Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Datenverarbeitung in Python mit NumPy, Matplotlib und Pandas

Proseminar „Softwareentwicklung in der Wissenschaft“

vorgelegt von

Larissa Lach

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	7299048
Betreuer:	Jannek Squar

Hamburg, 31.08.2020

Inhaltsverzeichnis

1 Einleitung

Python ist eine Programmiersprache mit klarer Syntax und einfacher Lesbarkeit. Mithilfe der bereits vorhandenden Standardbibliothek können also schon in wenigen Zeilen eine Menge an Programmen geschrieben werden. Da Python ohne Erweiterungen jedoch in der Ausführung von komplexen Anwendungen zu den langsamsten Programmiersprachen gehört, sollte man sogenannte Python Pakete verwenden.

Für Python existiert bereits eine große Sammlung an Add-on-Paketen, welche die Programmiersprache in verschiedenen Hinsichten verbessert. In Abbildung 1.1 sind vier der beliebtesten Pakete für die Datenverarbeitung in Zusammenhang zueinander dargestellt. Python bietet die grundlegende Datenstruktur für alle Pakete. Numpy bringt mehrdimensionale Arrays mit sich und ermöglicht einfaches Arbeiten mit Matrizen. Auf SciPy wird in dieser Ausarbeitung nicht genauer eingegangen. Dieses Paket ist hauptsächlich für die Erweiterung der Leistungsfähigkeit von Numpy zuständig. Matplotlib ermöglicht die grafische Darstellung von Daten und Pandas erleichtert die Verarbeitung von Datentabellen mit verschiedenen Datentypen.

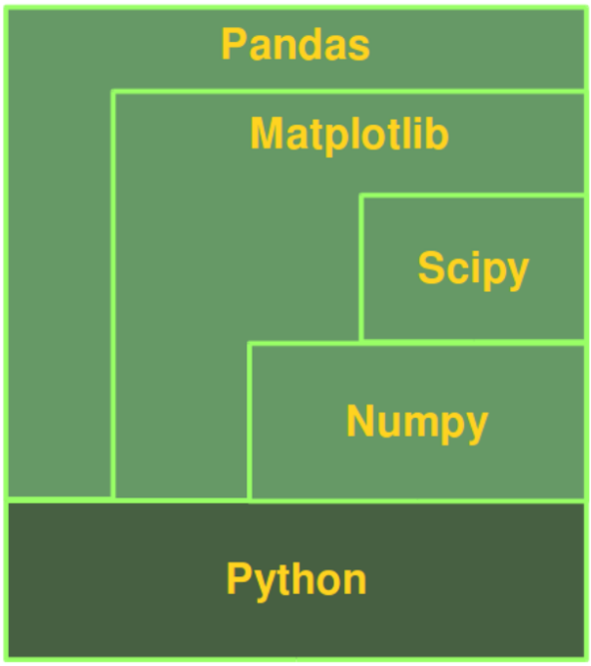

Abbildung 1.1: Zusammenhang zwischen den Paketen

2 NumPy (numerisches Python)

2.1 Allgemeines

NymPy erweitert Python um numerische Funktionen, welche ursprünglich für wissenschaftliche Berechnungen entworfen wurden. Das Paket ist aber auch für andere Funktionen der Datenverarbeitung nützlich, wie zum Beispiel zur Bildbearbeitung.

Da Numpy größtenteils in C geschrieben ist, beschleunigt die Nutzung von Numpy die Ausführung von Python Programmen sehr. Aus diesem Grund und dadurch, dass es den Quelltext sehr verkürzt, wird Numpy auch als Backend vieler anderer Pakete verwendet. Zusätzlich benötigt das mehrdimensionale Array von Numpy deutlich weniger Speicherplatz, da sie zusammenhängend gespeichert werden. Die Listen von Python hingegen nutzen Referenzen auf Speicherblöcke, was eben deutlich mehr Speicher beansprucht. Ein weiterer Vorteil besteht darin, dass Numpy viele bereits implementierte Funktionen mit sich bringt, welche das Arbeiten mit Daten deutlich vereinfacht.

Zum einen können Arrays in allen möglichen Arten erstellt werden, wie zum Beispiel mit zufälligen Zahlen oder eine Einheitsmatrix. Außerdem gibt es Allgemeine Funktionen, wie für den Zugriff auf Größe, Form, Dimension und weiteres. Zudem gibt es Funktionen um auf bestimmte Daten zuzugreifen oder zu ändern. Gebräuchliche Array-Algorithmen, wie Sortieren, sind ebenfalls berteits implementiert. Zusätzlich gibt es noch diverse mathematische Funktionen, wie Numpy-Operationen, welche komplexe Berechnungen auf ganzen Arrays ausführen oder Funktionen der linearen Algebra.

