

Berechnungen mit NumPy-Arrays: universelle Funktionen

Bislang haben wir uns mit den praktischen Grundlagen von NumPy befasst. In den folgenden Kapiteln werden wir der Frage auf den Grund gehen, warum NumPy in Pythons Data-Science-Welt von so herausragender Bedeutung ist. Fakt ist, dass es eine einfache und flexible Schnittstelle für optimierte Berechnungen mit Daten-Arrays bereitstellt.

Berechnungen mit NumPy-Arrays können äußerst schnell, aber auch sehr langsam sein. Für eine hohe Geschwindigkeit ist die Verwendung von vektorisierten Operationen entscheidend, die im Allgemeinen durch NumPys *universelle Funktionen* (engl. *Universal Functions* oder auch *UFuncs*) implementiert sind. Dieses Kapitel erläutert die Notwendigkeit von NumPys UFuncs, die sehr viel effizientere wiederholte Berechnungen mit Array-Elementen ermöglichen. Anschließend werden einige der gebräuchlichsten und praktischsten arithmetischen UFuncs vorgestellt, die im NumPy-Paket verfügbar sind.

Langsame Schleifen

Pythons Standardimplementierung (die auch als *CPython* bezeichnet wird) führt einige Operationen sehr langsam aus. Das liegt zum Teil daran, dass es sich bei Python um eine dynamische, interpretierte Programmiersprache handelt: Datentypen sind flexibel (dynamische Typisierung), wodurch Befehlsfolgen – anders als in Sprachen wie C oder Fortran – nicht zu hocheffizienter Maschinensprache kompiliert werden können. In jüngerer Zeit gab es verschiedene Anläufe, dieser Schwäche Herr zu werden: Das PyPy-Projekt (<http://pypy.org>), eine Python-Implementierung mit Just-in-Time-Compiler, und das Cython-Projekt (<http://cython.org>), das Python-Code in kompilierbaren C-Code konvertiert, sowie das Numba-Projekt (<http://numba.pydata.org>), das Python-Codeschnipsel in schnellen LLVM-Bytecode umwandelt, sind bekannte Beispiele. Diese Ansätze haben ihre Stärken und Schwächen, man kann aber mit Sicherheit sagen, dass noch keiner dieser drei Ansätze der Standard-CPython-Engine den Einfluss und die Verbreitung streitig gemacht hat.

Dass Python relativ langsam ist, macht sich insbesondere dann bemerkbar, wenn viele kleine Operationen wiederholt ausgeführt werden – beispielsweise Schleifen,

die auf alle Elemente eines Arrays zugreifen müssen, um diese zu verarbeiten. Nehmen wir beispielsweise an, wir möchten den Kehrwert aller in einem Array gespeicherten Werte berechnen. Ein schnörkelloser Ansatz könnte dann folgendermaßen aussehen:

```
In [1]: import numpy as np
        rng = np.random.default_rng(seed=1701)

        def compute_reciprocals(values):
            output = np.empty(len(values))
            for i in range(len(values)):
                output[i] = 1.0 / values[i]
            return output

        values = rng.integers(1, 10, size=5)
        compute_reciprocals(values)
Out[1]: array([0.11111111, 0.25          , 1.          , 0.33333333, 0.125        ])
```

Diese Implementierung erscheint einem C- oder Java-Programmierer vermutlich ganz natürlich. Wenn wir jedoch die Ausführungszeit dieses Codes mit einer umfangreichen Eingabe messen, müssen wir – vielleicht etwas überrascht – feststellen, dass dieser Vorgang sehr langsam ist! Wir messen die Zeit mit IPythons `%timeit`-Magic (siehe Abschnitt »Profiling und Timing von Code« auf Seite 45):

```
In [2]: big_array = rng.integers(1, 100, size=1000000)
        %timeit compute_reciprocals(big_array)
Out[2]: 2.61 s ± 192 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Die Ausführung dieser Millionen von Operationen und das Speichern des Resultats dauern mehrere Sekunden! In Anbetracht der Tatsache, dass selbst Mobiltelefone schon Rechenleistungen aufbringen, die sich in Gigaflops (eine Milliarde numerische Rechenoperationen pro Sekunde) bemessen, erscheint dieses Ergebnis absurd langsam. Es stellt sich heraus, dass nicht die Rechenoperationen selbst den Engpass verursachen, sondern die Typüberprüfung und die Funktionsaufrufe, die Python bei jedem Durchlauf der Schleife erledigen muss. Bei jeder Berechnung eines Kehrwerts überprüft Python zunächst den Objekttyp und muss dann die richtige Funktion ermitteln, die für diesen Typ anzuwenden ist. Bei kompiliertem Code wäre diese Typinformation vor der Ausführung des Codes bekannt, und das Ergebnis könnte sehr viel effizienter berechnet werden.

Kurz vorgestellt: UFuncs

NumPy stellt für viele Operationen komfortable Schnittstellen bereit, um auf genau diese Art statisch typisierter kompilierter Routinen zuzugreifen. Man spricht hier von einer *vektorierten* Operation. Um einfache Operationen wie die hier gezeigte elementweise Division zu vektorisieren, reicht es aus, Pythons arithmetische Operatoren direkt auf das Array-Objekt anzuwenden. Dieser vektorierte Ansatz sorgt dafür, dass die Schleife in die kompilierte Schicht verschoben wird, die NumPy zugrunde liegt, was zu einer deutlich schnelleren Ausführung führt.

Vergleichen Sie die beiden folgenden Ergebnisse:

