O'REILLY®

2nd Edition

Python Data Science Handbook

Essential Tools for Working with Data

Early Release

RAW & UNEDITED

Jake VanderPlas

Python Data Science Handbook

2ND EDITION

Essential Tools for Working with Data

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jake VanderPlas

Python Data Science Handbook

by Jake VanderPlas

Copyright © 2022 Jake VanderPlas. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman

Development Editor: Jill Leonard

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

December 2022: Second Edition

Revision History for the Early Release

- 2022-01-18: First Release
- 2022-03-29: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098121228 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Python Data Science Handbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-12116-7

[LSI]

Preface

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

What Is Data Science?

This is a book about doing data science with Python, which immediately begs the question: what is *data science*? It’s a surprisingly hard definition to nail down, especially given how ubiquitous the term has become. Vocal critics have variously dismissed the term as a superfluous label (after all, what science doesn’t involve data?) or a simple buzzword that only exists to salt resumes and catch the eye of overzealous tech recruiters.

In my mind, these critiques miss something important. Data science, despite its hype-laden veneer, is perhaps the best label we have for the cross-disciplinary set of skills that are becoming increasingly important in many applications across industry and academia. This cross-disciplinary piece is key: in my mind, the best existing definition of data science is illustrated by Drew Conway’s Data Science Venn Diagram, first published on his blog in September 2010:

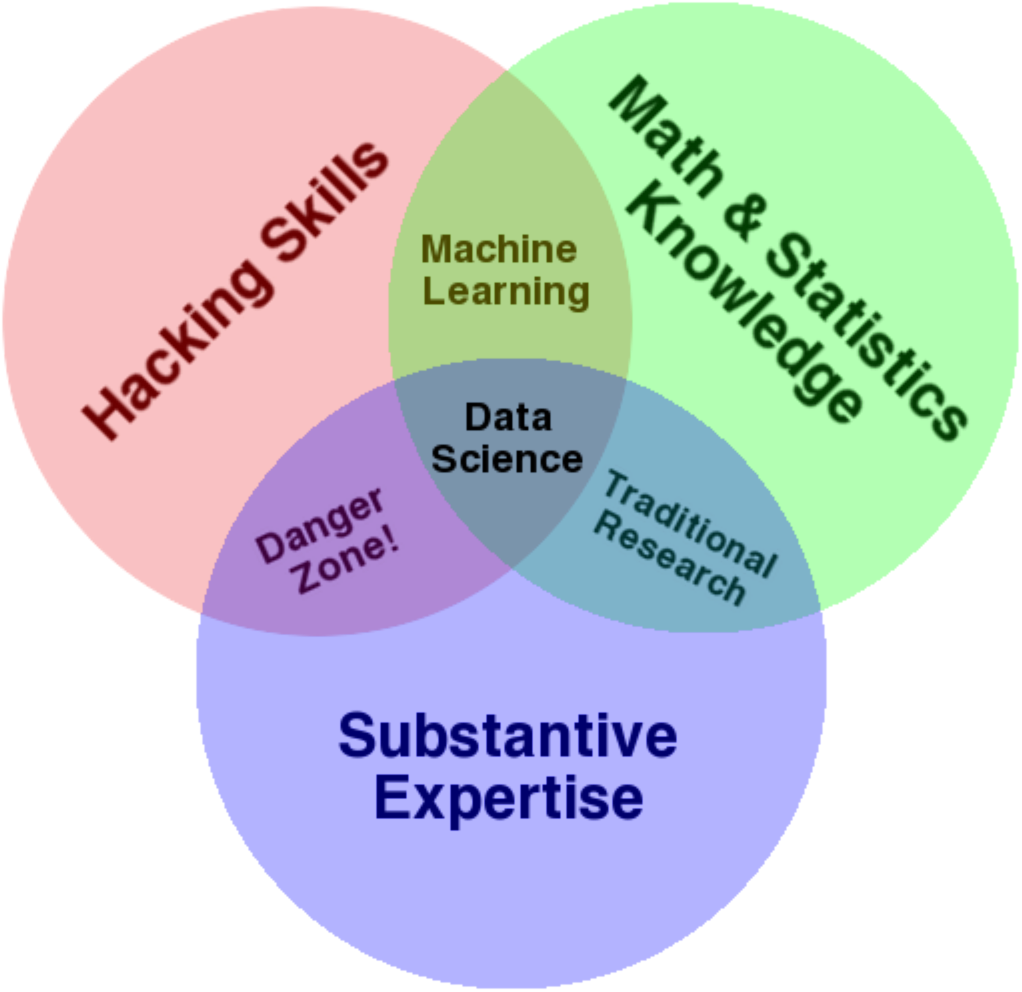


Figure P-1. Drew Conway's Data Science Venn Diagram

(Source: Drew Conway. Used by permission.)

While some of the intersection labels are a bit tongue-in-cheek, this diagram captures the essence of what I think people mean when they say “data science”: it is fundamentally an *interdisciplinary* subject. Data science comprises three distinct and overlapping areas: the skills of a *statistician* who knows how to model and summarize datasets (which are growing ever larger); the skills of a *computer scientist* who can design and use algorithms to efficiently store, process, and visualize this data; and the *domain expertise*—what we might think of as “classical” training in a subject—necessary both to formulate the right questions and to put their answers in context.

With this in mind, I would encourage you to think of data science not as a new domain of knowledge to learn, but a new set of skills that you can apply within your current area of expertise. Whether you are reporting election results, forecasting stock returns, optimizing online ad clicks, identifying microorganisms in microscope photos, seeking new classes of astronomical objects, or working with data in any other field, the goal of this book is to give you the ability to ask and answer new questions about your chosen subject area.

Who Is This Book For?

In my teaching both at the University of Washington and at various tech-focused conferences and meetups, one of the most common questions I have heard is this: “how should I learn Python?” The people asking are generally technically minded students, developers, or researchers, often with an already strong background in writing code and using computational and numerical tools. Most of these folks don’t want to learn Python *per se*, but want to learn the language with the aim of using it as a tool for data-intensive and computational science. While a large patchwork of videos, blog posts, and tutorials for this audience is available online, I’ve long been frustrated by the lack of a single good answer to this question; that is what inspired this book.

