



Jake  
VanderPlas

# Data Science mit Python

Das **Handbuch** für den Einsatz von  
IPython, Jupyter, NumPy, Pandas, Matplotlib, Scikit-Learn

# Inhaltsverzeichnis

	<b>Einleitung</b> .....	13
	<b>Über den Autor</b> .....	18
<b>1</b>	<b>Mehr als normales Python: IPython</b> .....	19
1.1	Shell oder Notebook? .....	19
1.1.1	Die IPython-Shell starten .....	20
1.1.2	Das Jupyter-Notebook starten .....	20
1.2	Hilfe und Dokumentation in IPython .....	21
1.2.1	Mit ? auf die Dokumentation zugreifen .....	22
1.2.2	Mit ?? auf den Quellcode zugreifen .....	23
1.2.3	Module mit der Tab-Vervollständigung erkunden .....	24
1.3	Tastaturkürzel in der IPython-Shell .....	26
1.3.1	Tastaturkürzel zum Navigieren .....	27
1.3.2	Tastaturkürzel bei der Texteingabe .....	27
1.3.3	Tastaturkürzel für den Befehlsverlauf .....	28
1.3.4	Sonstige Tastaturkürzel .....	29
1.4	Magische Befehle in IPython .....	29
1.4.1	Einfügen von Codeblöcken mit %paste und %cpaste .....	29
1.4.2	Externen Code ausführen mit %run .....	31
1.4.3	Messung der Ausführungszeit von Code mit %timeit .....	31
1.4.4	Hilfe für die magischen Funktionen anzeigen mit ?, %magic und %lsmagic .....	32
1.5	Verlauf der Ein- und Ausgabe .....	32
1.5.1	Die IPython-Objekte In und Out .....	33
1.5.2	Der Unterstrich als Abkürzung und vorhergehende Ausgaben .....	34
1.5.3	Ausgaben unterdrücken .....	34
1.5.4	Weitere ähnliche magische Befehle .....	35
1.6	IPython und Shell-Befehle .....	35
1.6.1	Kurz vorgestellt: die Shell .....	36
1.6.2	Shell-Befehle in IPython .....	37
1.6.3	Werte mit der Shell austauschen .....	37
1.7	Magische Befehle für die Shell .....	38
1.8	Fehler und Debugging .....	39
1.8.1	Exceptions handhaben: %xmode .....	39
1.8.2	Debugging: Wenn das Lesen von Tracebacks nicht ausreicht .....	41
1.9	Profiling und Timing von Code .....	44
1.9.1	Timing von Codeschnipseln: %timeit und %time .....	45
1.9.2	Profiling kompletter Skripte: %prun .....	46

1.9.3	Zeilenweises Profiling mit %lprun . . . . .	47
1.9.4	Profiling des Speicherbedarfs: %memit und %mprun . . . . .	48
1.10	Weitere IPython-Ressourcen . . . . .	50
1.10.1	Quellen im Internet . . . . .	50
1.10.2	Bücher . . . . .	50
2	<b>Einführung in NumPy</b> . . . . .	51
2.1	Die Datentypen in Python. . . . .	52
2.1.1	Python-Integers sind mehr als nur ganzzahlige Werte . . . . .	53
2.1.2	Python-Listen sind mehr als nur einfache Listen. . . . .	54
2.1.3	Arrays feststehenden Typs in Python . . . . .	56
2.1.4	Arrays anhand von Listen erzeugen . . . . .	56
2.1.5	Neue Arrays erzeugen . . . . .	57
2.1.6	NumPys Standarddatentypen . . . . .	58
2.2	Grundlagen von NumPy-Arrays. . . . .	59
2.2.1	Attribute von NumPy-Arrays . . . . .	60
2.2.2	Indizierung von Arrays: Zugriff auf einzelne Elemente . . . . .	61
2.2.3	Slicing: Teilmengen eines Arrays auswählen. . . . .	62
2.2.4	Arrays umformen . . . . .	65
2.2.5	Arrays verketten und aufteilen. . . . .	66
2.3	Berechnungen mit NumPy-Arrays: universelle Funktionen . . . . .	68
2.3.1	Langsame Schleifen . . . . .	68
2.3.2	Kurz vorgestellt: UFuncs . . . . .	70
2.3.3	NumPys UFuncs im Detail . . . . .	70
2.3.4	UFunc-Features für Fortgeschrittene . . . . .	75
2.3.5	UFuncs: mehr erfahren . . . . .	77
2.4	Aggregationen: Minimum, Maximum und alles dazwischen . . . . .	77
2.4.1	Summieren der Werte eines Arrays . . . . .	77
2.4.2	Minimum und Maximum . . . . .	78
2.4.3	Beispiel: Durchschnittliche Größe der US-Präsidenten . . . . .	80
2.5	Berechnungen mit Arrays: Broadcasting. . . . .	82
2.5.1	Kurz vorgestellt: Broadcasting . . . . .	82
2.5.2	Für das Broadcasting geltende Regeln . . . . .	84
2.5.3	Broadcasting in der Praxis . . . . .	87
2.6	Vergleiche, Maskierungen und boolesche Logik . . . . .	88
2.6.1	Beispiel: Regentage zählen. . . . .	89
2.6.2	Vergleichsoperatoren als UFuncs . . . . .	90
2.6.3	Boolesche Arrays verwenden . . . . .	91
2.6.4	Boolesche Arrays als Maskierungen . . . . .	94
2.7	Fancy Indexing . . . . .	97
2.7.1	Fancy Indexing im Detail . . . . .	97
2.7.2	Kombinierte Indizierung . . . . .	98
2.7.3	Beispiel: Auswahl zufälliger Punkte . . . . .	99
2.7.4	Werte per Fancy Indexing modifizieren . . . . .	101
2.7.5	Beispiel: Daten gruppieren . . . . .	102

2.8	Arrays sortieren .....	104
2.8.1	Schnelle Sortierung in NumPy: np.sort und np.argsort.....	105
2.8.2	Teilsortierungen: Partitionierung .....	107
2.8.3	Beispiel: k nächste Nachbarn .....	107
2.9	Strukturierte Daten: NumPys strukturierte Arrays .....	112
2.9.1	Strukturierte Arrays erzeugen .....	113
2.9.2	Erweiterte zusammengesetzte Typen .....	114
2.9.3	Record-Arrays: strukturierte Arrays mit Pfiff .....	115
2.9.4	Weiter mit Pandas .....	115
<b>3</b>	<b>Datenbearbeitung mit Pandas .....</b>	<b>117</b>
3.1	Pandas installieren und verwenden .....	117
3.2	Kurz vorgestellt: Pandas-Objekte .....	118
3.2.1	Das Pandas-Series-Objekt .....	118
3.2.2	Das Pandas-DataFrame-Objekt .....	122
3.2.3	Das Pandas-Index-Objekt .....	126
3.3	Daten indizieren und auswählen .....	127
3.3.1	Series-Daten auswählen .....	127
3.3.2	DataFrame-Daten auswählen .....	131
3.4	Mit Pandas-Daten arbeiten .....	135
3.4.1	UFuncs: Indexerhaltung .....	136
3.4.2	UFuncs: Indexanpassung .....	137
3.4.3	UFuncs: Operationen mit DataFrame und Series .....	139
3.5	Handhabung fehlender Daten .....	140
3.5.1	Überlegungen zu fehlenden Daten .....	141
3.5.2	Fehlende Daten in Pandas .....	141
3.5.3	Mit null-Werten arbeiten .....	145
3.6	Hierarchische Indizierung .....	149
3.6.1	Mehrfach indizierte Series .....	149
3.6.2	Methoden zum Erzeugen eines MultiIndex .....	153
3.6.3	Indizierung und Slicing eines MultiIndex .....	156
3.6.4	Multi-Indizes umordnen .....	159
3.6.5	Datenaggregationen mit Multi-Indizes .....	162
3.7	Datenmengen kombinieren: concat und append .....	164
3.7.1	Verkettung von NumPy-Arrays .....	165
3.7.2	Einfache Verkettungen mit pd.concat .....	165
3.8	Datenmengen kombinieren: Merge und Join .....	169
3.8.1	Relationale Algebra .....	170
3.8.2	Join-Kategorien .....	170
3.8.3	Angabe der zu verknüpfenden Spalten .....	173
3.8.4	Mengenarithmetik bei Joins .....	176
3.8.5	Konflikte bei Spaltennamen: das Schlüsselwort suffixes .....	177
3.8.6	Beispiel: Daten von US-Bundesstaaten .....	178
3.9	Aggregation und Gruppierung .....	183
3.9.1	Planetendaten .....	183

3.9.2	Einfache Aggregationen in Pandas .....	184
3.9.3	GroupBy: Aufteilen, Anwenden und Kombinieren .....	186
3.10	Pivot-Tabellen .....	195
3.10.1	Gründe für Pivot-Tabellen .....	195
3.10.2	Pivot-Tabellen von Hand erstellen. ....	196
3.10.3	Die Syntax von Pivot-Tabellen .....	197
3.10.4	Beispiel: Geburtenraten .....	199
3.11	Vektorisierte String-Operationen .....	204
3.11.1	Kurz vorgestellt: String-Operationen in Pandas .....	204
3.11.2	Liste der Pandas-Stringmethoden .....	206
3.11.3	Beispiel: Rezeptdatenbank .....	211
3.12	Zeitreihen verwenden .....	215
3.12.1	Kalenderdaten und Zeiten in Python .....	215
3.12.2	Zeitreihen in Pandas: Indizierung durch Zeitangaben .....	219
3.12.3	Datenstrukturen für Zeitreihen in Pandas .....	220
3.12.4	Häufigkeiten und Abstände. ....	222
3.12.5	Resampling, zeitliches Verschieben und geglättete Statistik .....	224
3.12.6	Mehr erfahren. ....	229
3.12.7	Beispiel: Visualisierung von Fahrradzahlungen in Seattle .....	229
3.13	Leistungsstarkes Pandas: eval() und query() .....	236
3.13.1	Der Zweck von query() und eval(): zusammengesetzte Ausdrücke .....	236
3.13.2	Effiziente Operationen mit pandas.eval() .....	237
3.13.3	DataFrame.eval() für spaltenweise Operationen .....	239
3.13.4	Die DataFrame.query()-Methode .....	241
3.13.5	Performance: Verwendung von eval() und query() .....	242
3.14	Weitere Ressourcen. ....	242
4	<b>Visualisierung mit Matplotlib. ....</b>	<b>245</b>
4.1	Allgemeine Tipps zu Matplotlib. ....	246
4.1.1	Matplotlib importieren .....	246
4.1.2	Stil einstellen. ....	246
4.1.3	show() oder kein show()? – Anzeige von Diagrammen .....	246
4.1.4	Grafiken als Datei speichern .....	248
4.2	Zwei Seiten derselben Medaille .....	250
4.3	Einfache Liniendiagramme .....	251
4.3.1	Anpassen des Diagramms: Linienfarben und -stile .....	254
4.3.2	Anpassen des Diagramms: Begrenzungen. ....	256
4.3.3	Diagramme beschriften .....	258
4.4	Einfache Streudiagramme .....	260
4.4.1	Streudiagramme mit plt.plot() erstellen .....	260
4.4.2	Streudiagramme mit plt.scatter() erstellen .....	263
4.4.3	plot kontra scatter: eine Anmerkung zur Effizienz .....	265
4.5	Visualisierung von Fehlern. ....	265
4.5.1	Einfache Fehlerbalken .....	265
4.5.2	Stetige Fehler .....	267

4.6	Dichtediagramme und Konturdiagramme .....	268
4.6.1	Visualisierung einer dreidimensionalen Funktion.....	268
4.7	Histogramme, Binnings und Dichte .....	272
4.7.1	Zweidimensionale Histogramme und Binnings .....	274
4.8	Anpassen der Legende.....	277
4.8.1	Legendenelemente festlegen .....	279
4.8.2	Legenden mit Punktgrößen .....	280
4.8.3	Mehrere Legenden .....	282
4.9	Anpassen von Farbskalen .....	283
4.9.1	Farbskala anpassen .....	284
4.9.2	Beispiel: Handgeschriebene Ziffern .....	288
4.10	Untergeordnete Diagramme.....	290
4.10.1	plt.axes: Untergeordnete Diagramme von Hand erstellen.....	290
4.10.2	plt.subplot: Untergeordnete Diagramme in einem Raster anordnen .....	292
4.10.3	plt.subplots: Das gesamte Raster gleichzeitig ändern .....	293
4.10.4	plt.GridSpec: Kompliziertere Anordnungen .....	294
4.11	Text und Beschriftungen .....	296
4.11.1	Beispiel: Auswirkungen von Feiertagen auf die Geburtenzahlen in den USA .....	296
4.11.2	Transformationen und Textposition .....	299
4.11.3	Pfeile und Beschriftungen .....	300
4.12	Achsenmarkierungen anpassen .....	303
4.12.1	Vorrangige und nachrangige Achsenmarkierungen .....	304
4.12.2	Markierungen oder Beschriftungen verbergen.....	305
4.12.3	Anzahl der Achsenmarkierungen verringern oder erhöhen .....	306
4.12.4	Formatierung der Achsenmarkierungen.....	307
4.12.5	Zusammenfassung der Formatter- und Locator-Klassen.....	310
4.13	Matplotlib anpassen: Konfigurationen und Stylesheets .....	311
4.13.1	Diagramme von Hand anpassen .....	311
4.13.2	Voreinstellungen ändern: rcParams .....	312
4.13.3	Stylesheets .....	314
4.14	Dreidimensionale Diagramme in Matplotlib.....	318
4.14.1	Dreidimensionale Punkte und Linien .....	319
4.14.2	Dreidimensionale Konturdiagramme .....	320
4.14.3	Drahtgitter- und Oberflächendiagramme .....	322
4.14.4	Triangulation von Oberflächen .....	323
4.15	Basemap: geografische Daten verwenden .....	326
4.15.1	Kartenprojektionen .....	328
4.15.2	Zeichnen eines Kartenhintergrunds .....	332
4.15.3	Daten auf einer Karte anzeigen .....	334
4.15.4	Beispiel: Kalifornische Städte.....	335
4.15.5	Beispiel: Oberflächentemperaturen.....	337
4.16	Visualisierung mit Seaborn.....	339
4.16.1	Seaborn kontra Matplotlib .....	339
4.16.2	Seaborn-Diagramme.....	341

4.17	Weitere Ressourcen . . . . .	357
4.17.1	Matplotlib . . . . .	357
4.17.2	Weitere Grafikbibliotheken für Python. . . . .	357
5	<b>Machine Learning</b> . . . . .	359
5.1	Was ist Machine Learning? . . . . .	360
5.1.1	Kategorien des Machine Learnings . . . . .	360
5.1.2	Qualitative Beispiele für Machine-Learning-Anwendungen . . . . .	361
5.1.3	Zusammenfassung. . . . .	369
5.2	Kurz vorgestellt: Scikit-Learn . . . . .	369
5.2.1	Datenrepräsentierung in Scikit-Learn . . . . .	370
5.2.2	Scikit-Learns Schätzer-API. . . . .	372
5.2.3	Anwendung: Handgeschriebene Ziffern untersuchen . . . . .	380
5.2.4	Zusammenfassung. . . . .	385
5.3	Hyperparameter und Modellvalidierung . . . . .	385
5.3.1	Überlegungen zum Thema Modellvalidierung . . . . .	385
5.3.2	Auswahl des besten Modells . . . . .	389
5.3.3	Lernkurven . . . . .	396
5.3.4	Validierung in der Praxis: Rastersuche . . . . .	399
5.3.5	Zusammenfassung . . . . .	401
5.4	Merkmalerstellung . . . . .	401
5.4.1	Kategoriale Merkmale. . . . .	402
5.4.2	Texte als Merkmale. . . . .	403
5.4.3	Bilder als Merkmale . . . . .	404
5.4.4	Abgeleitete Merkmale. . . . .	405
5.4.5	Vervollständigung fehlender Daten. . . . .	407
5.4.6	Pipelines mit Merkmalen. . . . .	408
5.5	Ausführlich: Naive Bayes-Klassifikation . . . . .	409
5.5.1	Bayes-Klassifikation . . . . .	409
5.5.2	Gauß'sche naive Bayes-Klassifikation . . . . .	410
5.5.3	Multinomiale naive Bayes-Klassifikation . . . . .	413
5.5.4	Einsatzgebiete für naive Bayes-Klassifikation. . . . .	416
5.6	Ausführlich: Lineare Regression . . . . .	417
5.6.1	Einfache lineare Regression. . . . .	417
5.6.2	Regression der Basisfunktion . . . . .	419
5.6.3	Regularisierung. . . . .	423
5.6.4	Beispiel: Vorhersage des Fahrradverkehrs . . . . .	427
5.7	Ausführlich: Support Vector Machines . . . . .	432
5.7.1	Gründe für Support Vector Machines. . . . .	433
5.7.2	Support Vector Machines: Maximierung des Randbereichs . . . . .	434
5.7.3	Beispiel: Gesichtserkennung . . . . .	443
5.7.4	Zusammenfassung Support Vector Machines . . . . .	447
5.8	Ausführlich: Entscheidungsbäume und Random Forests . . . . .	448
5.8.1	Gründe für Random Forests . . . . .	448
5.8.2	Schätzerensembles: Random Forests . . . . .	454
5.8.3	Random-Forest-Regression . . . . .	455