In dem Beispiel in der Abbildung 2.1 wird gezeigt, wie man Numpy ersteinmal importieren muss. Danach wird ein 3-dimensionales Array erstellt und der Datentyp auf int16 gestellt. Dies ist in diesem Falle sinnvoll, da in dem Beispiel definitiv nicht mehr benötigt wird und somit Speicher gespart werden kann. In den Zeilen danach werden ein paar allgemeine Funktionen beispielhaft angewandt, um ein kleinen Überblick über Numpy-Arrays zu verschaffen. Auf der rechten Seite sind die Lösugen dazu zu sehen.

Ein weiterer interessanter Aspekt, welcher mit Numpy möglich ist, ist das Maskieren von Arrays. Die Maskierung von Arrays ist sinnvoll, wenn sie fehlende oder ungültige Einträge beinhalten. Das Modul numpy.ma bringt eine Menge an Funktionen mit, mit denen man sehr einfach maskierte Arrays erstellen kann. Ein maskiertes Array ist in diesem Fall einfach eine Kombination aus einer Maske und einem Numpy-Array. Die Maske ist ein Array, welches aus booleschen Werten besteht. Dabei bestimmt jeder Wert die Gültigkeit des Wertes, in dem zugeordneten Numpy-Array an der gleichen Stelle. Wenn ein Wert in dem Array ungültig ist, steht in der Maske eine 1 also True. Ist der Wert gültig, steht in der Maske eine 0 also False.

```
1  import numpy as np
2
3  arr = np.array([[[1, 2, 3],
4                   [4, 5, 6]],
5                  [[7, 8, 9],
6                   [10, 11, 12]]],
7                   dtype="int16")
8  dimension = arr.ndim
9  form = arr.shape
10 typ = arr.dtype
11 element = arr[1, 0, 2]
12 reihe = arr[0, 0, :]
13 reshape = arr.reshape((3, 4))
```

```
dimension:  3

form:       (2, 2, 3)

typ:        int16

element:    9

reihe:      [1 2 3]

reshape:    [[1  2  3  4]
             [5  6  7  8]
             [9 10 11 12]]
```

Abbildung 2.1: Beispiel Array und Funktionen

2.2 uFunc (universelle Funktion)

Im vorherigen Absatz wurden bereits die Funktionen, welche Numpy mit sich bringt, angesprochen. Die meisten dieser Funktionen gehören zu der Klasse numpy.ufunc. Diese Klasse ist größtenteils in C implementiert, was zu dem bereits genannten Geschwindigkeitsvorteil führt.

Außerdem werden, dank dieser Klasse, keine Python-for-Schleifen oder if-Abfragen mehr benötigt. Dafür gibt es drei Gründe. Zum einen arbeitet Ufunc mit Vektorisierung. Schleifenblöcke werden also durch speziellen Code optimiert, wodurch Ufunc zu einer schnellen, elementweisen und vektorisierten Array-Operation wird. Aus der Vektorisierung resultiert eine Zeiteinsparung, da moderne CPUs bereits für solche Operationen optimiert sind.

Zum anderen verwendet Ufunc eine feste Anzahl spezifischer Eingaben und erzeugt eine feste Anzahl spezifischer Ausgaben. Es sind also auch mehrere Eingaben möglich.

Zuletzt nutzt Ufunc Broadcasting, was benötigt wird um mit Eingaben verschiedener Formen arbeiten zu können. Um dies zu verdeutlichen, wird in Abbildung 2.2 als Beispiel aufgezeigt was Numpy macht, wenn zwei Arrays verschiedener Formen addiert werden sollen. Man hat ein Array x welches die Form (4,1) und die Dimension 2 hat und ein Array y, welches die Form (5) und die Dimension 1 hat. Verwendet man nun numpy.add(x, y) werden x und y zuerst gebroadcastet, sodass sie beide die selbe Form haben, wie es auf der rechten Seite in Abbildung 2.2 als x und y dargestellt ist. Danach wird dann elementweise addiert. Versucht man die beiden Arrays ohne Numpy zu addieren, werden die Elemente, wie mit zz gezeigt, lediglich aneinandergereiht.

```
1  import numpy as np
2
3  x = [[0],
4       [1],
5       [2],
6       [3]]
7
8  y = [1, 1, 1, 1, 1]
9
10
11 z = np.add(x, y)
12
13 zz = x + y
```

```
x = [[1 1 1 1 1]
     [1 1 1 1 1]
     [1 1 1 1 1]
     [1 1 1 1 1]]
y = [[0 0 0 0 0]
     [1 1 1 1 1]
     [2 2 2 2 2]
     [3 3 3 3 3]]
z = [[1 1 1 1 1]
     [2 2 2 2 2]
     [3 3 3 3 3]
     [4 4 4 4 4]]
zz = [[0],[1],[2],[3], 1, 1, 1, 1, 1]
```

Abbildung 2.2: Verdeutlichung von Broadcasting anhand von Arrayaddition

2.3 Wetterdaten Beispiel

Um Numpy kurz zu verdeutlichen, wird im Folgenden ein Beispiel aufgezeigt, welches in den nächsten Kapiteln noch fortgeführt wird.

