen unterteilt. Diese Zellen können entweder Kommentare oder ausführbaren Code enthalten. In der Abbildung 2 sehen Sie eine einfache Oberfläche die aus zwei

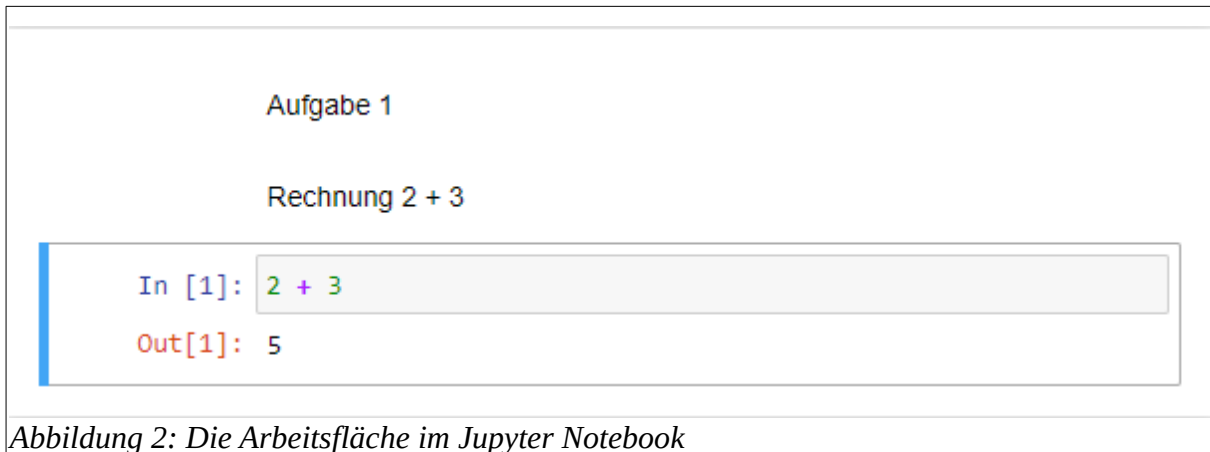


Abbildung 2: Die Arbeitsfläche im Jupyter Notebook

Kommentarzellen sowie einer Codezelle (umrahmt mit blauer Markierung). Die Kommentarzellen enthalten in der Regel Arbeitsanweisungen bzw. Kommentare, die in der folgenden Codezeile implementiert ist oder zu implementieren sind. Die Codezeilen können einzeln oder als gesamtes Notizbuch ausgeführt werden. Links von der Codezelle ist ein Schrittzähler sichtbar. Dieser (Notebook) globale Zähler zeigt in welchem Arbeitsschritt die Codezelle zuletzt ausgeführt wurde. In der Abbildung 3 wurde die Codezeile im 4tem Arbeitsschritt ausgeführt. Dabei wird dieser Zähler bei jeder Ausführung hochgezählt, auch wenn die Zelle gerade in dem vorhergehenden Schritt ausgeführt wurde. In der Abbildung wurde die Zelle (und nur diese Zelle) 4 mal ausgeführt.

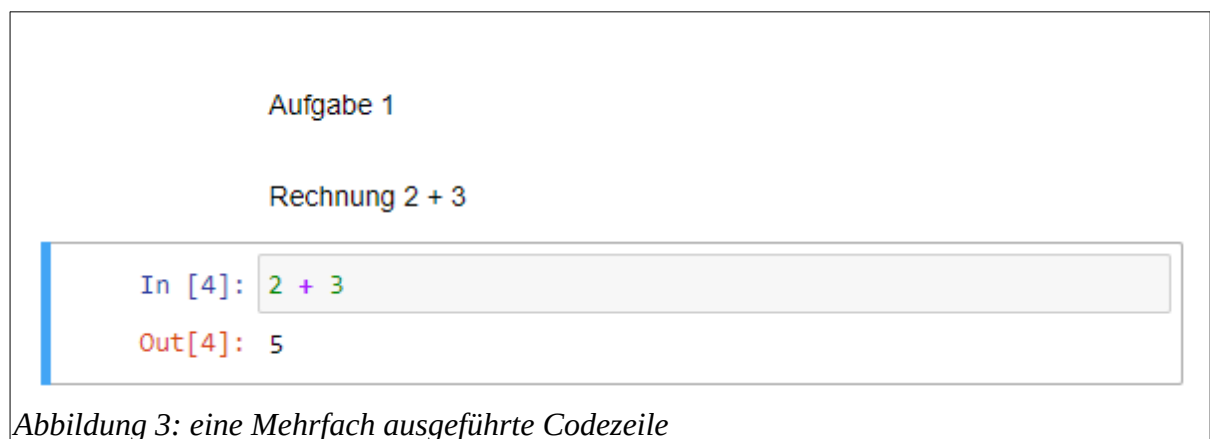


Abbildung 3: eine Mehrfach ausgeführte Codezeile

In die Codezellen wird der eigentliche Python Quelltext eingefügt. Wird eine Code Zeile ausgeführt erscheint das Ergebnis (Abbildung 3), oder eine Fehlermeldung (Abbildung 4), in dem „Out[]“ Bereich der Zelle.

```
In [2]: 2 + 3r
        File "<ipython-input-2-e7c5f43310db>", line 1
          2 + 3r
            ^
        SyntaxError: invalid syntax
```

Abbildung 4 Fehlermeldung in eine Zelle

1.1.2 Menü

1.1.3 Kernel














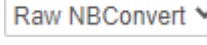

1.1.4 Werkzeugleiste

Über der Arbeitsfläche befindet sich die Werkzeugleiste, in der die gebräuchlichsten Befehle zur



Abbildung 5: Jupyter Notebook Werkzeugleiste

Konfiguration des Notebooks angebracht sind. Im Einzelnen sind das:

-  Speichern – Notebook wird gespeichert.
-  Eine Zelle hinzufügen – eine neue Arbeitszelle (Kommentar oder Codezeile) wird dem Notebook hinzugefügt.
-  Selektierte Zellen ausschneiden – entfernt die aktuell ausgewählten Zellen aus dem Notebook.
-  Selektierte Zellen kopieren – kopiert die aktuell ausgewählten Zellen in die Zwischenablage.
-  Zellen aus der Zwischenablage einfügen – fügt die in der Zwischenablage befindlichen Zellen in das Notebook ein.
-  Selektierte Zellen nach oben verschieben – die Selektierten Zellen werden eine Stufe nach oben verschoben
-  Selektierte Zellen nach unten verschieben – die Selektierten Zellen werden eine Stufe nach unten verschoben
-  Zelle Ausführen – die aktuelle Zeile wird ausgeführt, die folgende Zeile wird danach automatisch ausgewählt.
-  Stopp – die Ausführung des Kernel wird gestoppt.
-  Kernel Rücksetzen – der Kernel wird zurückgesetzt, alle Ergebnisse werden gelöscht, alle Ausgaben bleiben erhalten
-  Kernel Rücksetzen und ausführen – der Kernel wird zurückgesetzt, alle Ergebnisse werden gelöscht anschließend werden alle Zellen neu ausgeführt.
-  Oder  oder  – Zeigt bzw. Ändert den Typ der aktuellen Zelle.
-  Befehlspalette – hinter diesem Icon verbirgt sich eine Liste mit weiteren Notebook Befehlen. Sie wird beim Klicken auf dieses Icon geöffnet.

1.1.4.1 Befehlspalette

Durch einen Klick auf das „Tastatursymbol“ wird die Befehlspalette des Jupyter Notebooks geöffnet (Abbildung 6). Darin sind in einer Liste weitere Befehle, die der Kontrolle und Konfiguration des Notebooks dienen, enthalten. Über die Verwendung des Suchfeldes im oberen Teil der Liste lässt sich diese nach gewünschten Befehlen suchen (Abbildung 7).

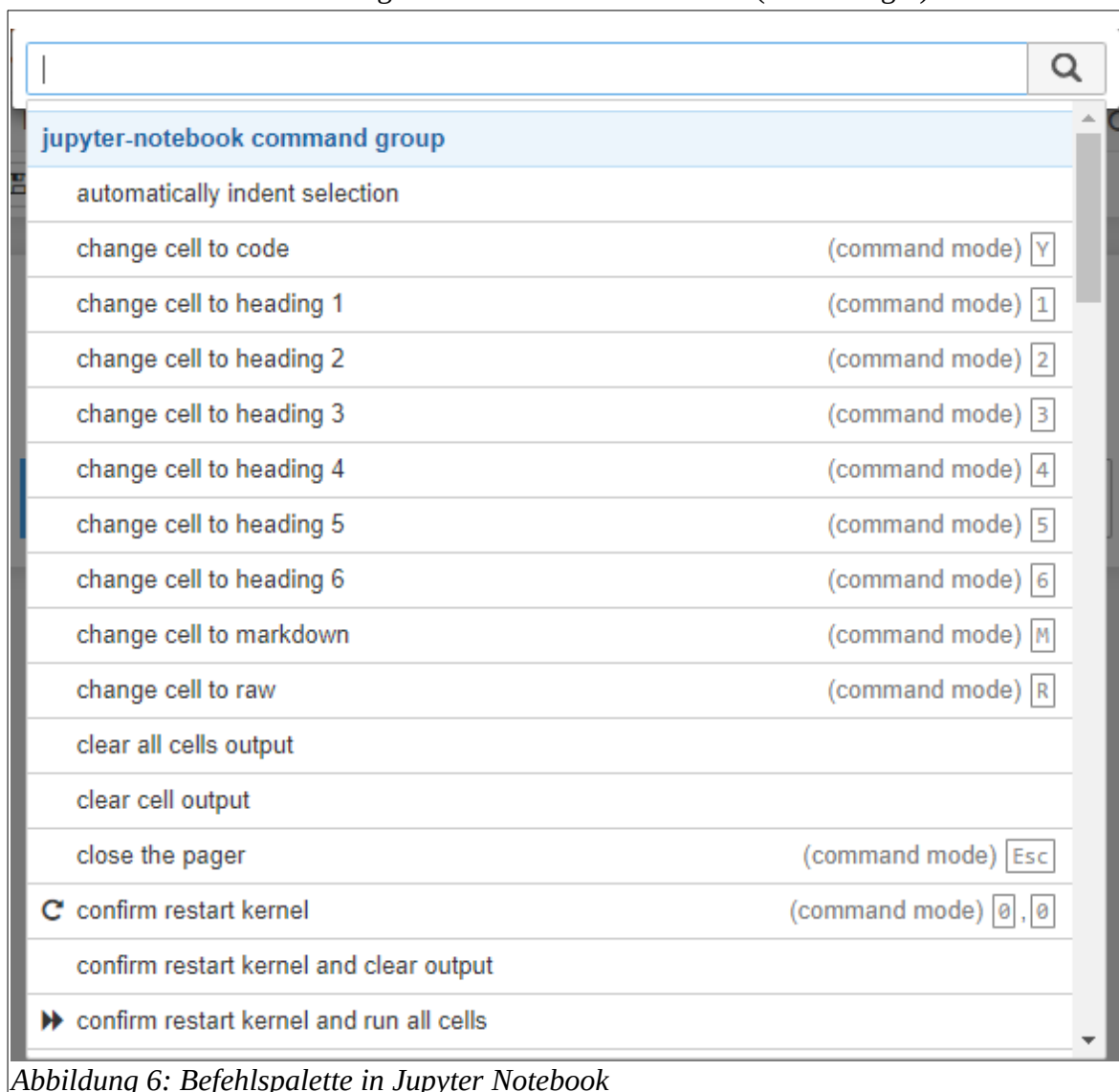


Abbildung 6: Befehlspalette in Jupyter Notebook

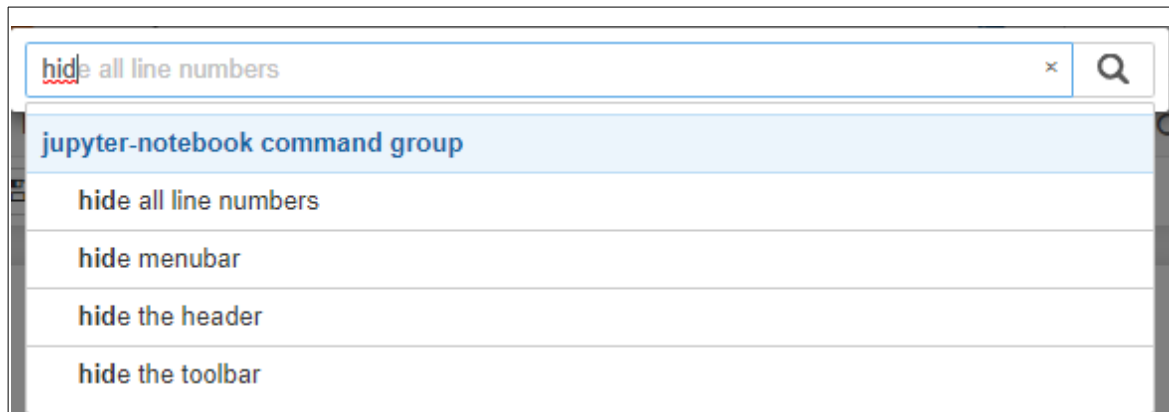


Abbildung 7: Befehlspalette durchsuchen

1.2 Allgemeiner Aufbau

2 Pandas

2.1 Pandas importieren

Bevor die Pandas-Bibliothek verwendet werden kann, muss diese zuerst importiert werden (Abbildung 8). Hat der Import der Bibliothek geklappt, kann das eigentliche Arbeiten mit dieser Bibliothek beginnen!

```
import pandas as pd
```

Abbildung 8 Pandas importieren

2.2 Daten

2.2.1 CVS Dateien

```
"Manufacturer","CPU","Technology","Package","Pins","Clock","Transistors", "Year"
"MOS","6502","NMOS", "PDIP",40 ,1.00,3510
"MOS","6502A","NMOS", "PDIP",40,2.00,3510
"MOS","6502B","NMOS", "PDIP",40,3.0,,
"Intel","8008","PMOS","PDIP",18,,3500, 1972
"Intel","8080","NMOS",CDIP,40,,6000, 1974
"Zilog","Z80","NMOS",CDIP,40,2.5,8500,1976
"Motorola","6800","NMOS" ,"PDIP",40, 1.0, 4100
"Motorola","6809","NMOS" ,"PDIP",40, 1.0, 9000,1978
"Motorola","68000","HMOS" ,"PDIP",64, 10.0, 68000,1979
```

Pseudocode 1 eine CSV-Datei

ML sind Daten, viele Daten, sehr viele Daten. Wie kommen nun diese Daten in das System „rein“? Die Antwort auf diese Frage heißt für gewöhnlich „CSV“ - „Comma-separated values“. Es handelt sich dabei um Textdateien, in denen Daten tabellarisch, durch Kommas getrennt, gespeichert sind. In dem Pseudocode 1 ist so eine Datei zu sehen. Die oberste Zeile dieser Datei beinhaltet die Namen der Spalten. In den folgenden Zeilen sind dann die einzelnen Datensätze definiert. Die einzelnen Daten eines Datensatzes sind dabei durch Separatoren getrennt. Dies können wie hier Kommas sein, aber auch Semikolons etc. sind möglich.

2.2.2 Pandas DataFrame

Die Struktur DataFrame speichert die Daten in tabellarischer Form als eine 2-dimensionale Tabelle. Genauso wie in einer CSV-Datei entspricht hier jede Zeile einem komplettem Datensatz (Abbildung 9).

```
print(cpu)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	"PDIP"	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	"PDIP"	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	"PDIP"	40	3.0	NaN	NaN
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0
4	Intel	8080	NMOS	CDIP	40	NaN	6000.0	1974.0
5	Zilog	Z80	NMOS	CDIP	40	2.5	8500.0	1976.0
6	Motorola	6800	NMOS	PDIP	40	1.0	4100.0	NaN
7	Motorola	6809	NMOS	PDIP	40	1.0	9000.0	1978.0
8	Motorola	68000	HMOS	PDIP	64	10.0	68000.0	1979.0

Abbildung 9 CPU Datensatz als Instanz der Pandas Klasse DataFrame

2.2.3 Eine DataFrame anlegen

Eine neue leere DataFrame-Instanz kann mit dem **.DataFrame({})**-Konstruktor angelegt werden (Abbildung 10). Sollte das neue DataFrame bereits von Anfang an mit Daten erstellt werden, kann dies z.B. mit Dictionarys erfolgen (Abbildung 11). Dabei wird der Schlüsselname als Name der entsprechenden Spalte und die dem Schlüssel zugewiesene Liste (oben) oder Tupel (unten) zu den Daten der Spalte. Da die Sets ungeordnet sind, funktioniert das mit den Sets nicht, da das Ergebnis, welche Daten in welchen Zeilen gespeichert wurden, rein zufällig wäre.