The book is not meant to be an introduction to Python or to programming in general; I assume the reader has familiarity with the Python language, including defining functions, assigning variables, calling methods of objects, controlling the flow of a program, and other basic tasks. Instead it is meant to help Python users learn to use Python’s data science stack–libraries such as IPython, NumPy, Pandas, Matplotlib, Scikit-Learn, and related tools–to effectively store, manipulate, and gain insight from data.

Why Python?

Python has emerged over the last couple decades as a first-class tool for scientific computing tasks, including the analysis and visualization of large datasets. This may have come as a surprise to early proponents of the Python language: the language itself was not specifically designed with data analysis or scientific computing in mind. The usefulness of Python for data science stems primarily from the large and active ecosystem of third-party packages: *NumPy* for manipulation of homogeneous array-based data, *Pandas* for manipulation of heterogeneous and labeled data, *SciPy* for common scientific computing tasks, *Matplotlib* for publication-quality visualizations, *IPython* for interactive execution and sharing of code, *Scikit-*

Learn for machine learning, and many more tools that will be mentioned in the following pages.

If you are looking for a guide to the Python language itself, I would suggest the sister project to this book, “https://github.com/jakevdp/WhirlwindTourOfPython[A Whirlwind Tour of the Python Language]”. This short report provides a tour of the essential features of the Python language, aimed at data scientists who already are familiar with one or more other programming languages.

Outline of the Book

Each chapter of this book focuses on a particular package or tool that contributes a fundamental piece of the Python Data Sciece story.

1. IPython and Jupyter: these packages provide the computational environment in which many Python-using data scientists work. . NumPy: this library provides the `ndarray` for efficient storage and manipulation of dense data arrays in Python.
2. Pandas: this library provides the `DataFrame` for efficient storage and manipulation of labeled/columnar data in Python.
3. Matplotlib: this library provides capabilities for a flexible range of data visualizations in Python.
4. Scikit-Learn: this library provides efficient & clean Python implementations of the most important and established machine learning algorithms.

The PyData world is certainly much larger than these five packages, and is growing every day. With this in mind, I make every attempt through these pages to provide references to other interesting efforts, projects, and packages that are pushing the boundaries of what can be done in Python. Nevertheless, these five are currently fundamental to much of the work being done in the Python data science space, and I expect they will remain important even as the ecosystem continues growing around them.

Using Code Examples

Supplemental material (code examples, figures, etc.) is available for download at *http://github.com/jakevdp/PythonDataScienceHandbook/*. This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example:

Example P-1.

The Python Data Science Handbook by Jake VanderPlas (O'Reilly). Copyright 2016 Jake VanderPlas, 978-1-491-91205-8.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Installation Considerations

Installing Python and the suite of libraries that enable scientific computing is straightforward . This section will outline some of the considerations when setting up your computer.

Though there are various ways to install Python, the one I would suggest for use in data science is the Anaconda distribution, which works similarly whether you use Windows, Linux, or Mac OS X. The Anaconda distribution comes in two flavors:

- Miniconda gives you the Python interpreter itself, along with a command-line tool called `conda` which operates as a cross-platform package manager geared toward Python packages, similar in spirit to the apt or yum tools that Linux users might be familiar with.
- Anaconda includes both Python and conda, and additionally bundles a suite of other pre-installed packages geared toward scientific computing. Because of the size of this bundle, expect the installation to consume several gigabytes of disk space.

Any of the packages included with Anaconda can also be installed manually on top of Miniconda; for this reason I suggest starting with Miniconda.

To get started, download and install the Miniconda package–make sure to choose a version with Python 3–and then install the core packages used in this book:

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

Throughout the text, we will also make use of other more specialized tools in Python's scientific ecosystem; installation is usually as easy as typing **`conda install packagename`**. For more information on conda, including information about creating and using conda environments (which I would *highly* recommend), refer to conda's online documentation.

Chapter 1. IPython: Beyond Normal Python

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. You can find preliminary code and notebook files on GitHub.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

There are many options for development environments for Python, and I’m often asked which one I use in my own work. My answer sometimes surprises people: my preferred environment is IPython plus a text editor (in my case, Emacs or VSCode depending on my mood). IPython (short for *Interactive Python*) was started in 2001 by Fernando Perez as an enhanced Python interpreter, and has since grown into a project aiming to provide, in Perez’s words, “Tools for the entire life cycle of research computing.” If Python is the engine of our data science task, you might think of IPython as the interactive control panel.

As well as being a useful interactive interface to Python, IPython also provides a number of useful syntactic additions to the language; we’ll cover the most useful of these additions here. In addition, IPython is closely tied with the Jupyter project, which provides a browser-based notebook that is useful for development, collaboration, sharing, and even publication of data science results. The IPython notebook is actually a special case of the broader Jupyter notebook structure, which encompasses notebooks for Julia, R, and other programming languages. As an example of the usefulness of the notebook format, look no further than the page you are reading: the entire manuscript for this book was composed as a set of IPython notebooks.

IPython is about using Python effectively for interactive scientific and data-intensive computing. This chapter will start by stepping through some of the IPython features that are useful to the practice of data science, focusing especially on the syntax it offers beyond the standard features of Python. Next, we will go into a bit more depth on some of the more useful "magic commands" that can speed-up common tasks in creating and using data science code. Finally, we will touch on some of the features of the notebook that make it useful in understanding data and sharing results.

Shell or Notebook?

There are two primary means of using IPython that we'll discuss in this chapter: the IPython shell and the IPython notebook. The bulk of the material in this chapter is relevant to both, and the examples will switch between them depending on what is most convenient. In the few sections that are relevant to just one or the other, we will explicitly state that fact. Before we start, some words on how to launch the IPython shell and IPython notebook.