In diesem Beispiel sollen Wetterdaten in Arrays geladen und später als Diagramm dargestellt werden. Die Tabelle 2.1 zeigt zum Überblick alle Daten auf. Für dieses Beispiel werden minimale, maximale und mittlere Temperaturen, sowie Sonnenstunden der jeweiligen Tage verwendet. Die Daten der Spalten wurden ohne Spaltenbezeichnung als Zeilen in eine Textdatein namens “Daten.txt“ gespeichert und mit drei Leerzeichen voneinander getrennt.

Zu Beginn muss man wie zuvor Numpy importieren. Da die Daten zum Teil unterschiedliche Datentypen haben, muss man für die einzelnen Datenreihen einzelne Arrays erstellen, indem man die jeweiligen Daten mit numpy.loadtxt(...) ganz einfach aus der Textdatei lädt. Dies funktioniert so für einfache Zahlen. Da die Daten für die Spalte Datum, jedoch nicht den Typen int oder Ähnliches besitzen, muss man diese als String in ein Array laden. Um dies zu tun nutzt man die Funktion numpy.genfromtxt(...) und setzt den Datentypen auf String. Wie in Listing 2.1 dargestellt, müssen die Funktionen auf dieses Beispiel angepasst werden.

```
1  import numpy as np
2
3  celsiusmin = np.loadtxt("Daten.txt", delimiter="   ", max_rows=1,
       ↪ skiprows=1)
4  celsiusmax = np.loadtxt("Daten.txt", delimiter="   ", max_rows=1,
       ↪ skiprows=2)
5  celsiusmid = np.loadtxt("Daten.txt", delimiter="   ", max_rows=1,
       ↪ skiprows=3)
```

```
6 sonne = np.loadtxt("Daten.txt", delimiter="    ", max_rows=1,
    ↪ skiprows=4)
7
8 datum = np.genfromtxt("Daten.txt", dtype="str", delimiter="    ",
    ↪ max_rows=1)
```

Listing 2.1: Laden der Wetterdaten

Datum	Minimum Temp.	Maximum Temp.	Mittlere Temp.	Sonne
01.05.	6,50	14,50	10,50	4,10
02.05.	7,30	13,60	10,45	2,60
03.05.	5,60	14,80	10,20	2,20
04.05.	3,50	14,20	8,85	11,10
05.05.	1,30	14,00	7,65	9,90
06.05.	0,70	14,30	7,50	14,10
07.05.	4,00	15,10	9,55	6,60
08.05.	1,70	19,30	10,50	12,40
09.05.	5,80	23,00	14,40	8,70
10.05.	6,60	24,20	15,40	8,70
11.05.	3,50	11,60	7,55	5,20
12.05.	2,20	10,60	6,40	5,20
13.05.	3,50	11,50	7,50	4,20
14.05.	1,40	12,20	6,80	10,10
15.05.	-0,40	12,70	6,15	5,10
16.05.	5,80	15,90	10,85	6,10
17.05.	6,20	17,20	11,70	6,70
18.05.	11,10	17,00	14,05	0,10
19.05.	7,40	19,00	13,20	7,60
20.05.	4,40	20,00	12,20	12,20
21.05.	8,00	22,00	15,00	9,30
22.05.	12,20	22,20	17,20	1,70
23.05.	8,60	17,00	12,80	5,40
24.05.	8,50	16,00	12,25	4,60
25.05.	8,50	15,20	11,85	2,70
26.05.	8,20	20,30	14,25	6,70
27.05.	7,00	19,70	13,35	9,70
28.05.	5,30	20,00	12,65	13,60
29.05.	5,10	21,90	13,50	13,10
30.05.	7,50	21,20	14,35	11,90
31.05.	8,20	20,20	14,20	10,60

Tabelle 2.1: Wetterdaten

3 Matplotlib

3.1 Allgemeines

Matplotlib erweitert Python um grafische Darstellungsmöglichkeiten. In Kombination mit Numpy bildet Matplotlib eine sehr gute Alternative für MATLAB. Dies ist ein anderes kostenpflichtiges Programm, mit dem man Daten grafisch darstellen kann. Da Python ohnehin immer beliebter wird und dazu noch kostenlos ist, wird Matlab Nutzern der Umstieg leichter gemacht. Für Matplotlib gibt es eine Erweiterung, mit der man in Matplotlib ähnlich wie in Matlab programmieren kann. Im Folgenden wird jedoch mit der objektorientierten API gearbeitet.