```
In [3]: print(compute_reciprocals(values))
        print(1.0 / values)
Out[3]: [0.11111111 0.25          1.          0.33333333 0.125         ]
        [0.11111111 0.25          1.          0.33333333 0.125         ]
```

Wenn wir das mit der `big_array`-Ausführungszeit vergleichen, stellen wir fest, dass die Verarbeitung um Größenordnungen schneller abgeschlossen ist als die Python-Schleife:

```
In [4]: %timeit (1.0 / big_array)
Out[4]: 2.54 ms ± 383 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In NumPy sind vektorisierte Operationen durch UFuncs implementiert, deren Hauptaufgabe es ist, wiederholt Berechnungen mit Werten in NumPy-Arrays schnell auszuführen. UFuncs sind außerordentlich flexibel: Die letzte Operation betraf einen Skalar und ein Array, wir können jedoch auch zwei Arrays verwenden:

```
In [5]: np.arange(5) / np.arange(1, 6)
Out[5]: array([0.          , 0.5          , 0.66666667, 0.75          , 0.8          ])
```

UFunc-Operationen sind nicht auf eindimensionale Arrays beschränkt. Sie können auch auf mehrdimensionale Arrays angewendet werden:

```
In [6]: x = np.arange(9).reshape((3, 3))
        2 ** x
Out[6]: array([[ 1,  2,  4],
               [ 8, 16, 32],
               [64, 128, 256]])
```

Berechnungen, die die Vektorisierung durch UFuncs einsetzen, sind fast immer effizienter als ihre durch Python-Schleifen implementierten Pendanten, insbesondere bei größeren Arrays. Wenn Ihnen eine solche Schleife in einem NumPy-Skript begegnet, sollten Sie überlegen, ob sie durch einen vektorisierten Ausdruck ersetzt werden kann.

NumPys UFuncs im Detail

Es gibt zwei Varianten von UFuncs: *unäre*, die eine einzelne Eingabe verarbeiten, und *binäre*, die zwei Eingaben verarbeiten. Wir werden Beispiele für beide Funktionstypen betrachten.

Array-Arithmetik

Es erscheint ganz natürlich, NumPys UFuncs einzusetzen, weil sie von Python's systemeigenen arithmetischen Operatoren Gebrauch machen. Sie können die normalen Operatoren für Addition, Subtraktion, Multiplikation und Division verwenden:

```
In [7]: x = np.arange(4)
        print("x      =", x)
        print("x + 5  =", x + 5)
        print("x - 5  =", x - 5)
```

```

print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2) # Ganzzahldivision
Out[7]: x      = [0 1 2 3]
        x + 5   = [5 6 7 8]
        x - 5   = [-5 -4 -3 -2]
        x * 2   = [0 2 4 6]
        x / 2   = [0.  0.5 1.  1.5]
        x // 2  = [0 0 1 1]

```

Darüber hinaus gibt es unäre UFuncs für die Negation, einen `**`-Operator zum Potenzieren sowie einen `%`-Operator für die Berechnung des Modulos (also Teilungsrests):

```

In [8]: print("-x      =", -x)
        print("x ** 2 =", x ** 2)
        print("x % 2  =", x % 2)
Out[8]: -x      = [ 0 -1 -2 -3]
        x ** 2  = [0 1 4 9]
        x % 2   = [0 1 0 1]