Launching the IPython Shell

This chapter, like most of this book, is not designed to be absorbed passively. I recommend that as you read through it, you follow along and experiment with the tools and syntax we cover: the muscle-memory you build through doing this will be far more useful than the simple act of reading about it. Start by launching the IPython interpreter by typing **`ipython`** on the command-line; alternatively, if you've installed a distribution like Anaconda or EPD, there may be a launcher specific to your system (we'll discuss this more fully in "Help and Documentation in IPython").

Once you do this, you should see a prompt like the following:

```
Python 3.9.2 (v3.9.2:1a79785e3e, Feb 19 2021, 09:06:10)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.21.0 -- An enhanced Interactive Python. Type '?' for
help.

In [1]:
```

With that, you're ready to follow along.

Launching the Jupyter Notebook

The Jupyter notebook is a browser-based graphical interface to the IPython shell, and builds on it a rich set of dynamic display capabilities. As well as executing Python/IPython statements, the notebook allows the user to include formatted text, static and dynamic visualizations, mathematical equations, JavaScript widgets, and much more. Furthermore, these documents can be saved in a way that lets other people open them and execute the code on their own systems.

Though the IPython notebook is viewed and edited through your web browser window, it must connect to a running Python process in order to execute code. This process (known as a "kernel") can be started by running the following command in your system shell:

```
$ jupyter lab
```

This command will launch a local web server that will be visible to your browser. It immediately spits out a log showing what it is doing; that log will look something like this:

```
$ jupyter lab
[ServerApp] Serving notebooks from local directory:
/Users/jakevdp/PythonDataScienceHandbook
[ServerApp] Jupyter Server 1.4.1 is running at:
```

```
[ServerApp] http://localhost:8888/lab?token=dd852649
[ServerApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

Upon issuing the command, your default browser should automatically open and navigate to the listed local URL; the exact address will depend on your system. If the browser does not open automatically, you can open a window and manually open this address (*http://localhost:8888/lab/* in this example).

Help and Documentation in IPython

If you read no other section in this chapter, read this one: I find the tools discussed here to be the most transformative contributions of IPython to my daily workflow.

When a technologically-minded person is asked to help a friend, family member, or colleague with a computer problem, most of the time it’s less a matter of knowing the answer as much as knowing how to quickly find an unknown answer. In data science it’s the same: searchable web resources such as online documentation, mailing-list threads, and StackOverflow answers contain a wealth of information, even (especially?) if it is a topic you’ve found yourself searching before. Being an effective practitioner of data science is less about memorizing the tool or command you should use for every possible situation, and more about learning to effectively find the information you don’t know, whether through a web search engine or another means.

One of the most useful functions of IPython/Jupyter is to shorten the gap between the user and the type of documentation and search that will help them do their work effectively. While web searches still play a role in answering complicated questions, an amazing amount of information can be found through IPython alone. Some examples of the questions IPython can help answer in a few keystrokes:

- How do I call this function? What arguments and options does it have?

- What does the source code of this Python object look like?
- What is in this package I imported? What attributes or methods does this object have?

Here we'll discuss IPython's tools to quickly access this information, namely the `?` character to explore documentation, the `??` characters to explore source code, and the Tab key for auto-completion.

Accessing Documentation with ?

The Python language and its data science ecosystem is built with the user in mind, and one big part of that is access to documentation. Every Python object contains the reference to a string, known as a *doc string*, which in most cases will contain a concise summary of the object and how to use it. Python has a built-in `help()` function that can access this information and prints the results. For example, to see the documentation of the built-in `len` function, you can do the following:

```
In [1]: help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Depending on your interpreter, this information may be displayed as inline text, or in some separate pop-up window.

Because finding help on an object is so common and useful, IPython introduces the `?` character as a shorthand for accessing this documentation and other relevant information:

```
In [2]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

This notation works for just about anything, including object methods:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Signature: L.insert(index, object, /)
Docstring: Insert object before index.
Type:      builtin_function_or_method
```

or even objects themselves, with the documentation from their type:

```
In [5]: L?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty
list.
The argument must be an iterable if specified.
```

Importantly, this will even work for functions or other objects you create yourself! Here we'll define a small function with a docstring:

```
In [6]: def square(a):
   ....:     """Return the square of a."""
   ....:     return a ** 2
   ....:
```

Note that to create a docstring for our function, we simply placed a string literal in the first line. Because doc strings are usually multiple lines, by convention we used Python's triple-quote notation for multi-line strings.

Now we'll use the `?` mark to find this doc string:

```
In [7]: square?
Signature: square(a)
Docstring: Return the square of a.
File:      <ipython-input-6>
Type:      function
```

This quick access to documentation via docstrings is one reason you should get in the habit of always adding such inline documentation to the code you write!

Accessing Source Code with `??`

Because the Python language is so easily readable, another level of insight can usually be gained by reading the source code of the object you're curious about. IPython provides a shortcut to the source code with the double question mark (`??`):

```
In [8]: square??
Signature: square(a)
Source:
def square(a):
    """Return the square of a."""
    return a ** 2
File:      <ipython-input-6>
Type:      function
```

For simple functions like this, the double question-mark can give quick insight into the under-the-hood details.

If you play with this much, you'll notice that sometimes the `??` suffix doesn't display any source code: this is generally because the object in question is not implemented in Python, but in C or some other compiled extension language. If this is the case, the `??` suffix gives the same output as the `?` suffix. You'll find this particularly with many of Python's built-in objects and types, for example `len` from above:

```
In [9]: len??
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

Using `?` and/or `??` gives a powerful and quick interface for finding information about what any Python function or module does.

Exploring Modules with Tab-Completion

IPython’s other useful interface is the use of the tab key for auto-completion and exploration of the contents of objects, modules, and name-spaces. In the examples that follow, we’ll use `<TAB>` to indicate when the Tab key should be pressed.

Tab-completion of object contents

Every Python object has various attributes and methods associated with it. Like with the `help` function discussed before, Python has a built-in `dir` function that returns a list of these, but the tab-completion interface is much easier to use in practice. To see a list of all available attributes of an object, you can type the name of the object followed by a period (`.`) character and the Tab key:

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

To narrow-down the list, you can type the first character or several characters of the name, and the Tab key will find the matching attributes and methods:

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

In [10]: L.co<TAB>
              copy()   count()
```

If there is only a single option, pressing the Tab key will complete the line for you. For example, the following will instantly be replaced with `L.count`:

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

Though Python has no strictly-enforced distinction between public/external attributes and private/internal attributes, by convention a preceding underscore is used to denote such methods. For clarity, these private methods and special methods are omitted from the list by default, but it's possible to list them by explicitly typing the underscore:

```
In [10]: L._<TAB>
             __add__             __delattr__     __eq__
             __class__           __delitem__     __format__()
             __class_getitem__() __dir__()       __ge__
>
             __contains__        __doc__         __getattribute__
```

For brevity, we've only shown the first few columns of the output. Most of these are Python's special double-underscore methods (often nicknamed "dunder" methods).