Mit Matplotlib bleibt der Quelltext für verschidene Diagrammarten sehr kurz und damit sehr übersichtlich. Dazu kommen noch diverse Möglichkeiten zur optischen Formatierung der Darstellung, wie zum Beispiel die Einstellung der Legende oder der Linienarten und Farben. Ein kleiner Teil hiervon ist in Abbildung 3.2 dargestellt.

Um mit Matplotlib arbeiten zu können muss zuerst die Klasse matplotlib.pyplot importiert werden. Die Daten die an der y-Achse dargestellt werden sollen, werden als Liste oder Array, wie in Listing 3.1, übergeben. Hierbei ist es sinnvoll Numpy Arrays zu nutzen. Für die x-Achse wird standardmäßig der Index von 0 - X verwendet. Möchte man keinen Standardindex nutzen, kann man auch für die x-Achse ein Array oder eine Liste mit eigenen Daten übergeben. Abbildung 3.1 zeigt das Diagramm, welches in Listing 3.1 erstellt wurde.

Eine interessante Funktion von Matplotlib ist pyplot.subplot. Mit dieser Funktion ist es möglich mehrere Graphen verschiedener Typen oder mit verschieden y-Achsen gleichzeitig in einem Diagramm darstellen zu können, ohne dass sie übereinander liegen und es unübersichtlich wird. Ein Beispiel dafür ist in Abbildung 3.3 oben rechts zu sehen. In dieser Abbildung sind außerdem noch einige andere Diagrammarten dargestellt, um einen kleinen Überblick zu verschaffen, was mit Matplotlib alles möglich ist.

```
1 import matplotlib.pyplot as plt
2
3 plt.plot([-1, -4.5, 16, 23, 15, 59])
4 plt.show()
```

Listing 3.1: Erstellen eines simplen Diagramms

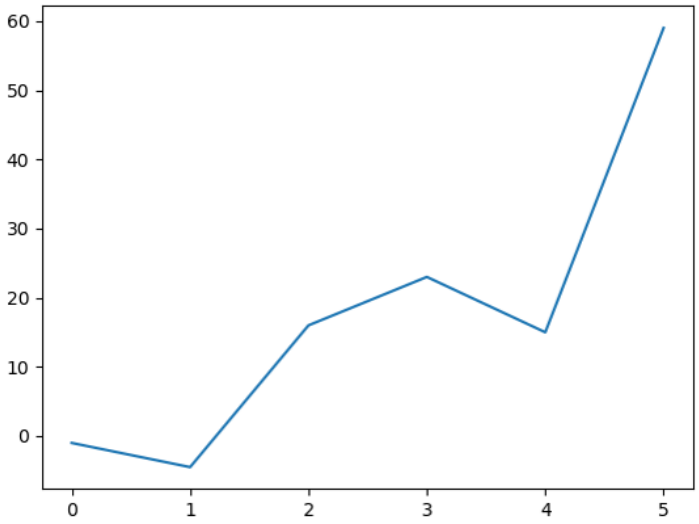

Abbildung 3.1: Beispieldiagramm

Zeichen	Beschreibung
'-'	(Bindestrich) durchgezogene Linie
'--'	(zwei Bindestriche) gestrichelte Linie
'-.'	Strichpunkt-Linie
':'	punktierte Linie
'.'	Punkt-Marker
','	Pixel-Marker
'o'	Kreis-Marker
'v'	Dreiecks-Marker, Spitze nach unten
'^'	Dreiecks-Marker, Spitze nach oben
'<'	Dreiecks-Marker, Spitze nach links
'>'	Dreiecks-Marker, Spitze nach rechts
'1'	tri-runter-Marker
'2'	tri-hoch-Marker
'3'	tri-links Marker
'4'	tri-rechts Marker
's'	quadratischer Marker
'p'	fünfeckiger Marker
'*'	Stern-Marker
'h'	Sechseck-Marker1
'H'	Sechseck-Marker2
'+'	Plus-Marker
'x'	x-Marker
'D'	rautenförmiger Marker
'd'	dünner rautenförmiger Marker
'|'	Marker in Form einer vertikalen Linie
'_'	Marker in Form einer horizontalen Linie

Zeichen	Farbe
'b'	blau
'g'	grün
'r'	rot
'c'	cyan
'm'	magenta
'y'	gelb
'k'	schwarz
'w'	weiß