```
df = pd.DataFrame({})  
print(df)
```

```
Empty DataFrame  
Columns: []  
Index: []
```

Abbildung 10 einem leeren DataFrame anlegen

```
df = pd.DataFrame({"Name": ["Apfel", "Banane", "Orange"],  
                  "Farbe": ("rot", "gelb", "orange")})  
print(df)
```

	Name	Farbe
0	Apfel	rot
1	Banane	gelb
2	Orange	orange

Abbildung 11: einem DataFrame mit Daten anlegen

2.2.4 Dimension der DataFrame ermitteln

Die Dimension einer DataFrame, also die Anzahl der Zeilen sowie der Spalten, die in dem DataFrame enthalten sind, kann über die Eigenschaft `shape` ermittelt werden. Die in der Abbildung 12 „gemessene“ DataFrame beinhaltet 9 Datensätze (Zeilen) à 8 Spalten.

```
print(cpu.shape)
```

```
(9, 8)
```

Abbildung 12
Dimension einer
DataFrame

2.2.5 Spaltentypen anzeigen

Über die **DataFrame** Eigenschaft **.dtypes** lassen sich die Typen der einzelnen Spalten eines **DataFrame** anzeigen (Abbildung 13).

```
cpu.dtypes
Manufacturer    object
CPU              object
Technology       object
Package          object
Pins             int64
Clock            float64
Transistors      float64
Year             float64
dtype: object
```

Abbildung 13 Spaltentypen eines DataFrame anzeigen

2.2.6 Informationen über DataFrame

Allgemeine Informationen über einen DataFrame lassen sich über die Methode **.info()** anzeigen (Abbildung 14). Es werden die Struktur des DataFrames, die Informationen über den Typ der einzelnen Spalten, die Anzahl der Datensätze sowie die Anzahl der Daten, die nicht NaN sind. Als letzte Angabe wird die Menge des Speicherplatzes, die das aktuelle DataFrame verwendet angegeben.

```
df["Gewicht"]=[181,100,160]
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Name        3 non-null      object
1   Farbe       3 non-null      object
2   Geschmack   3 non-null      object
3   Gewicht     3 non-null      int64
dtypes: int64(1), object(3)
memory usage: 224.0+ bytes
```

Abbildung 14 DataFrame Informationen

2.2.7 Zugriff auf Spalten

Auf einzelne Spalten kann über den Spaltennamen als Index zugegriffen werden (Abbildung 15).

```
print(df["Name"])

0    Apfel
1    Banane
2    Orange
Name: Name, dtype: object
```

Abbildung 15 Zugriff auf einzelne Spalten

2.2.8 Spalten Umbenennen

Mit dem Befehl **rename** lassen sich die Namen der Spalten umbenennen. Als Parameter ist als Dictionary der gegenwärtige Name der Spalte, sowie der zukünftig gewünschte anzugeben (Abbildung 16).

```
print(cpu.head(1))
cpu.rename(columns={"Pins":"Beine"},inplace = True)
print(cpu.head(1))
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN

	Manufacturer	CPU	Technology	Package	Beine	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN

Abbildung 16: Eine Spalte umbenennen

Mit diesem Befehl lassen sich bei Bedarf, auch mehrere Spalten gleichzeitig umbenennen (Abbildung 17).

```
print(cpu.head(1))
cpu.rename(columns={"Pins":"Beine", "Clock" : "Uhr"},inplace = True)
print(cpu.head(1))
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN

	Manufacturer	CPU	Technology	Package	Beine	Uhr	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN

Abbildung 17 Mehrere Spalten gleichzeitig umbenennen

2.2.9 Spalten hinzufügen

Um eine Spalte hinzufügen reicht es diese per Index zu adressieren und dieser Spalte die Werte über eine entsprechende Kollektion zuzuweisen (Abbildung 18). Ist die so adressierte Spalte nicht vorhanden, dann wird diese durch den Zugriff automatisch erstellt.

```
print(df)
df["Geschmack"]=["Sauer","Kein","Perfekt"]
print("\n",df)
```

	Name	Farbe
0	Apfel	rot
1	Banane	gelb
2	Orange	orange

	Name	Farbe	Geschmack
0	Apfel	rot	Sauer
1	Banane	gelb	Kein
2	Orange	orange	Perfekt

Abbildung 18: Spalten Hinzufügen

2.2.10 Datensätze einer Spalte ändern

Bei bereits vorhandenen Spalten werden bei exakt der gleichen Vorgehensweise die Daten, die sich in dieser Spalte befinden, mit neuen Daten überschrieben (Abbildung 19).

```
df["Geschmack"]=["geht so","fehlt","süß"]
print(df)
```

	Name	Farbe	Geschmack
0	Apfel	rot	geht so
1	Banane	gelb	fehlt
2	Orange	orange	süß

Abbildung 19: Datensätze einer Spalte ändern

2.2.11 Einzelne Datensätze ändern

Über den doppelten Indexzugriff lassen sich die Datensätze auch direkt ändern (Abbildung 20).

```
df["Geschmack"][1]="nicht vorhanden"
print(df)
```

	Name	Farbe	Geschmack
0	Apfel	rot	geht so
1	Banane	gelb	nicht vorhanden
2	Orange	orange	süß

Abbildung 20 Einzelne Datensätze ändern

2.2.12 Datensätze mit Schlüsselwert filtern

Mit Angabe der entsprechenden Spalte, ist es möglich eine Liste aller Datensätze die dem gegebenen Suchkriterium entsprechen zu erstellen. Das Suchkriterium wird als eine Auswertungsoperation ($=$, \neq , $<$ usw.) mit dem zugehörigen/gesuchten Wert/Eigenschaft angegeben. Wie sie in der Abbildung 21 sehen können, ist das Ergebnis eine Liste mit Wahr/Falsch Werten. Dabei markiert ein Wahr Wert die Spalten die dem gewünschten Suchkriterium entsprechen. Diese Liste kann wiederum benutzt werden um aus einer Bestehenden DataFrame eine neue DataFrame zu erstellen in der nur die gewünschten Datensätze, also alle die in der Liste als „Wahr“ markiert sind, vorkommen (Abbildung 22). Das ganze lässt sich, wie in der Abbildung zu sehen ist, in einer Anweisung kaskadieren.

```
cpu["Package"] == "CDIP"
```

0	False
1	False
2	False
3	False
4	True
5	True
6	False
7	False
8	False

Name: Package, dtype: bool

Abbildung 21 Datensätze Auswählen

```
cdips = cpu[cpu["Package"] == "CDIP"]
print (cdips)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
4	Intel	8080	NMOS	CDIP	40	NaN	6000.0	1974.0
5	Zilog	Z80	NMOS	CDIP	40	2.5	8500.0	1976.0

Abbildung 22 Datensätze aus einer DataFrame filtern

2.2.13 CSV-Daten einlesen

Zum Einlesen der CSV-Daten ist in Pandas die Funktion `.read_csv()` zuständig. Sie lädt die CSV Daten und baut daraus eine entsprechende Instanz der Pandas Datenstruktur **DataFrame**. Fehlende Daten werden dabei durch sog. NaN (Not a Number) ersetzt. **ToDo Delimiter /sep**

```
cpu = pd.read_csv("D:/HS/ml/jupyter/cpu.csv")
print(type(cpu))
```

```
<class 'pandas.core.frame.DataFrame'>
```

Abbildung 23 Daten in Pandas laden

2.2.14 Kopfdaten anzeigen

Mit der Funktion **head(n)** lassen sich die ersten n Datensätze in dem entsprechenden **DataFrame** anzeigen (Abbildung 24). Beinhaltet der **DataFrame** weniger Datensätze als ausgegeben werden sollten, werden nur die vorhandenen Datensätze angezeigt. Wird diese Funktion ohne Parameter aufgerufen, werden 5 Datensätze angezeigt. Bei einem Negativen Index werden die letzten n Datensätze ausgelassen.


```
cpu.head(20)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	"PDIP"	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	"PDIP"	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	"PDIP"	40	3.0	NaN	NaN
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0
4	Intel	8080	NMOS	CDIP	40	NaN	6000.0	1974.0
5	Zilog	Z80	NMOS	CDIP	40	2.5	8500.0	1976.0
6	Motorola	6800	NMOS	PDIP	40	1.0	4100.0	NaN
7	Motorola	6809	NMOS	PDIP	40	1.0	9000.0	1978.0
8	Motorola	68000	HMOS	PDIP	64	10.0	68000.0	1979.0

Abbildung 24: Die ersten n Datensätze (Zeilen) mit `head()` anzeigen

Die Methode liefert ein Objekt der Klasse **DataFrame** zurück (Abbildung 25)

```
head = cpu.head(1)
print(type(head))
print(head)
```

```
<class 'pandas.core.frame.DataFrame'>
  Manufacturer  CPU Technology  Package  Pins  Clock  Transistors  "Year"
0      MOS    6502      NMOS    "PDIP"   40    1.0      3510.0    NaN
```

Abbildung 25 Rückgabe der Methode `head()`

2.2.15 Rumpfdatensätze anzeigen

Genauso wie die Kopfdaten mit der Methode **head()**, lassen sich die Rumpfdaten mit der Methode **tail()** anzeigen (Abbildung 26). Bei einem negativen Index werden die ersten n Datensätze ausgelassen.

```
cpu.tail(2)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
7	Motorola	6809	NMOS	PDIP	40	1.0	9000.0	1978.0
8	Motorola	68000	HMOS	PDIP	64	10.0	68000.0	1979.0

Abbildung 26 Die Letzten n Datensätze (Zeilen) mit `tail()` anzeigen

2.2.16 Eigenschaft values

Die Eigenschaft values eines DataFrames gibt die numpy-Repräsentation des DataFrames zurück.

Es wird allerdings empfohlen statt `to_numpy()` [?]

```
values = cpu.head(4).values
print(type(values))
print(values)

<class 'numpy.ndarray'>
[[ 'MOS' '6502' 'NMOS' 'PDIP' 40 1.0 3510.0 nan]
 [ 'MOS' '6502A' 'NMOS' 'PDIP' 40 2.0 3510.0 nan]
 [ 'MOS' '6502B' 'NMOS' 'PDIP' 40 3.0 nan nan]
 [ 'Intel' '8008' 'PMOS' 'PDIP' 18 nan 3500.0 1972.0]]
```

Abbildung 27

2.3 Daten kopieren

Weder die Methode `.head()` noch `.tail()` erzeugen keine echte Kopie eines DataFrames. Sie benutzen immer noch die Datenbasis des Originals. Wie Sie in der Abbildung 28 sehen, kann dies bei unvorsichtiger Änderungen der Tochterdatensätze die Originalbasis in Mitleidenschaft ziehen. Wie Sie in dieser Abbildung sehen können führte die Änderung an der **table**-Variable dazu, dass jetzt die ersten 3 Datensätzen ohne NaNs sind, die folgenden aber noch NaNs enthalten. So was kann unter Umständen dazu führen, dass die Datenbasis inkonsistent wird.

Erst bei der Verwendung der `.copy()`-Methode werden die Daten tatsächlich auch kopiert

```
table = cpu.head(3)
print(table)
table.fillna(-1, inplace=True)
print(table)
print(cpu)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	-1.0
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	-1.0
2	MOS	6502B	NMOS	PDIP	40	3.0	-1.0	-1.0
	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	-1.0
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	-1.0
2	MOS	6502B	NMOS	PDIP	40	3.0	-1.0	-1.0
	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	-1.0
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	-1.0
2	MOS	6502B	NMOS	PDIP	40	3.0	-1.0	-1.0
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0
4	Intel	8080	NMOS	CDIP	40	NaN	6000.0	1974.0
5	Zilog	Z80	NMOS	CDIP	40	2.5	8500.0	1976.0
6	Motorola	6800	NMOS	PDIP	40	1.0	4100.0	NaN
7	Motorola	6809	NMOS	PDIP	40	1.0	9000.0	1978.0
8	Motorola	68000	HMOS	PDIP	64	10.0	68000.0	1979.0

C:\Users\tavin\AppData\Roaming\Python\Python38\site-packages\pandas\core\frame.py:4317: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
return super().fillna()

Abbildung 28 Unerwünschte Veränderung der ursprünglichen Daten

Erst bei der Verwendung der `.copy()`-Methode werden die Daten tatsächlich auch kopiert (Abbildung 29), sodass die ursprüngliche Datenbasis unverändert bleibt.

```
table = cpu.head(3).copy()
print(table)
table.fillna(-1, inplace=True)
print(table)
print(cpu.head(3))
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	-1.0
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	-1.0
2	MOS	6502B	NMOS	PDIP	40	3.0	-1.0	-1.0
	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN

Abbildung 29 Unerwünschte Veränderung der ursprünglichen Daten mit `copy()` verhindern

2.4 Datensätze zählen

`.count`

`.describe`

```
df = pd.DataFrame({"Name": ["Apfel", "Banane", "Orange", "Zitrone"],
                   "Farbe": ("rot", "gelb", "orange", "gelb"),
                   "Geschmack": ["Sauer", np.nan, "Perfekt", "Sauer"],
                   "Gewicht": (185, 190, 160, 80)})

print(df.count())
```

```
Name      4
Farbe      4
Geschmack  3
Gewicht    4
dtype: int64
```

Abbildung 30 Anzahl der gültigen Daten in den Spalten zählen

Über die Angabe des Parameters **axis** lassen sich auch die gültigen Werte der einzelnen Datensätze zählen

```
print(df.count(axis = 1))
```

```
0    4
1    3
2    4
3    4
dtype: int64
```

Abbildung 31 Anzahl der Daten in einer Spalte zählen

2.4.1 Einmalige Datensätze Zählen

Mit der Methode **.unique()** kann untersucht werden wie viele verschiedene Datensätze in einer Spalte vorhanden sind. Die Methode liefert ein array mit in den jedes Datum nur einmal vorhanden ist (Ähnlich einem Set) die zurückgegebenen Daten sind nicht sortiert (Abbildung 32 & 33).

```
df = pd.DataFrame({"Name": ["Apfel", "Kiwi", "Banane", "Zitrone", "Orange"],
                  "Farbe": ("grün", "braun", "gelb", "gelb", "orange")})
print(df)
df["Farbe"].unique()
```

	Name	Farbe
0	Apfel	grün
1	Kiwi	braun
2	Banane	gelb
3	Zitrone	gelb
4	Orange	orange

```
array(['grün', 'braun', 'gelb', 'orange'], dtype=object)
```

Abbildung 32 Funktionsweise der Methode **.unique()**

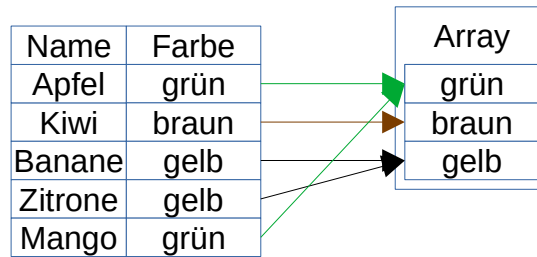


Abbildung 33: Funktionsweise der Methode `.unique()`

2.4.2 Werte sortieren

Um Werte nach einer bestimmten Spalte zu Sortieren kann die Methode `sort_values()` bemüht werden. Diese kann die Werte in Abhängigkeit von dem Attribut „ascending“ der der Größe nach auf- oder absteigend Sortieren. Als rückgabe parameter wird ein neuer DtaFrame mit der entsprechend sortierten Datensätze zurückgeliefert. Der Quellen DataFrame bleibt unangetastet und behält dabei seine ursprüngliche Sortierung.

```
fruchte =pd.DataFrame({"Name":["Apfel","Birne",
                              "Banane","mango"],
                      "Gewicht":(185,190,100,400)})
fruchte.sort_values("Gewicht")
```

	Name	Gewicht
2	Banane	100
0	Apfel	185
1	Birne	190
3	mango	400

```
fruchte.sort_values("Gewicht", ascending=False)
```

	Name	Gewicht
3	mango	400
1	Birne	190
0	Apfel	185
2	Banane	100

Abbildung 34

2.4.3 Spalten gruppiert und Auswerten

Eine Weitere Sehr mächtige Funktion ist die **groupby()** Methode. Diese gruppiert gleiche Datensätze aller Spalten zusammen. Als Parameter wird die Spalte angegeben, dessen Werte zusammen bearbeitet werden sollten. In der Abbildung 35 Ist es die Spalte „Name“ also werden alle Datensätze in

```
fruchte =pd.DataFrame({"Name":["Apfel","Birne",
                              "Apfel","mango"],
                      "Gewicht":(185,190,160,400)})
fruchte.groupby(["Name"]).max()
```

Gewicht	
Name	
Apfel	185
Birne	190
mango	400

Abbildung 35 Spalten gruppiert auswerten

Durch diese Spalte gruppiert und auf diese Gruppen wird die Funktion **max()** angewandt. Somit entstehen hier zuerst 3 Gruppen - alle Äpfel, alle Birnen und alle Mangos werden zusammengefügt. Anschließend wird das maximale Gewicht in jeder dieser Gruppe bestimmt. Da es zwei Datensätze mit den Namen „Apfel“ gibt werden dessen Gewichte (185 sowie 160) genommen und aus denen das Maximum ermittelt. Als Rückgabewert wird ein neuer Dataframe mit den Maximas zurückgegeben (Abbildung 36).