Tab completion when importing

Tab completion is also useful when importing objects from packages. Here we'll use it to find all possible imports in the `itertools` package that start with `co`:

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

Similarly, you can use tab-completion to see which imports are available on your system (this will change depending on which third-party scripts and modules are visible to your Python session):

```
In [10]: import <TAB>
            abc                 anyio
            activate_this       appdirs
            aifc                appnope        >
            antigravity         argon2

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

Beyond tab completion: wildcard matching

Tab completion is useful if you know the first few characters of the object or attribute you're looking for, but is little help if you'd like to match characters at the middle or end of the word. For this use-case, IPython provides a means of wildcard matching for names using the `*` character.

For example, we can use this to list every object in the namespace that ends with `Warning`:

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

Notice that the `*` character matches any string, including the empty string.

Similarly, suppose we are looking for a string method that contains the word `find` somewhere in its name. We can search for it this way:

```
In [11]: str.*find*?
str.find
str.rfind
```

I find this type of flexible wildcard search can be useful for finding a particular command when getting to know a new package or reacquainting myself with a familiar one.

Keyboard Shortcuts in the IPython Shell

If you spend any amount of time on the computer, you've probably found a use for keyboard shortcuts in your workflow. Most familiar perhaps are the Cmd-C and Cmd-V (or Ctrl-C and Ctrl-V) for copying and pasting in a wide variety of programs and systems. Power-users tend to go even further:

popular text editors like Emacs, Vim, and others provide users an incredible range of operations through intricate combinations of keystrokes.

The IPython shell doesn’t go this far, but does provide a number of keyboard shortcuts for fast navigation while typing commands. These shortcuts are not in fact provided by IPython itself, but through its dependency on the GNU Readline library: as such, some of the following shortcuts may differ depending on your system configuration. Also, while some of these shortcuts do work in the browser-based notebook, this section is primarily about shortcuts in the IPython shell.

Once you get accustomed to these, they can be very useful for quickly performing certain commands without moving your hands from the “home” keyboard position. If you’re an Emacs user or if you have experience with Linux-style shells, the following will be very familiar. We’ll group these shortcuts into a few categories: *navigation shortcuts*, *text entry shortcuts*, *command history shortcuts*, and *miscellaneous shortcuts*.

Navigation shortcuts

While the use of the left and right arrow keys to move backward and forward in the line is quite obvious, there are other options that don’t require moving your hands from the “home” keyboard position:

Keystroke	Action
`Ctrl-a`	Move cursor to the beginning of the line
`Ctrl-e`	Move cursor to the end of the line
`Ctrl-b` or the left arrow key	Move cursor back one character
`Ctrl-f` or the right arrow key	Move cursor forward one character

Text Entry Shortcuts

While everyone is familiar with using the Backspace key to delete the previous character, reaching for the key often requires some minor finger gymnastics, and it only deletes a single character at a time. In IPython there are several shortcuts for removing some portion of the text you’re typing. The most immediately useful of these are the commands to delete entire lines of text. You’ll know these have become second-nature if you find yourself using a combination of Ctrl-b and Ctrl-d instead of reaching for Backspace to delete the previous character!

Keystroke	Action
Backspace key	Delete previous character in line
`Ctrl-d`	Delete next character in line
`Ctrl-k`	Cut text from cursor to end of line
`Ctrl-u`	Cut text from beginning of line to cursor
`Ctrl-y`	Yank (i.e. paste) text that was previously cut
`Ctrl-t`	Transpose (i.e., switch) previous two characters

Command History Shortcuts

Perhaps the most impactful shortcuts discussed here are the ones IPython provides for navigating the command history. This command history goes beyond your current IPython session: your entire command history is stored in a SQLite database in your IPython profile directory. The most straightforward way to access these is with the up and down arrow keys to step through the history, but other options exist as well:

Keystroke	Action
`Ctrl-p` (or the up arrow key)	Access previous command in history
`Ctrl-n` (or the down arrow key)	Access next command in history
`Ctrl-r`	Reverse-search through command history

The reverse-search can be particularly useful. Recall that in the previous section we defined a function called `square`. Let’s reverse-search our Python history from a new IPython shell and find this definition again. When you press Ctrl-r in the IPython terminal, you’ll see the following prompt:

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

If you start typing characters at this prompt, IPython will auto-fill the most recent command, if any, that matches those characters:

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

At any point, you can add more characters to refine the search, or press Ctrl-r again to search further for another command that matches the query. If you followed along in the previous section, pressing Ctrl-r twice more gives:

```
In [1]:
(reverse-i-search)`sqa': def square(a):
    """Return the square of a"""
    return a ** 2
```

Once you have found the command you're looking for, press Return and the search will end. We can then use the retrieved command, and carry-on with our session:

```
In [1]: def square(a):
    """Return the square of a"""
    return a ** 2

In [2]: square(2)
Out[2]: 4
```

Note that Ctrl-p/Ctrl-n or the up/down arrow keys can also be used to search through history, but only by matching characters at the beginning of the line. That is, if you type **`def`** and then press Ctrl-p, it would find the most recent command (if any) in your history that begins with the characters `def`.

Miscellaneous Shortcuts

Finally, there are a few miscellaneous shortcuts that don't fit into any of the preceding categories, but are nevertheless useful to know:

Keystroke	Action
`Ctrl-l`	Clear terminal screen
`Ctrl-c`	Interrupt current Python command
`Ctrl-d`	Exit IPython session

The Ctrl-c in particular can be useful when you inadvertently start a very long-running job.

While some of the shortcuts discussed here may seem a bit tedious at first, they quickly become automatic with practice. Once you develop that muscle memory, I suspect you will even find yourself wishing they were available in other contexts.

IPython Magic Commands

The previous two sections showed how IPython lets you use and explore Python efficiently and interactively. Here we'll begin discussing some of the enhancements that IPython adds on top of the normal Python syntax. These are known in IPython as *magic commands*, and are prefixed by the `%` character. These magic commands are designed to succinctly solve various common problems in standard data analysis. Magic commands come in two flavors: *line magics*, which are denoted by a single `%` prefix and operate on a single line of input, and *cell magics*, which are denoted by a double `%%` prefix and operate on multiple lines of input. We'll demonstrate and discuss a few brief examples here, and come back to more focused discussion of several useful magic commands later in the chapter.

Pasting Code Blocks: `%paste` and `%cpaste`

When working in the IPython interpreter, one common gotcha is that pasting multi-line code blocks can lead to unexpected errors, especially when indentation and interpreter markers are involved. A common case is that you find some example code on a website and want to paste it into your interpreter. Consider the following simple function:

~~~~
[>> DEF DONOTHING(X)]
====
...     return x
====

``` The code is formatted as it would appear in the Python
interpreter,
and if you copy and paste this directly into older IPython
versions, you
get an error:
~~~~

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

In the direct paste, the interpreter is confused by the
additional
prompt characters. But never fear-IPython's `%paste` magic
function is designed to handle this exact type of multi-line,
marked-up
input:

```ipython In [3]: %paste

[>> DEF DONOTHING(X)]
====
...     return x
====

== - End pasted text -
~~~

The %paste command both enters and executes the code, so now the function is ready to be used:

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

A command with a similar intent is %cpaste, which opens up an interactive multiline prompt in which you can paste one or more chunks of code to be executed in a batch:

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

These magic commands, like others we'll see, make available functionality that would be difficult or impossible in a standard Python interpreter.

Running External Code: `%run`

As you begin developing more extensive code, you will likely find yourself working in both IPython for interactive exploration, as well as a text editor to store code that you want to reuse. Rather than running this code in a new window, it can be convenient to run it within your IPython session. This can be done with the `%run` magic.

For example, imagine you've created a `myscript.py` file with the following contents:

```
#
------------------------------------
-
# file: myscript.py

def square(x):
    """square a number"""
    return x ** 2

for N in range(1, 4):
    print(f"{N} squared is {square(N)}")
```

You can execute this from your IPython session as follows:

```
In [6]: %run myscript.py
1 squared is 1
2 squared is 4
3 squared is 9
```

Note also that after you've run this script, any functions defined within it are available for use in your IPython session:

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

There are several options to fine-tune how your code is run; you can see the documentation in the normal way, by typing **`%run?`** in the IPython interpreter.

Timing Code Execution: `%timeit`

Another example of a useful magic function is `%timeit`, which will automatically determine the execution time of the single-line Python statement that follows it. For example, we may want to check the performance of a list comprehension:

```
In [8]: %timeit L = [n ** 2 for n in range(1000)]
430 µs ± 3.21 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

The benefit of `%timeit` is that for short commands it will automatically perform multiple runs in order to attain more robust results. For multi line statements, adding a second `%` sign will turn this into a cell magic that can handle multiple lines of input. For example, here's the equivalent construction with a `for`-loop:

```
In [9]: %%timeit
   ...: L = []
   ...: for n in range(1000):
   ...:     L.append(n ** 2)
   ...:
484 µs ± 5.67 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

We can immediately see that list comprehensions are about 10% faster than the equivalent `for`-loop construction in this case. We'll explore `%timeit` and other approaches to timing and profiling code in "Profiling and Timing Code".

Help on Magic Functions: `?`, `%magic`, and `%lsmagic`

Like normal Python functions, IPython magic functions have docstrings, and this useful documentation can be accessed in the standard manner. So, for example, to read the documentation of the `%timeit` magic simply type this:

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

Documentation for other functions can be accessed similarly. To access a general description of available magic functions, including some examples, you can type this:

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

For a quick and simple list of all available magic functions, type this:

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

Finally, I’ll mention that it is quite straightforward to define your own magic functions if you wish. We won’t discuss it here, but if you are interested, see the references listed in “More IPython Resources”.

Input and Output History

Previously we saw that the IPython shell allows you to access previous commands with the up and down arrow keys, or equivalently the Ctrl-p/Ctrl-n shortcuts. Additionally, in both the shell and the notebook, IPython exposes several ways to obtain the output of previous commands, as well as string versions of the commands themselves. We’ll explore those here.

IPython’s `In` and `Out` Objects

By now I imagine you’re becoming familiar with the `In [1]:/Out[1]:` style prompts used by IPython. But it turns out that these are not just pretty decoration: they give a clue as to how you can access previous inputs and outputs in your current session. Imagine you start a session that looks like this:

```
In [1]: import math

In [2]: math.sin(2)
Out[2]: 0.9092974268256817

In [3]: math.cos(2)
Out[3]: -0.4161468365471424
```

We've imported the built-in `math` package, then computed the sine and the cosine of the number 2. These inputs and outputs are displayed in the shell with `In`/`Out` labels, but there's more–IPython actually creates some Python variables called `In` and `Out` that are automatically updated to reflect this history:

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

In [5]: Out
Out[5]:
{2: 0.9092974268256817,
 3: -0.4161468365471424,
 4: ['', 'import math', 'math.sin(2)', 'math.cos(2)', 'In',
'Out']}
```

The `In` object is a list, which keeps track of the commands in order (the first item in the list is a place-holder so that `In[1]` can refer to the first command):

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

The `Out` object is not a list but a dictionary mapping input numbers to their outputs (if any):