Abbildung 3.2: Caption

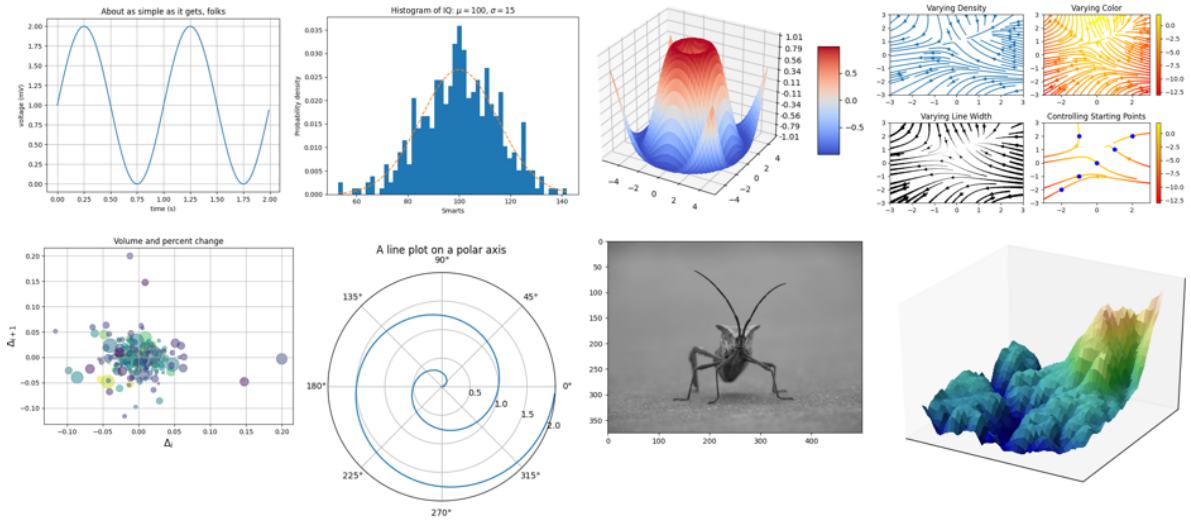

Abbildung 3.3: Caption

3.2 Wetterdaten Beispiel

Um Matplotlib nun auch besser zu verdeutlichen, wird im Folgenden das Beispiel aus Abschnitt 2.3 fortgeführt. Zusätzlich zu Numpy muss jetzt auch die Klasse pyplot von Matplotlib importiert werden. Nach dem Laden der Daten in Listing 3.2 wird noch ein Array erstellt, um die x-Achse später ordentlich formatieren zu können. Als nächstes werden zwei Subplots (ax1 und ax2) erstellt und formatiert. Wenn der erste Subplot definiert wird, bekommt auch die x-Achse einen Titel. Mit dem ersten Subplot werden die Sonnenstunden als Balkendiagramm und mit dem zweiten alle Daten die in °C angegeben sind als Liniendiagramm dargestellt. Um die Daten in einer Legende anzuzeigen, benötigen sie jeweils ein eigenes Label.

Danach folgen noch allgemeine Formatierungen, wie das Stezen des Diagrammtitels. Außerdem wird in diesem Beispiel der Bereich zwischen den Daten der maximalen und minimalen Temperaturen leicht durchsichtig gefärbt. Damit an der x Achse nicht 0-30 steht, sondern das jeweilige Datum werden hier die Beschriftungen der Striche zu den Daten gewechselt, die unter datum gespeichert sind. Zuletzt wurde noch die Legende unten rechts eingefügt

Wird dieses Programm nun ausgeführt entsteht das Diagramm, welches in Abbildung 3.4 gezeigt wird.

```
1  import numpy as np
2  from matplotlib import pyplot as plt
3
4  celsiusmin = np.loadtxt("Daten.txt", delimiter="    ", max_rows=1,
       ↪ skiprows=1)
5  celsiusmax = np.loadtxt("Daten.txt", delimiter="    ", max_rows=1,
       ↪ skiprows=2)
6  celsiusmid = np.loadtxt("Daten.txt", delimiter="    ", max_rows=1,
       ↪ skiprows=3)
7  sonne = np.loadtxt("Daten.txt", delimiter="    ", max_rows=1,
       ↪ skiprows=4)
8  datum = np.genfromtxt("Daten.txt", dtype="str", delimiter="    ",
       ↪ max_rows=1)
9
10 x = np.arange(0, 31, 1)
11
12 fig, ax1 = plt.subplots()
13
14 ax1.set_xlabel('Datum')
15 ax1.set_ylabel('Sonnenschein in Std.')
16 ax1.bar(x, sonne, color='indianred')
17
18 ax2 = ax1.twinx()  # zweite Achse mit der gleichen X-Achse
19
20 ax2.set_ylabel('°C')
21 ax2.plot(x, celsiusmin, 'blue', label='minimale Temperatur')
22 ax2.plot(x, celsiusmax, 'red', label='maximale Temperatur')
23 ax2.plot(x, celsiusmid, 'orange', label='mittlere Temperatur')
24
25 plt.title('Wetterüberblick')
26 plt.fill_between(x, celsiusmin, celsiusmax, color='blue',
       ↪ alpha=0.05)
27 plt.xticks(x, datum)
28 plt.legend(loc='lower right')
29
30 plt.show()
```