5.8.4	Beispiel: Random Forest zur Klassifikation handgeschriebener Ziffern . . . . .	457
5.8.5	Zusammenfassung Random Forests . . . . .	459
5.9	Ausführlich: Hauptkomponentenanalyse . . . . .	460
5.9.1	Kurz vorgestellt: Hauptkomponentenanalyse . . . . .	460
5.9.2	Hauptkomponentenanalyse als Rauschfilter . . . . .	467
5.9.3	Beispiel: Eigengesichter . . . . .	469
5.9.4	Zusammenfassung Hauptkomponentenanalyse . . . . .	472
5.10	Ausführlich: Manifold Learning . . . . .	473
5.10.1	Manifold Learning: »HELLO« . . . . .	473
5.10.2	Multidimensionale Skalierung (MDS) . . . . .	475
5.10.3	MDS als Manifold Learning . . . . .	477
5.10.4	Nichtlineare Einbettungen: Wenn MDS nicht funktioniert . . . . .	479
5.10.5	Nichtlineare Mannigfaltigkeiten: lokal lineare Einbettung . . . . .	480
5.10.6	Überlegungen zum Thema Manifold-Methoden . . . . .	482
5.10.7	Beispiel: Isomap und Gesichter . . . . .	483
5.10.8	Beispiel: Visualisierung der Strukturen in Zifferndaten . . . . .	487
5.11	Ausführlich: k-Means-Clustering . . . . .	490
5.11.1	Kurz vorgestellt: der k-Means-Algorithmus . . . . .	490
5.11.2	k-Means-Algorithmus: Expectation-Maximization . . . . .	492
5.11.3	Beispiele . . . . .	497
5.12	Ausführlich: Gauß'sche Mixture-Modelle . . . . .	503
5.12.1	Gründe für GMM: Schwächen von k-Means . . . . .	503
5.12.2	EM-Verallgemeinerung: Gauß'sche Mixture-Modelle . . . . .	507
5.12.3	GMM als Dichteschätzung . . . . .	511
5.12.4	Beispiel: GMM zum Erzeugen neuer Daten verwenden . . . . .	515
5.13	Ausführlich: Kerndichteschätzung . . . . .	518
5.13.1	Gründe für Kerndichteschätzung: Histogramme . . . . .	518
5.13.2	Kerndichteschätzung in der Praxis . . . . .	522
5.13.3	Beispiel: Kerndichteschätzung auf Kugeloberflächen . . . . .	524
5.13.4	Beispiel: Nicht ganz so naive Bayes-Klassifikation . . . . .	527
5.14	Anwendung: Eine Gesichtserkennungs-Pipeline . . . . .	532
5.14.1	HOG-Merkmale . . . . .	533
5.14.2	HOG in Aktion: eine einfache Gesichtserkennung . . . . .	534
5.14.3	Vorbehalte und Verbesserungen . . . . .	539
5.15	Weitere Machine-Learning-Ressourcen . . . . .	541
5.15.1	Machine Learning in Python . . . . .	541
5.15.2	Machine Learning im Allgemeinen . . . . .	541
	<b>Stichwortverzeichnis . . . . .</b>	<b>543</b>

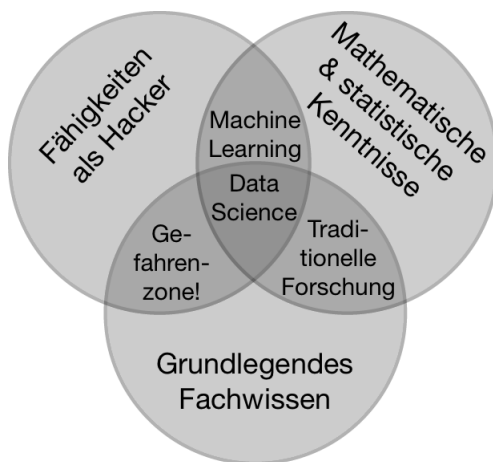


# Einleitung

## Was ist Data Science?

In diesem Buch geht es darum, Data Science mithilfe von Python zu betreiben, daher stellt sich unmittelbar die Frage: Was ist *Data Science* überhaupt? Das genau zu definieren, erweist sich als überraschend schwierig, insbesondere in Anbetracht der Tatsache, wie geläufig dieser Begriff inzwischen geworden ist. Von lautstarken Kritikern wird dieser Begriff mitunter als eine überflüssige Bezeichnung abgetan (denn letzten Endes kommt keine Wissenschaft ohne Daten aus) oder für ein leeres Schlagwort gehalten, das lediglich dazu dient, Lebensläufe aufzupolieren, um die Aufmerksamkeit übereifriger Personalverantwortlicher zu erlangen.

Meiner Ansicht nach übersehen diese Kritiker dabei einen wichtigen Punkt. Trotz des mit dem Begriff einhergehenden Hypes ist Data Science wohl die beste Beschreibung für fachübergreifende Fähigkeiten, die in vielen industriellen und akademischen Anwendungsbereichen immer wichtiger werden. Entscheidend ist hier die Interdisziplinarität: Ich halte Drew Conways Venn-Diagramm, das er im September 2010 erstmals in seinem Blog veröffentlichte, für die beste Definition von Data Science (siehe Abbildung 0.1).



**Abb. 0.1:** Das Venn-Diagramm zur Data Science von Drew Conway

Zwar sind einige der Bezeichnungen für die Schnittmengen nicht ganz ernst gemeint, aber dennoch erfasst dieses Diagramm das Wesentliche dessen, was gemeint ist, wenn man von »Data Science« spricht: Es handelt sich um ein grundlegend interdisziplinäres Thema. Data Science umfasst drei verschiedene und sich überschneidende Bereiche: die Aufgaben

eines Statistikers, der (immer größer werdende) Datenmengen modellieren und zusammenfassen kann, die Arbeit des Informatikers, der Algorithmen für die effiziente Speicherung, Verarbeitung und Visualisierung dieser Daten entwerfen kann, und das erforderliche Fachwissen – das wir uns als das »klassisch« Erlernte eines Fachgebiets vorstellen können –, um sowohl die angemessenen Fragen zu stellen als auch die Antworten im richtigen Kontext zu bewerten.

Das habe ich im Sinn, wenn ich Sie dazu auffordere, Data Science nicht als ein neu zu erlernendes Fachwissensgebiet zu begreifen, sondern als neue Fähigkeiten, die Sie im Rahmen Ihres vorhandenen Fachwissens anwenden können. Ob Sie über Wahlergebnisse berichten, Aktienrenditen vorhersagen, Mausclicks auf Onlinewerbung optimieren, Mikroorganismen auf Mikroskopbildern identifizieren, nach neuen Arten astronomischer Objekte suchen oder mit irgendwelchen anderen Daten arbeiten: Ziel dieses Buchs ist es, Ihnen die Fähigkeit zu vermitteln, neuartige Fragen über das von Ihnen gewählte Fachgebiet zu stellen und diese zu beantworten.

## An wen richtet sich das Buch?

Sowohl in meinen Vorlesungen an der Universität Washington als auch auf verschiedenen technisch orientierten Konferenzen und Treffen wird mir am häufigsten diese Frage gestellt: »Wie kann man Python am besten erlernen?« Bei den Fragenden handelt es sich im Allgemeinen um technisch interessierte Studenten, Entwickler oder Forscher, die oftmals schon über umfangreiche Erfahrung mit dem Schreiben von Code und der Verwendung von rechnergestützten und numerischen Tools verfügen. Die meisten dieser Personen möchten Python erlernen, um die Programmiersprache als Tool für datenintensive und rechnergestützte wissenschaftliche Aufgaben zu nutzen. Für diese Zielgruppe ist eine Vielzahl von Lernvideos, Blogbeiträgen und Tutorials online verfügbar. Allerdings frustriert mich bereits seit geraumer Zeit, dass es auf obige Frage keine wirklich eindeutige und gute Antwort gibt – und das war der Anlass für dieses Buch.

Das Buch ist nicht als Einführung in Python oder die Programmierung im Allgemeinen gedacht. Ich setze voraus, dass der Leser mit der Programmiersprache Python vertraut ist. Dazu gehören das Definieren von Funktionen, die Zuweisung von Variablen, das Aufrufen der Methoden von Objekten, die Steuerung des Programmablaufs und weitere grundlegende Aufgaben. Das Buch soll vielmehr Python-Usern dabei helfen, die zum Betreiben von Data Science verfügbaren Pakete zu nutzen – Bibliotheken wie IPython, NumPy, Pandas, Matplotlib, Scikit-Learn und ähnliche Tools –, um Daten effektiv zu speichern, zu handhaben und Einblick in diese Daten zu gewinnen.

## Warum Python?

Python hat sich in den letzten Jahrzehnten zu einem erstklassigen Tool für wissenschaftliche Berechnungen entwickelt, insbesondere auch für die Analyse und Visualisierung großer Datensätze. Die ersten Anhänger der Programmiersprache Python dürfte das ein wenig überraschen: Beim eigentlichen Design der Sprache wurde weder der Datenanalyse noch wissenschaftlichen Berechnungen besondere Beachtung geschenkt.

Dass sich Python für die Data Science als so nützlich erweist, ist vor allem dem großen und aktiven Ökosystem der Programmpakete von Drittherstellern zu verdanken: Da gibt es

NumPy für die Handhabung gleichartiger Array-basierter Daten, Pandas für die Verarbeitung verschiedenartiger und gekennzeichneter Daten, SciPy für gängige wissenschaftliche Berechnungen, Matplotlib für druckreife Visualisierungen, IPython für die interaktive Ausführung und zum Teilen von Code, Scikit-Learn für Machine Learning sowie viele weitere Tools, die später im Buch noch Erwähnung finden.

Falls Sie auf der Suche nach einer Einführung in die Programmiersprache Python sind, empfehle ich das dieses Buch ergänzende Projekt *A Whirlwind Tour of the Python Language* (<https://github.com/jakevdp/WhirlwindTourOfPython>). Hierbei handelt es sich um eine Tour durch die wesentlichen Features der Sprache Python, die sich an Data Scientists richtet, die bereits mit anderen Programmiersprachen vertraut sind.

## Python 2 kontra Python 3

In diesem Buch wird die Syntax von Python 3 verwendet, die Spracherweiterungen enthält, die mit Python 2 inkompatibel sind. Zwar wurde Python 3 schon 2008 veröffentlicht, allerdings verbreitete sich diese Version insbesondere in den Communitys von Wissenschaft und Webentwicklung nur langsam. Das lag vor allem daran, dass die Anpassung vieler wichtiger Pakete von Drittherstellern an die neue Sprachversion Zeit benötigte. Seit Anfang 2014 gibt es jedoch stabile Versionen der für die Data Science wichtigsten Tools, die sowohl mit Python 2 als auch mit Python 3 kompatibel sind, daher wird in diesem Buch die neuere Syntax von Python 3 genutzt. Allerdings funktionieren die meisten Codeabschnitte dieses Buchs ohne Änderungen auch in Python 2. Wenn Py2-inkompatible Syntax verwendet wird, weise ich ausdrücklich darauf hin.

## Inhaltsübersicht

Alle Kapitel in diesem Buch konzentrieren sich auf ein bestimmtes Paket oder Tool, das für die mit Python betriebene Data Science von grundlegender Bedeutung ist.

### *IPython und Jupyter (Kapitel 1)*

Diese Pakete bieten eine Umgebung für Berechnungen, die von vielen Data Scientists genutzt wird, die Python einsetzen.

### *NumPy (Kapitel 2)*

Diese Bibliothek stellt das `ndarray`-Objekt zur Verfügung, das ein effizientes Speichern und die Handhabung dicht gepackter Datenarrays in Python ermöglicht.

### *Pandas (Kapitel 3)*

Diese Bibliothek verfügt über das `DataFrame`-Objekt, das ein effizientes Speichern und die Handhabung gekennzeichneter bzw. spaltenorientierter Daten in Python gestattet.

### *Matplotlib (Kapitel 4)*

Diese Bibliothek ermöglicht flexible und vielfältige Visualisierungen von Daten in Python.

### *Scikit-Learn (Kapitel 5)*

Diese Bibliothek stellt eine effiziente Implementierung der wichtigsten und gebräuchlichsten Machine-Learning-Algorithmen zur Verfügung.

Natürlich umfasst die PyData-Welt viel mehr als diese fünf Pakete – und sie wächst mit jedem Tag weiter. Ich werde mich im Folgenden daher bemühen, Hinweise auf andere interessante Projekte, Bestrebungen und Pakete zu geben, die die Grenzen des mit Python Machbaren erweitern. Dessen ungeachtet sind die fünf genannten Pakete derzeit für viele der mit Python möglichen Aufgaben der Data Science von grundlegender Bedeutung, und ich erwarte, dass sie wichtig bleiben, auch wenn das sie umgebende Ökosystem weiterhin wächst.

## Verwendung der Codebeispiele

Unter <https://github.com/jakevdp/PythonDataScienceHandbook> steht ergänzendes Material (Codebeispiele, Abbildungen usw.) zum Herunterladen zur Verfügung. Das Buch soll Ihnen helfen, Ihre Arbeit zu erledigen. Den im Buch aufgeführten Code können Sie generell in Ihren eigenen Programmen und der Dokumentation verwenden. Sie brauchen uns nicht um Erlaubnis zu fragen, solange Sie nicht erhebliche Teile des Codes nutzen. Wenn Sie beispielsweise ein Programm schreiben, das einige der im Buch aufgeführten Codeschnipsel verwendet, benötigen Sie dafür keine Erlaubnis. Der Verkauf oder Vertrieb einer CD-ROM, die Codebeispiele aus dem Buch enthält, bedarf hingegen einer Genehmigung. Das Beantworten von Fragen durch Verwendung von Zitaten oder Beispielcode aus diesem Buch muss nicht extra genehmigt werden. Die Verwendung erheblicher Teile des Beispielcodes in der Dokumentation Ihres eigenen Produkts erfordert jedoch eine Genehmigung.

Wir freuen uns über Quellennennungen, machen sie jedoch nicht zur Bedingung. Üblich ist die Nennung von Titel, Autor(en), Verlag, Erscheinungsjahr und ISBN, also beispielsweise »Data Science mit Python« von Jake VanderPlas (mitp Verlag 2017), ISBN 978-3-95845-695-2.

## Installation der Software

Die Installation von Python und der für wissenschaftliche Berechnungen erforderlichen Bibliotheken ist unkompliziert. In diesem Abschnitt finden Sie einige Überlegungen, denen Sie bei der Einrichtung Ihres Computers Beachtung schenken sollten.

Es gibt zwar verschiedene Möglichkeiten, Python zu installieren, allerdings empfehle ich zum Betreiben von Data Science die Anaconda-Distribution, die unter Windows, Linux und macOS auf ähnliche Weise funktioniert. Es gibt zwei Varianten der Anaconda-Distribution:

- Miniconda (<http://conda.pydata.org/miniconda.html>) besteht aus dem eigentlichen Python-Interpreter und einem Kommandozeilenprogramm namens *conda*, das als plattformübergreifender Paketmanager für Python-Pakete fungiert. Das Programm arbeitet in ähnlicher Weise wie die Tools *apt* oder *yum*, die Linux-Usern bekannt sein dürften.
- Anaconda (<https://www.continuum.io/downloads>) enthält sowohl Python als auch *conda* und darüber hinaus eine Reihe vorinstallierter Pakete, die für wissenschaftliche Berechnungen konzipiert sind. Aufgrund der Größe dieser Pakete müssen Sie davon ausgehen, dass die Installation mehrere Gigabyte Speicherplatz auf der Festplatte belegt.

Alle in Anaconda enthaltenen Pakete können auch nachträglich der Miniconda-Installation hinzugefügt werden. Daher empfehle ich, mit Miniconda anzufangen.

Laden Sie zunächst das Miniconda-Paket herunter und installieren Sie es. Vergewissern Sie sich, dass Sie eine Version auswählen, die Python 3 enthält. Installieren Sie dann die in diesem Buch verwendeten Pakete:

```
[~]$ conda install numpy pandas scikit-learn matplotlib seaborn ipython-notebook
```

Wir werden im gesamten Buch noch weitere, spezialisiere Tools verwenden, die zum wissenschaftlich orientierten Ökosystem in Python gehören. Für gewöhnlich ist zur Installation lediglich eine Eingabe wie `conda install paketname` erforderlich. Weitere Informationen über conda, beispielsweise über das Erstellen und Verwenden von conda-Umgebungen (die ich nur nachdrücklich empfehlen kann), finden Sie in der Onlinedokumentation (<http://conda.pydata.org/docs/>).

## Konventionen dieses Buchs

In diesem Buch gelten die folgenden typografischen Konventionen:

### *Kursive Schrift*

Kennzeichnet neue Begriffe, Dateinamen und Dateinamenserweiterungen.

### Nicht proportionale Schrift

Wird für URLs, Programmlistings und im Fließtext verwendet, um Programmbestandteile wie Variablen- oder Funktionsbezeichnungen, Datenbanken, Datentypen Umgebungsvariablen, Anweisungen und Schlüsselwörter zu kennzeichnen.

### **Fette nicht proportionale Schrift**

Kommandos oder sonstiger Text, der vom User buchstabengetreu eingegeben werden soll.