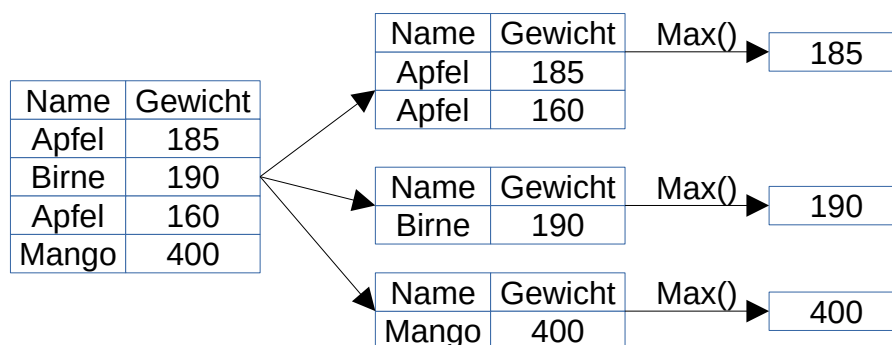


Abbildung 36: Funktionsweise der groupby() Methode

2.4.4 Einzelne Spalten auswerten

Einzelne spalten in einer DataFrame lassen sich auch alleine „für sich“ Auswerten. Je nach dem typ der Abfrage Das Ergebnis solch einer Auswertung kann ein elementarer Datentyp wie Float, oder aber auch ein Feld mit zu jeder Spalte zugehörendem Wert (Abbildung 37). In der Abbildung 37 im Fall zwei und drei wurde eine liste mit spalten zurückgegeben , als Objekt des Typs **Series**, in dem verzeichnet wurde ob die entsprechende Zeile das Kriterium der Bedingung entspricht (Wert „True“) oder nicht (Wert „False“). Dabei entspricht jede Zeile des zurückgegebenen Objekt einer Zeile in dem DataFrame.

df["Gewicht"] > 160)

Name	Gewicht	> 160 ?	Series
Apfel	185	→	True
Birne	190	→	True
Apfel	160	→	False
Mango	400	→	True

Abbildung 38: Funktionsweise einer Spaltenauswertung

Wie in der Abbildung 39 zu sehen ist, lassen sich derartige Abfragen auch über logische Auswertung miteinander kombinieren. Hier wurden alle Früchte die schwerer als 100 aber leichter als 190 Gramm durch Entsprechende **und** Verknüpfung ausgewählt und daraus ein neuer Dataframe erstellt. Bei der Erstellung des neuem Dataframes wurden nur die Zeilen übernommen, die bei dem in den erstellten **Series** Objekten beide Zeile/Zellen dem Wert Wahr („True“) beinhalteten.

```
print(type(df["Gewicht"].mean()))
print(df["Gewicht"].mean())
```

```
<class 'float'>
153.75
```

```
df["Geschmack"].str.startswith("S")
```

```
0      True
1      NaN
2     False
3      True
Name: Geschmack, dtype: object
```

```
print(type(df["Gewicht"] > 160))
print(df["Gewicht"] > 160)
```

```
<class 'pandas.core.series.Series'>
0      True
1      True
2     False
3     False
Name: Gewicht, dtype: bool
```

Abbildung 37 Spalten einer Dataframe auswerten.

```
print(df)
df[(df["Gewicht"] > 100) & (df["Gewicht"] < 190)]
```

	Name	Farbe	Geschmack	Gewicht
0	Apfel	rot	Sauer	185
1	Banane	gelb	NaN	190
2	Orange	orange	Perfekt	160
3	Zitrone	gelb	Sauer	80

	Name	Farbe	Geschmack	Gewicht
0	Apfel	rot	Sauer	185
2	Orange	orange	Perfekt	160

Abbildung 39 Spaltenauswertung kombinieren

2.5 NaNs

2.5.1 NaNs definieren

Sollten Werte als NaN definiert werden, so ist das über die numpy Bibliothek möglich (Abbildung 40).

```
import numpy as np
liste = ["Apfel", np.nan, "Orange", "Zitrone"]
print(liste)
```

```
['Apfel', nan, 'Orange', 'Zitrone']
```

Abbildung 40 ein Wert als NaN definieren

2.5.2 NaNs anzeigen

Die NaNs (Not a Number) entsprechen den fehlenden Daten in dem Datensatz. Damit mit den Datensätzen gearbeitet werden kann, müssen diese NaNs entfernt werden, dazu später. Zuerst schauen uns wir an, ob überhaupt NaNs in dem DataFrame vorhanden sind. Die Methode **.isna()** wandelt alle Ausdrücke in dem DataFrame, die nicht NaN sind, zu dem Wert Falsch (False) und alle NaNs zu dem Wert Wahr (**True**) (Abbildung 41). Als Gegenstück zu **.isna()** gibt es noch die Methode **.notna()** bei der die Auswertung umgekehrt ist.

```
table = cpu.head(4).isna()
print(type(table))

print(cpu.head(4))
print(table)
```

```
<class 'pandas.core.frame.DataFrame'>
```

	lanufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	"Year"
0	False	False	False	False	False	False	False	True
1	False	False	False	False	False	False	False	True
2	False	False	False	False	False	False	True	True
3	False	False	False	False	False	True	False	False

Abbildung 41: NaNs Anzeigen

Die beiden Funktionen **.isna()** und **.isnull()** sind identisch. Sie stammen noch aus der Sprache R, wo die beiden Werte **null** und **na** unterschiedliche Bedeutung haben. Diese gibt es in Python bzw. numpy, dort gibt es nur die NaNs. Gleiches gilt für die **.notna()** und **.notnull()**-Methoden.

2.5.3 Feststellen, ob NaNs vorhanden sind

Die Methode **.any()** prüft, ob einer der Elemente eines DataFrames den Wert Wahr (*True*) Aufweist. Da die DataFrame ein zweidimensionales Feld ist, prüft eine einfache Anwendung vom **.any()** zuerst in welchen Spalten NaNs vorhanden sind (Abbildung 42 oben). Erst die zweifache Anwendung von der **.any()** Methode liefert die gewünschte Auswertung über alle Felder in einem Wert (Abbildung 42 unten). Damit das funktioniert muss zuerst eine Umwandlung der Werte der DataFrames in boolesche Ausdrücke mit **.isna()** erfolgen.

```
table = cpu.isna()
print(table.any())
```

Manufacturer	False
CPU	False
Technology	False
Package	False
Pins	False
Clock	True
Transistors	True
"Year"	True
dtype:	bool

```
print(table.any().any())
```

True

Abbildung 42

2.5.4 NaNs zählen

Um die Menge der NaNs in einem DataFrame zu zählen gibt es mehrere Möglichkeiten. Die erste wäre die von der **.isna()**-Methode erstellte *values* Eigenschaft zu nutzen (über die **.to_numpy()** Methode), um direkt die Summe aller *True* Vorkommnisse in dem *ndarray* Objekt zu bilden (Abbildung 43).

```
nanCount = cpu.head(4).isna().to_numpy().sum()
print(nanCount)
```

5

Abbildung 43 NaNs zählen

Eine andere Möglichkeit, mit der man die NaNs zählen könnte, wäre die Methode **.sum()** zweimal hintereinander einzusetzen. Die erste Anwendung von **.sum()** zählt die Anzahl der NaNs pro Spalte (Abbildung 44), die zweite summiert schließlich das Vorkommen der NaNs in den einzelnen Spalten zu einem Gesamtergebnis (Abbildung 45).

```
nanCount = cpu.head(4).isna().sum().sum()
print(nanCount)
```

5

Abbildung 45 NaNs zählen

```
nanCount = cpu.head(4).isna().sum()
print(nanCount)
```

Manufacturer	0
CPU	0
Technology	0
Package	0
Pins	0
Clock	1
Transistors	1
"Year"	3
dtype:	int64

Abbildung 44 NaNs pro Spalte zählen

2.5.5 NaNs auffüllen

Da wir jetzt herausgefunden haben, dass wir NaN Werte in unserem Datensatz haben, können wir jetzt darüber nachdenken, wie wir diese loswerden. Eine Möglichkeit wäre, die NaNs selbst mit „sinnvollen“ Werten aufzufüllen. Es ist genau das, was die Funktion **.fillna()** macht.

```
table = cpu.head(3).copy()
print(table)
table.fillna(-1, inplace = True)
print(table)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	-1.0
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	-1.0
2	MOS	6502B	NMOS	PDIP	40	3.0	-1.0	-1.0

Abbildung 46 NaNs mit einem Wert (-1) Auffüllen

Normalerweise wird bei der Methode **.fillna()** ein neuer Dataframe erstellt. Ist dies nicht erwünscht so ist der Parameter **inplace** auf **True** zu setzen. Dadurch wird der ursprüngliche Datensatz geändert. Dass dies zu Problemen führen kann haben wir bereits in der Abbildung 28 gesehen.

2.5.6 NaNs Automatisch auffüllen

Über den Parameter **method**, lassen sich die NaNs über festgelegte Muster füllen. Ist der Parameter **ffill**, oder **pad** so werden die NaNs „nach vorne“ - mit dem zuletzt gültigen Wert aufgefüllt (Abbildung 47). Bitte beachten Sie das in dem Beispiel die ersten 3 Datensätze nicht aufgefüllt werden könnten, da es hier keinem vorhergehenden Wert gab.

```
cpu.fillna(method="ffill", inplace = True)
print(cpu[{"Manufacturer", "CPU", "Year"}])
```

	CPU	Manufacturer	Year
0	6502	MOS	NaN
1	6502A	MOS	NaN
2	6502B	MOS	NaN
3	8008	Intel	1972.0
4	8080	Intel	1974.0
5	Z80	Zilog	1976.0
6	6800	Motorola	1976.0
7	6809	Motorola	1978.0
8	68000	Motorola	1979.0

Abbildung 47 Auffüllen mit dem Wert des Vorgängers

Ist der Parameter hingegen auf **bfill** oder **backfill** eingestellt (Abbildung 48), so wird der Datensatz des Nachfolgers (backward - rückwärts) als Füllwert herangezogen. Beachte das der Datensatz 6 jetzt mit dem wert 1978, dem Wert seines Nachfolgers aufgefüllt wird

```
cpu.fillna(method="bfill", inplace = True)
print(cpu[{"Manufacturer", "CPU", "Year"}])
```

	CPU	Manufacturer	Year
0	6502	MOS	1972.0
1	6502A	MOS	1972.0
2	6502B	MOS	1972.0
3	8008	Intel	1972.0
4	8080	Intel	1974.0
5	Z80	Zilog	1976.0
6	6800	Motorola	1978.0
7	6809	Motorola	1978.0
8	68000	Motorola	1979.0

Abbildung 48 Auffüllen mit dem Wert des Nachfolgers

2.5.7 Menge der zu füllenden Datensätze limitieren

Über dem Parameter **limit**, lässt sich wiederum bestimmen wie viele der folgenden oder vorausgegangenen Datensätze ausgefüllt werden sollten. So ist im Beispiel (Abbildung 49) von den drei aufeinander folgenden Datensätzen nur in dem Datensatz 2 der NaN Wert aufgefüllt. Da die Zählung bei den Nächsten gültigen Wert wieder von vorne anfängt ist hier auch in dem Datensatz 6 der NaN Wert ersetzt worden.

```
cpu.fillna(method="bfill", inplace = True ,limit=1)
print(cpu[{"Manufacturer", "CPU", "Year"}])
```

	CPU	Manufacturer	Year
0	6502	MOS	NaN
1	6502A	MOS	NaN
2	6502B	MOS	1972.0
3	8008	Intel	1972.0
4	8080	Intel	1974.0
5	Z80	Zilog	1976.0
6	6800	Motorola	1976.0
7	6809	Motorola	1978.0
8	68000	Motorola	1979.0

Abbildung 49 Auffüllung limitieren

2.5.8 Anwendungsfall: NaNs mit Durchschnittswert füllen

Aus welchem Grund auch immer wollen wir jetzt die fehlenden Jahresangaben mit dem Durchschnittswert füllen (Abbildung 50). Dazu wird wieder die Methode `.fillna()` verwendet, allerdings wird diese diesmal nur auf die Jahresspalte angewendet. Die Anweisung `„table[„Year“]“` wählt dabei die entsprechende Spalte in dem DataFrame, in der die NaNs durch den Durchschnittswert ersetzt werden sollten. Als Parameter geben wir den durch die Methode `.mean()` berechneten Wert dieser Spalte. Bei der Berechnung des Durchschnittswerts werden die NaNs ausgelassen. Da wir die ganze Tabelle wieder haben wollen, müssen wir mit dem Parameter `inplace` arbeiten, da wir sonst nur die Jahresspalte als Rückgabeparameter bekommen würden.

```
table = cpu.copy()
print(table.head(4))

table["Year"].fillna(table["Year"].mean(), inplace = True)

print(table.head(4))
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0
	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	1975.8
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	1975.8
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	1975.8
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0

Abbildung 50 NaNs mit Durchschnittswerten füllen

2.5.9 Datensätze mit NaNs entfernen

Die andere Möglichkeit NaNs aus dem DataFrame loszuwerden wäre die entsprechenden Datensätze die NaNs enthalten zu löschen. Hierzu dient die Methode `.dropna()`. Sie löst alle Datensätze (Zeilen) die mindestens einem NaN Wert enthalten. Wie sie in der Abbildung 51 sehen können, sind von den 6 Datensätzen nach Anwendung von `.dropna()` 5 rausgeflogen, da sie NaNs enthalten haben.

```
table = cpu.head(6).copy()
print(table)
table.dropna(inplace = True)
print(table)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0
4	Intel	8080	NMOS	CDIP	40	NaN	6000.0	1974.0
5	Zilog	Z80	NMOS	CDIP	40	2.5	8500.0	1976.0

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
5	Zilog	Z80	NMOS	CDIP	40	2.5	8500.0	1976.0

Abbildung 51: Datensätze mit NaNs entfernen

2.5.10 Spalten mit NaNs entfernen

Die Methode `.dropna()` kann durch die Angabe des Parameters `axis = 1` dazu gebracht werden, statt Zeilen die Spalten, in denen NaNs enthalten sind, zu löschen (Abbildung 52).