Listing 3.2: Wetterdaten mit Matplotlib darstellen

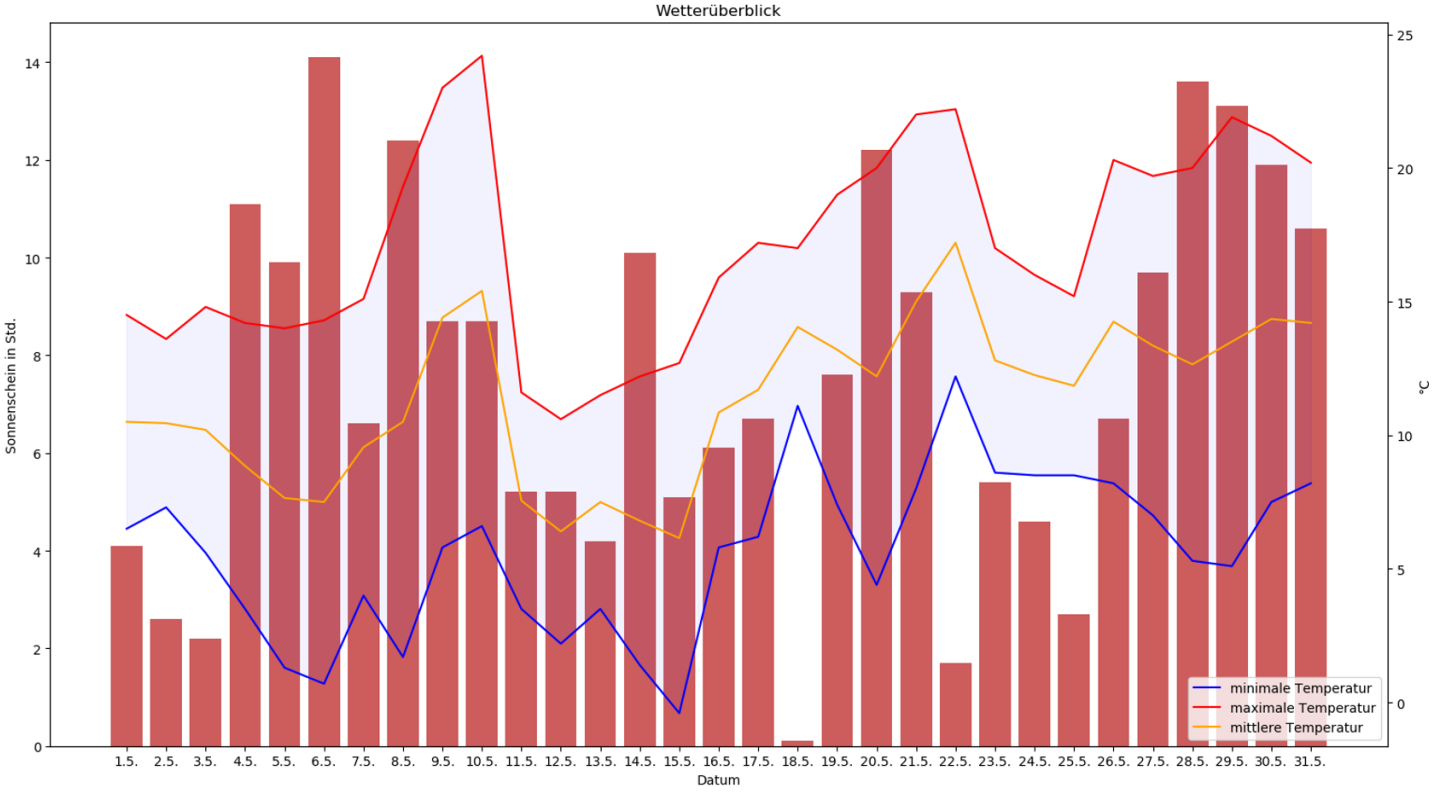

Abbildung 3.4: Wetterdiagramm

4 Pandas

4.1 Allgemeines

Der Begriff “Pandas“ steht für “Python and data analysis“ und “panal data“. Das Paket dient der Datenmanipulation und Datenanalyse. Es erweitert Python um Datenstrukturen und Funktionen zur Verarbeitung von Datentabellen.