```

Außerdem können Sie diese Operatoren nach Belieben miteinander verknüpfen. Dabei wird die übliche Rangfolge der Operatoren beachtet:

```

In [9]: -(0.5*x + 1) ** 2
Out[9]: array([-1. , -2.25, -4. , -6.25])

```

All diese arithmetischen Operatoren sind eigentlich nur komfortable »Verpackungen«, sogenannte Wrapper, für bestimmte in NumPy integrierte Funktionen. Beispielsweise ist der `+`-Operator ein Wrapper für die `add`-Funktion:

```

In [10]: np.add(x, 2)
Out[10]: array([2, 3, 4, 5])

```

In Tabelle 6-1 sind die in NumPy implementierten arithmetischen Operatoren zusammengefasst.

Tabelle 6-1: In NumPy implementierte arithmetische Operatoren

Operator	UFunc-Pendant	Beschreibung
+	<code>np.add</code>	Addition (z.B. $1 + 1 = 2$)
-	<code>np.subtract</code>	Subtraktion (z.B. $3 - 2 = 1$)
-	<code>np.negative</code>	unäre Negation (z.B. -2)
*	<code>np.multiply</code>	Multiplikation (z.B. $2 * 3 = 6$)
/	<code>np.divide</code>	Division (z.B. $3 / 2 = 1.5$)
//	<code>np.floor_divide</code>	Ganzzahldivision (z.B. $3 // 2 = 1$)
**	<code>np.power</code>	Potenzierung (z.B. $2 ** 3 = 8$)
%	<code>np.mod</code>	Modulo (Teilungsrest) (z.B. $9 \% 4 = 1$)

Darüber hinaus gibt es boolesche und bitweise arbeitende Operatoren, die wir in Kapitel 9 erkunden werden.

Absolutwert

NumPy kennt nicht nur Pythons integrierte arithmetische Operatoren, sondern auch die integrierte Absolutwertfunktion:

```
In [11]: x = np.array([-2, -1, 0, 1, 2])
          abs(x)
Out[11]: array([2, 1, 0, 1, 2])
```

Die entsprechende NumPy-UFunc heißt `np.absolute`, für die ebenso das Alias `np.abs` verwendet werden kann:

```
In [12]: np.absolute(x)
Out[12]: array([2, 1, 0, 1, 2])

In [13]: np.abs(x)
Out[13]: array([2, 1, 0, 1, 2])
```

Diese UFunc kann auch komplexe Zahlen verarbeiten. In diesem Fall liefert sie den Betrag einer komplexen Zahl (die Länge des Vektors in der komplexen Zahlenebene).

```
In [14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
          np.abs(x)
Out[14]: array([5., 5., 2., 1.])
```

Trigonometrische Funktionen

NumPy stellt eine Vielzahl praktischer UFuncs zur Verfügung. Zu den für Data Scientists nützlichsten gehören die trigonometrischen Funktionen. Wir definieren zunächst ein Array mit Winkeln:

```
In [15]: theta = np.linspace(0, np.pi, 3)
```

Nun können wir ein paar trigonometrische Funktionen dieser Werte berechnen:

```
In [16]: print("theta      = ", theta)
          print("sin(theta) = ", np.sin(theta))
          print("cos(theta) = ", np.cos(theta))
          print("tan(theta) = ", np.tan(theta))
Out[16]: theta      = [0.          1.57079633  3.14159265]
          sin(theta) = [0.00000000e+00  1.00000000e+00  1.2246468e-16]
          cos(theta) = [ 1.0000000e+00  6.123234e-17 -1.0000000e+00]
          tan(theta) = [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

Die Werte werden mit der begrenzten Präzision des Computers berechnet, daher sind einige der Werte, die eigentlich null sein sollten, nicht immer exakt null. Die trigonometrischen Umkehrfunktionen sind ebenfalls verfügbar:

```
In [17]: x = [-1, 0, 1]
          print("x          = ", x)
          print("arcsin(x) = ", np.arcsin(x))
          print("arccos(x) = ", np.arccos(x))
          print("arctan(x) = ", np.arctan(x))
Out[17]: x          = [-1, 0, 1]
          arcsin(x) = [-1.57079633  0.          1.57079633]
          arccos(x) = [ 3.14159265  1.57079633  0.          ]
          arctan(x) = [-0.78539816  0.          0.78539816]
```

Potenzen und Logarithmen

Potenzen und Exponentialfunktion sind weitere gängige Operationen, die als NumPy-UFuncs zur Verfügung stehen:

```
In [18]: x = [1, 2, 3]
         print("x =", x)
         print("e^x =", np.exp(x))
         print("2^x =", np.exp2(x))
         print("3^x =", np.power(3., x))
Out[18]: x      = [1, 2, 3]
         e^x = [ 2.71828183  7.3890561 20.08553692]
         2^x = [2.  4.  8.]
         3^x = [ 3.  9. 27.]
```

Die Umkehrfunktionen der Exponentialfunktionen – Logarithmen – sind ebenfalls verfügbar. `np.log` liefert den natürlichen Logarithmus. Sollten Sie es vorziehen, Logarithmen zur Basis 2 oder 10 zu verwenden, stehen diese ebenfalls bereit:

```
In [19]: x = [1, 2, 4, 10]
         print("x      =", x)
         print("ln(x)  =", np.log(x))
         print("log2(x) =", np.log2(x))
         print("log10(x) =", np.log10(x))
Out[19]: x      = [1, 2, 4, 10]
         ln(x)  = [0.          0.69314718  1.38629436  2.30258509]
         log2(x) = [0.          1.          2.          3.32192809]
         log10(x) = [0.          0.30103   0.60205999  1.          ]
```

Es gibt außerdem spezielle Versionen, die auch bei sehr kleinen Eingaben noch genaue Ergebnisse liefern:

```
In [20]: x = [0, 0.001, 0.01, 0.1]
         print("exp(x) - 1 =", np.expm1(x))
         print("log(1 + x) =", np.log1p(x))
Out[20]: exp(x) - 1 = [0.          0.0010005  0.01005017  0.10517092]
         log(1 + x) = [0.          0.0009995  0.00995033  0.09531018]
```

Diese Funktionen liefern für sehr kleine Werte von `x` präzisere Ergebnisse als die Funktionen `np.log` und `np.exp`.

Spezialisierte UFuncs

Es gibt eine Vielzahl weiterer UFuncs in NumPy, dazu gehören hyperbolische trigonometrische Funktionen, bitweise Arithmetik, Vergleichsoperatoren, Konvertierungsfunktionen von Bogenmaß in Grad, Rundung, Teilungsrest und vieles mehr. Das Stöbern in der NumPy-Dokumentation fördert eine Menge interessanter Funktionalität zutage.

Das Submodul `scipy.special` ist eine weitere hervorragende Quelle für spezialisierte oder weniger bekannte UFuncs. Wenn Sie irgendeine ungewöhnliche mathematische Funktion auf Ihre Daten anwenden möchten, stehen die Chancen nicht schlecht, dass sie in `scipy.special` implementiert ist. Es gibt viel zu viele Funktio-

nen, um sie alle aufzuführen, aber der nachstehende Codeabschnitt zeigt einige, die in der Statistik auftauchen könnten:

```
In [21]: from scipy import special

In [22]: # Gamma-Funktion (verallgemeinerte Fakultäts-
# funktion) und verwandte Funktionen
x = [1, 5, 10]
print("gamma(x)      =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2)    =", special.beta(x, 2))
Out[22]: gamma(x)      = [1.0000e+00 2.4000e+01 3.6288e+05]
ln|gamma(x)| = [ 0.          3.17805383 12.80182748]
beta(x, 2)    = [0.5          0.03333333 0.00909091]

In [23]: # Fehlerfunktion (gaußsches Fehlerintegral)
# komplementäre und inverse Fehlerfunktion
x = np.array([0, 0.3, 0.7, 1.0])
print("erf(x)      =", special.erf(x))
print("erfc(x)     =", special.erfc(x))
print("erfinv(x)   =", special.erfinv(x))
Out[23]: erf(x)      = [0.          0.32862676 0.67780119 0.84270079]
erfc(x) = [1.          0.67137324 0.32219881 0.15729921]
erfinv(x) = [0.          0.27246271 0.73286908          inf]
```

Sowohl in NumPy als auch in `scipy.special` steht eine immense Menge weiterer UFuncs zur Verfügung. Die Dokumentation beider Pakete ist online verfügbar, daher wird eine Internetsuche nach relevanten Informationen mit Begriffen wie »gamma function python« sicher erfolgreich sein.

UFunc-Features für Fortgeschrittene

Viele NumPy-User verwenden UFuncs, ohne sich jemals mit allen verfügbaren Features befassen zu haben. Ich werde nun einige der spezialisierten Funktionalitäten von UFuncs etwas genauer betrachten.

Ausgabe festlegen

Bei umfangreichen Berechnungen ist es manchmal nützlich, das Array angeben zu können, in dem das Ergebnis der Berechnung gespeichert werden soll. Zu diesem Zweck können Sie bei allen UFuncs das `out`-Argument der jeweiligen Funktion verwenden:

```
In [24]: x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
Out[24]: [ 0. 10. 20. 30. 40.]
```

Sie können dafür sogar Array-Views verwenden. Wir könnten beispielsweise die Ergebnisse einer Berechnung in jedem zweiten Element eines angegebenen Arrays speichern:

```
In [25]: y = np.zeros(10)
         np.power(2, x, out=y[::2])
         print(y)
Out[25]: [ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

Hätten wir stattdessen `y[::2] = 2 ** x` geschrieben, wäre ein temporäres Array erzeugt worden, um die Ergebnisse der Berechnungen von `2 ** x` aufzunehmen, gefolgt von einer zweiten Operation, die diese Werte in das `y`-Array kopiert. Bei einer so einfachen Berechnung ergibt sich dabei kaum ein Unterschied, aber bei sehr großen Arrays kann die Menge des durch die Verwendung von `out` eingesparten Arbeitsspeichers durchaus von Bedeutung sein.

Aggregationen

Bei binären UFuncs können Aggregationen direkt anhand des Objekts berechnet werden. Wenn wir zum Beispiel ein Array durch eine bestimmte Operation verkleinern (*reduzieren*) möchten, können wir dazu die `reduce`-Methode einer UFunc verwenden. Dabei wird die Operation wiederholt auf die Elemente eines Arrays angewendet, bis nur noch ein einziges Ergebnis verbleibt.

Beispielsweise liefert der Aufruf von `reduce` der UFunc `add` die Summe aller Elemente des Arrays zurück:

```
In [26]: x = np.arange(1, 6)
         np.add.reduce(x)
Out[26]: 15
```

Auf ähnliche Weise gibt der Aufruf von `reduce` der UFunc `multiply` das Produkt aller Elemente des Arrays zurück:

```
In [27]: np.multiply.reduce(x)
Out[27]: 120
```

Sollen dabei alle Zwischenergebnisse der Berechnung gespeichert werden, können wir stattdessen `accumulate` verwenden:

```
In [28]: np.add.accumulate(x)
Out[28]: array([ 1,  3,  6, 10, 15])

In [29]: np.multiply.accumulate(x)
Out[29]: array([ 1,  2,  6, 24, 120])
```

Beachten Sie, dass es für diese besonderen Fälle eigene NumPy-Funktionen gibt, um solche Ergebnisse zu erzielen (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), die wir in Kapitel 7 näher betrachten werden.

Dyadische Produkte

Jede UFunc kann mit der `outer`-Methode die Ausgabe aller Paare zweier verschiedener Eingaben berechnen (dyadisches Produkt, wird auch als äußeres Produkt bezeichnet). Auf diese Weise ist es mit nur einer Zeile Code möglich, beispielsweise eine Multiplikationstabelle zu erzeugen:


```
In [30]: x = np.arange(1, 6)
         np.multiply.outer(x, x)
Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 2,  4,  6,  8, 10],
                [ 3,  6,  9, 12, 15],
                [ 4,  8, 12, 16, 20],
                [ 5, 10, 15, 20, 25]])
```

Die Methoden `ufunc.at` und `ufunc.reduceat`, die wir in Kapitel 10 eingehender betrachten, sind hier ebenfalls sehr hilfreich.

Wir werden außerdem sehen, wie UFuncs Arrays verschiedener Größe und unterschiedlicher Shapes verknüpfen können – eine Reihe von Operationen, die als *Broadcasting* bezeichnet werden. Dieses Thema ist so wichtig, dass ihm ein eigenes Kapitel gewidmet ist (siehe Kapitel 8).

UFuncs: mehr erfahren

Weitere Informationen über UFuncs (inklusive einer vollständigen Liste der verfügbaren Funktionen) finden Sie in der Dokumentation auf den NumPy (<http://www.numpy.org>)- und SciPy (<http://www.scipy.org>)-Websites.