```
table = cpu.head(3).copy()
print(table)
table.dropna(axis = 1, inplace = True)
print(table)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN

	Manufacturer	CPU	Technology	Package	Pins	Clock
0	MOS	6502	NMOS	PDIP	40	1.0
1	MOS	6502A	NMOS	PDIP	40	2.0
2	MOS	6502B	NMOS	PDIP	40	3.0

Abbildung 52: Spalten mit NaNs entfernen

2.5.11 Spalten entfernen

Sollten einzelne oder mehrere Spalten aus dem DataFrame entfernt werden, kann dies mit der **.drop()** Methode erledigt werden (Abbildung 53). Man kann dabei seinen destruktiven Neigungen entweder einzeln oder gebündelt mit einer Kollektion nachgehen. Zu beachten ist dabei, dass hier die Achse angegeben werden muss, da **.drop()** in der Standardeinstellung nicht Spalten, sondern Zeilen entfernt.

```
table = cpu.drop("Package", axis = 1)
print(table.head(2))
table = cpu.drop({"CPU", "Pins", "Clock", "Year"}, axis = 1)
print(table.head(2))
```

	Manufacturer	CPU	Technology	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	40	2.0	3510.0	NaN

	Manufacturer	Technology	Package	Transistors
0	MOS	NMOS	PDIP	3510.0
1	MOS	NMOS	PDIP	3510.0

Abbildung 53: Spalten mit drop() entfernen

Auch mit der Methode **.pop()** lassen sich einzelne Spalten aus einem DataFrame entfernen (Abbildung 54). Als Rückgabe bekommt man von der **.pop()** Methode in diesem Fall die entfernte Spalte zurück.

```
cpu2 = cpu.copy()
table = cpu2.pop("Package")
print(table.head(2))
print(cpu2.head(2))
```

0	PDIP
1	PDIP

Name: Package, dtype: object

	Manufacturer	CPU	Technology	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	40	2.0	3510.0	NaN

Abbildung 54 Einzelne Spalten mit pop() entfernen

2.5.12 Datensätze entfernen

Die Methode **.drop()** kann auch verwendet werden, oder besser gesagt ist das eher ihre Hauptaufgabe, um einzelne Datensätze zu entfernen (Abbildung 55). Es kann ein Datensatz, oder mehrere Datensätze gleichzeitig über eine Liste, aus dem DataFrame entfernt werden.

```
table = cpu.copy()
print(table.head(4))
table = table.drop({1,3})
print(table.head(4))
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0
	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
4	Intel	8080	NMOS	CDIP	40	NaN	6000.0	1974.0
5	Zilog	Z80	NMOS	CDIP	40	2.5	8500.0	1976.0

Abbildung 55 Datensätze mit **.drop()** entfernen

Wie sie in der Abbildung sehen können, wird bei einer Entfernung von Zeilen die (sichtbare) Indizierung der Datensätze nicht geändert.

2.5.13 Datensätze selektiv entfernen

Um Datensätze selektiv zu entfernen ist zuerst eine Liste mit den Datensätzen zu erstellen, die das Löschkriterium erfüllen. Diese Liste kann z.B. wie in der Abbildung 56 zu sehen ist, eine Liste aller Prozessoren mit maximal 18 Pins sein. Die Anweisung **.gt()** (Größer als) erstellt solch eine für alle

```
table = cpu.copy()
print(cpu.head(4))

pins=cpu["Pins"].gt(19)
table=cpu.drop(table.index[pins])

print("\n")
print(table)
```

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
0	MOS	6502	NMOS	PDIP	40	1.0	3510.0	NaN
1	MOS	6502A	NMOS	PDIP	40	2.0	3510.0	NaN
2	MOS	6502B	NMOS	PDIP	40	3.0	NaN	NaN
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0

	Manufacturer	CPU	Technology	Package	Pins	Clock	Transistors	Year
3	Intel	8008	PMOS	PDIP	18	NaN	3500.0	1972.0

Abbildung 56: Datensätze selektiv entfernen

Einträge der Spalte Pins (Abbildung 57). Aus dieser Liste wird dann über die Methode **.index()** eine Liste aller Zeilen, die den Kriterien entsprechen, generiert. Beachten Sie, dass in der erstellten Liste der Datensatz 3 fehlt, da das der einzige nicht wahre Datensatz mit weniger als 19 Pins ist. Jetzt können mit **.drop()** über genau diese Indexliste alle Datensätze mit mehr als 18 Pins gelöscht werden. Zum Schluss bleibt in dem DataFrame nur noch ein Datensatz übrig, da alle anderen CPUs mehr als 18 Pins Gehäuse haben.

```
print(pins)
print(table.index[pins])
```

```
0    True
1    True
2    True
3   False
4    True
5    True
6    True
7    True
8    True
Name: Pins, dtype: bool
Int64Index([0, 1, 2, 4, 5, 6, 7, 8], dtype='int64')
```

Abbildung 57 Indexliste generieren

2.6 Mathematische Operationen

2.6.1 Summe ermitteln

Summen aller Werte der Spalten in einem DataFrame können mittels `.sum()` ermittelt werden (Abbildung 58). Dies kann entweder für alle Spalten eines DataFrames oder nur für einzelne ausgewählte Spalten erfolgen. Wie Sie der Abbildung entnehmen können, werden nicht nur numerische Werte summiert, sondern auch Zeichenketten. Inwiefern dies Sinn ergibt, dürfen Sie selbst entscheiden.

```
print(cpu.sum())
print("\nSume aller Transistoren - ",cpu["Transistors"].sum())
```

Manufacturer	MOSMOSMOSIntelIntelZilogMotorolaMotorolaMotorola
CPU	65026502A6502B80088080Z806800680968000
Technology	NMOSNMOSNMOSPMOSNMOSNMOSNMOSNMOSH MOS
Package	PDIPPDIPPDIPDIPCDIPCDIPDIPDIPDIP
Pins	362
Clock	20.5
Transistors	106120
Year	9879
dtype:	object

```
Sume aller Transistoren - 106120.0
```

Abbildung 58 Summen bilden mit `.sum()`

2.6.2 Maximum und Minimum bilden

Genauso wie die Summen mit `.sum()` lassen sich die Maxima mit der Methode `.max()` (Abbildung 59) sowie Minima mit der Methode `.min()` (Abbildung 60) der einzelnen Spalten bilden. Die Auswertung von Zeichenketten erfolgt dabei lexikografisch, sodass hier nicht bedeutet, dass Zilog die besten und Intel die schlechtesten Prozessoren herstellt. Hier liegt der Buchstabe „I“ vor dem „Z“ im ASCII Zeichensatz.

```
print(cpu.max())
print("\nMaximum aller Transistoren - ",cpu["Transistors"].max())
```

Manufacturer	Zilog
CPU	Z80
Technology	PMOS
Package	PDIP
Pins	64
Clock	10
Transistors	68000
Year	1979

dtype: object

Maximum aller Transistoren - 68000.0

Abbildung 59 Maximas ermitteln mit `max()`

```
print(cpu.min())
print("\nMinimum aller Transistoren - ",cpu["Transistors"].min())
```

Manufacturer	Intel
CPU	6502
Technology	HMOS
Package	CDIP
Pins	18
Clock	1
Transistors	3500
Year	1972

dtype: object

Minimum aller Transistoren - 3500.0

Abbildung 60 Minimas ermitteln mit `min()`

2.6.3 Differenz bilden

Eine Differenz kann nur zwischen zwei numerischen Werten errechnet werden.

```
df = pd.DataFrame({})
df["CPU"] = cpu["CPU"]
df["Transistors"] = cpu["Transistors"]
df["Difference"] = cpu["Transistors"].diff()
print(df)
```

	CPU	Transistors	Difference
0	6502	3510.0	NaN
1	6502A	3510.0	0.0
2	6502B	NaN	NaN
3	8008	3500.0	NaN
4	8080	6000.0	2500.0
5	Z80	8500.0	2500.0
6	6800	4100.0	-4400.0
7	6809	9000.0	4900.0
8	68000	68000.0	59000.0

Abbildung 61: Differenz bilden

2.6.4 Addition und Subtraktion

Auch eine Addition, Subtraktion bzw. weitere mathematische Operationen lassen sich nur mit numerischen Werten durchführen.

```
df = pd.DataFrame({})
df["CPU"] = cpu["CPU"]
df["Transistors"] = cpu["Transistors"]
df["+ 1"] = cpu["Transistors"].add(1)
df["- 1"] = cpu["Transistors"].sub(1)
print(df)
```

	CPU	Transistors	+ 1	- 1
0	6502	3510.0	3511.0	3509.0
1	6502A	3510.0	3511.0	3509.0
2	6502B	NaN	NaN	NaN
3	8008	3500.0	3501.0	3499.0
4	8080	6000.0	6001.0	5999.0
5	Z80	8500.0	8501.0	8499.0
6	6800	4100.0	4101.0	4099.0
7	6809	9000.0	9001.0	8999.0
8	68000	68000.0	68001.0	67999.0

Abbildung 62 Addition und Subtraktion

2.6.5 Spalten indexieren

Für viele Operationen ist es wichtig, dass die Datensätze indexiert vorliegen. In der Abbildung 63 sehen Sie zuerst den DataFrame nicht indexiert, bzw. „automatisch“ numerisch indexiert (0..3), so wie er entstanden ist. Mit der Methode `.set_index()` unter der Angabe der Spalte, die als Index dienen sollte, wird die Spalte ausgewählt die als der neue Index dienen Sollte.

```
df = pd.DataFrame({"Name": ["Apfel", "Banane", "Orange", "Zitrone"],
                  "Farbe": ["rot", "gelb", "orange", "gelb"],
                  "Geschmack": ["Sauer", "Kein", "Perfekt", "Sauer"],
                  "Gewicht": (185, 190, 160, 80)})

print(df, "\n")
print(df.set_index("Farbe"))
```

	Name	Farbe	Geschmack	Gewicht
0	Apfel	rot	Sauer	185
1	Banane	gelb	Kein	190
2	Orange	orange	Perfekt	160
3	Zitrone	gelb	Sauer	80

	Name	Geschmack	Gewicht
Farbe			
rot	Apfel	Sauer	185
gelb	Banane	Kein	190
orange	Orange	Perfekt	160
gelb	Zitrone	Sauer	80

Abbildung 63 DataFrame indexieren.

2.6.6 DataFrames zusammenfügen

Will man zwei DataFrame Objekte zusammenfügen, so kann dies mit der Methode `.join()` erfolgen (Abbildung 64). Falls es in den beiden DataFrames Spalten mit gleichem Namen gibt, so sind die beiden Suffixe, um den die Namen der gleichlautenden Spalten der alten DataFrames in dem neuen DataFrame erweitert werden, anzugeben. In der Abbildung ist das in den beiden Spalten „Name“ der Fall. So wurde aus den ersten Spalten der beiden DataFrames in der Abbildung 64 die „neuen“ Spalten „Name_1“ sowie „Name_2“ gebildet. Zu beachten ist, dass die Zeilen einerseits „blind“ (in Reihenfolge der Datensätze in den beiden DataFrames) zusammengesetzt werden, was besonders gut im zweiten und dritten Datensatz zu sehen ist (Banane & Birne, Orange & Banane). Andererseits diktiert das erste DataFrame die Größe des neuen DataFrames. Ist das erste DataFrame größer als das zweite, so wird ein Fehler gemeldet. Umgekehrt werden die „überschüssigen“ Datensätze ignoriert.

2.6.7 Dataframes indiziert zusammensetzen

Eine weitaus klügere Art DataFrames zusammenzusetzen kann über Indexierung der beiden ursprünglichen DataFrames erfolgen. Dazu muss zuerst der DataFrame mit der Methode `.setIndex()` indiziert werden (Abbildung 65). Als Parameter wird der Name der Spalte angegeben, die als Index dienen sollte. Wie Sie in der Abbildung 65 sehen können, wird die entsprechende Spalte jetzt von den

```
print(frucht1.set_index('Name'))
```

	Farbe	Gewicht
Name		
Apfel	rot	185
Banane	gelb	190
Orange	orange	100

Abbildung 65 DataFrames Indexieren

anderen abgesetzt betrachtet. Sind beide DataFrames indiziert, so können diese jetzt nach den Indizes sortiert zusammengesetzt werden (Abbildung 66). Besteht eine Überlappung in den Spaltennamen (Indizes ausgeschlossen), ist wieder eine Angabe von Suffixen notwendig. Wie in der Abbildung ersichtlich, wurden diesmal die Indizes des ersten DataFrames mit den Indizes des zweiten DataFrames verknüpft und die entsprechenden Datensätze mit den Daten des zweiten DataFrame ergänzt. Da es für den Datensatz „Orange“ keinem Datensatz in dem zweiten DataFrame gegeben hat, wurde der Wert für die Spalte „Gewicht“ bei Orange als NaN definiert.

```
frucht = frucht1.set_index('Name').join(frucht2.set_index('Name'))
print(frucht)
```

	Farbe	Gewicht
Name		
Apfel	rot	185.0
Banane	gelb	100.0
Orange	rot	NaN

Abbildung 66 Verknüpfung von DataFrames über Indexe

Über den Parameter „how“ lässt sich steuern ob linkes (Default) oder rechtes (Abbildung 67) DataFrame dem Index bestimmt.

```
frucht = frucht1.set_index('Name').join(frucht2.set_index('Name'), how = 'right')
print(frucht)
```

	Farbe	Gewicht
Name		
Apfel	rot	185
Birne	NaN	190
Banane	rot	100

Abbildung 67 Verknüpfung von DataFrames mit rechten Masterindex

Des Weiteren erlaubt dieser Parameter noch die Möglichkeit einer kompletten Vereinigung der beiden DataFrames (Abbildung 68) oder Bildung einer Schnittmenge (Abbildung 69). Bei der

Bildung der Vereinigung können unter Umständen unvollständige Datensätze entstehen, da die nur in einem Datensatz vorhandenen Datensätze an den „neuen“ Stellen mit NaNs gefüllt werden.

```
frucht = frucht1.set_index('Name').join(frucht2.set_index('Name'), how = 'outer')
print(frucht)
```

	Farbe	Gewicht
Name		
Apfel	rot	185.0
Banane	rot	100.0
Birne	NaN	190.0
Orange	gelb	NaN

Abbildung 68 Verknüpfung von DataFrames als Vereinigungsmenge

```
frucht = frucht1.set_index('Name').join(frucht2.set_index('Name'), how = 'inner')
print(frucht)
```

	Farbe	Gewicht
Name		
Apfel	rot	185
Banane	rot	100

Abbildung 69 Verknüpfung von DataFrames als Schnittmenge

2.6.8 DataFrames expandieren

Sollten einzelne Datenfelder in einem DataFrame Listen, Tuples etc. enthalten, so müssen diese gegebenenfalls vor der Verarbeitung expandiert werden. In der Abbildung 70 ist so ein Fall enthalten da dort einerseits Äpfel und Kiwi und andererseits Banane & Zitrone zusammengefasst sind. In so einem Fall kann die Methode **.explode()** angewandt werden, sodass alle Listen in der Spalte, auf die die **.explode()**-Methode angewandt wurden, zu Einzelwerten aufgelöst wurden.

```
df = pd.DataFrame({"Name": [ ["Apfel", "Kiwi"], ["Banane", "Zitrone"], "Orange"],
                          "Farbe": ("grün", "gelb", "orange")})
print(df)
df.explode("Name")
```

	Name	Farbe
0	[Apfel, Kiwi]	grün
1	[Banane, Zitrone]	gelb
2	Orange	orange

	Name	Farbe
0	Apfel	grün
0	Kiwi	grün
1	Banane	gelb
1	Zitrone	gelb
2	Orange	orange

Abbildung 70: Dataframes expandieren

Dabei werden die entsprechenden Datensätze der Menge der Listenmitglieder entsprechend vervielfacht, sodass die einzelnen Listenmitglieder in diese Kopien eingesetzt werden können. Auf diese Weise sind in dem Beispiel aus drei Datensätzen letztendlich fünf Datensätze entstanden. Sollten mehrere Spalten Listen enthalten, so ist auf jede dieser Spalten die Methode `.explode()` anzuwenden.

2.6.9 Datensätze nach Bezeichnern Selektieren

Eine Möglichkeit Datensätze nach Bezeichnern zu selektieren stellt die Methode `.loc()` dar (Abbildung 71). Damit die Suche funktioniert, muss die entsprechende Spalte zuerst indiziert werden (falls sie es noch nicht bereits ist). Danach kann nach dem gewünschten Bezeichner in der Spalte gesucht werden.