Pandas bringt zwei verschiedene Datenstrukturen mit sich, zum einen Series und zum anderen DataFrame. Eine Series besteht aus einem Index und quasi einer Spalte Daten. Wenn kein bestimmter Index übergeben wird, wird der Standardindex (0 - x) genutzt. Die Daten werden als eindimensionales Array gespeichert, welches nur Daten eines Datentyps beinhaltet. Ein Dataframe basiert auf Tabellen. Hierbei existieren sowohl Zeilen, als auch Spaltenindex. Jede Spalte besteht dabei aus einem eindeutigen Datentypen, wobei verschiedene Spalten auch verschiedene Typen haben können.

Um mehrere Series in einen Dataframe umzuwandeln, gibt es eine Formel von Pandas. Sind x, y und z Series nutzt man pandas.concat([x, y, z], axis=1) und x, y und z sind in einem Dataframe zusammengefasst. Weitere wichtige Funktionen von Pandas sind unter anderem pandas.read*(), um Dateien zu importieren bzw zu lesen oder pandas.to*(), um Daten zu exportieren. Für das * muss dabei der jeweilige Dateityp angegeben werden.

4.2 Umgang mit fehlenden Daten

Fehlende Daten sind ein großes Problem in der Datenanalyse. So können sie beispielsweise sehr schnell beim Addieren zweier Tabellen oder Series entstehen. Wenn ein Index zum Beispiel nicht in beiden Seriesobjekten vorkommt, so wird der entsprochende Wert auf NaN gesetzt.

In Pandas gibt es zwei Funktionen, um auf fehlende Daten zu prüfen: isnull() und notnull(). Wurden sie gefunden, kann man auf zwei verschiedene Arten fortfahren. Man kann allen fehlenden Daten mit der Funktion fillna() einen bestimmten Wert zuweisen. Mit der Funktion dropna() kann die gesamte Zeile und mit dropna(axis=1) die gesamte Spalte gelöscht werden, in der ein NaN vorhanden ist.

Außerdem kann man, wie in dem Beispiel in der Tabelle 4.1, eine Anforderung angeben, dass nur bestimmte Zeilen oder Spalten gelöscht werden sollen. In dem Beispiel sollen alle Zeilen gelöscht werden, in denen weniger als fünf Daten tatsächlich existieren.

dropna(thresh=5, axis=0)

	A	B	C	D	E	F
0	NaN	NaN	14.2	14.3	NaN	NaN
1	14.5	14.5	14.0	15.0	14.5	NaN
2	14.6	NaN	14.8	15.3	14.0	14.2
3	NaN	14.9	15.7	NaN	14.0	15.3
4	15.2	15.2	14.6	15.3	15.5	14.9
5	NaN	15.8	15.9	16.9	16.0	16.2

Tabelle 4.1: Beispiel für dropna()

4.3 Wetterdaten Beispiel

Auch für Pandas wird im Folgenden noch einmal das Beispiel mit den Wetterdaten angepasst. Für Pandas wurden die Daten, wie in Tabelle 2.1, in einer Exceldatei gespeichert und mit der funktion pandas.read_excel() in eine Variable geladen. Wie die Daten abgespeichert werden, ist in Listing 4.2 zu erkennen. Mit Pandas muss nicht mehr jeder Datensatz einzeln gespeichert werden, da man in einem Dataframe über den Index auf die einzelnen Zeilen und Spalten zugreifen kann. Der Rest des Quelltexts bleibt ähnlich, wie in Abschnitt 3.2, nur das Laden der Daten in die Graphen benötigt die Angabe der Spaltenbezeichnung.

Das Diagramm, welches dabei entsteht, ist das selbe, wie in Abschnitt 3.2, also in Abbildung 3.4, dargestellt.

```
1  import numpy as np
2  import pandas as pd
3  from matplotlib import pyplot as plt
4
5  daten = pd.read_excel('Daten.xlsx')
6
7  x = np.arange(0, 31, 1)
8
9  fig, ax1 = plt.subplots()
10
11 ax1.set_xlabel('Datum')
12 ax1.set_ylabel('Sonnenschein in Std.')
13 ax1.bar(x, daten['Sonnenschein'], color='indianred')
14
15 ax2 = ax1.twinx()  # zweite Achse mit der gleichen X-Achse
16
17 ax2.set_ylabel('°C')
18 ax2.plot(x, daten['minimale Temp.'], 'blue', label='minimale
     ↪ Temperatur')
19 ax2.plot(x, daten['maximale Temp.'], 'red', label='maximale
```

```
    ↪ Temperatur')
20 ax2.plot(x, daten['mittlere Temp.'], 'orange', label='mittlere
    ↪ Temperatur')
21
22 plt.title('Wetterüberblick')
23 plt.fill_between(x, daten['minimale Temp.'], daten['maximale
    ↪ Temp.'], color='blue', alpha=0.05)
24 plt.xticks(x, daten['Datum'])
25 plt.legend(loc='lower right')
26
27 plt.show()
```