Außerdem können Sie jederzeit in IPython auf Informationen zugreifen, indem Sie die Pakete importieren und, wie in Kapitel 1 beschrieben, die Tab-Vervollständigung bzw. die Hilfsfunktion mit dem Fragezeichen verwenden.

Inhalt

Einleitung	13
-------------------------	-----------

Teil I: Mehr als normales Python: Jupyter

1 Der Einstieg in IPython und Jupyter	21
Die IPython-Shell starten	21
Das Jupyter Notebook starten	22
Hilfe und Dokumentation in IPython	22
Tastaturkürzel in der IPython-Shell	27
2 Erweiterte interaktive Features	31
Magische Befehle in IPython	31
Verlauf der Ein- und Ausgabe	33
IPython und Shell-Befehle	36
3 Debugging und Profiling	41
Fehler und Debugging	41
Profiling und Timing von Code	45
Weitere IPython-Ressourcen	50

Teil II: Einführung in NumPy

4 Die Datentypen in Python	55
Python-Integer sind mehr als nur ganzzahlige Werte	56
Python-Listen sind mehr als nur einfache Listen	57
Arrays feststehenden Typs in Python	59
Arrays anhand von Listen erzeugen	59
Neue Arrays erzeugen	60
NumPys Standarddatentypen	61
5 Grundlagen von NumPy-Arrays	63
Attribute von NumPy-Arrays	63
Indizierung von Arrays: Zugriff auf einzelne Elemente	64
Slicing: Teilmengen eines Arrays auswählen	65
Arrays umformen	68
Arrays verketteten und aufteilen	69

6	Berechnungen mit NumPy-Arrays: universelle Funktionen	71
	Langsame Schleifen	71
	Kurz vorgestellt: UFuncs	72
	NumPys UFuncs im Detail	73
	UFunc-Features für Fortgeschrittene	77
	UFuncs: mehr erfahren	79
7	Aggregationen: Minimum, Maximum und alles dazwischen	81
	Summieren der Werte eines Arrays	81
	Minimum und Maximum	82
	Beispiel: Durchschnittliche Größe der US-Präsidenten	84
8	Berechnungen mit Arrays: Broadcasting	87
	Kurz vorgestellt: Broadcasting	87
	Für das Broadcasting geltende Regeln	89
	Broadcasting in der Praxis	91
9	Vergleiche, Maskierungen und boolesche Logik	95
	Beispiel: Regentage zählen	95
	Vergleichsoperatoren als UFuncs	96
	Boolesche Arrays verwenden	98
	Boolesche Arrays als Maskierungen	100
	Verwendung der Schlüsselwörter »and« bzw. »or« und der Operatoren & bzw.	101
10	Fancy Indexing	103
	Fancy Indexing im Detail	103
	Kombinierte Indizierung	105
	Beispiel: Auswahl zufälliger Punkte	105
	Werte per Fancy Indexing modifizieren	107
	Beispiel: Daten gruppieren	108
11	Arrays sortieren	111
	Schnelle Sortierung in NumPy: np.sort und np.argsort	112
	Nach Zeilen und Spalten sortieren	112
	Teilsortierungen: Partitionierung	113
	Beispiel: k nächste Nachbarn	113
12	Strukturierte Daten: NumPys strukturierte Arrays	117
	Strukturierte Arrays erzeugen	118
	Erweiterte zusammengesetzte Typen	119
	Record-Arrays: strukturierte Arrays mit Pfiff	120
	Weiter mit Pandas	120

Teil III: Datenbearbeitung mit Pandas

13 Kurz vorgestellt: Pandas-Objekte	125
Das Pandas-Series-Objekt	125
Das Pandas-DataFrame-Objekt	128
Das Pandas-Index-Objekt	131
14 Daten indizieren und auswählen	133
Series-Daten auswählen	133
DataFrame-Daten auswählen	136
15 Mit Pandas-Daten arbeiten	141
UFuncs: Indexerhaltung	141
UFuncs: Indexanpassung	142
UFuncs: Operationen mit DataFrame und Series	144
16 Handhabung fehlender Daten	147
Kompromisse beim Umgang mit fehlenden Daten	147
Fehlende Daten in Pandas	148
Pandas nullfähige Datentypen	151
Mit Nullwerten arbeiten	152
17 Hierarchische Indizierung	157
Mehrfach indizierte Series	157
Methoden zum Erzeugen eines MultiIndex	161
Indizierung und Slicing eines MultiIndex	163
Multi-Indizes umordnen	166
18 Datenmengen kombinieren: concat und append	171
Verkettung von NumPy-Arrays	172
Einfache Verkettungen mit pd.concat	172
19 Datenmengen kombinieren: merge und join	177
Relationale Algebra	177
Join-Kategorien	178
Angabe der zu verknüpfenden Spalten	180
Mengenarithmetik bei Joins	183
Konflikte bei Spaltennamen: das Schlüsselwort suffixes	184
Beispiel: Daten von US-Bundesstaaten	185
20 Aggregation und Gruppierung	191
Planetendaten	191
Einfache Aggregationen in Pandas	192
GroupBy: Aufteilen, Anwenden und Kombinieren	194