### *Kursive nicht proportionale Schrift*

Text, der durch eigene Werte oder durch kontextabhängige Werte zu ersetzen ist.

## Die Webseite zum Buch

Der Verlag hält auf seiner Website weiteres Material zum Buch bereit. Unter <http://www.mitp.de/695> können Sie sich Beispielcode herunterladen.

# Mehr als normales Python: IPython

Für Python stehen viele verschiedene Entwicklungsumgebungen zur Verfügung, und häufig werde ich gefragt, welche ich für meine eigenen Arbeiten verwende. Einige Leute überrascht die Antwort: Meine bevorzugte Entwicklungsumgebung ist IPython (<http://ipython.org/>) in Kombination mit einem Texteditor (entweder Emacs oder Atom – das hängt von meiner Stimmung ab). IPython (Abkürzung für *Interactive Python*) wurde 2001 von Fernando Perez in Form eines erweiterten Python-Interpreters ins Leben gerufen und hat sich seither zu einem Projekt entwickelt, das es sich zum Ziel gesetzt hat, »Tools für den gesamten Lebenszyklus in der forschenden Informatik« – so Perez' eigene Worte – bereitzustellen. Wenn man Python als Motor einer Aufgabe von Data Science betrachtet, können Sie sich IPython als die interaktive Steuerkonsole dazu vorstellen.

IPython ist nicht nur eine nützliche interaktive Schnittstelle zu Python, sondern stellt darüber hinaus eine Reihe praktischer syntaktischer Erweiterungen der Sprache bereit. Die nützlichsten dieser Erweiterungen werden wir gleich erörtern. IPython ist außerdem sehr eng mit dem Jupyter-Projekt verknüpft (<http://jupyter.org>), das ein browserbasiertes sogenanntes Notebook zur Verfügung stellt, das bei der Entwicklung, der Zusammenarbeit, dem Teilen und sogar der Veröffentlichung von Ergebnissen der Data Science gute Dienste leistet. Tatsächlich ist das IPython-Notebook eigentlich ein Sonderfall der umfangreicheren Jupyter-Notebook-Struktur, die Notebooks für Julia, R und andere Programmiersprachen umfasst. Um ein Beispiel für die Nützlichkeit dieses Notebook-Formats zu geben: Betrachten Sie einfach nur die Seite, die Sie gerade lesen. Das vollständige Manuskript dieses Buchs wurde in Form einer Reihe von IPython-Notebooks verfasst.

Bei IPython geht es darum, Python effizient für wissenschaftliche und datenintensive Berechnungen interaktiv einsetzen zu können. In diesem Kapitel werden wir zunächst einige der Features von IPython betrachten, die sich in der Praxis der Data Science als nützlich erweisen. Der Schwerpunkt liegt hierbei auf der bereitgestellten Syntax, die mehr zu bieten hat als die Standardfeatures von Python. Anschließend werden wir uns etwas eingehender mit einigen der sehr nützlichen »magischen Befehle« befassen, die gängige Aufgaben bei der Erstellung und Verwendung des Data-Science-Codes beschleunigen können. Zum Abschluss erörtern wir dann einige der Features des Notebooks, die dem Verständnis der Daten und dem Teilen der Ergebnisse dienen können.

## 1.1 Shell oder Notebook?

Es gibt im Wesentlichen zwei verschiedene Methoden, IPython zu verwenden, die wir in diesem Kapitel betrachten: die IPython-Shell und das IPython-Notebook. Ein Großteil des Inhalts dieses Kapitels betrifft beide, und die Beispiele verwenden im Wechsel Shell und Notebook – je nachdem, was am praktischsten ist. In den Abschnitten, die lediglich für eines der beiden Verfahren von Bedeutung sind, werde ich ausdrücklich darauf hinweisen.

Doch zunächst einmal folgen einige Hinweise zum Starten der IPython-Shell und zum Öffnen eines Notebooks.

### 1.1.1 Die IPython-Shell starten

Wie die meisten Teile dieses Buchs sollte dieses Kapitel nicht passiv gelesen werden. Ich empfehle Ihnen, während der Lektüre mit den vorgestellten Tools und der angegebenen Syntax herumzuexperimentieren. Die durch das Nachvollziehen der Beispiele erworbenen Fingerfertigkeiten werden sich als sehr viel nützlicher erweisen, als wenn Sie nur darüber lesen. Geben Sie auf der Kommandozeile **ipython** ein, um den Python-Interpreter zu starten. Sollten Sie eine Distribution wie Anaconda oder EPD (*Enthought Python Distribution*) installiert haben, können Sie möglicherweise alternativ einen systemspezifischen Programmstarter verwenden. (Wir erörtern das ausführlicher in Abschnitt 1.2, »Hilfe und Dokumentation in IPython«.)

Nach dem Start des Interpreters sollte Ihnen eine Eingabeaufforderung wie die folgende angezeigt werden:

```
IPython 4.0.1 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra
              details.

In [1]:
```

Nun können Sie fortfahren.

### 1.1.2 Das Jupyter-Notebook starten

Das Jupyter-Notebook ist eine browserbasierte grafische Schnittstelle für die Python-Shell und besitzt eine große Vielfalt dynamischer Anzeigemöglichkeiten. Neben der Ausführung von Python-/IPython-Anweisungen gestattet das Notebook dem User das Einfügen von formatiertem Text, statischen und dynamischen Visualisierungen, mathematischen Formeln, JavaScript-Widgets und vielem mehr. Darüber hinaus können die Dokumente in einem Format gespeichert werden, das es anderen Usern ermöglicht, sie auf ihren eigenen Systemen zu öffnen und den Code auszuführen.

Das IPython-Notebook wird zwar in einem Fenster Ihres Webbrowsers angezeigt und bearbeitet, allerdings ist eine Verbindung zu einem laufenden Python-Prozess erforderlich, um Code auszuführen. Geben Sie in Ihrer System-Shell folgenden Befehl ein, um diesen Prozess (der als »Kernel« bezeichnet wird) zu starten:

```
$ jupyter notebook
```

Dieser Befehl startet einen lokalen Webserver, auf den Ihr Browser zugreifen kann. Er gibt sofort einige Meldungen aus, die zeigen, was vor sich geht. Dieses Log sieht in etwa folgendermaßen aus:

```
$ jupyter notebook
[NotebookApp] Serving notebooks from local directory: /Users/jakevdp/...
[NotebookApp] 0 active kernels
[NotebookApp] The IPython Notebook is running at: http://localhost:8888/
[NotebookApp] Use Control-C to stop this server and shut down all kernels...
```

Nach der Eingabe des Befehls sollte sich automatisch Ihr Standardbrowser öffnen und die genannte lokale URL anzeigen. Die genaue Adresse ist von Ihrem System abhängig. Öffnet sich Ihr Browser nicht automatisch, können Sie von Hand ein Browserfenster öffnen und die Adresse (in diesem Beispiel `http://localhost:8888/`) eingeben.

## 1.2 Hilfe und Dokumentation in IPython

Auch wenn Sie die anderen Abschnitte dieses Kapitels überspringen, sollten Sie doch wenigstens diesen lesen: Ich habe festgestellt, dass die hier erläuterten IPython-Tools den größten Einfluss auf meinem alltäglichen Arbeitsablauf haben.

Wenn ein technologisch interessierter Mensch darum gebeten wird, einem Freund, Familienmitglied oder Kollegen bei einem Computerproblem zu helfen, geht es meistens gar nicht darum, die Lösung zu kennen, sondern zu wissen, wie man schnell eine noch unbekannte Lösung findet. Mit Data Science verhält es sich genauso: Durchsuchbare Webressourcen wie Onlinedokumentationen, Mailinglisten und auf Stackoverflowbusiness.com gefundene Antworten enthalten jede Menge Informationen, auch (und gerade?) wenn es sich um ein Thema handelt, nach dem Sie selbst schon einmal gesucht haben. Für einen leistungsfähigen Praktiker der Data Science geht es weniger darum, das in jeder erdenklichen Situation einzusetzende Tool oder den geeigneten Befehl auswendig zu lernen, sondern vielmehr darum, zu wissen, wie man die benötigten Informationen schnell und einfach findet – sei es nun mithilfe einer Suchmaschine oder auf anderem Weg.

Zwischen dem User und der erforderlichen Dokumentation sowie den Suchvorgängen, die ein effektives Arbeiten ermöglichen, klafft eine Lücke. Diese zu schließen, ist eine der nützlichsten Funktionen von IPython/Jupyter. Zwar spielen Suchvorgänge im Web bei der Beantwortung komplizierter Fragen nach wie vor eine Rolle, allerdings stellt IPython bereits eine bemerkenswerte Menge an Informationen bereit. Hier einige Beispiele für Fragen, bei deren Beantwortung IPython nach einigen wenigen Tastendrücken hilfreich sein kann:

- Wie rufe ich eine bestimmte Funktion auf? Welche Argumente und Optionen besitzt sie?
- Wie sieht der Quellcode eines bestimmten Python-Objekts aus?
- Was ist in einem importierten Paket enthalten? Welche Attribute oder Methoden besitzt ein Objekt?

Wir erörtern nun die IPython-Tools für den schnellen Zugriff auf diese Informationen, nämlich das Zeichen `?` zum Durchsuchen der Dokumentation, die beiden Zeichen `??` zum Erkunden des Quellcodes und die `Tab`-Taste, die eine automatische Vervollständigung ermöglicht.



## 1.2.1 Mit ? auf die Dokumentation zugreifen

Die Programmiersprache Python und das für die Data Science geeignete Ökosystem schenken dem User große Beachtung. Dazu gehört insbesondere der Zugang zur Dokumentation. Alle Python-Objekte enthalten einen Verweis auf einen String, den sogenannten *Docstring*, der wiederum in den meisten Fällen eine kompakte Übersicht über das Objekt und dessen Verwendung enthält. Python verfügt über eine integrierte `help()`-Funktion, die auf diese Informationen zugreift und sie ausgibt. Um beispielsweise die Dokumentation der integrierten Funktion `len` anzuzeigen, können Sie Folgendes eingeben:

```
In [1]: help(len)
Help on built-in function len in module builtins:
len(...)
    len(object) -> integer
    Return the number of items of a sequence or mapping.
```

Je nachdem, welchen Interpreter Sie verwenden, wird der Text auf der Konsole oder in einem eigenen Fenster ausgegeben.

Die Suche nach der Hilfe für ein Objekt ist äußerst nützlich und geschieht sehr häufig. Daher verwendet IPython das Zeichen `?` als Abkürzung für den Zugriff auf die Dokumentation und weitere wichtige Informationen:

```
In [2]: len?
Type:          builtin_function_or_method
String form: <built-in function len>
Namespace:    Python builtin
Docstring:
len(object) -> integer
Return the number of items of a sequence or mapping.
```

Diese Schreibweise funktioniert praktisch mit allem, auch mit Objektmethoden:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Type:          builtin_function_or_method
String form: <built-in method insert of list object at 0x1024b8ea8>
Docstring:    L.insert(index, object) -- insert object before index
```

Und sogar mit Objekten selbst – dann wird die Dokumentation des Objekttyps angezeigt:

```
In [5]: L?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
```

```
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Wichtig zu wissen ist, dass das ebenfalls mit Funktionen und anderen von Ihnen selbst erzeugten Objekten funktioniert:

```
In [6]: def square(a):
....:     """a zum Quadrat zurückgeben."""
....:     return a ** 2
....:
```

Beachten Sie hier, dass wir zum Erstellen des Docstrings unserer Funktion einfach eine literale Zeichenkette in die erste Zeile eingegeben haben. Da Docstrings für gewöhnlich mehrzeilig sind, haben wir gemäß Konvention Pythons Schreibweise für mehrzeilige Strings mit dreifachem Anführungszeichen verwendet.

Nun verwenden wir das Zeichen `?`, um diesen Docstring anzuzeigen:

```
In [7]: square?
Type:      function
String form: <function square at 0x103713cb0>
Definition: square(a)
Docstring: a zum Quadrat zurückgeben.
```

Dieser schnelle Zugriff auf die Dokumentation via Docstring ist einer der Gründe dafür, dass Sie sich angewöhnen sollten, den von Ihnen geschriebenen Code immer zu dokumentieren!

## 1.2.2 Mit ?? auf den Quellcode zugreifen

Da die Programmiersprache Python sehr leicht verständlich ist, können Sie für gewöhnlich tiefere Einblicke gewinnen, wenn Sie sich den Quellcode eines Objekts ansehen, das Sie interessiert. Mit einem doppelten Fragezeichen (`??`) stellt IPython eine Abkürzung für den Zugriff auf den Quellcode zur Verfügung:

```
In [8]: square??
Type:      function
String form: <function square at 0x103713cb0>
Definition: square(a)
Source:
def square(a):
    "a zum Quadrat zurückgeben."
    return a ** 2
```

Bei so einfachen Funktionen wie dieser können Sie mit dem doppelten Fragezeichen einen schnellen Blick darauf werfen, was unter der Haube vor sich geht.

Sollten Sie damit weiter herumexperimentieren, werden Sie feststellen, dass ein angehängtes `??` manchmal überhaupt keinen Quellcode anzeigt. Das liegt im Allgemeinen daran, dass das fragliche Objekt nicht in Python implementiert ist, sondern in C oder einer anderen kompilierten Erweiterungssprache. In diesem Fall liefert `??` dieselbe Ausgabe wie `?`. Das kommt insbesondere bei vielen in Python fest integrierten Objekten und Typen vor, wie beispielsweise bei der vorhin erwähnten Funktion `len`:

```
In [9]: len??
Type:      builtin_function_or_method
String form: <built-in function len>
Namespace: Python builtin
Docstring:
len(object) -> integer
Return the number of items of a sequence or mapping.
```

Der Einsatz von `?` und/oder `??` bietet eine schnelle und leistungsfähige Schnittstelle für das Auffinden von Informationen darüber, was in einer Python-Funktion oder einem Python-Modul eigentlich geschieht.

### 1.2.3 Module mit der Tab-Vervollständigung erkunden

IPython besitzt eine weitere nützliche Schnittstelle: die Verwendung der `[Tab]`-Taste zur automatischen Vervollständigung und zum Erkunden des Inhalts von Objekten, Modulen und Namensräumen. In den folgenden Beispielen wird durch `<TAB>` angezeigt, dass die `[Tab]`-Taste gedrückt werden muss.

#### Tab-Vervollständigung des Inhalts von Objekten

Jedes Python-Objekt besitzt verschiedene Attribute und Methoden, die ihm zugeordnet sind. Neben dem bereits erläuterten `help` verfügt Python über eine integrierte `dir`-Funktion, die eine Liste dieser Attribute und Methoden ausgibt. Allerdings ist es in der Praxis viel einfacher, die Tab-Vervollständigung zu verwenden. Um eine Liste aller verfügbaren Attribute anzuzeigen, geben Sie einfach den Namen des Objekts ein, gefolgt von einem Punkt `(.)` und der `[Tab]`-Taste:

```
In [10]: L.<TAB>
L.append  L.copy    L.extend  L.insert  L.remove  L.sort
L.clear   L.count    L.index   L.pop     L.reverse
```

Um die Anzahl der Treffer in der Liste zu verringern, geben Sie einfach den ersten oder mehrere Buchstaben des Namens ein. Nach dem Betätigen der `[Tab]`-Taste werden dann nur noch die übereinstimmenden Attribute und Methoden angezeigt:

```
In [10]: L.c<TAB>
L.clear  L.copy  L.count
In [10]: L.co<TAB>
L.copy  L.count
```

Wenn der Treffer eindeutig ist, wird die Zeile durch ein weiteres Drücken der `[Tab]`-Taste vervollständigt. Die folgende Eingabe wird beispielsweise sofort zu `L.count` vervollständigt:

```
In [10]: L.cou<TAB>
```

Python erzwingt zwar keine strenge Unterscheidung zwischen öffentlichen/externen und privaten/internen Attributen, allerdings gibt es die Konvention, Letztere durch einen vorangestellten Unterstrich zu kennzeichnen. Der Einfachheit halber werden die privaten und besonderen Methoden in der Liste standardmäßig weggelassen. Es ist jedoch möglich, sie durch ausdrückliche Eingabe des Unterstrichs anzuzeigen:

```
In [10]: L.<TAB>
L.__add__      L.__gt__      L.__reduce__
L.__class__    L.__hash__    L.__reduce_ex__
```

Wir zeigen hier nur kurz die ersten paar Zeilen der Ausgabe. Bei den meisten handelt es sich um Pythons spezielle Methoden, deren Namen mit einem doppelten Unterstrich beginnen (oft auch bezeichnet mit dem Spitznamen »dunder«-Methoden).

### Tab-Vervollständigung beim Importieren

Auch beim Importieren von Objekten eines Pakets erweist sich die Tab-Vervollständigung als nützlich. Hier verwenden wir sie, um alle möglichen Importe des `itertools`-Pakets zu finden, deren Namen mit `co` beginnen:

```
In [10]: from itertools import co<TAB>
combinations                compress
combinations_with_replacement  count
```

Auf ähnliche Weise können Sie die Tab-Vervollständigung einsetzen, um zu prüfen, welche Importe für Ihr System verfügbar sind (das hängt davon ab, welche Skripte und Module von Drittherstellern für Ihre Python-Sitzung zugänglich sind):

```
In [10]: import <TAB>
Display all 399 possibilities? (y or n)
Crypto                dis                py_compile
Cython                distutils         pyc1br
...                  ...                ...

diff1ib               pwd                zmq
In [10]: import h<TAB>
hashlib               hmac                http
heapq                 html                hus1
```

(Der Kürze halber sind hier wieder nicht alle 399 importierbaren Pakete und Module aufgeführt, die auf meinem System verfügbar sind.)