```
In [7]: print(Out[2])
.9092974268256817
```

Note that not all operations have outputs: for example, import statements and print statements don't affect the output. The latter may be surprising, but makes sense if you consider that print is a function that returns None; for brevity, any command that returns None is not added to Out.

Where this can be useful is if you want to interact with past results. For example, let's check the sum of sin(2) ** 2 and cos(2) ** 2 using the previously-computed results:

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

The result is 1.0 as we'd expect from the well-known trigonometric identity. In this case, using these previous results probably is not necessary, but it can become very handy if you execute a very expensive computation and want to reuse the result!

Underscore Shortcuts and Previous Outputs

The standard Python shell contains just one simple shortcut for accessing previous output; the variable _ (i.e., a single underscore) is kept updated with the previous output; this works in IPython as well:

```
In [9]: print(_)
.0
```

But IPython takes this a bit further—you can use a double underscore to access the second-to-last output, and a triple underscore to access the third-to-last output (skipping any commands with no output):

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

In [11]: print(___)
.9092974268256817
```

IPython stops there: more than three underscores starts to get a bit hard to count, and at that point it's easier to refer to the output by line number.

There is one more shortcut we should mention, however–a shorthand for `Out[X]` is `_X` (i.e., a single underscore followed by the line number):

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

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

Suppressing Output

Sometimes you might wish to suppress the output of a statement (this is perhaps most common with the plotting commands that we'll explore in [Link to Come]). Or maybe the command you're executing produces a result that you'd prefer not like to store in your output history, perhaps so that it can be deallocated when other references are removed. The easiest way to suppress the output of a command is to add a semicolon to the end of the line:

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

The result is computed silently, and the output is neither displayed on the screen nor stored in the `Out` dictionary:

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

Related Magic Commands

For accessing a batch of previous inputs at once, the `%history` magic command is very helpful. Here is how you can print the first four inputs:

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

As usual, you can type `%history?` for more information and a description of options available. Other similar magic commands are `%rerun` (which will re-execute some portion of the command history) and `%save` (which saves some set of the command history to a file). For more information, I suggest exploring these using the `?` help functionality discussed in "Help and Documentation in IPython".

IPython and Shell Commands

When working interactively with the standard Python interpreter, one of the frustrations is the need to switch between multiple windows to access Python tools and system command-line tools. IPython bridges this gap, and gives you a syntax for executing shell commands directly from within the IPython terminal. The magic happens with the exclamation point: anything appearing after `!` on a line will be executed not by the Python kernel, but by the system command-line.

The following assumes you're on a Unix-like system, such as Linux or Mac OSX. Some of the examples that follow will fail on Windows, which uses a different type of shell by default, though if you use the *Windows Subsystem for Linux* the examples here should run correctly. If you're unfamiliar with shell commands, I'd suggest reviewing the Shell Tutorial put together by the always excellent Software Carpentry Foundation.

Quick Introduction to the Shell

A full intro to using the shell/terminal/command-line is well beyond the scope of this chapter, but for the uninitiated we will offer a quick introduction here. The shell is a way to interact textually with your computer. Ever since the mid 1980s, when Microsoft and Apple introduced the first versions of their now ubiquitous graphical operating systems, most computer users have interacted with their operating system through familiar clicking of menus and drag-and-drop movements. But operating systems existed long before these graphical user interfaces, and were primarily controlled through sequences of text input: at the prompt, the user would type a command, and the computer would do what the user told it to. Those early prompt systems are the precursors of the shells and terminals that most data scientists still use today.

Someone unfamiliar with the shell might ask why you would bother with this, when many results can be accomplished by simply clicking on icons and menus. A shell user might reply with another question: why hunt icons and click menus when you can accomplish things much more easily by typing? While it might sound like a typical tech preference impasse, when moving beyond basic tasks it quickly becomes clear that the shell offers much more control of advanced tasks, though admittedly the learning curve can be intimidating.

As an example, here is a sample of a Linux/OSX shell session where a user explores, creates, and modifies directories and files on their system (`osx:~ $` is the prompt, and everything after the `$` sign is the typed command; text that is preceded by a # is meant just as description, rather than something you would actually type in):

```
osx:~ $ echo "hello world"             # echo is like Python's
print function
hello world

osx:~ $ pwd                            # pwd = print working
directory
/home/jake                             # this is the "path" that
we're sitting in

osx:~ $ ls                             # ls = list working
directory contents
notebooks  projects

osx:~ $ cd projects/                   # cd = change directory

osx:projects $ pwd
/home/jake/projects

osx:projects $ ls
datasci_book   mpld3   myproject.txt
```

```
osx:projects $ mkdir myproject          # mkdir = make new
directory

osx:projects $ cd myproject/

osx:myproject $ mv ../myproject.txt ./  # mv = move file. Here
we're moving the
                                        # file myproject.txt from
one directory
                                        # up (../) to the current
directory (./)
osx:myproject $ ls
myproject.txt
```

Notice that all of this is just a compact way to do familiar operations (navigating a directory structure, creating a directory, moving a file, etc.) by typing commands rather than clicking icons and menus. With just a few commands (`pwd`, `ls`, `cd`, `mkdir`, and `cp`) you can do many of the most common file operations. It's when you go beyond these basics that the shell approach becomes really powerful.

Shell Commands in IPython

Any standard shell command can be used directly in IPython by prefixing it with the `!` character. For example, the `ls`, `pwd`, and `echo` commands can be run as follows:

```
In [1]: !ls
myproject.txt

In [2]: !pwd
/home/jake/projects/myproject

In [3]: !echo "printing from the shell"
printing from the shell
```

Passing Values to and from the Shell

Shell commands can not only be called from IPython, but can also be made to interact with the IPython namespace. For example, you can save the output of any shell command to a Python list using the assignment operator:

```
In [4]: contents = !ls

In [5]: print(contents)
['myproject.txt']

In [6]: directory = !pwd

In [7]: print(directory)
['/Users/jakevdp/notebooks/tmp/myproject']
```

These results are not returned as lists, but as a special shell return type defined in IPython:

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

This looks and acts a lot like a Python list, but has additional functionality, such as the `grep` and `fields` methods and the `s`, `n`, and `p` properties that allow you to search, filter, and display the results in convenient ways. For more information on these, you can use IPython's built-in help features.