21	Pivot-Tabellen	203
	Gründe für Pivot-Tabellen	203
	Pivot-Tabellen von Hand erstellen	204
	Die Syntax von Pivot-Tabellen	204
	Beispiel: Geburtenraten	207
22	Vektorisierte String-Operationen	213
	Kurz vorgestellt: String-Operationen in Pandas	213
	Liste der Pandas-String-Methoden	214
	Beispiel: Rezeptdatenbank	218
23	Zeitreihen verwenden	223
	Kalenderdaten und Zeiten in Python	223
	Zeitreihen in Pandas: Indizierung durch Zeitangaben	227
	Datenstrukturen für Zeitreihen in Pandas	227
	Gleichförmige Sequenzen: <code>pd.date_range</code>	228
	Häufigkeiten und Abstände	229
	Resampling, zeitliches Verschieben und geglättete Statistik	231
	Beispiel: Visualisierung von Fahrradzahlungen in Seattle	236
24	Leistungsstarkes Pandas: <code>eval</code> und <code>query</code>	243
	Der Zweck von <code>query</code> und <code>eval</code> : zusammengesetzte Ausdrücke	243
	Effiziente Operationen mit <code>pandas.eval</code>	244
	<code>DataFrame.eval</code> für spaltenweise Operationen	246
	Die <code>DataFrame.query</code> -Methode	248
	Performance: Wann <code>eval</code> und <code>query</code> verwendet werden sollten	248
	Weitere Ressourcen	249

Teil IV: Visualisierung mit Matplotlib

25	Allgemeine Tipps zu Matplotlib	253
	Matplotlib importieren	253
	Stil einstellen	253
	<code>show</code> oder kein <code>show</code> ? – Anzeige von Diagrammen	253
26	Einfache Liniendiagramme	261
	Anpassen des Diagramms: Linienfarben und -stile	264
	Anpassen des Diagramms: Begrenzungen	266
	Diagramme beschriften	268
	Stolpersteine in Matplotlib	270

27 Einfache Streudiagramme	271
Streudiagramme mit plt.plot erstellen	271
Streudiagramme mit plt.scatter erstellen	273
plot kontra scatter: eine Anmerkung zur Effizienz	276
Visualisierung von Messunsicherheiten	276
28 Dichtediagramme und Konturdiagramme	281
Visualisierung einer dreidimensionalen Funktion	281
Histogramme, Binnings und Dichte	285
Zweidimensionale Histogramme und Binnings	287
29 Anpassen der Legende	291
Legendenelemente festlegen	293
Legenden mit Punktgrößen	295
Mehrere Legenden	296
30 Anpassen von Farbskalen	299
Farbskala anpassen	300
Beispiel: Handgeschriebene Ziffern	304
31 Untergeordnete Diagramme	307
plt.axes: untergeordnete Diagramme von Hand erstellen	307
plt.subplot: untergeordnete Diagramme in einem Raster anordnen	309
plt.subplots: das gesamte Raster gleichzeitig ändern	310
plt.GridSpec: kompliziertere Anordnungen	312
32 Text und Beschriftungen	315
Beispiel: Auswirkungen von Feiertagen auf die Geburtenzahlen in den USA	315
Transformationen und Textposition	317
Pfeile und Beschriftungen	319
33 Achsenmarkierungen anpassen	323
Vorrangige und nachrangige Achsenmarkierungen	323
Markierungen oder Beschriftungen verbergen	325
Anzahl der Achsenmarkierungen verringern oder erhöhen	326
Formatierung der Achsenmarkierungen	328
Zusammenfassung der Formatter- und Locator-Klassen	330
34 Matplotlib anpassen: Konfigurationen und Stylesheets	333
Diagramme von Hand anpassen	333
Voreinstellungen ändern: rcParams	335