## Mehr als die Tab-Vervollständigung: Suche mit Wildcards

Die Tab-Vervollständigung ist nützlich, wenn Ihnen die ersten paar Buchstaben des Namens eines Objekts oder Attributs bekannt sind, das Sie suchen, hilft aber nicht weiter, wenn Sie nach übereinstimmenden Zeichen in der Mitte oder am Ende einer Bezeichnung suchen. Für diesen Anwendungsfall hält IPython mit dem Zeichen `*` eine Suche mit Wildcards bereit.

Wir können das Zeichen beispielsweise verwenden, um alle Objekte im Namensraum anzuzeigen, deren Namen auf `Warning` enden:

```
In [10]: *Warning?
BytesWarning          RuntimeError
DeprecationWarning    SyntaxWarning
FutureWarning         UnicodeWarning
ImportWarning         UserWarning
PendingDeprecationWarning Warning
ResourceWarning
```

Beachten Sie, dass das Zeichen `*` mit allen Strings übereinstimmt – auch mit einer leeren Zeichenkette.

Nehmen wir nun an, wir suchten nach einer Stringmethode, die an irgendeiner Stelle das Wort `find` enthält. Auf diese Weise können wir sie finden:

```
In [10]: str.*find*?
str.find
str.rfind
```

Ich finde diese Art der flexiblen Suche mit Wildcards äußerst nützlich, um einen bestimmten Befehl zu finden, wenn ich ein neues Paket erkunde oder mich mit einem bereits bekannten erneut vertraut mache.

## 1.3 Tastaturkürzel in der IPython-Shell

Auch wenn Sie nur wenig Zeit mit dem Computer verbringen, werden Sie vermutlich bereits festgestellt haben, dass sich das eine oder andere Tastaturkürzel für Ihren Arbeitsablauf als nützlich erweist. Am bekanntesten sind vielleicht `Cmd-C` und `Cmd-V` (bzw. `Strg-C` und `Strg-V`) zum Kopieren und Einfügen in vielen ganz verschiedenen Programmen und Systemen. Erfahrene User gehen oft sogar noch einen Schritt weiter: Gängige Texteditoren wie Emacs, Vim und andere stellen dem User einen immensen Umfang an verschiedenen Funktionen durch komplizierte Kombinationen von Tastendrücken zur Verfügung.

Ganz so weit geht die IPython-Shell nicht, dennoch bietet sie einige Tastaturkürzel zum schnellen Navigieren beim Eingeben von Befehlen. Diese Tastaturkürzel stellt tatsächlich nicht IPython selbst zur Verfügung, sondern die von dem Programm genutzte GNU-Readline-Bibliothek. Aus diesem Grund unterscheiden sich einige Tastaturkürzel auf Ihrem Sys-

tem abhängig von der Konfiguration womöglich von den nachstehend aufgeführten. Einige der Tastaturkürzel funktionieren eventuell auch im browserbasierten Notebook, allerdings geht es in diesem Abschnitt vornehmlich um die Tastaturkürzel in der IPython-Shell.

Sobald Sie sich daran gewöhnt haben, sind sie äußerst nützlich, um schnell bestimmte Befehle auszuführen, ohne die Hände von der Tastatur nehmen zu müssen. Sollten Sie Emacs-User sein oder Erfahrung mit Linux-Shells haben, wird Ihnen das Folgende bekannt vorkommen. Wir gruppieren die Befehle nach bestimmten Kategorien: Tastaturkürzel zum Navigieren, Tastaturkürzel bei der Texteingabe, Tastaturkürzel für den Befehlsverlauf und sonstige Tastaturkürzel.

### 1.3.1 Tastaturkürzel zum Navigieren

Dass die nach links und rechts weisenden Pfeiltasten dazu dienen, den Cursor in der Zeile rückwärts bzw. vorwärts zu bewegen, ist ziemlich offensichtlich, es gibt jedoch weitere Möglichkeiten, die es nicht erforderlich machen, Ihre Hände aus der gewohnten Schreibposition zu bewegen (siehe Tabelle 1.1).

Tastaturkürzel	Beschreibung
<code>Strg</code> - <code>A</code>	Bewegt den Cursor an den Zeilenanfang.
<code>Strg</code> - <code>E</code>	Bewegt den Cursor an das Zeilenende.
<code>Strg</code> - <code>B</code> (oder Pfeil nach links)	Bewegt den Cursor ein Zeichen rückwärts.
<code>Strg</code> - <code>F</code> (oder Pfeil nach rechts)	Bewegt den Cursor ein Zeichen vorwärts.

**Tabelle 1.1:** Tastaturkürzel zum Navigieren

### 1.3.2 Tastaturkürzel bei der Texteingabe

Jedem User ist die Verwendung der Rückschritttaste zum Löschen des zuvor eingegebenen Zeichens bekannt, allerdings sind oftmals einige Fingerverrenkungen erforderlich, um sie zu erreichen, und außerdem löscht sie beim Betätigen jeweils nur ein einzelnes Zeichen. In IPython gibt es einige Tastaturkürzel zum Löschen bestimmter Teile der eingegebenen Textzeile. Sofort nützlich sind die Befehle zum Löschen der ganzen Textzeile. Sie sind Ihnen in Fleisch und Blut übergegangen, wenn Sie feststellen, dass Sie die Tastenkombinationen `Strg`-`B` und `Strg`-`D` verwenden, anstatt die Rückschritttaste zu benutzen, um das zuvor eingegebene Zeichen zu löschen!

Tastaturkürzel	Beschreibung
Rückschritttaste	Zeichen links vom Cursor löschen.
<code>Strg</code> - <code>D</code>	Zeichen rechts vom Cursor löschen.
<code>Strg</code> - <code>K</code>	Text von der Cursorposition bis zum Zeilenende ausschneiden.
<code>Strg</code> - <code>U</code>	Text vom Zeilenanfang bis zur Cursorposition ausschneiden.
<code>Strg</code> - <code>Y</code>	Zuvor ausgeschnittenen Text einfügen.
<code>Strg</code> - <code>T</code>	Die beiden zuletzt eingegebenen Zeichen vertauschen.

**Tabelle 1.2:** Tastaturkürzel bei der Texteingabe

### 1.3.3 Tastaturkürzel für den Befehlsverlauf

Unter den hier aufgeführten von IPython bereitgestellten Tastaturkürzeln dürften diejenigen zur Navigation im Befehlsverlauf die größten Auswirkungen haben. Der Befehlsverlauf umfasst nicht nur die aktuelle IPython-Sitzung, Ihr gesamter Befehlsverlauf ist in einer SQLite-Datenbank im selben Verzeichnis gespeichert, in dem sich auch Ihr IPython-Profil befindet. Die einfachste Zugriffsmöglichkeit auf Ihren Befehlsverlauf ist das Betätigen der Pfeiltasten nach oben und unten, mit denen Sie ihn schrittweise durchblättern können, es stehen aber noch andere Möglichkeiten zur Verfügung (siehe Tabelle 1.3).

Tastaturkürzel	Beschreibung
<code>[Strg]-[P]</code> (oder Pfeiltaste nach oben)	Vorhergehenden Befehl im Verlauf auswählen.
<code>[Strg]-[N]</code> (oder Pfeiltaste nach unten)	Nachfolgenden Befehl im Verlauf auswählen.
<code>[Strg]-[R]</code>	Rückwärtssuche im Befehlsverlauf.

**Tabelle 1.3:** Tastaturkürzel für den Befehlsverlauf

Die Rückwärtssuche kann besonders praktisch sein. Wie Sie wissen, haben wir im vorigen Abschnitt eine Funktion namens `square` definiert. Durchsuchen Sie nun in einer neuen IPython-Shell den Befehlsverlauf nach dieser Definition. Wenn Sie im IPython-Terminal die Tastenkombination `[Strg]-[R]` drücken, wird Ihnen die folgende Eingabeaufforderung angezeigt:

```
In [1]:
(reverse-i-search) `':
```

Beginnen Sie nun damit, Zeichen einzugeben, zeigt IPython den zuletzt eingegebenen Befehl an (sofern vorhanden), der mit den eingegebenen Zeichen übereinstimmt:

```
In [1]:
(reverse-i-search) `squ': square??
```

Sie können jederzeit weitere Zeichen eingeben, um die Suche zu verfeinern, oder drücken Sie erneut `[Strg]-[R]`, um nach einem weiter zurückliegenden Befehl zu suchen, der zur Suchanfrage passt. Wenn Sie die Eingaben im letzten Abschnitt nachvollzogen haben, wird nach zweimaligem Betätigen von `[Strg]-[R]` Folgendes angezeigt:

```
In [1]:
(reverse-i-search) `squ': def square(a):
    """a zum Quadrat zurückgeben."""
    return a ** 2
```

Sobald Sie den gesuchten Befehl gefunden haben, können Sie die Suche mit der `[Enter]`-Taste beenden. Jetzt können Sie den gefundenen Befehl ausführen und die Sitzung fortsetzen:

```
In [1]: def square(a):
    """a zum Quadrat zurückgeben."""
```

```

    return a ** 2
In [2]: square(2)
Out[2]: 4

```

Sie können auch die Tastenkombinationen `[Strg]-[P]`/`[Strg]-[N]` oder die Pfeiltasten nach oben und unten verwenden, um den Befehlsverlauf zu durchsuchen, allerdings werden dann bei der Suche lediglich die Zeichen am Anfang der Eingabezeile berücksichtigt. Wenn Sie also **def** eingeben und dann `[Strg]-[P]` drücken, wird der zuletzt eingegebene Befehl im Verlauf angezeigt (falls vorhanden), der mit den Zeichen **def** beginnt.

### 1.3.4 Sonstige Tastaturkürzel

Darüber hinaus gibt es einige weitere nützliche Tastaturkürzel, die sich keiner der bisherigen Kategorien zuordnen lassen (siehe Tabelle 1.4).

Tastaturkürzel	Beschreibung
<code>[Strg]-[L]</code>	Terminalanzeige löschen.
<code>[Strg]-[C]</code>	Aktuellen Python-Befehl abbrechen.
<code>[Strg]-[D]</code>	Python-Sitzung beenden.

**Tabelle 1.4:** Sonstige Tastaturkürzel

Insbesondere der Befehl `[Strg]-[C]` kann sich als nützlich erweisen, wenn Sie versehentlich einen sehr zeitaufwendigen Job gestartet haben.

Einige der hier vorgestellten Befehle mögen auf den ersten Blick vielleicht uninteressant erscheinen, Sie werden sie aber mit etwas Übung wie im Schlaf benutzen. Haben Sie sich diese Fingerfertigkeiten einmal angeeignet, werden Sie sich sogar wünschen, dass diese Befehle auch an anderer Stelle zur Verfügung stünden.

## 1.4 Magische Befehle in IPython

Die beiden letzten Abschnitte zeigen, wie IPython es Ihnen ermöglicht, Python effektiv und interaktiv zu verwenden und zu erkunden. Nun kommen wir zu einigen Erweiterungen, die IPython der normalen Python-Syntax hinzufügt. Diese werden in IPython als »magische« Befehle oder Funktionen bezeichnet, und ihnen wird ein `%`-Zeichen vorangestellt. Die magischen Befehle sind dazu gedacht, verschiedene gängige Aufgaben, die bei einer Standarddatenanalyse immer wieder vorkommen, kurz und bündig zu erledigen. Von den magischen Befehlen/Funktionen (den sogenannten *Magics*) gibt es zwei Varianten: *Line-Magics*, denen ein einzelnes `%` vorangestellt wird und die jeweils eine einzelne Zeile verarbeiten, sowie *Cell-Magics*, die durch ein vorangestelltes `%%` gekennzeichnet sind und mehrzeilige Eingaben verarbeiten. Wir werden einige kurze Beispiele betrachten und befassen uns dann später in diesem Kapitel mit einer eingehenderen Erläuterung verschiedener nützlicher Magics.

### 1.4.1 Einfügen von Codeblöcken mit `%paste` und `%cpaste`

Beim Einsatz des IPython-Interpreters gibt es häufig das Problem, dass es beim Einfügen mehrzeiliger Codeblöcke zu unerwarteten Fehlern kommt, vor allem wenn der Text Einrückungen enthält.



kungen und vom Interpreter als Markierungen verwendete Zeichen enthält. Häufig ist das der Fall, wenn man auf einer Website Beispielcode entdeckt, den man im Interpreter einfügen möchte. Betrachten Sie die folgende einfache Funktion:

```
>>> def donothing(x):
...     return x
```

Der Code ist so formatiert, wie er im Python-Interpreter angezeigt werden soll. Wenn Sie ihn jedoch kopieren und direkt in IPython einfügen, erscheint eine Fehlermeldung:

```
In [2]: >>> def donothing(x):
...:     ...     return x
...:
...:
File "<ipython-input-20-5a66c8964687>", line 2
...     return x
                ^
SyntaxError: invalid syntax
```

Beim direkten Einfügen gerät der Interpreter durch die zusätzlich vorhandenen Zeichen zur Eingabeaufforderung durcheinander. Aber keine Sorge – IPythons magische Funktion `%paste` ist dafür ausgelegt, genau diesen Typ mehrzeiliger mit Textauszeichnungen versehener Eingaben korrekt zu handhaben:

```
In [3]: %paste
>>> def donothing(x):
...     return x

## -- End pasted text --
```

Der `%paste`-Befehl fügt den Code ein und führt ihn aus, die Funktion kann nun also verwendet werden:

```
In [4]: donothing(10)
Out[4]: 10
```

Der Befehl `%cpaste` hat einen ganz ähnlichen Zweck und zeigt eine interaktive mehrzeilige Eingabeaufforderung an, in der Sie einen oder mehrere Codeschnipsel einfügen und der Reihe nach ausführen lassen können:

```
In [5]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:>>> def donothing(x):
...     return x
:--
```

Diese magischen Befehle – und andere, auf die wir später noch zu sprechen kommen – stellen eine Funktionalität bereit, die mit einem herkömmlichen Python-Interpreter nur sehr schwer zu erzielen oder sogar unmöglich wäre.

### 1.4.2 Externen Code ausführen mit %run

Wenn Sie damit anfangen, umfangreicheren Code zu entwickeln, werden Sie vermutlich feststellen, dass Sie sowohl IPython für interaktive Erkundungen als auch einen Texteditor zum Speichern von Code einsetzen, den Sie wiederverwenden möchten. Oft ist es praktisch, den Code nicht in einem neuen Fenster, sondern innerhalb der laufenden IPython-Sitzung auszuführen. Zu diesem Zweck gibt es den magischen Befehl `%run`.

Nehmen wir beispielsweise an, Sie haben eine Datei namens *myscript.py* angelegt, die folgenden Inhalt hat:

```
#-----  
# file: myscript.py  
def square(x):  
    """Quadrieren einer Zahl"""  
    return x ** 2  
for N in range(1, 4):  
    print(N, "zum Quadrat ist", square(N))
```

Sie können diese Datei wie folgt in Ihrer IPython-Sitzung ausführen:

```
In [6]: %run myscript.py  
1 zum Quadrat ist 1  
2 zum Quadrat ist 4  
3 zum Quadrat ist 9
```

Beachten Sie hier außerdem, dass nach der Ausführung dieses Skripts die darin definierten Funktionen in Ihrer IPython-Sitzung verfügbar sind:

```
In [7]: square(5)  
Out[7]: 25
```

Es stehen verschiedene Möglichkeiten zur Verfügung, genauer einzustellen, wie Ihr Code ausgeführt wird. Sie können sich durch die Eingabe von `%run?` wie gewohnt die Dokumentation im IPython-Interpreter anzeigen lassen.

### 1.4.3 Messung der Ausführungszeit von Code mit %timeit

Ein weiteres Beispiel einer nützlichen magischen Funktion ist `%timeit`, die automatisch die Ausführungszeit einer einzelnen Python-Anweisung ermittelt, die dem Befehl übergeben wird. Wir könnten beispielsweise die Performance einer Listenabstraktion wie folgt ermitteln:

```
In [8]: %timeit L = [n ** 2 for n in range(1000)]  
1000 loops, best of 3: 325 µs per loop
```

Die `%timeit`-Funktion hat den Vorteil, dass sie bei kurzen Befehlen automatisch mehrere Durchläufe ausführt, um aussagekräftigere Ergebnisse zu erhalten. Bei mehrzeiligen

Anweisungen macht das Hinzufügen eines zweiten %-Zeichens den Befehl zu einem Cell-Magic, das mehrzeilige Eingaben verarbeiten kann. Hier ist beispielsweise der entsprechende Code mit einer `for`-Schleife:

```
In [9]: %%timeit
...: L = []
...: for n in range(1000):
...:     L.append(n ** 2)
...:
1000 loops, best of 3: 373 µs per loop
```

Wir können sofort feststellen, dass die Listenabstraktion in diesem Fall rund 10 % schneller ist als die entsprechende `for`-Schleife. Wir werden uns mit `%timeit` und anderen Ansätzen für das Timing und Profiling von Code in Abschnitt 1.9, »Profiling und Timing von Code«, noch eingehender befassen.