```
df = pd.DataFrame({"Name": ["Apfel", "Banane", "Orange", "Zitrone"],
                  "Farbe": ("rot", "gelb", "orange", "gelb"),
                  "Geschmack": ["Sauer", "Kein", "Perfekt", "Sauer"],
                  "Gewicht": (185, 190, 160, 80)})

df = df.set_index("Farbe")
print(df.loc["gelb"])
```

	Name	Geschmack	Gewicht
Farbe			
gelb	Banane	Kein	190
gelb	Zitrone	Sauer	80

Abbildung 71: Datensätze suchen mit loc

Es können nicht nur nach einem, sondern nach mehreren Ausdrücken gleichzeitig gesucht werden. Die Angabe der gesuchten Begriffe kann dabei entweder über Listen oder Tupel stattfinden (Abbildung 72).

```
print(df.loc[{"rot", "orange"}])
```

	Name	Geschmack	Gewicht
Farbe			
orange	Orange	Perfekt	160
rot	Apfel	Sauer	185

Abbildung 72 Suchen mit Liste

Neben einzelnen Bezeichnern können auch Bereiche von einem bis zu einem anderen Bezeichner ausgewählt werden. (Abbildung 73). Allerdings müssen die beiden Bezeichner einmalig sein, also nur einmal in der gesamten Spalte vorkommen.

```
print(df.loc["rot":"orange"])
```

	Name	Geschmack	Gewicht
Farbe			
rot	Apfel	Sauer	185
gelb	Banane	Kein	190
orange	Orange	Perfekt	160

Abbildung 73 Bereiche auswählen

Die Ausgabe lässt sich weiter einschränken, wenn die Spalte, aus der die Daten angezeigt werden sollen, mit angegeben wird. Wie in der Abbildung 74 zu sehen ist, sind alle rote Früchte (in der Dataframe) sauer.

```
print(df.loc["rot","Geschmack"])
```

Sauer

Abbildung 74 Ausgabe einschränken

Ein weitere Möglichkeit besteht darin, Daten nach vorgegebenen Kriterien vorzuselektieren. Wie Sie in der Abbildung 75 sehen können, wurden hier alle Früchte vorselektiert, die ein Mindestgewicht von über 160 Gramm aufweisen.

```
print(df.loc[df["Gewicht"] > 160])
```

	Name	Geschmack	Gewicht
Farbe			
rot	Apfel	Sauer	185
gelb	Banane	Kein	190

Abbildung 75 Vorselektion

2.6.10 Selektion mit Wahrheitsliste

Mit der Funktion `.loc()` lassen sich auch Elemente mittels einer Liste selektieren, in der die zu selektierenden Datensätze (Zeilen) als *Wahr (True)* und die Datensätze, die ausgelassen werden sollten, als *Unwahr (False)* markiert werden (Abbildung 76). Wie Sie in der Abbildung sehen können, wurden in der unteren Ausgabe der Daten zwei Datensätze ausgelassen (1 und 3). Das sind auch exakt die Stellen, bei denen in der übergebenen Liste der Wert *unwahr* steht. Zu beachten ist, dass die Wahrheitsliste genau die Länge haben muss wie viele Datensätze (Zeilen) sich in dem DataFrame

```
print(df.loc[[True,True,True,True,True]])
```

	Name	Geschmack	Gewicht
Farbe			
rot	Apfel	Sauer	185
gelb	Banane	Kein	190
orange	Orange	Perfekt	160
gelb	Zitrone	Sauer	80
grün	Kiwi	Sauer	90

```
print(df.loc[[False,True,False,True,True]])
```

	Name	Geschmack	Gewicht
Farbe			
gelb	Banane	Kein	190
gelb	Zitrone	Sauer	80
grün	Kiwi	Sauer	90

Abbildung 76 Selektieren mit Wahrheitsliste

befinden. Ist die Liste kürzer oder länger als die Anzahl der Datensätze, wird eine Fehlermeldung ausgegeben.

2.6.11 Datensätze nach Position selektieren

Neben der Möglichkeit Datensätze nach dem Inhalt zu selektieren, ist es auch möglich die Datensätze mit **.iloc[]** nach ihrer vorkommen Reihenfolge in dem Dataframe zu selektieren.

```
df = pd.DataFrame({"Name": ["Apfel", "Banane", "Orange", "Zitrone", "Kiwi"],
                  "Farbe": ("rot", "gelb", "orange", "gelb", "grün"),
                  "Geschmack": ["Sauer", "Kein", "Perfekt", "Sauer", "Sauer"],
                  "Gewicht": (185, 190, 160, 80, 90)})

df.iloc[0]
```

Name	Farbe	Geschmack	Gewicht
Apfel	rot	Sauer	185

Name: 0, dtype: object

Abbildung 77 einem Datensatz mit .iloc[] selektieren

Die Auswahl der Datensätze kann auch über eine Liste erfolgen (Abbildung 78). Zu beachten ist, dass bei einer direkten Listenangabe eine zusätzliche Klammerung für die Liste selbst notwendig ist.

```
df.iloc[[0,3]]
```

	Name	Farbe	Geschmack	Gewicht
0	Apfel	rot	Sauer	185
3	Zitrone	gelb	Sauer	80

Abbildung 78 Mehrere Datensätze mit iloc[] selektieren

Auch eine Auswahl über Bereichsangaben ist möglich (Abbildung 79). In diesem Fall wird der gewünschte Bereich wie üblich innerhalb der eckigen Klammern angegeben.

```
df.iloc[0:2]
```

	Name	Farbe	Geschmack	Gewicht
0	Apfel	rot	Sauer	185
1	Banane	gelb	Kein	190

Abbildung 79 Bereiche mit iloc[] selektieren

Wie bei der Methode `.loc[]` kann auch bei `.iloc[]` eine Wahrheitsliste zur Auswahl der Datensätze verwendet werden (Abbildung 80). Die Liste muss exakt genauso viele Mitglieder aufweisen wie die Anzahl der Zeilen in dem DataFrame.

```
df.iloc[[True,False, False, True, False]]
```

	Name	Farbe	Geschmack	Gewicht
0	Apfel	rot	Sauer	185
3	Zitrone	gelb	Sauer	80

Abbildung 80 `.iloc[]` Auswahl über Wahrheitsliste

```
df.iloc[lambda x: x.index % 2 > 0]
```

	Name	Farbe	Geschmack	Gewicht
1	Banane	gelb	Kein	190
3	Zitrone	gelb	Sauer	80

Abbildung 81 `.iloc[]` Selektion mit Funktionen

2.6.12 Selektion von Zeilen und Spalten

Mit der Methode `.iloc[]` lassen sich nicht nur vollständige Datensätze bzw. Zeilen sondern auch einzelne Datenfelder auswählen (Abbildung 82). Die Auswahl erfolgt über die Angabe der gewünschten Zeile gefolgt von der Angabe der Spalte.

```
print ("die",df.iloc[3,0],"ist",df.iloc[3,2])
```

die Zitrone ist Sauer

Abbildung 82 Auswahl eines Datenfeldes mit `.iloc[]`

Neben einzelnen Datenfeldern lassen sich auch Kombinationen aus mehreren Spalten sowie Zeilen gleichzeitig auswählen (Abbildung 83). Die Angabe der gewünschten Datenfelder erfolgt über zwei Listen, zuerst die Liste der Zeilen gefolgt von der Liste der Spalten.

```
columns = list(df.columns.values.tolist())
print(columns)
```

```
geschmack = columns.index("Geschmack")
df.iloc[[0,2],[1,geschmack]]
```

```
['Name', 'Farbe', 'Geschmack', 'Gewicht']
```

	Farbe	Geschmack
0	rot	Sauer
2	orange	Perfekt

Abbildung 83 Mehrere Datenfelder auswählen

2.6.13 Funktionen auf Datenfeldern ausführen

Bei einem DataFrame besteht die Möglichkeit über die Funktion `.apply()` eine Funktion auf die Werte entweder der gesamten DataFrame oder nur eines Teils davon (Abbildung 84). Sollte das gesamte DataFrame bearbeitet werden, so ist die Achse anhand welcher die Daten an die Funktion übergeben werden (zeilen- oder spaltenweise) anzugeben. Zu beachten ist, dass die Funktion unterschiedliche Datentypen hat, je nachdem, ob sie auf das gesamte DataFrame oder nur auf eine Spalte bzw. Zeile angewandt wird. Bei der Bearbeitung eines DataFrames werden die einzelnen Spalten oder Zeilen als Objekte der Klasse **Series** an die Funktion weitergeleitet, bei der Verarbeitung der Spalte oder Zeile als die tatsächlichen Objekte selbst, die in den Datenfeldern gespeichert sind. So wird „rot“ von der Funktion in drei Datenfeldern aufgespalten, da hier in einem String iteriert wird. Weiterhin kann die aufgerufene Funktion auch die Geometrie des DataFrames ändern, indem Sie zusätzliche Datenfelder initialisiert und zurückgibt.

```
def funktionchen(data):
    value = []
    print(type(data))
    for iteration in data:
        value.append(iteration + "chen")
    return(pd.Series(value))

df = pd.DataFrame({"Name": ["Apfel", "Banane", "Orange"],
                  "Farbe": ("rot", "gelb", "orange")})

print(df.apply(funktionchen, axis = 1), "\n")
print(df["Farbe"].apply(funktionchen))
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
      0      1
0  Apfelchen  rotchen
1  Bananechen  gelbchen
2  Orangechen  orangechen

<class 'str'>
<class 'str'>
<class 'str'>
      0      1      2      3      4      5
0  rchen  ochen  tchen  NaN  NaN  NaN
1  gchen  echen  lchen  bchen  NaN  NaN
2  ochen  rchen  achen  nchen  gchen  echen
```

Abbildung 84 Funktionen auf Datenfeldern ausführen

2.7 Datensätze Plotten

Die Datensätze einer DataFrame lassen sich direkt ohne Umwege auf dem Bildschirm plotten. Dabei ist für jedem Plot Typ eine eigene Funktion zuständig, wobei folgende Plot Typen verfügbar sind:

- `bar()` - Balkengrafik
- `barh` – Horizontale Balkengrafik
- `hist()` - Plottet ein Histogramm (Verteilung der Daten)
- `boxplot()` - Kastengrafik
- `linie()` - Liniendiagramm
- `kde()` oder `density()` - Dichtediagramm
- `area()` - Flächendiagramm
- `pie()` – Kuchen bzw. Torten bzw. Kreisdiagramm.
- `scatter()` - Streudiagramm
- `hexbin()` - Streudiagramm mit hexagonalen Elementen

```
table = pd.DataFrame([[2,1],[0,2],[2,3],[2,3]])  
table.rename(columns={0:"x", 1:"y"},inplace = True)  
table.plot.bar()
```

<AxesSubplot:>

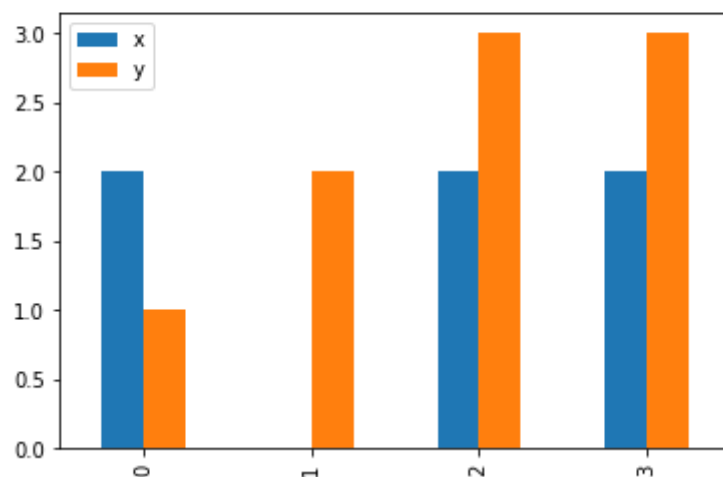


Abbildung 85 Ein Plot (Balkenplot)

2.7.1 Balkengrafik

Der Balkengrafik Plot, kann entweder als ein vertikaler (Abbildung 86 links) oder ein horizontaler (Abbildung 86 rechts) dargestellt werden. Dabei entspricht ein Balken exakt einem Wert in dem Dataframe. Werte die in den Unterfeldern an der gleichen Stellen liegen werden mit der gleichen Farbe gezeichnet als in dem Beispiel ist - [2 = blau , 1 = Orange, 4 = Grün], [0 = blau , 2 = Orange, 4 = Grün], usw.

```
table = pd.DataFrame([[2,1,4],[0,2,4],[2,3,5],[2,3,3],
                     [2,2,2],[0,4,2],[2,1,1],[1,3,1]])
table.plot.bar()
table.plot.barh()
```

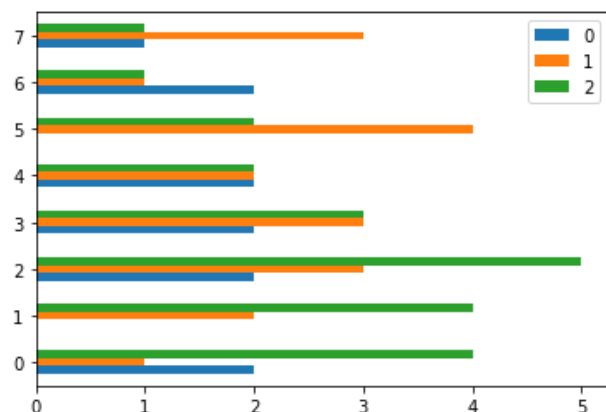
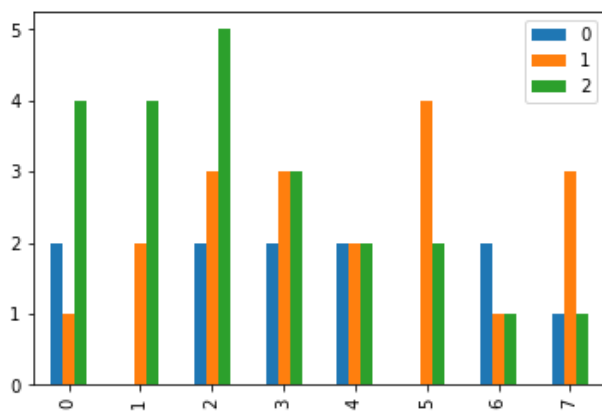


Abbildung 86 vertikale und horizontale Balkengrafik

2.7.2 Histogramm

```
table = pd.DataFrame([[2,1,4],[0,2,4],[2,3,5],[2,3,3],
                     [2,2,2],[0,4,2],[2,1,1],[1,3,1]])
table.plot.hist()
```

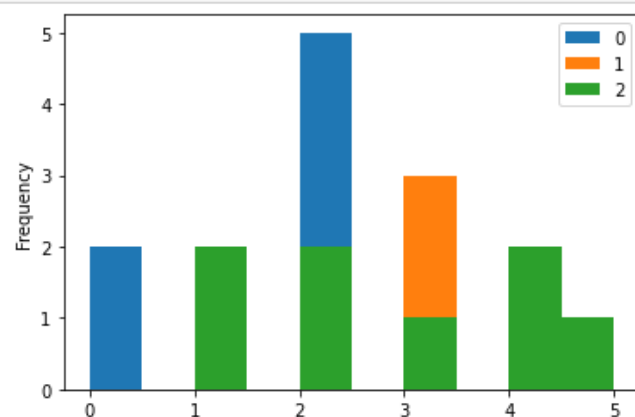


Abbildung 87: ein Histogramm

Im Unterschied zu Balkengrafik, werden bei einem Histogramm zwar die Daten auch als Balken dargestellt, aber diesmal zeigen diese nicht die einzelnen Daten, sondern die Verteilung dieser in dem Datensatz. So kommt auf der 0ten Stelle z.b. 2 mal die 0 (erster blauer Balken) und 5 mal die 2 (dritter Balken). Je nachdem wie viele der Balken dargestellt werden sollten können hier auch

Bereiche statt einzelnen Werten abgebildet sein. Ein Nachteil dieser Darstellung ist, dass sich hier die Balken überschreiben können. So fehlt z.b. der blaue Balken für die 1 des 0ten Datensatz da dieser durch den grünen überschrieben wird.

2.7.3 Dichte-Diagramm

Ein Dichte-Diagramm ist im Grunde genommen ein Histogramm mit geglätteten Kanten. Was bei der Betrachtung beider Diagramme die den gleichen Sachverhalt darstellen nicht schwer zu erkennen ist (Abbildung 88).