Communication in the other direction–passing Python variables into the shell–is possible using the `{varname}` syntax:

```
In [9]: message = "hello from Python"

In [10]: !echo {message}
hello from Python
```

The curly braces contain the variable name, which is replaced by the variable's contents in the shell command.

Shell-Related Magic Commands

If you play with IPython's shell commands for a while, you might notice that you cannot use `!cd` to navigate the filesystem:

```
In [11]: !pwd
/home/jake/projects/myproject
```

```
In [12]: !cd ..

In [13]: !pwd
/home/jake/projects/myproject
```

The reason is that shell commands in the notebook are executed in a temporary subshell that does not maintain state from command to command. If you'd like to change the working directory in a more enduring way, you can use the `%cd` magic command:

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

In fact, by default you can even use this without the `%` sign:

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

This is known as an `automagic` function, and the ability to execute such commands without an explicit `%` can be toggled with the `%automagic` magic function.

Besides `%cd`, other available shell-like magic functions are `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm`, and `%rmdir`, any of which can be used without the `%` sign if `automagic` is on. This makes it so that you can almost treat the IPython prompt as if it's a normal shell:

```
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
```

This access to the shell from within the same terminal window as your Python session lets you more naturally combine Python and the shell in your workflows with fewer context switches.

Errors and Debugging

Code development and data analysis always require a bit of trial and error, and IPython contains tools to streamline this process. This section will briefly cover some options for controlling Python's exception reporting, followed by exploring tools for debugging errors in code.

Controlling Exceptions: `%xmode`

Most of the time when a Python script fails, it will raise an Exception. When the interpreter hits one of these exceptions, information about the cause of the error can be found in the *traceback*, which can be accessed from within Python. With the `%xmode` magic function, IPython allows you to control the amount of information printed when the exception is raised. Consider the following code:

```
def func1(a, b):
    return a / b

def func2(x):
    a = x
    b = x - 1
    return func1(a, b)


func2(1)
```

```
ZeroDivisionError: division by zero
```

Calling `func2` results in an error, and reading the printed trace lets us see exactly what happened. In the default mode, this trace includes several lines showing the context of each step that led to the error. Using the `%xmode` magic function (short for *Exception mode*), we can change what information is printed.

`%xmode` takes a single argument, the mode, and there are three possibilities: `Plain`, `Context`, and `Verbose`. The default is `Context`, and gives output like that just shown before. `Plain` is more compact and gives less information:

```
%xmode Plain
```

```
Exception reporting mode: Plain
```

```
func2(1)
```

```
 ZeroDivisionError: division by zero
```

The `Verbose` mode adds some extra information, including the arguments to any functions that are called:

```
%xmode Verbose
```

```
Exception reporting mode: Verbose
```

```
func2(1)
```

```
 ZeroDivisionError: division by zero
```

This extra information can help narrow-in on why the exception is being raised. So why not use the `Verbose` mode all the time? As code gets complicated, this kind of traceback can get extremely long. Depending on the context, sometimes the brevity of `Plain` or `Context` mode is easier to work with.

Debugging: When Reading Tracebacks Is Not Enough

The standard Python tool for interactive debugging is `pdb`, the Python debugger. This debugger lets the user step through the code line by line in order to see what might be causing a more difficult error. The IPython-enhanced version of this is `ipdb`, the IPython debugger.

There are many ways to launch and use both these debuggers; we won't cover them fully here. Refer to the online documentation of these two utilities to learn more.

In IPython, perhaps the most convenient interface to debugging is the `%debug` magic command. If you call it after hitting an exception, it will automatically open an interactive debugging prompt at the point of the exception. The `ipdb` prompt lets you explore the current state of the stack, explore the available variables, and even run Python commands!

Let's look at the most recent exception, then do some basic tasks–print the values of `a` and `b`, and type `quit` to quit the debugging session:

```
%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

```
The interactive debugger allows much more than this, though–we
can even
step up and down through the stack and explore the values of
variables
there:

[source, python]
```

%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

```
This allows you to quickly find out not only what caused the
error, but
what function calls led up to the error.

If you'd like the debugger to launch automatically whenever
an exception is raised, you can use the `%pdb` magic function to
turn on
this automatic behavior:

[source, python]
```

%xmode Plain %pdb on func2(1)

Exception reporting mode: Plain Automatic pdb calling has been turned ON 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
```

Finally, if you have a script that you'd like to run from the beginning in interactive mode, you can run it with the command `%run -d`, and use the `next` command to step through the lines of code interactively.

Partial list of debugging commands

There are many more available commands for interactive debugging than we've listed here; the following table contains a description of some of the more common and useful ones:

Command	Description
`l(ist)`	Show the current location in the file
`h(elp)`	Show a list of commands, or find help on a specific command
`q(uit)`	Quit the debugger and the program
`c(ontinue)`	Quit the debugger, continue in the program
`n(ext)`	Go to the next step of the program
`<enter>`	Repeat the previous command
`p(rint)`	Print variables
`s(tep)`	Step into a subroutine
`r(eturn)`	Return out of a subroutine

For more information, use the `help` command in the debugger, or take a look at `ipdb`'s online documentation.

Profiling and Timing Code

In the process of developing code and creating data processing pipelines, there are often trade-offs you can make between various implementations. Early in developing your algorithm, it can be counterproductive to worry about such things. As Donald Knuth famously quipped, “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”

But once you have your code working, it can be useful to dig into its efficiency a bit. Sometimes it’s useful to check the execution time of a given command or set of commands; other times it’s useful to examine a multiline process and determine where the bottleneck lies in some complicated series of operations. IPython provides access to a wide array of functionality for this kind of timing and profiling of code. Here we’ll discuss the following IPython magic commands:

- `%time`: Time the execution of a single statement
- `%timeit`: Time repeated execution of a single statement for more accuracy
- `%prun`: Run code with the profiler
- `%lprun`: Run code with the line-by-line profiler
- `%memit`: Measure the memory use of a single statement
- `%mprun`: Run code with the line-by-line memory profiler

The last four commands are not bundled with IPython–you’ll need to get the `line_profiler` and `memory_profiler` extensions, which we will discuss in the following sections.