### 1.4.4 Hilfe für die magischen Funktionen anzeigen mit `?`, `%magic` und `%lsmagic`

Wie normale Python-Funktion besitzen auch IPythons magische Funktionen Docstrings, und auf diese nützliche Dokumentation kann man wie gewohnt zugreifen. Um also beispielsweise die Dokumentation des magischen Befehls `%timeit` zu lesen, geben Sie einfach Folgendes ein:

```
In [10]: %timeit?
```

Auf die Dokumentation anderer Funktionen wird auf ähnliche Weise zugegriffen. Zur Anzeige einer allgemeinen Beschreibung der verfügbaren magischen Funktionen inklusive einiger Beispiele geben Sie nachstehenden Befehl ein:

```
In [11]: %magic
```

Und so zeigen Sie schnell und einfach eine Liste aller zur Verfügung stehenden magischen Funktionen an:

```
In [12]: %lsmagic
```

Abschließend möchte ich noch erwähnen, dass es ganz einfach möglich ist, eigene magische Funktionen zu definieren. Wir werden darauf an dieser Stelle nicht weiter eingehen, aber wenn Sie daran interessiert sind, werfen Sie einen Blick auf die Hinweise in Abschnitt 1.10, »Weitere IPython-Ressourcen«.

## 1.5 Verlauf der Ein- und Ausgabe

Sie wissen bereits, dass die IPython-Shell es ermöglicht, mit den Pfeiltasten nach oben und unten (oder mit den entsprechenden Tastaturkürzeln `[Strg]-P`/`[Strg]-N`) auf frühere Befehle zuzugreifen. Sowohl in der Shell als auch in Notebooks bietet IPython darüber

hinaus verschiedene Möglichkeiten, die Ausgabe vorhergehender Befehle oder reine Textversionen der Befehle selbst abzurufen. Das sehen wir uns nun genauer an.

### 1.5.1 Die IPython-Objekte In und Out

Die von IPython verwendeten Ausgaben der Form `In[1]:/Out[1]:` dürften Ihnen inzwischen hinlänglich vertraut sein. Dabei handelt es sich jedoch keinesfalls nur um hübsche Verzierungen, vielmehr geben sie einen Hinweis darauf, wie Sie auf vorhergehende Ein- und Ausgaben Ihrer aktuellen Sitzung zugreifen können. Nehmen wir an, Sie starten eine Sitzung, die folgendermaßen aussieht:

```
In [1]: import math
In [2]: math.sin(2)
Out[2]: 0.9092974268256817
In [3]: math.cos(2)
Out[3]: -0.4161468365471424
```

Wir importieren das integrierte `math`-Paket und berechnen dann den Sinus und den Kosinus von 2. Diese Ein- und Ausgaben werden in der Shell mit `In/Out`-Labeln angezeigt. Das ist jedoch noch nicht alles – tatsächlich erzeugt IPython verschiedene Python-Variablen namens `In` und `Out`, die automatisch aktualisiert werden und so den Verlauf widerspiegeln:

```
In [4]: print(In)
['', 'import math', 'math.sin(2)', 'math.cos(2)', 'print(In)']

In [5]: Out
Out[5]: {2: 0.9092974268256817, 3: -0.4161468365471424}
```

Das `In`-Objekt ist eine Liste, die über die Reihenfolge der Befehle Buch führt (das erste Element der Liste ist ein Platzhalter, sodass `In[1]` auf den ersten Befehl verweist):

```
In [6]: print(In[1])
import math
```

Das `Out`-Objekt hingegen ist keine Liste, sondern ein Dictionary, in dem die Eingabenummern den jeweiligen Ausgaben (falls vorhanden) zugeordnet sind:

```
In [7]: print(Out[2])
0.9092974268256817
```

Beachten Sie hier, dass nicht alle Operationen eine Ausgabe erzeugen. Beispielsweise haben die `import`- und `print`-Anweisungen keine Auswirkung auf die Ausgabe. Letzteres überrascht vielleicht etwas, ergibt jedoch Sinn, wenn man bedenkt, dass `print` eine Funktion ist, die `None` zurückliefert. Kurz und bündig: Alle Befehle, die `None` zurückgeben, werden nicht zum `Out`-Dictionary hinzugefügt.

Das kann sich als nützlich erweisen, wenn Sie die letzten Ergebnisse verwenden möchten. Berechnen Sie beispielsweise die Summe von  $\sin(2) ** 2$  und  $\cos(2) ** 2$  unter Zuhilfenahme der zuvor errechneten Ergebnisse:

```
In [8]: Out[2] ** 2 + Out[3] ** 2
Out[8]: 1.0
```

Das Ergebnis lautet 1.0, wie es gemäß der wohlbekannten trigonometrischen Gleichung auch zu erwarten ist. In diesem Fall wäre es eigentlich gar nicht notwendig, die vorhergehenden Ergebnisse zu verwenden, allerdings kann es ungemein praktisch sein, wenn Sie eine sehr zeitaufwendige Berechnung ausführen und das Ergebnis wiederverwenden möchten!

## 1.5.2 Der Unterstrich als Abkürzung und vorhergehende Ausgaben

Die normale Python-Shell besitzt nur eine einfache Abkürzung für den Zugriff auf vorherige Ausgaben. Die Variable `_` (ein einfacher Unterstrich) enthält das Ergebnis der jeweils letzten Ausgabe. In IPython funktioniert das ebenfalls:

```
In [9]: print(_)
1.0
```

Allerdings geht IPython noch einen Schritt weiter – Sie können außerdem einen doppelten Unterstrich verwenden, um auf die vorletzte Ausgabe zuzugreifen, oder einen dreifachen, um auf die drittletzte Ausgabe zuzugreifen (wobei alle Befehle ohne Ausgabe übersprungen werden):

```
In [10]: print(__)
-0.4161468365471424
In [11]: print(___)
0.9092974268256817
```

Hier ist in IPython jedoch Schluss: Mehr als drei Unterstriche sind etwas schwierig abzuzählen, und es ist einfacher, die Nummer der Ausgabe zu verwenden.

Eine weitere Abkürzung soll an dieser Stelle noch Erwähnung finden: `_X` (ein einfacher Unterstrich, gefolgt von der Ausgabennummer) ist die Abkürzung für `Out[X]`:

```
In [12]: Out[2]
Out[12]: 0.9092974268256817
In [13]: _2
Out[13]: 0.9092974268256817
```

## 1.5.3 Ausgaben unterdrücken

Manchmal ist es erwünscht, die Ausgaben einer Anweisung zu unterdrücken (am häufigsten kommt das vielleicht bei den Befehlen zum Erstellen von Diagrammen vor, mit denen wir uns in Kapitel 4 befassen werden). Oder der auszuführende Befehl liefert ein Ergebnis, das Sie lieber nicht im Verlauf der Ausgabe speichern möchten, möglicherweise damit der

dadurch belegte Speicherplatz wieder freigegeben wird, wenn es keine weiteren Referenzen mehr darauf gibt. Die einfachste Methode zum Unterdrücken der Ausgabe ist das Anhängen eines Semikolons an das Zeilenende:

```
In [14]: math.sin(2) + math.cos(2);
```

Beachten Sie hier, dass das Ergebnis zwar berechnet, aber weder auf dem Bildschirm angezeigt noch im Out-Dictionary gespeichert wird:

```
In [15]: 14 in Out
Out[15]: False
```

### 1.5.4 Weitere ähnliche magische Befehle

Mit dem magischen Befehl `%history` kann man auf mehrere vorhergehende Eingaben gleichzeitig zugreifen. So können Sie die ersten vier Eingaben anzeigen:

```
In [16]: %history -n 1-4
1: import math
2: math.sin(2)
3: math.cos(2)
4: print(In)
```

Sie können wie gewohnt **%history?** eingeben, um weitere Informationen und eine Beschreibung der möglichen Optionen anzuzeigen. `%rerun` (erneute Ausführung eines Teils des Befehlsverlaufs) und `%save` (zum Speichern eines Teils des Befehlsverlaufs in einer Datei) sind weitere ähnliche magische Befehle. Wenn Sie an zusätzlichen Informationen interessiert sind, können Sie die in Abschnitt 1.2, »Hilfe und Dokumentation in IPython«, beschriebene Hilfsfunktion `?` verwenden, um diese Befehle zu erkunden.

## 1.6 IPython und Shell-Befehle

Wenn man einen normalen Python-Interpreter interaktiv verwendet, muss man sich damit herumärgern, dass man gezwungen ist, zwischen mehreren Fenstern hin und her zu schalten, um auf Python-Tools und Kommandozeilenprogramme des Systems zuzugreifen. IPython schließt diese Lücke und stellt eine Syntax zum Ausführen von Shell-Befehlen direkt im IPython-Terminal bereit. Möglich macht das ein Ausrufezeichen: Jeglicher nach einem `!` stehender Text in einer Zeile wird nicht vom Python-Kernel, sondern von der Kommandozeile des Systems ausgeführt.

Im Folgenden wird vorausgesetzt, dass Sie ein unixoides System wie Linux oder macOS verwenden. Einige der Beispiele werden unter Windows fehlschlagen, das standardmäßig eine andere Art von Shell verwendet. Allerdings ist mittlerweile eine native Version der Shell Bash verfügbar, sodass dies kein Problem mehr darstellt. Sollten Ihnen Shell-Befehle nicht geläufig sein, empfiehlt sich die Lektüre des Shell-Tutorials, das von der ausgezeichneten Software Carpentry Foundation zusammengestellt wurde (<http://swcarpentry.github.io/shell-novice/>).

## 1.6.1 Kurz vorgestellt: die Shell

Eine vollständige Einführung in die Arbeit mit Shell, Terminal oder Kommandozeile geht weit über den Rahmen dieses Kapitels hinaus. An dieser Stelle folgt lediglich eine Kurzeinführung für Leser, die über gar keine Kenntnisse auf diesem Gebiet verfügen. Die Shell bietet eine Möglichkeit, per Texteingabe mit dem Computer zu interagieren. Seit Mitte der 1980er-Jahre, als Apple und Microsoft die ersten Versionen der heute allgegenwärtigen grafischen Betriebssysteme vorstellten, interagieren die meisten User mit ihrem Betriebssystem durch das vertraute Klicken auf Menüpunkte und durch Verschieben von Objekten mit der Maus. Nun gab es jedoch schon viel früher, lange bevor die grafischen Benutzeroberflächen entwickelt wurden, Betriebssysteme, die vornehmlich durch Texteingaben gesteuert wurden: Der User gibt auf der Kommandozeile einen Befehl ein, und der Computer führt aus, was der User ihm befohlen hat. Diese ersten Kommandozeilensysteme sind die Vorgänger der Shells und Terminals, die viele Data Scientists auch heute noch verwenden.

Wenn man mit der Shell nicht vertraut ist, mag man fragen, warum man sich diese Mühe machen sollte, wenn man doch mit ein paar Mausklicks auf Symbole und Menüs schon so viel erreichen kann. Ein Shell-User könnte mit einer Gegenfrage antworten: Warum irgendwelchen Symbolen nachjagen und Menüpunkte anklicken, wenn man seine Ziele durch Texteingaben viel einfacher erreichen kann? Zunächst hört sich das nach einer dieser typischen Pattsituationen zweier Lager mit unterschiedlichen Präferenzen an. Wenn jedoch mehr als nur grundlegende Arbeiten zu erledigen sind, wird schnell deutlich, dass die Shell bei anspruchsvolleren Aufgaben viel mehr Steuerungsmöglichkeiten bietet, wenngleich die Lernkurve den durchschnittlichen Computeruser zugegebenermaßen einschüchtern kann.

Nachstehend finden Sie als Beispiel eine Linux/macOS-Shell-Sitzung, in der ein User Dateien und Verzeichnisse auf dem System erkundet, anlegt und modifiziert (**bash:~ \$** ist die Eingabeaufforderung, und alles hinter dem **\$**-Zeichen ist der eingegebene Befehl; die Texte, denen ein **#** vorausgeht, sind lediglich Beschreibungen und müssen nicht eingetippt werden):

```
bash:~ $ echo "Hallo Welt" #echo entspricht Pythons print
Hallo Welt
bash:~ $ pwd # pwd = print working directory
# (Arbeitsverzeichnis ausgeben)
/home/jake
bash:~ $ ls # ls = list; Inhalt des Verzeichnisses ausgeben
notebooks projects
bash:~ $ cd projects/ # cd = change directory
# (Verzeichnis wechseln)
bash:projects $ pwd
/home/jake/projects
bash:projects $ ls
datasci_book mpld3 myproject.txt
bash:projects $ mkdir myproject # mkdir = make directory
# (Verzeichnis anlegen)
```

```
bash:projects $ cd myproject/  
bash:myproject $ mv ../myproject.txt ./ # mv = move file  
                # (Datei verschieben)  
                # Hier bewegen wir die Datei myproject.txt  
                # in einer höheren Verzeichnisebene (../)  
                # in das aktuelle Arbeitsverzeichnis (./)  
bash:myproject $ ls  
myproject.txt
```

Wie Sie sehen, handelt es sich hier lediglich um eine kompakte Art und Weise, gängige Operationen (Navigieren in einer Verzeichnisstruktur, Anlegen eines Verzeichnisses, Datei verschieben usw.) durch die Eingabe von Befehlen auszuführen, statt Symbole und Menüs anzuklicken. Beachten Sie, dass sich die gebräuchlichsten Dateioperationen mit einigen wenigen Befehlen (`pwd`, `ls`, `cd`, `mkdir` und `cp`) erledigen lassen. Die wahre Leistungsfähigkeit der Shell zeigt sich vor allem aber dann, wenn man anspruchsvollere als diese grundlegenden Aufgaben erledigen möchte.

## 1.6.2 Shell-Befehle in IPython

Sie können sämtliche in der Kommandozeile verfügbaren Befehle in IPython verwenden, indem Sie ihnen ein `!`-Zeichen voranstellen.. So können die Befehle `ls`, `pwd` und `echo` beispielsweise folgendermaßen ausgeführt werden:

```
In [1]: !ls  
myproject.txt  
In [2]: !pwd  
/home/jake/projects/myproject  
In [3]: !echo "Textausgabe per Shell"  
Textausgabe per Shell
```

## 1.6.3 Werte mit der Shell austauschen

Shell-Befehle können nicht nur von IPython aus aufgerufen werden, sie können auch mit dem IPython Namensraum interagieren. So können Sie beispielsweise die Ausgabe eines beliebigen Shell-Befehls mit dem Zuweisungsoperator in einer Python-Liste speichern:

```
In [4]: inhalt = !ls  
In [5]: print(inhalt)  
['myproject.txt']  
In [6]: verzeichnis = !pwd  
In [7]: print(verzeichnis)  
['/Users/jakevdp/notebooks/tmp/myproject']
```

Beachten Sie hier, dass das Ergebnis nicht als Liste zurückgegeben wird, sondern als ein in IPython definierter spezieller Rückgabetypp für Shells:



```
In [8]: type(directory)
IPython.utils.text.SList
```

Dieser Typ sieht zwar wie eine Python-Liste aus und verhält sich auch sehr ähnlich, verfügt aber über zusätzliche Funktionalität, wie z.B. über die `grep`- und `fields`-Methoden sowie die Eigenschaften `s`, `n` und `p`, die es Ihnen erlauben, die Ergebnisse auf komfortable Art und Weise zu durchsuchen, zu filtern und anzuzeigen. Weitere Informationen dazu finden Sie in IPythons integrierter Hilfsfunktion.

Die Kommunikation in der anderen Richtung, also die Übergabe von Python-Variablen an die Shell, wird durch die Syntax `{varname}` ermöglicht:

```
In [9]: meldung = "Hallo von Python"
In [10]: !echo {meldung}
Hallo von Python
```

Der Variablenname wird von den geschweiften Klammern eingeschlossen und wird im Shell-Befehl durch den Inhalt der Variablen ersetzt.

## 1.7 Magische Befehle für die Shell

Wenn Sie ein Weilchen mit IPythons Shell-Befehlen herumexperimentiert haben, ist Ihnen vielleicht aufgefallen, dass es nicht möglich ist, mit `!cd` im Dateisystem zu navigieren:

```
In [11]: !pwd
/home/jake/projects/myproject
In [12]: !cd ..
In [13]: !pwd
/home/jake/projects/myproject
```

Das liegt daran, dass die Shell-Befehle in einem Notebook in einer temporären Subshell ausgeführt werden. Wenn Sie das Arbeitsverzeichnis dauerhaft ändern möchten, steht Ihnen dafür der magische Befehl `%cd` zur Verfügung:

```
In [14]: %cd ..
/home/jake/projects
```

Tatsächlich können Sie den Befehl standardmäßig sogar ohne das `%`-Zeichen aufrufen:

```
In [15]: cd myproject
/home/jake/projects/myproject
```

Man spricht hier von einer `automagic`-Funktion, deren Verhalten mit der Funktion `%automagic` geändert werden kann.