```
table.plot.kde()
table.plot.hist(alpha=0.5)
```

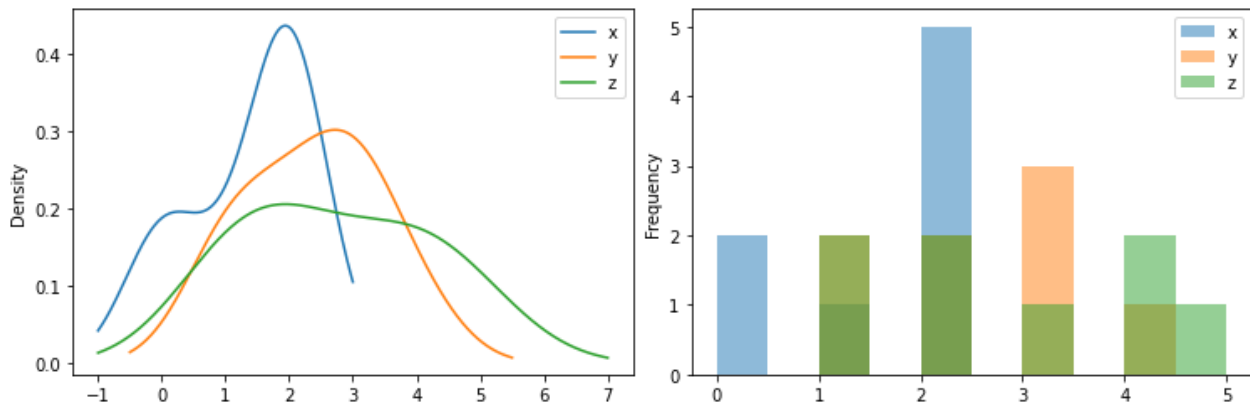


Abbildung 88 Dichte-Diagramm und Histogramm mit gleichen Daten

2.7.4 Liniendiagramm

Bei einer Liniendiagramm werden die einzelnen Spalten nebeneinander als Linien dargestellt. Die Darstellung auf der y Achse erfolgt dabei in absoluten Werten bezüglich des Ursprungs (Wert 0).

```
table = pd.DataFrame([[2,1,4],[0,2,4],[2,3,5],[2,3,3],
                      [2,2,2],[0,4,2],[2,1,1],[1,3,1]])
table.plot.line()
```

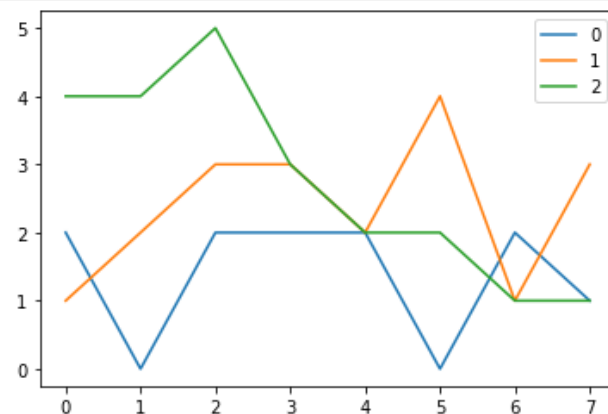


Abbildung 89 Liniengrafik

2.7.5 Area Plot

Eine ähnliche Darstellung wie ein Liniendiagramm zeigt das Flächendiagramm. Der Unterschied ist, das die Werte (Vertikal) jetzt in Bezug auf dem Wert der vorausgehenden Spalte dargestellt werden. So ist der Wert 7 in dem Ersten Zeile der zweiten Spalte als Absolut 4 zu interpretieren da zuerst die Werte für die nullte Spalte (2) sowie die erste Spalte (1) gezeichnet wurden. Somit ist der Ursprungspunkt für diesem Wert der Wert 3 auf der Y Achse.

```
table = pd.DataFrame([[2,1,4],[0,2,4],[2,3,5],[2,3,3],
                     [2,2,2],[0,4,2],[2,1,1],[1,3,1]])
table.plot.area()
```

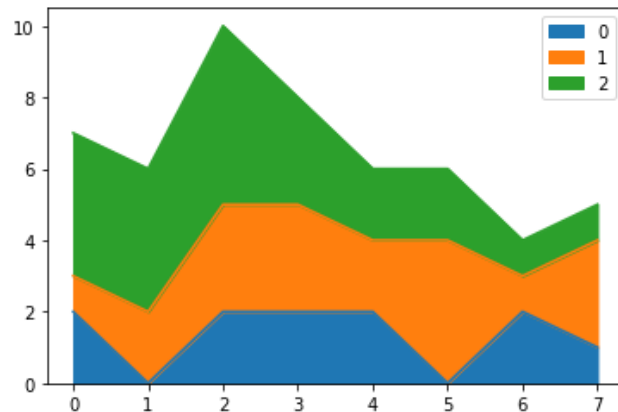


Abbildung 90 Ein Flächendiagramm

2.7.6 Boxplot

Ein Boxplot oder Kastengrafik zeigt bereits eine statistische Auswertung der Daten. Die Box selbst beschreibt den Bereich zwischen den ersten und dritten Viertel der Daten, also den Bereich in dem mindestens 50% der Daten einer Spalte liegen. Die grüne Linie zeigt den Median der Werte einer Spalte. Die beiden Begrenzungslinien sind als $1.5 \cdot \text{IQR}$ ($\text{IQR} = Q3 - Q1$) definiert. Ausreißer, die außerhalb dieser Begrenzungslinien liegen, werden schließlich als Kreise in das Diagramm eingezeichnet.

```
table = pd.DataFrame([[2,1,4],[2,2,4],[3,3,5],[3,3,3],
                     [3,2,2],[3,4,2],[4,1,1],[0,3,1]])
table.plot.box()
```

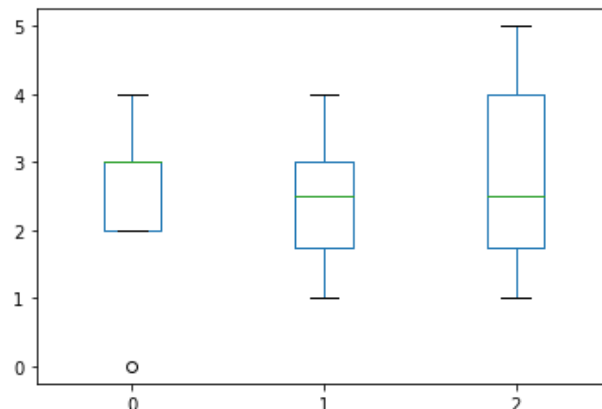


Abbildung 91 Eine Kastengrafik

2.7.7 Kuchendiagramm

Im Unterschied zu den anderen Plotarten kann bei dem Kuchendiagramm nur eine einzelne Spalte dargestellt werden. Deswegen muss die darzustellende Spalte als Parameter (y Achse) angegeben werden (Abbildung 92). Die Ausgaberrichtung erfolgt gegen den Uhrzeigersinn.

```
table = pd.DataFrame([[2,1,4],[0,2,4],[2,3,5],[2,3,3],
                     [2,2,2],[0,4,2],[2,1,1],[1,3,1]])
table.plot.pie(y=0)
```

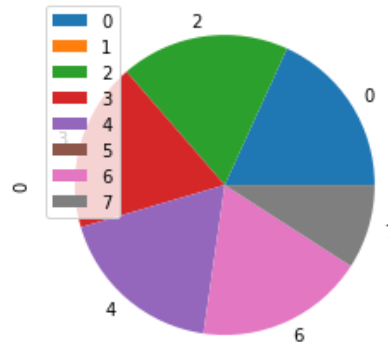


Abbildung 92 Kuchendiagramm.

2.7.8 Scatter

Ein Streudiagramm, auch Punktwolke genannt, zeigt die Verteilung von Wertepaaren in einem 2D Raum. Als Parameter werden die Spalten für die x sowie die y Achse benötigt (Abbildung 93 Oben). Über einem dritten Parameter (c) kann die farbliche Darstellung der Punkte eingeschaltet werden. Standardmäßig werden die Punkte in Graustufen angezeigt (Abbildung 93 links). Über den Parameter **colormap** kann aber auch eine andere vordefinierte Farbgebung gewählt werden (Abbildung 93 rechts).

```
table.plot.scatter(y=0, x=1, c = 2)
table.plot.scatter(y=0, x=1, c = 2, colormap='RdBu')
```

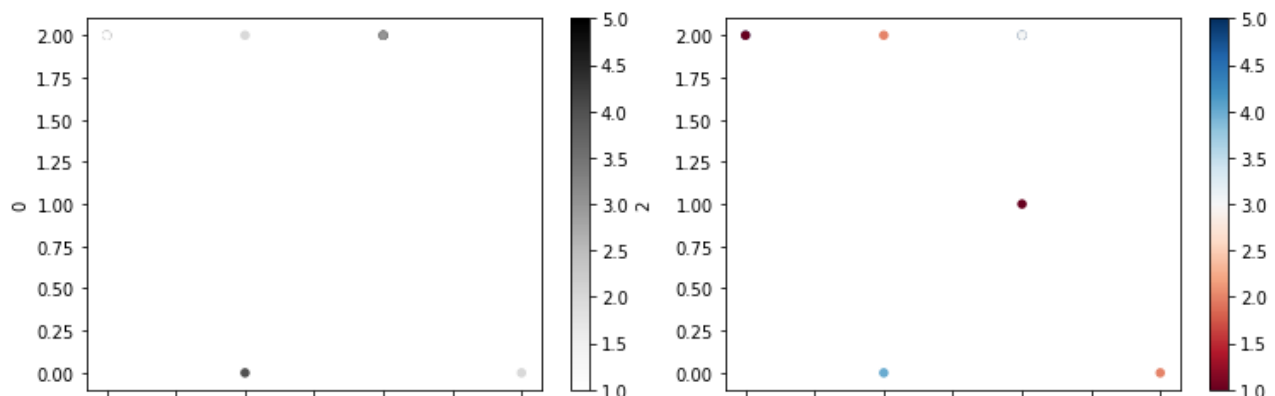


Abbildung 93 Streudiagramm

2.7.9 HexBin

Ähnlich wie ein Streudiagramm, sieht der HexBin Plot aus. Der eigentliche Unterschied ist, dass statt Punkten hier hexagonale Flächen gezeichnet werden. Durch den Parameter **gridsize** lässt sich die Menge der darstellbaren Hexagone steuern. Je größer die **gridsize** desto mehr Hexagone werden in dem Plot dargestellt. In der Abbildung 94 links ist ein hexbin Plot mit **gridsize 20**, rechts mit **gridsize 2** zu sehen. Außerdem müssen noch die Spalten angegeben werden die als x sowie y Werte interpretiert werden sollten.

```
table.plot.hexbin(y=0, x=1, gridsize=20, cmap="viridis")
table.plot.hexbin(y=0, x=1, gridsize=2, cmap="viridis")
```

```
<AxesSubplot:xlabel='y', ylabel='x'>
```

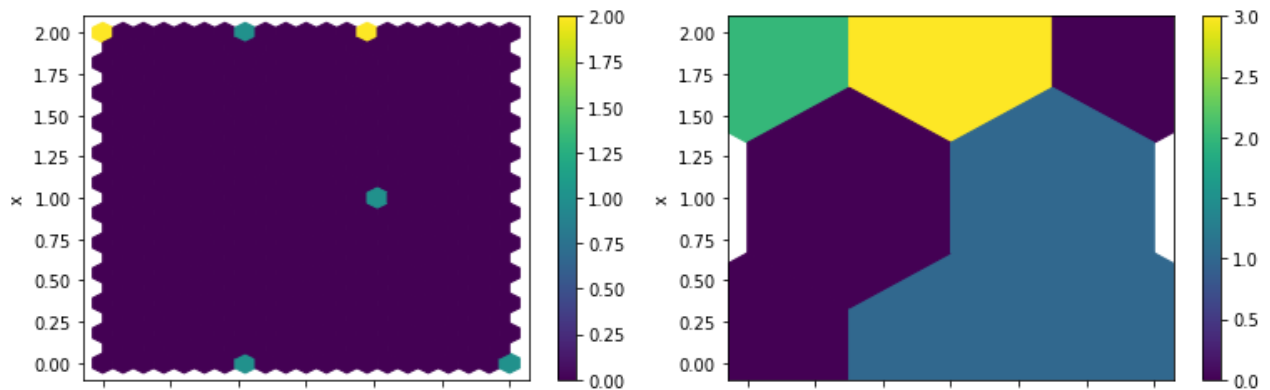


Abbildung 94 Ein Hexbin Plot

2.7.10 Plots logarithmisch Skalieren

Plots können automatisch entweder über die x-, die y- oder über beide Achsen bei der Darstellung skaliert werden (Abbildung 95). Die entsprechenden Parameter sind **logx** für die x- sowie **logy** für die y-Achse. Sollten beide Achsen gleichzeitig logarithmisch skaliert werden, dann kann auch einfachheitshalber der **loglog** Parameter angewandt werden. Zu beachten ist, dass nicht jeder Plot auch logarithmisch skaliert werden kann – bei einem Torten-/Kuchen-/Kreisdiagramm macht dies nicht sonderlich viel Sinn, so dass diese Parameter dort auch keine Wirkung entfalten.

```
table.plot.kde(logx=True)
table.plot.kde(logy=True)
```

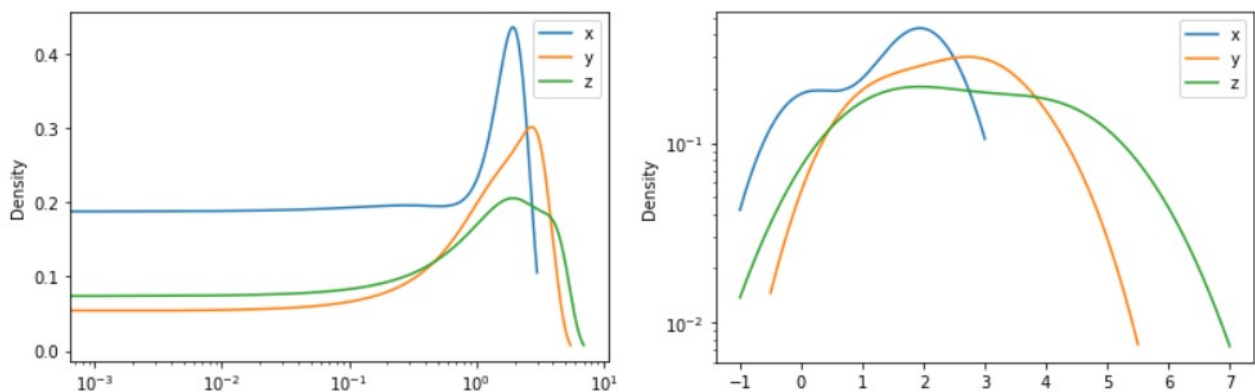


Abbildung 95: Plots logarithmisch skalieren

2.7.11 Linienformat festlegen

Der Parameter **style** legt den Format, mit dem die Linien gezeichnet werden sollen, fest (Abbildung 96). Neben dem Linientyp (Gepunktet, Strichpunkt usw.) lässt sich auch die Farbe festlegen mit der die Linie gezeichnet werden soll. Der Parameterformat entspricht dem der Mathplotlib. Der Parameter hat nur bei den Plots eine Wirkung, die auch tatsächlich Linien Zeichnen (**line()**, **kde()**, **area()**)

```
table.plot.line(style = ["--", "-.", ":k"])
```

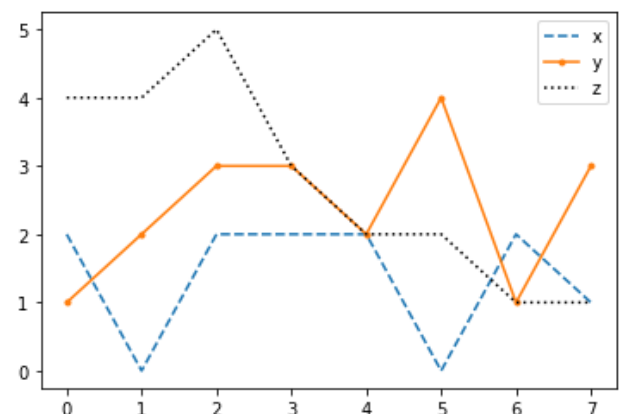


Abbildung 96: Linien Formatieren

2.7.12 Legende

Über den Parameter **legend** lässt sich die legende entweder ganz ausschalten (`legend = False`), oder die Reihenfolge der Elemente in der Legende umdrehen (`legend = reverse`) (Abbildung 97). Bei einer umgedrehten Reihenfolge werden die Spaltennamen von der Letzten bis zu ersten von Oben nach Unten angezeigt.