Listing 4.1: Wetterdaten mit Pandas darstellen

.	Datum	minimale Temp.	maximale Temp.	mittlere Temp.	Sonnenschein
0	01.5.	6.5	14.5	10.50	4.1
1	02.5.	7.3	13.6	10.45	2.6
2	03.5.	5.6	14.8	10.20	2.2
3	04.5.	3.5	14.2	8.85	11.1
4	05.5.	1.3	14.0	7.65	9.9
5	06.5.	0.7	14.3	7.50	14.1
6	07.5.	4.0	15.1	9.55	6.6
7	08.5.	1.7	19.3	10.50	12.4
8	09.5.	5.8	23.0	14.40	8.7
9	10.5.	6.6	24.2	15.40	8.7
10	11.5.	3.5	11.6	7.55	5.2
11	12.5.	2.2	10.6	6.40	5.2
12	13.5.	3.5	11.5	7.50	4.2
13	14.5.	1.4	12.2	6.80	10.1
14	15.5.	−0.4	12.7	6.15	5.1
15	16.5.	5.8	15.9	10.85	6.1
16	17.5.	6.2	17.2	11.70	6.7
17	18.5.	11.1	17.0	14.05	0.1
18	19.5.	7.4	19.0	13.20	7.6
19	20.5.	4.4	20.0	12.20	12.2
20	21.5.	8.0	22.0	15.00	9.3
21	22.5.	12.2	22.2	17.20	1.7
22	23.5.	8.6	17.0	12.80	5.4
23	24.5.	8.5	16.0	12.25	4.6
24	25.5.	8.5	15.2	11.85	2.7
25	26.5.	8.2	20.3	14.25	6.7
26	27.5.	7.0	19.7	13.35	9.7
27	28.5.	5.3	20.0	12.65	13.6
28	29.5.	5.1	21.9	13.50	13.1
29	30.5.	7.5	21.2	14.35	11.9
30	31.5.	8.2	20.2	14.20	10.6

Listing 4.2: Wie die Daten abgespiechert werden

5 Zusammenfassung

Verschiedene Pakete für Python bringen also verschiedene Vorteile mit sich. Wenn man mit Python arbeiten möchte, ist es also sinnvoll, sich ordentlich zu informieren, welche Pakete dem Programm nutzen könnten.

Numpy ist dabei wohl das beliebteste und meist genutzte Paket in der Datenverarbeitung, da man einfach einen kürzeren Quelltext erlangen kann, welcher dazu noch leichter zu verstehen ist. Dazu kommt noch der Geschwindigkeitsvorteil durch die in C geschriebene Klasse numpy.ufunc, ohne die die Ausführung von Python Programmen meist deutlich langsamer ist, als bei anderen Programmiersprachen.

Matplotlib ist in Verbindung mit Numpy eine sehr gute und dazu noch kostenlose Alternative für Matlab. Mit diesem Paket ist es möglich, schnell, einfach und mit übersichtlichen, leicht zu verstehenden Quelltext auch komplexe Diagramme darstellen zu lassen. Außerdem ist die Vielfalt an Diagrammarten und Gestaltungsmöglichkeiten sehr groß, wodurch Matplotlib für viele Anwendungsbereiche sinnvoll ist.

Pandas ist besonders sinnvoll, wenn man mit verschiedenen Datentypen arbeitet und alle Daten in einem Objekt speichern möchte. Neben den Paketen, welche in dieser Ausarbeitung erläutert wurden, gibt es natürlich noch viele weitere.

6 Literaturverzeichnis

(1) Numerisches Python: Einführung in Numpy
https://www.python-kurs.eu/numpy.php

(2) Introduction to Numpy
https://www.w3schools.com/python/numpy_intro.asp

(3) Offizielle NumPy Dokumentation
https://numpy.org/doc/stable/

(4) Numerisches Python: Einführung in Matplotlib
https://www.python-kurs.eu/matplotlib.php

(5) Matplotlib: Python plotting - offizielle Website von Matplotlib
https://matplotlib.org/

(6) Offizielle Matplotlib Dokumentation
https://matplotlib.org/contents.html

(7) Numerisches Python: Einführung in Pandas
https://www.python-kurs.eu/pandas.php

(8) Offizielle Pandas Dokumentation
https://pandas.pydata.org/docs/