35	Dreidimensionale Diagramme in Matplotlib	341
	Dreidimensionale Punkte und Linien	342
	Dreidimensionale Konturdiagramme	343
	Drahtgitter- und Oberflächendiagramme	344
	Triangulation von Oberflächen	346
	Beispiel: Visualisierung eines Möbiusbands	348
36	Visualisierung mit Seaborn	351
	Seaborn-Diagramme	352
	Kategoriale Diagramme	356
	Beispiel: Ergebnisse eines Marathonlaufs	359
	Weiterführende Ressourcen	365
	Weitere Grafikbibliotheken für Python	366

Teil V: Machine Learning

37	Was ist Machine Learning?	369
	Kategorien des Machine Learning	369
	Qualitative Beispiele für Machine-Learning-Anwendungen	370
	Zusammenfassung	379
38	Kurz vorgestellt: Scikit-Learn	381
	Datenrepräsentierung in Scikit-Learn	381
	Die Estimator-API	384
	Anwendung: Handgeschriebene Ziffern untersuchen	392
	Zusammenfassung	397
39	Hyperparameter und Modellvalidierung	399
	Überlegungen zum Thema Modellvalidierung	399
	Auswahl des besten Modells	403
	Lernkurven	410
	Validierung in der Praxis: Rastersuche	414
	Zusammenfassung	415
40	Feature Engineering	417
	Kategoriale Features	417
	Texte als Features	418
	Bilder als Features	420
	Abgeleitete Features	420
	Vervollständigung fehlender Daten	422
	Feature-Pipelines	423

41 Ausführlich: Naive Bayes-Klassifikation	425
Bayes-Klassifikation	425
Gaußsche naive Bayes-Klassifikation	426
Multinomiale naive Bayes-Klassifikation	429
Einsatzgebiete für naive Bayes-Klassifikation	432
42 Ausführlich: Lineare Regression	435
Einfache lineare Regression	435
Regression der Basisfunktion	437
Regularisierung	441
Beispiel: Vorhersage des Fahrradverkehrs	445
43 Ausführlich: Support Vector Machines	451
Gründe für Support Vector Machines	451
Support Vector Machines: Maximierung des Randbereichs	453
Beispiel: Gesichtserkennung	461
Zusammenfassung	465
44 Ausführlich: Entscheidungsbäume und Random Forests	467
Gründe für Random Forests: Entscheidungsbäume	467
Estimator-Ensembles: Random Forests	471
Random-Forest-Regression	473
Beispiel: Random Forest zur Klassifikation handgeschriebener Ziffern	475
Zusammenfassung	477
45 Ausführlich: Hauptkomponentenanalyse	479
Hauptkomponentenanalyse: ein Überblick	479
Hauptkomponentenanalyse als Rauschfilter	487
Beispiel: Eigengesichter	489
Zusammenfassung	492
46 Ausführlich: Manifold Learning	493
Manifold Learning: »HELLO«	494
Multidimensionale Skalierung	495
Nichtlineare Mannigfaltigkeiten: lokal lineare Einbettung	500
Überlegungen zum Thema Manifold-Methoden	502
Beispiel: Isomap und Gesichter	503
Beispiel: Visualisierung der Strukturen in Zifferndaten	507
47 Ausführlich: k-Means-Clustering	511
Kurz vorgestellt: der k-Means-Algorithmus	511
Expectation-Maximization	513
Beispiele	518

48 Ausführlich: Gaußsche Mixture-Modelle	525
Gründe für GMM: Schwächen von k-Means	525
EM-Verallgemeinerung: gaußsche Mixture-Modelle	528
Wahl des Kovarianztyps	532
GMM als Dichteschätzung	532
Beispiel: GMM zum Erzeugen neuer Daten verwenden	536
49 Ausführlich: Kerndichteschätzung	539
Gründe für Kerndichteschätzung: Histogramme	539
Kerndichteschätzung in der Praxis	543
Auswahl der Bandbreite durch Kreuzvalidierung	544
Beispiel: Nicht ganz so naive Bayes-Klassifikation	545
50 Anwendung: Eine Gesichtserkennungspipeline	551
HOG-Features	552
HOG in Aktion: eine einfache Gesichtserkennung	553
Vorbehalte und Verbesserungen	557
Weitere Machine-Learning-Ressourcen	559
Index	561