Neben `%cd` gibt es für die Shell noch die magischen Funktionen/Befehle `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm` und `%rmdir`, die alle auch ohne das `%`-Zeichen ver-

wendbar sind, sofern `automagic` eingeschaltet ist. Auf diese Weise können Sie die Kommandozeile in IPython fast wie eine normale Shell verwenden:

```
In [16]: mkdir tmp
In [17]: ls
myproject.txt tmp/
In [18]: cp myproject.txt tmp/
In [19]: ls tmp
myproject.txt
In [20]: rm -r tmp
```

Dieser Zugriff auf die Shell im selben Terminalfenster, in dem Ihre Python-Sitzung läuft, bedeutet für Sie beim Schreiben von Python-Code, dass Sie sehr viel weniger vom Interpreter zur Shell und wieder zurück wechseln müssen.

## 1.8 Fehler und Debugging

Bei der Entwicklung von Code und der Datenanalyse spielen Versuch und Irrtum auch immer eine gewisse Rolle. IPython bringt einige Tools mit, um diesen Vorgang zu optimieren. In diesem Abschnitt werden wir uns kurz mit einigen Optionen zur Konfiguration der Fehlerberichterstattung (Exceptions) in IPython befassen. Anschließend erkunden wir die Tools für das Debuggen von Fehlern im Code.

### 1.8.1 Exceptions handhaben: `%xmode`

Wenn ein Python-Skript fehlschlägt, wird in den meisten Fällen eine *Exception* ausgelöst. Trifft der Interpreter auf eine solche Exception, finden sich Informationen über die Fehlerursache im sogenannten *Traceback*, auf das Sie von Python aus zugreifen können. Mit der magischen Funktion `%xmode` erhalten Sie von IPython die Möglichkeit, den Umfang der Informationen festzulegen, die ausgegeben werden, wenn eine Exception ausgelöst wird. Betrachten Sie den folgenden Code:

```
In[1]: def func1(a, b):
        return a / b

        def func2(x):
            a = x
            b = x-1
            return func1(a, b)
In[2]: func2(1)
-----
ZeroDivisionError          Traceback (most recent call last)

<ipython-input-2-b2e110f6fc8f> in <module>()
----> 1 func2(1)
```

```
<ipython-input-1-d849e34d61fb> in func2(x)
      5     a = x
      6     b = x-1
----> 7     return func1(a, b)

<ipython-input-1-d849e34d61fb> in func1(a, b)
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero
```

Der Aufruf von `func2` verursacht einen Fehler, und im `Traceback` können wir genau sehen, was passiert ist. Standardmäßig enthält das `Traceback` einige Zeilen, die den Kontext der einzelnen Schritte zeigen, die zu dem Fehler geführt haben. Mit der magischen Funktion `%xmode` (kurz für *exception mode*) können wir ändern, welche Informationen ausgegeben werden.

Die `%xmode`-Funktion nimmt ein einzelnes Argument entgegen, den Modus, für den es drei mögliche Werte gibt: `Plain`, `Context` und `Verbose`. Voreingestellt ist der Modus `Context`, der zu der obigen Ausgabe führt. Der Modus `Plain` sorgt für eine kompaktere Ausgabe und liefert weniger Informationen:

```
In[3]: %xmode Plain
Exception reporting mode: Plain
In[4]: func2(1)
-----
Traceback (most recent call last):

  File "<ipython-input-4-b2e110f6fc8f>", line 1, in <module>
    func2(1)

  File "<ipython-input-1-d849e34d61fb>", line 7, in func2
    return func1(a, b)

  File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero
```

Im Modus `Verbose` werden einige zusätzliche Informationen ausgegeben, unter anderem die Argumente aller aufgerufenen Funktionen:

```

In[5]: %xmode Verbose
Exception reporting mode: Verbose
In[6]: func2(1)
-----
ZeroDivisionError                                Traceback (most recent call last)

<ipython-input-6-b2e110f6fc8f> in <module>()
----> 1 func2(1)
      global func2 = <function func2 at 0x103729320>
<ipython-input-1-d849e34d61fb> in func2(x=1)
      5     a = x
      6     b = x-1
----> 7     return func1(a, b)
      global func1 = <function func1 at 0x1037294d0>
      a = 1
      b = 0

<ipython-input-1-d849e34d61fb> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a / b
      a = 1
      b = 0
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero

```

Diese zusätzlichen Informationen können Ihnen dabei helfen einzugrenzen, warum eine Exception ausgelöst wird. Warum also nicht immer den **Verbose**-Modus nutzen? Wenn der Code komplizierter ist, kann diese Art des Tracebacks extrem umfangreich werden. Je nach Kontext lässt es sich manchmal besser mit der Knappheit des voreingestellten Modus arbeiten.

## 1.8.2 Debugging: Wenn das Lesen von Tracebacks nicht ausreicht

Das Standardtool für interaktives Debuggen ist `pdb`, der Python-Debugger. Mit diesem Debugger kann der User den Code Zeile für Zeile durchlaufen, um zu prüfen, was einen schwer zu findenden Fehler verursachen könnte. Die erweiterte IPython-Version heißt `ipdb`, das ist der IPython-Debugger.

Es gibt viele verschiedene Möglichkeiten, diese beiden Debugger zu starten und zu nutzen, die wir an dieser Stelle jedoch nicht vollständig abhandeln werden. Nutzen Sie die Online-dokumentationen dieser beiden Hilfsprogramme, wenn Sie mehr erfahren möchten.

In IPython ist der magische Befehl `%debug` wohl die komfortabelste Debugging-Schnittstelle. Wenn Sie ihn aufrufen, nachdem eine Exception ausgelöst wurde, wird automatisch eine interaktive Kommandozeile geöffnet, und zwar an der Stelle, an der die Exception auf-

getreten ist. Mit der `ipdb`-Kommandozeile können Sie den aktuellen Zustand des Stacks untersuchen, die Werte der verfügbaren Variablen anzeigen und sogar Python-Befehle ausführen!

Sehen wir uns also die letzte Exception einmal etwas genauer an. Wir führen einige grundlegende Aufgaben aus, nämlich die Ausgabe der Werte von `a` und `b`, und beenden die Debugging-Sitzung anschließend durch Eingabe von **quit**.

```
In[7]: %debug
> <ipython-input-1-d849e34d61fb>(2) func1()
      1 def func1(a, b):
----> 2     return a / b
      3

ipdb> print(a)
1
ipdb> print(b)
0
ipdb> quit
```

Der interaktive Debugger kann jedoch viel mehr als das – wir können sogar im Stack hinauf- und herabsteigen und die Werte der dort verfügbaren Variablen untersuchen:

```
In[8]: %debug
> <ipython-input-1-d849e34d61fb>(2) func1()
      1 def func1(a, b):
----> 2     return a / b
      3

ipdb> up
> <ipython-input-1-d849e34d61fb>(7) func2()
      5     a = x
      6     b = x-1
----> 7     return func1(a, b)

ipdb> print(x)
1
ipdb> up
> <ipython-input-6-b2e110f6fc8f>(1)<module>()
----> 1 func2(1)

ipdb> down
> <ipython-input-1-d849e34d61fb>(7) func2()
      5     a = x
      6     b = x-1
```

```
----> 7     return func1(a, b)

ipdb> quit
```

Auf diese Weise können Sie nicht nur schnell herausfinden, was den Fehler verursacht hat, sondern auch, welche Funktionsaufrufe zu dem Fehler führten.

Wenn der Debugger automatisch starten soll, sobald eine Exception ausgelöst wird, nutzen Sie die magische Funktion `%pdb`, um dieses Verhalten zu aktivieren:

```
In[9]: %xmode Plain
      %pdb on
      func2(1)
Exception reporting mode: Plain
Automatic pdb calling has been turned ON

Traceback (most recent call last):

  File "<ipython-input-9-569a67d2d312>", line 3, in <module>
    func2(1)

  File "<ipython-input-1-d849e34d61fb>", line 7, in func2
    return func1(a, b)

  File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero

> <ipython-input-1-d849e34d61fb>(2)func1()
   1 def func1(a, b):
----> 2     return a / b
     3

ipdb> print(b)
0
ipdb> quit
```

Und wenn Sie ein Skript von Anfang an im interaktiven Modus ausführen möchten, starten Sie es mit dem Befehl `%run -d` und können dann mit dem Befehl `next` die Codezeilen schrittweise interaktiv durchlaufen.

### Eine (unvollständige) Liste der Debugging-Befehle

Es gibt eine Vielzahl weiterer Befehle für das interaktive Debugging. Tabelle 1.5 enthält eine kurze Beschreibung einiger der gebräuchlicheren Befehle.

Befehl	Beschreibung
<code>list</code>	Anzeige der aktuellen Position in der Datei.
<code>h(elp)</code>	Liste der Befehle oder Hilfe für einen bestimmten Befehl anzeigen.
<code>q(uit)</code>	Debugger und Programm beenden.
<code>c(ontinue)</code>	Den Debugger beenden und das Programm weiter ausführen.
<code>n(ext)</code>	Mit dem nächsten Schritt des Programms fortfahren.
<code>&lt;enter&gt;</code>	Den vorherigen Befehl wiederholen.
<code>p(rint)</code>	Variablen ausgeben.
<code>s(tep)</code>	In eine Subroutine springen.
<code>r(eturn)</code>	Aus einer Subroutine zurückkehren.

Tabelle 1.5: Debugging-Befehle

Verwenden Sie den `help`-Befehl im Debugger, um weitere Informationen aufzurufen, oder werfen Sie einen Blick in die Onlinedokumentation (<https://github.com/gotcha/ipdb>).

## 1.9 Profiling und Timing von Code

Bei der Entwicklung des Codes und der Erstellung von Datenverarbeitungspipelines muss man sich häufig zwischen verschiedenen Implementierungen entscheiden. In der Frühphase der Entwicklung eines Algorithmus kann es jedoch kontraproduktiv sein, sich darüber schon Gedanken zu machen. Oder wie Donald Knuth bekanntermaßen geistreich anmerkte: »Wir sollten es in vielleicht 97% aller Fälle bleiben lassen, uns mit winzigen Verbesserungen zu befassen: Verfrühte Optimierung ist die Wurzel allen Übels.«

Sobald der Code jedoch funktioniert, kann es durchaus sinnvoll sein, die Effizienz zu überprüfen. Manchmal erweist es sich als nützlich, die Ausführungszeit eines bestimmten Befehls oder einer Befehlsfolge zu messen. In anderen Fällen ist es hilfreich, mehrzeilige Codeabschnitte zu untersuchen und herauszufinden, an welcher Stelle es in einer komplizierten Abfolge von Operationen zu einem Engpass kommt. IPython bietet eine Vielzahl von Funktionen für diese Art des Timings und Profilings von Code. Im Folgenden betrachten wir die nachstehenden magischen Befehle in IPython:

- `%time`: Die Ausführungszeit einer einzelnen Anweisung messen.
- `%timeit`: Die Ausführungszeit einer einzelnen Anweisung mehrfach messen, um aussagekräftigere Ergebnisse zu erhalten.
- `%prun`: Code mit dem Profiler ausführen.
- `%lprun`: Code mit dem Profiler zeilenweise ausführen.
- `%memit`: Den Speicherbedarf einer einzelnen Anweisung messen.
- `%mprun`: Code mit dem Memory-Profiler zeilenweise ausführen.

Die letzten vier dieser Befehle sind nicht Bestandteil von IPython – Sie müssen die Erweiterungen `line_profiler` und `memory_profiler` installieren, die wir in den nächsten Abschnitten eingehender betrachten.

## 1.9.1 Timing von Codeschnipseln: %timeit und %time

In Abschnitt 1.4, »Magische Befehle in IPython«, haben Sie das Line-Magic %timeit und das Cell-Magic %%timeit bereits kennengelernt. Mit %%timeit kann die für die wiederholte Ausführung einer Anweisung erforderliche Zeit gemessen werden:

```
In[1]: %timeit sum(range(100))
100000 loops, best of 3: 1.54 µs per loop
```

Da diese Operation außerordentlich schnell ist, wiederholt %timeit sie automatisch sehr oft. Bei langsameren Befehlen passt sich %timeit an und führt weniger Wiederholungen durch:

```
In[2]: %%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
1 loops, best of 3: 407 ms per loop
```

In manchen Fällen ist eine wiederholte Ausführung nicht die beste Möglichkeit. Soll beispielsweise eine Liste sortiert werden, führt eine wiederholte Ausführung womöglich in die Irre. Das Sortieren einer vorsortierten Liste erfolgt erheblich schneller als die Sortierung einer unsortierten Liste. Durch die wiederholte Ausführung wird das Ergebnis daher verzerrt:

```
In[3]: import random
L = [random.random() for i in range(100000)]
%timeit L.sort()
100 loops, best of 3: 1.9 ms per loop
```

In diesem Fall dürfte die magische Funktion %time die bessere Wahl sein. Sie ist ebenfalls gut geeignet für länger laufende Befehle, bei denen es unwahrscheinlich ist, dass systembedingte Verzögerungen das Ergebnis beeinflussen. Messen wir doch einmal die Ausführungszeit der Sortierung einer unsortierten und einer vorsortierten Liste:

```
In[4]: import random
L = [random.random() for i in range(100000)]
print("Sortieren einer unsortierten Liste:")
%time L.sort()
Sortieren einer unsortierten Liste:
CPU times: user 40.6 ms, sys: 896 µs, total: 41.5 ms
Wall time: 41.5 ms
In[5]: print("Sortieren einer schon sortierten Liste:")
%time L.sort()
Sortieren einer schon sortierten Liste:
```



```
CPU times: user 8.18 ms, sys: 10 µs, total: 8.19 ms
Wall time: 8.24 ms
```

Beachten Sie hier, wie viel schneller das Sortieren der vorsortierten Liste erfolgt, aber auch, wie viel länger das Timing mit `%time` im Vergleich zu `%timeit` dauert. Das liegt daran, dass `%timeit` hinter den Kulissen einige clevere Dinge anstellt, die verhindern, dass Systemaufrufe dem Timing in die Quere kommen. Beispielsweise wird unterbunden, dass nicht mehr benötigte Objekte entsorgt werden (Speicherbereinigung bzw. *Garbage Collection*), ein Vorgang, der das Timing beeinträchtigen könnte. Aus diesem Grund sind die mit `%timeit` ermittelten Resultate für gewöhnlich merklich schneller als die mit `%time` gemessenen.

Die Verwendung eines doppelten `%`-Zeichens (Cell-Magic-Syntax) ermöglicht es, mit `%time` und `%timeit` die Ausführungsdauer mehrzeiliger Skripten zu messen:

```
In[6]: %%time
        total = 0
        for i in range(1000):
            for j in range(1000):
                total += i * (-1) ** j
CPU times: user 504 ms, sys: 979 µs, total: 505 ms
Wall time: 505 ms
```

Weitere Informationen über `%time` und `%timeit` sowie die dafür verfügbaren Optionen finden Sie in der IPython-Hilfe (geben Sie auf der Kommandozeile `%time?` ein).

## 1.9.2 Profiling kompletter Skripte: `%prun`

Ein Programm besteht aus vielen einzelnen Anweisungen, und manchmal ist das Timing dieser Anweisungen in einem bestimmten Kontext wichtiger als das Timing der Anweisungen an sich. Python verfügt über einen integrierten Codeprofiler (mehr dazu können Sie in der Python-Dokumentation nachlesen), allerdings bietet IPython in Form der magischen Funktion `%prun` eine sehr viel komfortablere Möglichkeit, diesen Profiler zu verwenden.

Als Beispiel definieren wir eine einfache Funktion, die einige Berechnungen ausführt:

```
In[7]: def sum_of_lists(N):
        total = 0
        for i in range(5):
            L = [j ^ (j >> i) for j in range(N)]
            total += sum(L)
        return total
```

Nun können wir `%prun` aufrufen und einen Funktionsaufruf übergeben, um die Profiling-Ergebnisse anzuzeigen:

```
In[8]: %prun sum_of_lists(1000000)
```

Im Notebook wird die Ausgabe im Pager (seitenweise Anzeige) dargestellt und sieht in etwa folgendermaßen aus:

```
14 function calls in 0.714 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      5   0.599    0.120    0.599    0.120 <ipython-input-19>:4(<listcomp>)
      5   0.064    0.013    0.064    0.013 {built-in method sum}
      1   0.036    0.036    0.699    0.699 <ipython-input-19>:1(sum_of_lists)
      1   0.014    0.014    0.714    0.714 <string>:1(<module>)
      1   0.000    0.000    0.714    0.714 {built-in method exec}
```

Das Ergebnis ist eine Tabelle, die in der Reihenfolge der Gesamtdauer der Funktionsaufrufe zeigt, wo die meiste Zeit während der Ausführung verbracht wird. In diesem Fall benötigen die Listenabstraktionen innerhalb der `sum_of_lists`-Funktion den größten Teil der Zeit. An dieser Stelle können wir uns darüber Gedanken machen, welche Änderungen wir vornehmen könnten, um die Performance des Algorithmus zu verbessern.

Weitere Informationen über `%prun` und die dafür verfügbaren Optionen finden Sie in der IPython-Hilfe (geben Sie auf der Kommandozeile `%prun?` ein).