```
table.plot.bar(legend = "reverse")  
table.plot.bar(legend = False)
```

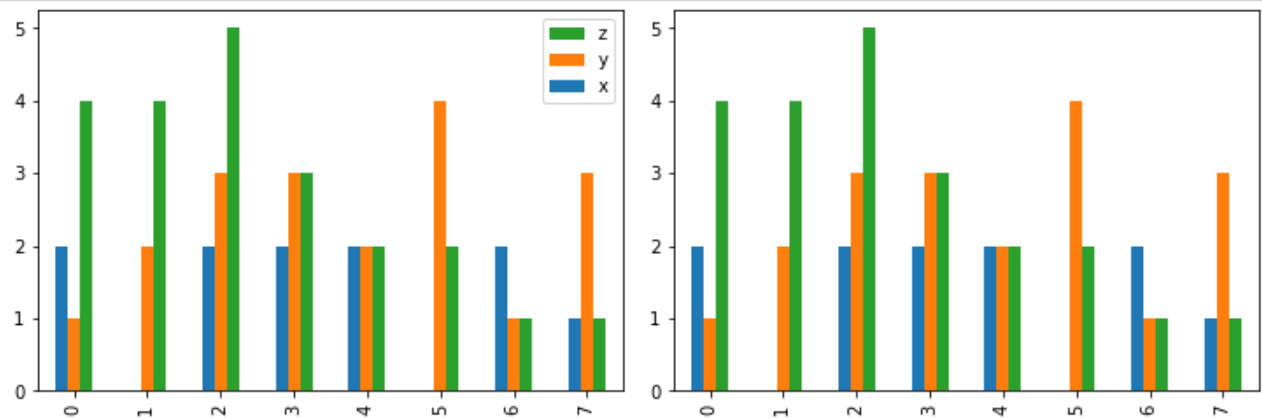


Abbildung 97 Legende eines Plots parametrisieren

2.8 Histogramme

Mittels eines Histogramm lässt sich grafisch die Verteilung der Daten innerhalb eines Datensatzes darstellen (Abbildung 98). Dabei wird die Häufigkeit mit deren ein Wert, oder ein Wertebereich in dem Datensatz vorkommt auf der y Achse plottet.

Gibt es in dem Dataframe mehrere Daten die numerische Inhalte beinhalten und es wird keine Spalte angegeben, so wird für jede dieser Spalten ein eigenes Histogramm gezeichnet (Abbildung 99). Die Spalten, die keine numerischen Werte beinhalten, werden dabei ignoriert. Sollte nur ein Subset der numerischen Features als Histogramme gezeichnet werden,

so können die entsprechenden Spalten als ein Set oder als eine Liste z.b. als :

`cpu[{"Pins","Year"}].hist()`

notiert werden. Bitte beachten

Sie die dazu notwendigen geschweiften oder eckigen Klammern, die hier den

notwendigen Set oder Liste

aus denangaben über die zu verwendete Spalten erzeugen.

Wird das Histogramm nicht direkt, sondern auf dem

Umweg über die plot Methode

ausgegeben, so werden

einzelnen Spalten nicht als

separate Histogramme,

sondern als ein gemeinsamer

Histogramm ausgegeben

(Abbildung 100).

```
cpu["Pins"].hist()
```

<AxesSubplot:>

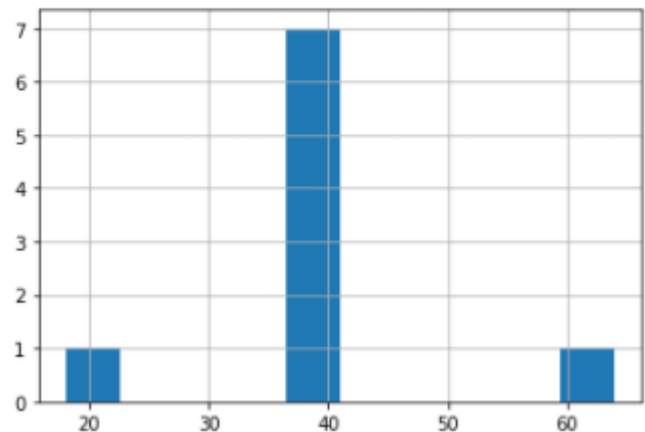


Abbildung 98 Ein Histogramm

```
cpu.hist()
```

```
array([[<AxesSubplot:title={ 'center': 'Pins' }>,  
       <AxesSubplot:title={ 'center': 'Clock' }>],  
       [<AxesSubplot:title={ 'center': 'Transistors' }>,  
       <AxesSubplot:title={ 'center': 'Year' }>]], dtype=object)
```

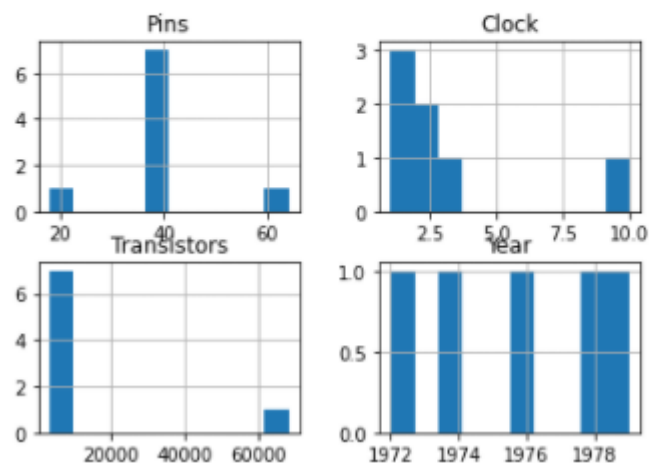


Abbildung 99 Mehrere Histogramme Automatisch erzeugen

```
cpu.plot.hist()
```

```
<AxesSubplot:ylabel='Frequency'>
```

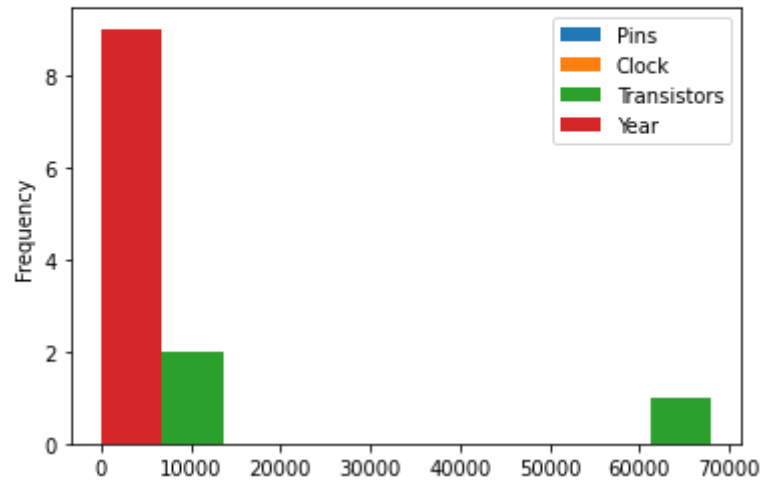


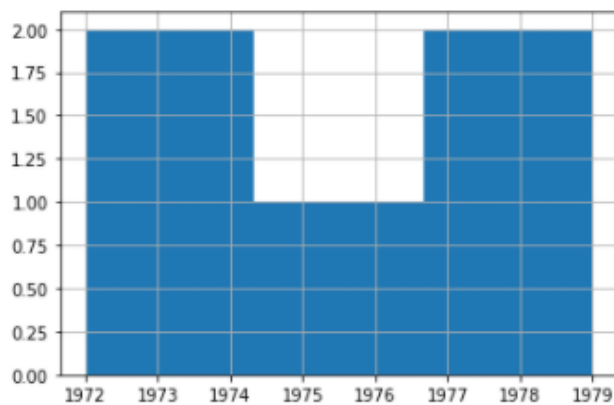
Abbildung 100

2.8.1 Die „Behälter“ eines Struktogramms

Es ist möglich die Anzahl der Spalten eines Histogramms durch den Parameter „bins“ (Behälter) zu bestimmen. Dabei werden die einzelnen Daten des Datenframes in diesen Behältern zusammengefasst. Durch den Parameter **bins** lässt sich die Menge der zu verwendeten Behälter für die Daten bestimmen. Dabei bestimmt die Spannweite der Daten deren Verteilung auf die einzelnen Bins. So können Histogramme mit gleichen Daten aber anderen Menge von Bins auch anders Aussehen (Abbildung 101).

```
cpu["Year"].hist(bins= 3)
```

<AxesSubplot:>



```
cpu["Year"].hist(bins= 5)
```

<AxesSubplot:>

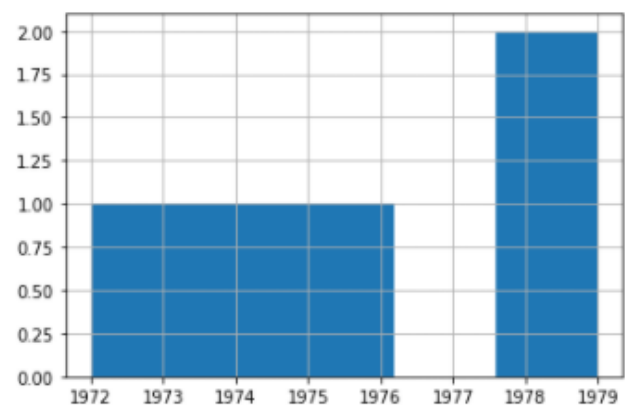


Abbildung 101 Histogramme mit unterschiedlichen Anzahl von „Bins“

Kommen in dem Datensatz Werte zwischen 1 und 20 und es sollten 10 Bins verwendet werden, so werden diese Werte auf Intervalle [0...2), [2...4), [4...6), [6...8), [8...10), [10...12), [12...14), [14...16), [16...18), [18...20] verteilt, wie es in der Abbildung 102 nicht schwer zu erkennen ist. Die Verteilung der Bins ist Standardmäßig linear.

```
values = (range(0, 21, 4))
df = pd.DataFrame(values)
df.hist(bins = 10)
```

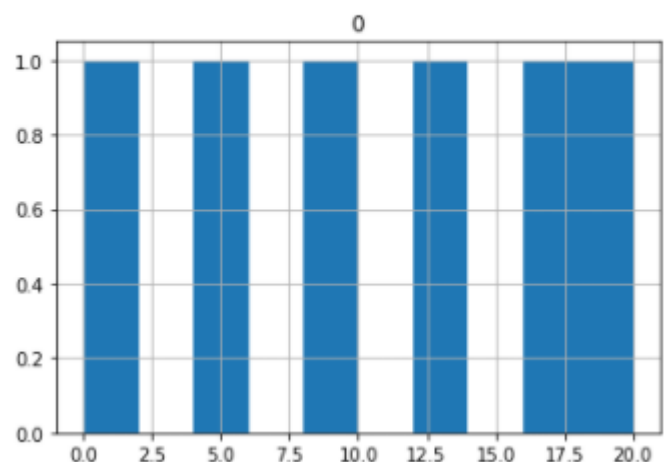


Abbildung 102 Verteilung der Werte auf die Bins

2.8.2 Nichtlineare Histogramme

Die Bins selbst, bzw. deren „Mitten“ lassen sich auch als ein Array von Werten angeben (Abbildung 103). In solch einem Fall werden die Bins an den Grenzwerten zwischen den in dem Array angegebenen Werten gebildet. So lässt sich die Verteilung der Daten auf die einzelnen Bins auch zielgerecht steuern.

```
bins_array =(range (1970,1984))
cpu["Year"].hist(bins = bins_array)
```

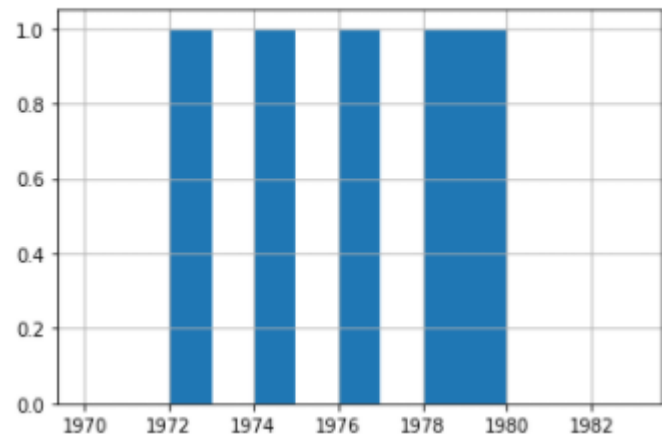


Abbildung 103 Bins eines Histogramms als Array vorgeben

Auf diese Art lassen sich auch Histogramme mit nichtlinearen Verteilungen erzeugen, indem die Intervalle selbst als eine nichtlineare Folge angegeben werden. Als ein Beispiel dazu können Sie der Abbildung 104 ein Histogramm mit einer logarithmischen Verteilung entnehmen. Hier wurden die Bins als eine logarithmische Folge (\log_{10}) angegeben, so dass die Verteilung auf die Bins auch logarithmisch getätigt wurde.

```
import numpy as np
values = (range (0, 21,4))
df = pd.DataFrame(values)
df.hist(bins=np.logspace(np.log10(1),np.log10(20.0), 10))
```

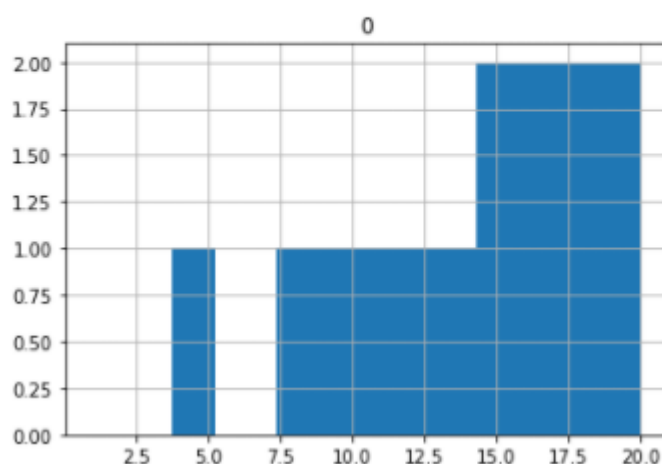


Abbildung 104: Ein nichtlineares Histogramm

2.9 1 aus n (One hot encoding)

Da die Machinelearning Algorithmen an sich nur mit numerischen Daten umgehen können, müssen alphanumerische Ausdrücke zuerst in Numerische Daten umgewandelt werden. Im Python stehen grundsätzlich zwei Möglichkeiten für solch eine automatische Umwandlung. Einerseits über eine Methode der DataFrame – der `get_dummies()` Methode, oder über sklearn Bibliothek mit der `OneHotEncoder` Klasse. Im diesem Kapitel wird zuerst nur die DataFrame Methode `get_dummies()` vorgestellt, die `OneHotEncoder` Klasse wird hingegen in dem Kapitel über die sklearn Bibliothek behandelt.

2.9.1 Einzelne Spalten umwandeln

Da die Machinelearning Algorithmen an sich nur mit numerischen Daten umgehen können, müssen alphanumerische Ausdrücke zuerst in Numerische Daten umgewandelt werden. Die Funktion `get_dummies` erzeugt aus einer gegebenen Spalte mit nicht numerischen Werten mehrere zusätzliche Spalten, deren Menge der Menge der unterschiedlichen Einträge in der Spalte entspricht (Abbildung 105) In die neuen Spalten wird an den Stellen der Wert 1 eingetragen an deren der entsprechende Bezeichner vorhanden war. Alle anderen neuen Spalten in der entsprechenden Zeile werden auf 0 gesetzt. Die Spalten selbst werden nach den ersetzten Einträgen benannt.

