Kapitel 1

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           pyclbr
...                 ...                 ...

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

(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                  RuntimeWarning
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
Strg-A	Bewegt den Cursor an den Zeilenanfang.
Strg-E	Bewegt den Cursor an das Zeilenende.
Strg-B (oder Pfeil nach links)	Bewegt den Cursor ein Zeichen rückwärts.
Strg-F (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.
Strg-D	Zeichen rechts vom Cursor löschen.
Strg-K	Text von der Cursorposition bis zum Zeilenende ausschneiden.
Strg-U	Text vom Zeilenanfang bis zur Cursorposition ausschneiden.
Strg-Y	Zuvor ausgeschnittenen Text einfügen.
Strg-T	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
Strg-P (oder Pfeiltaste nach oben)	Vorhergehenden Befehl im Verlauf auswählen.
Strg-N (oder Pfeiltaste nach unten)	Nachfolgenden Befehl im Verlauf auswählen.
Strg-R	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
Strg-L	Terminalanzeige löschen.
Strg-C	Aktuellen Python-Befehl abbrechen.
Strg-D	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 Einrü-

ckungen 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 einzeiligen 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 Ausgabenummer) 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 Rückgabetyp 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 Hilfefunktion.

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