### 1.9.3 Zeilenweises Profiling mit `%lprun`

Das Profiling der Funktionen mit `%prun` ist zwar brauchbar, aber manchmal ist es praktischer, ein zeilenweises Profiling vorzunehmen. Diese Funktionalität bringen Python und IPython nicht mit, aber es gibt ein Paket namens `line_profiler`, das über diese Fähigkeit verfügt. Verwenden Sie Pythons Paket-Tool `pip`, um das `line_profiler`-Paket zu installieren:

```
$ pip install line_profiler
```

Als Nächstes können Sie mit IPython die `line_profiler`-Erweiterung laden, die Bestandteil dieses Pakets ist:

```
In[9]: %load_ext line_profiler
```

Nun kann der Befehl `%lprun` ein zeilenweises Profiling aller Funktionen ausführen. Zu diesem Zweck müssen wir ausdrücklich festlegen, an welchen Funktionen wir interessiert sind:

```
In[10]: %lprun -f sum_of_lists sum_of_lists(5000)
```

Das Notebook übergibt wie vorhin das Ergebnis wieder dem Pager. Es sieht nun folgendermaßen aus:

```
Timer unit: 1e-06 s

Total time: 0.009382 s
File: <ipython-input-19-fa2be176cc3e>
Function: sum_of_lists at line 1
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
1					def sum_of_lists(N):
2	1	2	2.0	0.0	total = 0
3	6	8	1.3	0.1	for i in range(5):
4	5	9001	1800.2	95.9	L = [j ^ (j >> i)...
5	5	371	74.2	4.0	total += sum(L)
6	1	0	0.0	0.0	return total

Die am Anfang stehenden Informationen sind der Schlüssel für die Interpretation der Ergebnisse. Die Zeiten sind in Mikrosekunden angegeben, und es ist erkennbar, wo das Programm die meiste Zeit verbringt. Nun sind wir gegebenenfalls in der Lage, diese Informationen zu verwenden, um bestimmte Teile des Skripts zu modifizieren und es für den erwünschten Anwendungsfall leistungsfähiger zu machen.

Weitere Informationen über `%lprun` und die dafür verfügbaren Optionen finden Sie in der IPython-Hilfe (geben Sie auf der Kommandozeile `%lprun?` ein).

## 1.9.4 Profiling des Speicherbedarfs: `%memit` und `%mprun`

Ein anderer Aspekt des Profilings betrifft den Speicherbedarf einer Operation. Er lässt sich mit einer weiteren IPython-Erweiterung ermitteln, dem `memory_profiler`. Wie beim `line_profiler` muss die Erweiterung zunächst mit `pip` installiert werden:

```
$ pip install memory_profiler
```

Anschließend können wir die Erweiterung mit IPython laden:

```
In[12]: %load_ext memory_profiler
```

Die `memory_profiler`-Erweiterung bietet zwei nützliche magische Funktionen: das Magic `%memit` (das Pendant zu `%timeit` für die Messung des Speicherbedarfs) und die `%mprun`-Funktion (das Pendant zu `%lprun`). Die `%memit`-Funktion lässt sich ziemlich einfach verwenden:

```
In[13]: %memit sum_of_lists(1000000)
peak memory: 100.08 MiB, increment: 61.36 MiB
```

Diese Funktion verwendet also rund 100 MB Arbeitsspeicher.

Um den Speicherbedarf zeilenweise anzuzeigen, können wir das Magic `%mprun` einsetzen. Leider funktioniert es nur mit in separaten Modulen definierten Funktionen und nicht im Notebook selbst, daher verwenden wir zunächst einmal das `%%file`-Magic, um ein einfaches Modul namens `mprun_demo.py` zu erstellen, das die `sum_of_lists`-Funktion zum Inhalt hat und eine kleine Erweiterung enthält, die das Ergebnis des Speicher-Profilings besser veranschaulicht:

```
In[14]: %%file mprun_demo.py
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
        del L # Referenz auf L löschen
    return total

Overwriting mprun_demo.py
```

Nun können wir die neue Version dieser Funktion importieren und das zeilenweise Profiling des Speicherbedarfs starten:

```
In[15]: from mprun_demo import sum_of_lists
        %mprun -f sum_of_lists sum_of_lists(1000000)
```

Das dem Pager übergebene Ergebnis liefert eine Zusammenfassung des Speicherbedarfs der Funktion und sieht wie folgt aus:

```
Filename: ./mprun_demo.py
Line #   Mem usage   Increment   Line Contents
=====
      4    71.9 MiB      0.0 MiB      L = [j ^ (j >> i) for j in
range(N)]