```
f2 = fruit.drop("Farbe", axis = 1 )
farben = pd.get_dummies(fruit["Farbe"])
pd.concat([f2,farben] ,axis = 1).head(5)
```

	Name	Gewicht	Geschmack	Gelb	Grün	Rot
0	Apfel	150	Sauer	0	0	1
1	Apfel	140	Sauer	0	0	1
2	Zitrone	54	Sauer	0	1	0
3	Banane	156	Süß	1	0	0
4	Apfel	151	Süß	0	0	1

Abbildung 105 Eine Spalte in Numerische Werte umwandeln.

2.9.2 Mehrere Spalten gleichzeitig umwandeln

Die `get_dummies` Methode ist nicht darauf beschränkt nur einer Spalte zu arbeiten. Es können auch mehrere Spalten als Parameter in einem Set angegeben werden. Diese werden dann für sich einzeln bearbeitet und zum Schluss zu einer Tabelle zusammengefügt. In der Abbildung 106 wurden so die beiden Spalten „Farbe“ sowie „Geschmack“ die alphanumerische Ausdrücke enthalten haben automatisch zu Numerischen Ausdrücken umgewandelt (zweite Zeile). Da in der ersten Zeile die

```
f2 = fruit.drop({"Farbe", "Geschmack"}, axis = 1 )
farben = pd.get_dummies(fruit[{"Farbe", "Geschmack"}])
pd.concat([f2, farben], axis = 1).head(5)
```

	Name	Gewicht	Farbe_Gelb	Farbe_Grün	Farbe_Rot	Geschmack_Sauer	Geschmack_Süß
0	Apfel	150	0	0	1	1	0
1	Apfel	140	0	0	1	1	0
2	Zitrone	54	0	1	0	1	0
3	Banane	156	1	0	0	0	1
4	Apfel	151	0	0	1	0	1

Abbildung 106 `get_dummies` mit mehreren Spalten

beiden Spalten aus der Kopie der Original DataFrame bereits entfernt wurden, sind nach dem Zusammenfügen (Zeile drei) in dem Dataframe nur noch (bis auf die Spalte „Name“) numerische Ausdrücke enthalten. Bitte beachten Sie das hier zwei mal der Parameter `axis = 1` vorhanden ist. Dieser sorgt dafür das die entsprechenden Operationen (**drop** sowie **concat**) auf Spalten statt Zeilen angewandt werden.

3 Sklearn

Die scikit-learn Bibliothek beinhaltet die Tools die für Umsetzung von Aufgaben zu Datenvorhersage im Python notwendig sind.

3.1.1 Daten in Training- und Testdaten aufteilen

Da in der Datenvorhersage immer mit Training sowie Testdaten gearbeitet wird, so ist eine Funktion notwendig die bereits vorhandene Daten in genau die beiden Gruppen zufällig trennt. Diese Aufgabe erfüllt die ***train_test_split*** Funktion. Diese erzeugt aus einer, oder mehreren, Datentabellen jeweils ein zufällig verteiltes Training sowie einem Testdatensatz (Abbildung 107). Über dem Argument ***test_size*** lässt sich das Größenverhältnis der beiden Subsets bestimmen wird keine Größe angegeben so wird ein Standardwert von 0.25 (25%) verwendet. Alternativ dazu lässt sich auch mit dem Parameter ***train_size*** das Größenverhältnis des zu erstellenden Trainingsdatensatzes einstellen. Ein weiterer Wichtiger Parameter ist ***random_state*** dieser bestimmt dem Startwert der Zufallsfunktion. Ein gleicher Startwert erzeugt immer exakt die gleiche Verteilung der Datensätze so das die Funktion immer Identische Verteilung der Datensätze erzeugt. Wird kein Startwert angegeben, so wird der aktuelle Startwert des Zufallszahlengenerators verwendet, was jedes mal zu unterschiedlichen Verteilung der Datensätze auf die beiden Gruppen führt.

```
training , test = train_test_split(cpu, test_size = 0.5)
print("training data : \n", training)
print("Test data : \n", test )
```

```
training data :
  Manufacturer    CPU Technology Package Pins  Clock  Transistors   Year
5         Zilog    Z80         NMOS   CDIP   40    2.5      8500.0  1976.0
4         Intel   8080         NMOS   CDIP   40    NaN      6000.0  1974.0
1          MOS   6502A         NMOS   PDIP   40    2.0      3510.0   NaN
0          MOS   6502         NMOS   PDIP   40    1.0      3510.0   NaN
Test data :
  Manufacturer    CPU Technology Package Pins  Clock  Transistors   Year
7   Motorola   6809         NMOS   PDIP   40    1.0      9000.0  1978.0
8   Motorola  68000         HMOS   PDIP   64   10.0     68000.0  1979.0
3         Intel   8008         PMOS   PDIP   18    NaN      3500.0  1972.0
2          MOS   6502B         NMOS   PDIP   40    3.0         NaN   NaN
6   Motorola   6800         NMOS   PDIP   40    1.0      4100.0   NaN
```

Abbildung 107 Daten in Test sowie Trainingsdaten spalten

Für die meisten Operationen im Bereich der ML werden (mindestens) Datenpaare benötigt. Auch diese lassen sich direkt mit der ***train_test_split*** Funktion erzeugen. Im dem folgendem Beispiel (Abbildung 108) wurden zuerst 3 Datensätze aus dem erstellt. Danach werden alle drei

gleichzeitig in Training sowie Testdaten gespalten. Dies stellt sicher, die Spalten in den neu erstellten Tabellen genau zusammenpassen. So ist in dem Beispiel der 7te Datensatz aus der Ursprungstabelle an der ersten Stelle in den drei neuen Tabellen, so wie der erste an immer an der 4ten Stelle zu finden ist.

```
pins = cpu["Pins"]
clock = cpu["Clock"]
manufacturer = cpu["Manufacturer"]

cl_tr, cl_te, ma_tr, ma_te, pi_tr, pi_te = train_test_split(clock, manufacturer, pins)

print("clock : \n", cl_tr)
print("Manufacturer : \n", ma_tr)
```

```
clock :
 7      1.0
 8     10.0
 3      NaN
 1      2.0
 4      NaN
 5      2.5
Name: Clock, dtype: float64
Manufacturer :
 7      Motorola
 8      Motorola
 3          Intel
 1          MOS
 4          Intel
 5          Zilog
Name: Manufacturer, dtype: object
```

Abbildung 108 Mehrere Daten gleichzeitig in Test sowie Trainingsdaten spalten

Somit lassen sich die jetzt die entsprechende paare (Tripel usw.) aus der Datensätzen der Ursprungstabelle wieder zueinander zuordnen um diese z.b. für eine Regressionsrechnung zu verwenden.

3.2 Label Encoder

Im Unterschied zu dem OneHotEncoding wird beim LabelEncoder aus den vorhandenen Alphanumerischen Daten nur eine Spalte erstellt. Diese kodiert die Unterschiedlichen Alphanumerischen Ausdrücke mit verschiedenen numerischen Werten. So wird in der Abbildung 109 der Farbe rot der Wert von 2, der Farbe grün 1 und der Farbe gelb der Wert von 0 zugewiesen. Diese Umwandlung wird von der Methode **fit_transform** durchgeführt. Das Ergebnis ist ein Feld der Klasse **ndarray**, das direkt einer Dataframe als neue Spalte hinzugefügt werden kann.

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
fruit['Farben'] = label_encoder.fit_transform(fruit['Farbe'])
fruit.head(5)
```

	Name	Gewicht	Farbe	Geschmack	Farben
0	Apfel	150	Rot	Sauer	2
1	Apfel	140	Rot	Sauer	2
2	Zitrone	54	Grün	Sauer	1
3	Banane	156	Gelb	Süß	0
4	Apfel	151	Rot	Süß	2

Abbildung 109 Label Encoder

3.3 Logistische Regression

3.3.1 Eine Logistische Regression erstellen

jdshf

3.3.2 Fit

fit

3.3.3 predict

Predict

Accuracy Klasifizieren

sdfs

3.3.4 Genauigkeit berechnen

Accuracy Klasifizieren

3.3.5 Präzision berechnen

Die Genauigkeit der Vorhersage lässt sich mit der Funktion `precision_score`

3.3.6 Trefferquote ermitteln

`recall_score`

Table of Figures

Abbildung 1: ein Jupyter Notebook.....	5
Abbildung 2: Die Arbeitsfläche im Jupyter Notebook.....	6
Abbildung 3: eine Mehrfach ausgeführte Codezeile.....	6
Abbildung 4 Fehlermeldung in eine Zelle.....	7
Abbildung 5: Jupyter Notebook Werkzeugleiste.....	8
Abbildung 6: Befehlspalette in Jupyter Notebook.....	9
Abbildung 7: Befehlspalette durchsuchen.....	10
Abbildung 8 Pandas importieren.....	10
Abbildung 9 CPU Datensatz als Instanz der Pandas Klasse DataFrame.....	11
Abbildung 10 einem leeren DataFrame anlegen.....	12
Abbildung 11: einem DataFrame mit Daten anlegen.....	12
Abbildung 12 Dimension einer DataFrame.....	12
Abbildung 13 Spaltentypen eines DataFrame anzeigen.....	13
Abbildung 14 DataFrame Informationen.....	13
Abbildung 15 Zugriff auf einzelne Spalten.....	13
Abbildung 16: Eine Spalte umbenennen.....	14
Abbildung 17 Mehrere Spalten gleichzeitig umbenennen.....	14
Abbildung 18: Spalten Hinzufügen.....	14
Abbildung 19: Datensätze einer Spalte ändern.....	15
Abbildung 20 Einzelne Datensätze ändern.....	15
Abbildung 21 Datensätze Auswählen.....	15
Abbildung 22 Datensätze aus einer Dataframe filtern.....	16
Abbildung 23 Daten in Pandas laden.....	16
Abbildung 24: Die ersten n Datensätze (Zeilen) mit head() anzeigen.....	17
Abbildung 25 Rückgabe der Methode head().....	17
Abbildung 26 Die Letzten n Datensätze (Zeilen) mit head() anzeigen.....	17
Abbildung 27.....	18
Abbildung 28 Unerwünschte Veränderung der ursprünglichen Daten.....	19
Abbildung 29 Unerwünschte Veränderung der ursprünglichen Daten mit copy() verhindern.....	19
Abbildung 30 Anzahl der gültigen Daten in den Spalten zählen.....	19
Abbildung 31 Anzahl der Daten in einer Spalte zählen.....	20
Abbildung 32 Funktionsweise der Methode .unique().....	20
Abbildung 33: Funktionsweise der Methode .unique().....	21
Abbildung 34.....	21
Abbildung 35 Spalten gruppiert auswerten.....	22
Abbildung 36: Funktionsweise der groupby() Methode.....	22
Abbildung 37 Spalten einer Dataframe auswerten.....	23
Abbildung 38: Funktionsweise einer Spaltenauswertung.....	23
Abbildung 39 Spaltenauswertung kombinieren.....	23
Abbildung 40 ein Wert als NaN definieren.....	24
Abbildung 41: NaNs Anzeigen.....	24
Abbildung 42.....	25
Abbildung 43 NaNs zählen.....	25
Abbildung 44 NaNs pro Spalte zählen.....	25
Abbildung 45 NaNs zählen.....	25

Einführung in das Maschinelle Lernen

Abbildung 46 NaNs mit einem Wert (-1) Auffüllen.....	26
Abbildung 47 Auffüllen mit dem Wert des Vorgängers.....	26
Abbildung 48 Auffüllen mit dem Wert des Nachfolgers.....	27
Abbildung 49 Auffüllung limitieren.....	27
Abbildung 50 NaNs mit Durchschnittswerten füllen.....	28
Abbildung 51: Datensätze mit NaNs entfernen.....	29
Abbildung 52: Spalten mit NaNs entfernen.....	29
Abbildung 53: Spalten mit drop() entfernen.....	30
Abbildung 54 Einzelne Spalten mit pop() entfernen.....	30
Abbildung 55 Datensätze mit .drop() entfernen.....	31
Abbildung 56: Datensätze selektiv entfernen.....	32
Abbildung 57 Indexliste generieren.....	32
Abbildung 58 Summen bilden mit .sum().....	33
Abbildung 59 Maximas ermitteln mit max().....	34
Abbildung 60 Minimas ermitteln mit min().....	34
Abbildung 61: Differenz bilden.....	35
Abbildung 62 Addition und Subtraktion.....	35
Abbildung 63 DataFrame indexieren.....	36
Abbildung 64: DataFrames zusammenfügen.....	37
Abbildung 65 DataFrames Indexieren.....	37
Abbildung 66 Verknüpfung von DataFrames über Indexe.....	38
Abbildung 67 Verknüpfung von DataFrames mit rechten Masterindex.....	38
Abbildung 68 Verknüpfung von DataFrames als Vereinigungsmenge.....	38
Abbildung 69 Verknüpfung von DataFrames als Schnittmenge.....	39
Abbildung 70: Dataframes expandieren.....	39
Abbildung 71: Datensätze suchen mit loc.....	40
Abbildung 72 Suchen mit Liste.....	40
Abbildung 73 Bereiche auswählen.....	40
Abbildung 74 Ausgabe einschränken.....	41
Abbildung 75 Vorselektion.....	41
Abbildung 76 Selektieren mit Wahrheitsliste.....	41
Abbildung 77 einem Datensatz mit .iloc[] selektieren.....	42
Abbildung 78 Mehrere Datensätze mit iloc[] selektieren.....	42
Abbildung 79 Bereiche mit iloc[] selektieren.....	42
Abbildung 80 .iloc[] Auswahl über Wahrheitsliste.....	43
Abbildung 81 .iloc[] Selektion mit Funktionen.....	43
Abbildung 82 Auswahl eines Datenfeldes mit .iloc[].....	43
Abbildung 83 Mehrere Datenfelder auswählen.....	43
Abbildung 84 Funktionen auf Datenfeldern ausführen.....	44
Abbildung 85 Ein Plot (Balkenplot).....	45
Abbildung 86 vertikale und horizontale Balkengrafik.....	46
Abbildung 87: ein Histogramm.....	46
Abbildung 88 Dichte-Diagramm und Histogramm mit gleichen Daten.....	47
Abbildung 89 Liniengrafik.....	47
Abbildung 90 Ein Flächendiagramm.....	48
Abbildung 91 Eine Kastengrafik.....	48
Abbildung 92 Kuchendiagramm.....	49
Abbildung 93 Streudiagramm.....	49

Einführung in das Maschinelle Lernen

Abbildung 94 Ein Hexbin Plot.....	50
Abbildung 95: Plots logarithmisch skalieren.....	51
Abbildung 96 Linien Formatieren.....	51
Abbildung 97 Legende eines Plots parametrisieren.....	52
Abbildung 98 Ein Histogramm.....	53
Abbildung 99 Mehrere Histogramme Automatisch erzeugen.....	53
Abbildung 100.....	54
Abbildung 101 Histogramme mit unterschiedlichen Anzahl von „Bins“.....	55
Abbildung 102 Verteilung der Werte auf die Bins.....	55
Abbildung 103 Bins eines Histogramms als Array vorgeben.....	56
Abbildung 104: Ein nichtlineares Histogramm.....	56
Abbildung 105 Eine Spalte in Numerische Werte umwandeln.....	57
Abbildung 106 get_dummies mit mehreren Spalten.....	58
Abbildung 107 Daten in Test sowie Trainingsdaten spalten.....	59
Abbildung 108 Mehrere Daten gleichzeitig in Test sowie Trainingsdaten spalten.....	60
Abbildung 109 Label Encoder.....	61