Filename: ./mprun_demo.py
Line #   Mem usage   Increment   Line Contents
=====
      1     39.0 MiB      0.0 MiB      def sum_of_lists(N):
      2     39.0 MiB      0.0 MiB          total = 0
      3     46.5 MiB      7.5 MiB          for i in range(5):
      4     71.9 MiB     25.4 MiB              L = [j ^ (j >> i) for j in
range(N)]
      5     71.9 MiB      0.0 MiB              total += sum(L)
      6     46.5 MiB     -25.4 MiB              del L # Referenz auf L löschen
      7     39.1 MiB      -7.4 MiB          return total
```

Der Spalte Increment (Zunahme) können wir entnehmen, in welchem Maße die verschiedenen Zeilen den gesamten Speicherbedarf beeinflussen. Wie Sie sehen, ändert sich der Speicherbedarf beim Erstellen bzw. Löschen der Liste L um etwa 25 MB. Zusätzlich zu dem vom Python-Interpreter selbst belegten Speicherplatz werden also weitere 25 MB Arbeitsspeicher benötigt.

Weitere Informationen über `%memit` und `%mprun` sowie die dafür verfügbaren Optionen finden Sie in der IPython-Hilfe (geben Sie auf der Kommandozeile `%memit?` ein).

## 1.10 Weitere IPython-Ressourcen

In diesem Kapitel haben wir nur an der Oberfläche gekratzt, was den Einsatz von IPython zur Erledigung von Aufgaben der Data Science betrifft. In der Literatur und im Internet stehen sehr viel mehr Informationen zur Verfügung. Nachstehend sind einige Ressourcen aufgeführt, die Sie vielleicht nützlich finden.

### 1.10.1 Quellen im Internet

- **Die IPython-Website** (<http://ipython.org>): Die IPython-Website verlinkt zur Dokumentation, zu Beispielen, Tutorials und einer Vielzahl weiterer Ressourcen.
- **Die nbviewer-Website** (<http://nbviewer.ipython.org>): Diese Website zeigt statische Versionen von im Internet verfügbaren IPython-Notebooks an. Auf der Startseite finden Sie einige Beispiel-Notebooks, in denen Sie stöbern können, um zu erfahren, wofür andere User IPython verwenden!
- **Eine Auswahl interessanter IPython-Notebooks** (<http://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks/>): Die von nbviewer bereitgestellte kontinuierlich wachsende Liste interessanter Notebooks zeigt den Umfang und die Breite der numerischen Analysen, die mit IPython machbar sind. Sie finden hier alles – von kurzen Beispielen und Tutorials bis hin zu vollständig ausgearbeiteten Lehrgängen und im Notebook-Format vorliegenden Büchern.
- **Video-Tutorials**: Bei einer Suche im Internet werden Sie viele Video-Tutorials zum Thema IPython finden. Empfehlenswert sind insbesondere die Tutorials der PyCon-, SciPy- und PyData-Konferenzen von Fernando Perez und Brian Granger, die maßgeblich an der Entwicklung und Pflege von IPython und Jupyter beteiligt sind.

### 1.10.2 Bücher

- **Python for Data Analysis** (<http://bit.ly/python-for-data-analysis>): Wes McKinneys Buch enthält ein Kapitel, das die Verwendung von Python durch Data Scientists zum Thema hat. Ein großer Teil des vorgestellten Materials überschneidet sich zwar mit dem hier Erörterten, aber eine andere Perspektive einzunehmen, ist eigentlich immer hilfreich.
- **Learning IPython for Interactive Computing and Data Visualization** (<http://bit.ly/2eLCBB7>): Dieses kurze Buch von Cyrille Rossant bietet eine gute Einführung in die Verwendung von IPython zur Datenanalyse.
- **IPython Interactive Computing and Visualization Cookbook** (<http://bit.ly/2fCEtNE>): Dieses Buch, ebenfalls von Cyrille Rossant, ist eine umfangreichere und tiefer gehende Abhandlung der Verwendung von IPython in der Data Science. Trotz des Buchtitels geht es nicht nur um Python – das Buch befasst sich mit einem breiten Spektrum von Themen der Data Science.

Zu guter Letzt können Sie natürlich auch auf eigene Faust Hilfe finden: Das in Abschnitt 1.2, »Hilfe und Dokumentation in IPython«, beschriebene Hilfesystem in IPython kann äußerst praktisch sein, sofern Sie es gründlich und regelmäßig nutzen. Wenn Sie die Beispiele in diesem Buch (oder aus anderer Quelle) durcharbeiten, können Sie es einsetzen, um sich mit all den Tools vertraut zu machen, die IPython zu bieten hat.

# Stichwortverzeichnis

- 1-zu-1-Join 170
- ^ 93
- \_ (Unterstrich) 34
- \_\_ (doppelter Unterstrich) 34
- \_\_\_ (dreifacher Unterstrich) 34
- ;(Semikolon) 35
- !(Ausrufezeichen) 37
- ? 22
- ?? 23
- .matplotlibrc-Datei 314
- .mplstyle-Datei 314
- ) 34
- @ 241
- \* 26
- & 93, 95
- %%file 48
- %%timeit 32, 45
- %automagic 38
- %cat 38
- %cd 38
- %cp 38
- %cpaste 30
- %debug 41
- %env 38
- %history 35
- %history? 35
- %load\_ext 47
- %lprun 44
- %ls 38
- %man 38
- %matplotlib 247
- %memit 44, 48
- %mkdir 38
- %more 38
- %mprun 44, 48
- %mv 38
- %paste 30
- %pdb 43
- %prun 44, 46
- %pwd 38
- %rerun 35
- %rm 38
- %rmdir 38
- %run 31, 43
- %save 35
- %time 44, 45
- %timeit 31, 44, 45
- %xmode 39
- | 93, 95
- ~ 93
- A**
- Abbildung 252
- Abgeleitetes Merkmal 402
- Absolutwert 72
- accumulate() 76
- accuracy\_score() 377
- Achsenabschnitt 375, 417
- Achsenmarkierungen 303
  - formatieren 307
- Affine Transformation 462
- aggfunc-Schlüsselwort 198
- aggregate() 191
- Aggregatfunktion 79
- Aggregation 76
- Akaikes Informationskriterium (AIC) 514
- Algebra
  - relationale 170
- alpha-Schlüsselwort 263
- Altair 245
- Anaconda 16
- A-posteriori-Wahrscheinlichkeit 409
- append() 169
- apply()- 192
- argsort 106
- Arithmetische Operatoren 71
- Array
  - Arithmetik 71
  - aufteilen 67
  - boolesches 91
  - eindimensionale Teilmenge 63
  - kopieren 64
  - mehrdimensionale Teilmenge 63
  - nulldimensionales 82
  - sortieren 104
  - strukturiertes 112
  - umformen 65
  - verketten 66
  - Werte summieren 77
  - Zeilen und Spalten 63
  - zentrieren 87
- Array-Modul 56
- arrowprops-Dictionary 301
- asfreq() 225
- at() 102
- Atom 19
- Aufteilen, Anwenden, Kombinieren 186
- Ausdruck
  - regulärer 208
- Ausführungszeit messen 31
- Ausgabe
  - Dictionary 33
  - unterdrücken 35
- Äußerer Join 177
- Äußeres Produkt → Dyadisches Produkt
- automagic 38
- ax.add\_artist() 282
- ax.plot3d() 319
- ax.plot\_trisurf() 324
- ax.scatter3d() 319
- ax.set() 260
- ax.text() 297, 298
- Axes-Objekt 290
- B**
- Bagging 454
- BaggingClassifier 454
- Bandbreite 524
- BaseEstimator-Klasse 529
- Basemap 326
  - Bildauflösung 333
  - Daten anzeigen 334

- kartenspezifische Methoden 335
  - Zeichenfunktionen 332
- Bash 35
- Basisfunktion 419
  - Gauß'sche 421
  - radiale 439
- Basisvektor 465
- Bayes-Klassifikation 409
  - Gaussian naive 410
  - multinomiale naive 413
  - Vorteile 416
- Bayessches Informationskriterium (BIC) 514
- Beschriftungen verbergen 305
- Bestimmtheitsmaß 391
- Bias 390
- Bias-Varianz-Dilemma 390
- Big-O-Notation → Landau-Symbol
- Bin (Histogramm) 518
- Binning 274
- Bitweise Logikoperatoren 93
- Blickwinkel 321
- bluemarble() 332
- bmh-Stil 316
- Bogosort 105
- Bokeh 357
- Broadcasting 82
  - in der Praxis 87
  - Regeln 84
- Byte-Reihenfolge 59
- C**
  - C (Parameter) 442
  - C (Programmiersprache) 53
  - cd 37
  - Cell-Magic 29
  - ClassifierMixin-Klasse 529
  - Cluster
    - Freiheitsgrade 511
  - Clustering 360, 366
  - cmap-Argument 269
  - coef\_ 419
  - color-Schlüsselwort 254
  - columns-Attribut 123
  - Conway, Drew 13
  - copy() 64
  - CountVectorizer 403
  - cp 37
  - CPython 68
  - cross\_val\_score() 388
- cross\_validation-Modul 388
- cubehelix-Farbtabelle 286
- Cython-Projekt 69
- D**
  - dark\_background-Stil 317
  - DataFrame 117, 122
    - als Dictionary 123, 131
    - als NumPy-Array 122
    - als zweidimensionales Array 132
  - erzeugen 124
  - Indexanpassung 138
  - transponieren 133
  - DataFrame.eval() 240
  - DataFrameGroupBy-Objekt 187
  - datasets-Modul 500
  - Daten
    - fehlende vervollständigen 407
    - kategoriale 402
    - zurückgehaltene 385
  - datetime 215
  - datetime64-Datentyp 216
  - DatetimeIndex 218, 220
  - datetime-Modul 215
  - dateutil 215
  - dateutil-Modul 215
  - Debugger 41
  - DecisionTreeClassifier-Schätzer 450
  - describe() 185
  - Diagramm
    - Achsenmarkierung 303
    - anpassen 311
    - Begrenzungen 256
    - Beschriftung 258, 297
    - dreidimensionales 318
    - Effizienz 265
    - Farbe und Stil 254
    - Hintergrund 317
    - Pfeile 300
    - untergeordnetes 290
    - Zeitreihen 348
  - Diagrammmatrix 344
  - Dichtediagramm 268
  - Dichteschätzer 518
  - Dichteschätzung 511
  - DictVectorizer 402
  - Dimensionsreduktion 289, 360, 367, 381
  - dir 24
- Dispatch-Methode 189
- Distanzmatrix 475
- Distanzmetrik 526
- Docstring 22
- Doppelte Indizes 166
- Drahtgitterdiagramme 322
- drawmeridians() 332
- drawparallels() 332
- Drehmatrix 475
- Dreidimensionale Konturdiagramme 320
- Dreidimensionale Punkte und Linien 319
- Dreidimensionales Diagramm 318
- drop() 174
- dropna() 145
- dtype 60
- dtype-Schlüsselwort 57
- dunder-Methoden 25
- Dyadisches Produkt 76
- Dynamische Typisierung 52
- E**
  - echo 37
  - Eigengesichter 469
  - Einbettung
    - lokal lineare (LLE) 481
  - Emacs 19
  - EM-Algorithmus 492
  - Endianness 59, 141
  - Ensemble Learning 448
  - Ensemblemethode 448
  - Ensembleschätzer 459
  - Entscheidungsbaum 448
    - Überanpassung 452
  - Ereignisschleife 246
  - Erklärte Varianz 461
  - errorbar() 266
  - eval() 236
    - Attribute und Indizes 239
    - lokale Variablen 241
    - Operatoren 238
    - Rechenzeit 242
    - Spaltenzugriff 240
    - Speicherbedarf 242
    - Stringausdrücke 237
  - Event Loop → Ereignisschleife
  - Exception 39, 41
  - Exoplanet 183

Expectation-Maximization-Algorithmus 492  
Exponentialfunktion 73

## F

FacetGrid() 346  
Faktordiagramm 346  
Fancy Indexing 97  
    mit einfachen Indizes kombinieren 99  
    mit Maskierung kombinieren 99  
    mit Slicing kombinieren 99  
Farbauswahl 287  
Farbskala 270  
    anpassen 284  
Farbtabelle 269, 284  
    auswählen 284  
    diskrete Werte 288  
    divergente 284  
    qualitative 284  
    sequenzielle 284  
Feature Engineering → Merkmalerstellung  
Fehlende Daten 141  
Fehler  
    stetige 267  
    Visualisierung 265  
Fehlerbalken 265  
    horizontaler 266  
fig.add\_axes() 291  
fig.add\_subplot() 292  
fillna() 145  
filter() 192  
Fisher, Ronald 370  
fit\_intercept 375  
fit() 373, 530  
FiveThirtyEight-Stil 315  
Flag  
    ignore\_index 167  
    left\_index 174  
    right\_index 174  
    verify\_integrity 167  
for-Schleife 32  
freq-Argument 220  
Funktion  
    trigonometrische 72  
Funktionsaufruf 43

## G

Gamma-Funktion 74

Garbage Collection → Speichereinigung  
Gaussian-naive-Bayes-Klassifikation 410  
GaussianNB-Schätzer 411  
Gauß'sches Mixture-Modell (GMM) 379, 507  
Gaußkern 522  
Gaußprozess-Regression 267  
Gauß'sche Basisfunktion 421  
Geburtenraten 199  
Gelman, Andrew 199  
Generatives Modell 410  
Geografische Daten 326  
Gesichtserkennung 443, 534  
get\_dummies()-Methode 210  
get() 209  
Gewässergrenzen 332  
ggplot 245, 312  
ggplot-Stil 316  
Glättungsfunktion 522  
GMM → Gauß'sches Mixture-Modell (GMM)  
Gnomonische Projektion 330  
Granger, Brian 50  
grayscale-Stil 317  
grid\_search-Modul 400  
GridSearchCV 400  
GroupBy  
    Aggregation 191  
    Anwendung 192  
    Filter 191  
    Transformation 192  
GroupBy-Operation 186  
Gütefunktion 492

## H

Hard Negative Mining 540  
Hauptachse 461, 462  
Hauptkomponentenanalyse 378  
help() 22  
Hierarchische Indizierung 149  
hist() 273  
Histogram of Oriented Gradients (HOG) 533  
Histogramm 81, 102, 518  
Höhenwinkel 321  
Holdout-Daten → Zurückgehaltene Daten  
HoloViews 245  
Horizontalwinkel 321

how 147  
how-Schlüsselwort 176  
Hubble-Konstante 265  
Hunter, John 245  
Hyperparameter 374, 385  
  
**I**  
ignore\_index-Flag 167  
iloc-Attribut 130  
Imputation 402, 407  
Imputer-Klasse 408  
Index 119  
    als Array 126  
    als geordnete Menge 127  
Indexanpassung 137  
Indexer-Attribut 130  
Indexerhaltung 136  
Index-Objekt 126, 205  
IndexSlice-Objekt 159  
Indikatorvariable 210  
Indizes  
    doppelte 166  
Indizierung  
    hierarchische 149  
    partielle 157  
Innerer Join 176  
In-Objekt 33  
Integer-Variable 53  
intercept\_ 419  
ipdb (Debugger) 41  
IPython 19  
    Shell-Befehl 35  
IPython-Debugger 41  
IPython-Shell 20  
Iris-Datensammlung 264, 345, 377  
    Clustering 379  
    Dimensionalität 377  
isnull() 145  
Isoliniendiagramm → Konturdiagramm  
Isomap 483  
itemsize (Array) 61  
Iterator 388  
ix-Attribut 130

## J

jet-Farbtabelle 285  
Join  
    1-zu-1 170  
    äußerer 177  
    innerer 176  
    linker 177



- n-zu-1 171
- n-zu-n 172
- rechter 177
- join\_axes-Argument 168
- join\_axes-Parameter 168
- join() 175
- join-Parameter 168
- Jupyter-Notebook 20
- Jupyter-Projekt 19
- K**
- k nächste Nachbarn 107
- Kalenderdatum 215
- Kartenhintergrund 332
- Kartenprojektion 328
- Kartesisches Produkt 154
- Kategoriale Daten 402
- Kategoriale Merkmale 402
- KDE → Kerndichteschätzung
- Kegelprojektion 331
- Kepler-Mission 185
- Kerndichteschätzung 275, 522
  - Bandbreite 524
- Kerndichteschätzung (KDE) 518
- Kernel 439
- Kernel Density Estimation → Kerndichteschätzung
- kernel-Hyperparameter 441
- Kernel-Transformation 440
- Kernel-Trick 441
- keys-Schlüsselwort 167
- Klassifikation 360, 361
  - diskriminative 433
- k-Means-Algorithmus 490
- Knuth, Donald 44
- Konfusionsmatrix → Wahrheitsmatrix
- Konturdiagramm 268
  - gefülltes 270
- Kovarianztyp 511
- Kreuzvalidierung 387
- L**
- L1-Regularisierung 426
- L2-Regularisierung 425
- Label 360
- label-Attribut 280
- Lamberts winkeltreue Kegelprojektion 331
- Landau-Symbol 105, 111
- Landkarte 327
- LASSO-Regularisierung 426
- LaTeX 310
- latlon 334
- Leave-One-Out-Kreuzvalidierung (LOO) 389
- left\_index-Flag 174
- left\_on-Schlüsselwort 173
- Legende
  - anpassen 277
  - Elemente 279
  - mehrere 282
  - mit Punktgrößen 280
  - zweispaltige 278
- Lernkurve 396, 397
- LFW-Datenmenge 443, 483
- line\_profiler 44, 47
- Lineare Regression 373, 417
- Line-Magic 29
- linestyle-Schlüsselwort 254
- Linienbreite 262
- Liniendiagramm 251
- Linker Join 177
- lmpplot() 356
- loc-Attribut 130
- loc-Indexer 134
- Logarithmus 73
- Logikoperatoren
  - bitweise 93
- LogLocator 305
- Lokal lineare Einbettung (LLE) 481
- ls 37
- M**
- Machine Learning
  - Begriff 359
  - Kategorien 360
- Magic 29
- make-moons() 511
- Manifold Learning 473
- Mannigfaltigkeit 473
  - nichtlineare 480
- Marathonlauf 349
- Margin 434
- margins\_name-Schlüsselwort 199
- margins-Schlüsselwort 199
- Markierungen 261
- Markierungen verbergen 305
- Maskierung 90, 94
- Matplotlib
  - allgemeine Tipps 246
  - Galerie 303
- importieren 246
- objektorientierte Schnittstelle 251
- Schnittstelle im MATLAB-Stil 250
- Stil einstellen 246
- max() 78
- Maximum-Margin-Schätzer 435
- memory\_profiler 44
- Mercator-Projektion 329
- Merkmal 370
  - abgeleitetes 402
  - kategoriales 402
- Merkmalsstellung 401
- Merkmalsextraktion 533
- Merkmalsmatrix 370
- method-Argument 226
- min() 78
- MiniBatchKMeans 502
- Miniconda 16
- mkdir 37
- MNIST-Datenmenge 487
- Möbiusband 324
- Modell
  - Auswahl 389
  - Beurteilung 376
  - generatives 409
  - polynomiales 392
  - trainieren 362
- Modellkomplexität 393
- Modellparameter 362
- Modellvalidierung 385
- Modul
  - Array 56
  - cross\_validation 388
  - datasets 500
  - datetime 215
  - dateutil 215
  - grid\_search 400
  - mplot3d 319
  - mprun\_demo 48
  - reguläre Ausdrücke 208
  - style 314
- Mollweide-Projektion 330
- mplot3d-Modul 319
- mprun\_demo-Modul 48
- Multidimensionale Skalierung 475
- MultiIndex 150
  - erzeugen 153
  - Konstruktor 154
  - Sortierung 160

umordnen 159  
 Multi-Indexing 149  
 MultiIndex-Objekt 149  
 Multimenge 126  
 Multinomialverteilung 413  
 MultipleLocator() 308  
 Multiplikationstabelle 76  
 Muster 370

## N

Nachkommastelle 62  
 Namensraum 26, 37, 241  
 NaN 141, 143, 144  
 NA-Wert 141  
 nbytes (Array) 61  
 ncol() 278  
 ndarray-Objekt 56, 117  
 ndim (Array) 60  
 NetCDF-Format 337  
 newaxis 65, 86  
 Nichtlineare Einbettung 479  
 None 33, 144  
 None-Objekt 142  
 Normierung 519  
 notnull() 145  
 np.absolute 72  
 np.all() 92  
 np.any() 92  
 np.argmaxpartition() 107  
 np.concatenate 66  
 np.count\_nonzero() 92  
 np.dsplit 68  
 np.dstack 67  
 np.expm1 74  
 np.histogram() 274  
 np.histogramdd 275  
 np.hsplit 67  
 np.hstack 66  
 np.log 73  
 np.log1p 74  
 np.meshgrid 269  
 np.min-Funktion 78  
 np.partition() 107  
 np.recarray 115  
 np.sort 106  
 np.split 67  
 np.sum() 92  
 np.sum-Funktion 77  
 np.vsplit 67  
 np.vstack 66  
 np-max-Funktion 78  
 null 141  
 NullFormatter 305

Numba-Projekt 69  
 Numexpr-Paket 236  
 NumPy 51  
   Array aus Liste erzeugen 56  
   Array feststehenden Typs 55  
   Array neu erzeugen 57  
   Standarddatentypen 58  
 n-zu-1-Join 171  
 n-zu-n-Join 172

## O

Oberflächendiagramme 322  
 One-hot-Kodierung 402  
 on-Schlüsselwort 173  
 Operator  
   arithmetischer 71  
 Orthografische Projektion 330  
 out-Argument 75  
 Out-Dictionary 35  
 outer() 76  
 Out-Objekt 33

## P

pairwise\_distance 475  
 Pandas  
   DataFrame 122  
   Installation 117  
 pandas-datareader 224  
 Panel 163  
 Panel4D 163  
 Parameter C 442  
 PCA-Schätzer 461  
 pcolormesh() 338  
 pd.concat 164  
 pd.concat()-Funktion 165  
 pd.cut() 197  
 pd.date\_range() 221  
 pd.index 119  
 pd.merge() 169  
 pd.Panel 163  
 pd.Panel4D 163  
 pd.period\_range() 221  
 pd.qcut() 197  
 pd.read\_json 211  
 pd.rolling\_mean() 232  
 pd.timedelta\_range() 221  
 pd.to\_datetime()-Funktion 220  
 pdb (Debugger) 41  
 Perez, Fernando 19, 50, 245

Period 220  
 PeriodIndex 220  
 Perspektivische Projektion 330  
 Pfeile 300  
 pip 47  
 Pipeline 392, 408  
 pivot\_table() 197  
 Pivot-Tabelle 195  
   Syntax 197  
 Planetendaten 183  
 plot() 200  
 Plotly 357  
 plt.annotate() 301  
 plt.arrow() 301  
 plt.axes 290  
 plt.axis() 256  
 plt.xlabel() 271  
 plt.cm 286  
 plt.cm.get\_cmap() 288  
 plt.colorbar 270  
 plt.colorbar() 283  
 plt.contour() 269  
 plt.contourf() 270  
 plt.draw() 247  
 plt.errorbar() 266  
 plt.fill\_between 267  
 plt.FuncFormatter() 308  
 plt.gca() 250  
 plt.gcf() 250  
 plt.GridSpec() 294  
 plt.hexbin 275  
 plt.hist2d() 274  
 plt.hist() 519  
 plt.imshow() 271  
 plt.legend() 259, 277  
 plt.MaxLocator() 307  
 plt.NullFormatter() 305  
 plt.NullLocator() 305  
 plt.plot() 254  
 plt.rc() 312  
 plt.scatter 263  
 plt.show 246  
 plt.style 246  
 plt.style.available 314  
 plt.subplot 292  
 plt.subplots\_adjust() 292  
 plt.subplots() 293  
 plt.text() 297  
 Polarkoordinaten 323  
 Polynomiale Basisfunktion 420

Polynomiales Regressionsmodell 392  
 Potenz 73  
 predict\_proba() 412, 531  
 predict() 373, 376, 416  
 print 33  
 Produkt  
   kartesisches 154  
 Profiler 44, 46  
 Profiling  
   Funktionen 46  
   Speicherbedarf 48  
   zeilenweises 47  
 projection-Schlüsselwort 319  
 Projektion 328  
 Punktfarbe 263  
 Punktgröße 263  
 pwd 37  
 PyPy-Projekt 68

## Q

Quantil 81, 197  
 query() 182, 236  
   Rechenzeit 242  
   Speicherbedarf 242

## R

R (Programmiersprache) 141  
 Radiale Basisfunktion 439  
 Random Forest 448, 454  
 RandomForestClassifier 455  
 RandomForestRegressor 455  
 RandomizedPCA 470, 472  
 Rastersuche 399  
 Rauschfilter 467  
 rcParams 313  
 read\_csv()-Funktion 179  
 Rechter Join 177  
 Record-Arrays 112, 115  
 reduce() 76  
 Regentage zählen 89  
 Regression 360, 363  
   Basisfunktion 419  
   lineare 373, 417  
   Random Forest 455  
 Regulärer Ausdrück 208  
 Regularisierung 423  
 Relationale Algebra 170  
 re-Modul 208  
 resample() 225  
 Resampling 225  
 reset\_index() 162  
 reshape() 65

Rezeptdatenbank 211  
 Rezeptempfehlungssystem 213  
 Ridge-Regression 425  
 Ridge-Schätzer 425  
 right\_index-Flag 174  
 right\_on-Schlüsselwort 173  
 Robinson-Projektion 330

## S

savefig() 248  
 Schätzer  
   benutzerdefinierter 531  
 Schätzer-API 373  
 Schätzerensemble 454  
 Schleife 69  
 Schnittmenge 176  
 Scikit-Learn 369  
   Datenrepräsentierung 370  
   Schätzer-API 372  
 scipy.special 74  
 scipy-stats-Paket 275  
 Seaborn 339  
   Galerie 349  
 Seaborn-Diagrammtypen 341  
 Seaborn-Paket 183  
 Seaborn-Stil 318  
 Selection-Sort 104  
 Semikolon 35  
 Series  
   als Array 119  
   als Dictionary 120  
   erzeugen 121  
   Index 119  
 set\_index() 162  
 set() 340  
 Set-Datenstruktur 127  
 shadedrelief() 332  
 Shape 60  
 shape (Array) 60  
 sharex-Schlüsselwort 293  
 sharey-Schlüsselwort 293  
 Shell 36  
 Shell-Befehl 35  
 shift() 226  
 Sigma-Clipping 201  
 Silhouettenanalyse 495  
 Sinusoidal-Projektion 330  
 size (Array) 60  
 Skalar 82  
 Skalierung  
   multidimensionale 475

sklearn.feature\_extraction.FeatureHasher 403  
 sklearn.preprocessing.OneHotEncoder 403  
 slice() 209  
 Slice-Notation 62  
 Slicing 62  
 sns.distplot() 342  
 sns.factorplot() 348  
 sns.jointplot() 343, 347  
 sns.kdeplot() 341, 342  
 sns.pairgrid() 353  
 sns.pairplot() 345  
 sort\_index() 160  
 sort\_level() 160  
 Sortieren 105  
 Space Telescope Science Institute 245  
 SparsePCA 472  
 SpectralClustering-Schätzer 496  
 Speicherbereinigung 46  
 Stack 42  
 stack()-Methode 152  
 Steigung 417  
 Stereografische Projektion 330  
 Straffunktion 425  
 str-Attribut 205  
 Streudiagramm 260  
 Stringmethoden 206  
 String-Operationen 204  
 Stützvektor 437  
 style-Modul 314  
 Stylesheets 314  
 Suche mit Wildcards 26  
 suffixes-Schlüsselwort 178  
 sum() 77  
 Support Vector Machines 432

## T

Tab-Vervollständigung 24  
 Tastaturkürzel  
   Befehlsverlauf 28  
   navigieren 27  
   Texteingabe 27  
 Teilüberwachtes Lernen 360  
 TensorFlow 540  
 Testdatenmenge 377  
 Textklassifikation 413  
 TF-IDF-Maß 404  
 thresh 147

- Tikhonov-Regularisierung 425
- Timedelta 220
- TimedeltaIndex 220
- Timestamp 220
- Timestamp-Objekt 218
- to\_period() 220
- Traceback 39
- train\_test\_split() 377, 386
- Trainingsdatenmenge 377
- transform() 373
- Transformation 299
  - affine 462
  - inverse 463
- Transparenz 263
- Transponierung 133
- Triangulation 323
- Trigonometrische Funktionen 72
- Trigonometrische Umkehrfunktion 73
- tshift() 226
- tSNE-Algorithmus 499
- Typisierung
  - dynamische 52
- U**
- Überanpassung 391, 397
- Überwachtes Lernen 360
- UFuncs
  - binäre 70
  - spezialisierte 74
  - unäre 70
- UFuncs (universelle Funktionen) 68, 70
- unstack() 151
- Unteranpassung 390, 397
- Unterstrich 25, 34
- Unterstrich, doppelter 34
- Unterstrich, dreifacher 34
- Unüberwachtes Lernen 360
- US-Präsidenten 81
- V**
- validation\_curve 394
- Validierungskurve 391
- Validierungsmenge 387
- Variable
  - an Shell übergeben 38
  - Wert im Debugger anzeigen 42
- Varianz 391
  - erklärte 461
- Vega/Vega-Lite 358
- Vektorisierung 68, 70, 205, 402
- Verfrühte Optimierung 44
- Vergleichsoperatoren 90
- verify\_integrity-Flag 167
- Verkettung 165
- Vervollständigung fehlender Daten 407
- view\_init 321
- Violinendiagramm 354
- viridis-Farbtabelle 286
- Vispy 357
- Voreinstellungen ändern 312
- Vorhersage 362
- Vorkommenshäufigkeit 413
- W**
- Wahrheitsmatrix 383
- Weichzeichner 233
- Wickham, Hadley 186
- Wildcards 26
- Wrapper 71
- X**
- \_X (Out 34)
- Z**
- Zeichenerkennung 380
- Zeitdelta 215, 220
- Zeiteinheit 217
- Zeitintervall 215
- Zeitperiode 215, 220
- Zeitreihen 224
- Zeitstempel 215, 220
- Zielarray 371
- Ziffern
  - Klassifikation 382
- Zifferndaten 380
- Zurückgehaltene Daten 385
- Zwischenzeitanteil 352
- Zylinderprojektion 329
  - flächentreue 329
  - unechte 330