Timing Code Snippets: `%timeit` and `%time`

We saw the `%timeit` line-magic and `%%timeit` cell-magic in the introduction to magic functions in “IPython Magic Commands”; it can be used to time the repeated execution of snippets of code:

```
%timeit sum(range(100))
```

```
1.53 µs ± 47.8 ns per loop (mean ± std. dev. of 7 runs, 1000000
loops each)
```

Note that because this operation is so fast, `%timeit` automatically does a large number of repetitions. For slower commands, `%timeit` will automatically adjust and perform fewer repetitions:

```
%%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

```
536 ms ± 15.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

Sometimes repeating an operation is not the best option. For example, if we have a list that we'd like to sort, we might be misled by a repeated operation. Sorting a pre-sorted list is much faster than sorting an unsorted list, so the repetition will skew the result:

```
import random
L = [random.random() for i in range(100000)]
%timeit L.sort()
```

```
1.71 ms ± 334 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

For this, the `%time` magic function may be a better choice. It also is a good choice for longer-running commands, when short, system-related delays are unlikely to affect the result. Let's time the sorting of an unsorted and a presorted list:

```
import random
L = [random.random() for i in range(100000)]
print("sorting an unsorted list:")
%time L.sort()
```

```
sorting an unsorted list:
CPU times: user 31.3 ms, sys: 686 µs, total: 32 ms
Wall time: 33.3 ms
```

```
print("sorting an already sorted list:")
%time L.sort()
```

```
sorting an already sorted list:
CPU times: user 5.19 ms, sys: 268 µs, total: 5.46 ms
Wall time: 14.1 ms
```

Notice how much faster the presorted list is to sort, but notice also how much longer the timing takes with `%time` versus `%timeit`, even for the presorted list! This is a result of the fact that `%timeit` does some clever things under the hood to prevent system calls from interfering with the timing. For example, it prevents cleanup of unused Python objects (known as *garbage collection*) which might otherwise affect the timing. For this reason, `%timeit` results are usually noticeably faster than `%time` results.

For `%time` as with `%timeit`, using the double-percent-sign cell magic syntax allows timing of multiline scripts:

```
%%time
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

```
CPU times: user 655 ms, sys: 5.68 ms, total: 661 ms
Wall time: 710 ms
```

For more information on `%time` and `%timeit`, as well as their available options, use the IPython help functionality (i.e., type `%time?` at the IPython prompt).

Profiling Full Scripts: `%prun`

A program is made of many single statements, and sometimes timing these statements in context is more important than timing them on their own.

Python contains a built-in code profiler (which you can read about in the Python documentation), but IPython offers a much more convenient way to use this profiler, in the form of the magic function `%prun`.

By way of example, we'll define a simple function that does some calculations:

```
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
```

Now we can call `%prun` with a function call to see the profiled results:

```
%prun sum_of_lists(1000000)

         14 function calls in 0.932 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        5    0.808    0.162    0.808    0.162 <ipython-
 > input-7-f105717832a2>:4(<listcomp>)
        5    0.066    0.013    0.066    0.013 {built-in method
builtins.sum}
        1    0.044    0.044    0.918    0.918 <ipython-
 > input-7-f105717832a2>:1(sum_of_lists)
        1    0.014    0.014    0.932    0.932
<string>:1(<module>)
        1    0.000    0.000    0.932    0.932 {built-in method
builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable'
of
 > '_lsprof.Profiler' objects}
```

The result is a table that indicates, in order of total time on each function call, where the execution is spending the most time. In this case, the bulk of execution time is in the list comprehension inside `sum_of_lists`. From here, we could start thinking about what changes we might make to improve the performance in the algorithm.

For more information on `%prun`, as well as its available options, use the IPython help functionality (i.e., type `%prun?` at the IPython prompt).

Line-By-Line Profiling with `%lprun`

The function-by-function profiling of `%prun` is useful, but sometimes it's more convenient to have a line-by-line profile report. This is not built into Python or IPython, but there is a `line_profiler` package available for installation that can do this. Start by using Python's packaging tool, `pip`, to install the `line_profiler` package:

```
$ pip install line_profiler
```

Next, you can use IPython to load the `line_profiler` IPython extension, offered as part of this package:

```
%load_ext line_profiler
```

Now the `%lprun` command will do a line-by-line profiling of any function–in this case, we need to tell it explicitly which functions we're interested in profiling:

```
%lprun -f sum_of_lists sum_of_lists(5000)
```

```
Timer unit: 1e-06 s

Total time: 0.014803 s
File: <ipython-input-7-f105717832a2>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def
sum_of_lists(N):
     2         1          6.0      6.0      0.0      total = 0
     3         6         13.0      2.2      0.1      for i in
```

```
range(5):
     4         5      14242.0   2848.4     96.2          L = [j ^
(j >> i) for j
 > in range(N)]
     5         5        541.0    108.2      3.7          total +=
sum(L)
     6         1          1.0      1.0      0.0      return total
```

The information at the top gives us the key to reading the results: the time is reported in microseconds and we can see where the program is spending the most time. At this point, we may be able to use this information to modify aspects of the script and make it perform better for our desired use case.

For more information on `%lprun`, as well as its available options, use the IPython help functionality (i.e., type `%lprun?` at the IPython prompt).

Profiling Memory Use: `%memit` and `%mprun`

Another aspect of profiling is the amount of memory an operation uses. This can be evaluated with another IPython extension, the `memory_profiler`. As with the `line_profiler`, we start by `pip`-installing the extension:

```
$ pip install memory_profiler
```

Then we can use IPython to load the extension:

```
%load_ext memory_profiler
```

The memory profiler extension contains two useful magic functions: the `%memit` magic (which offers a memory-measuring equivalent of `%timeit`) and the `%mprun` function (which offers a memory-measuring equivalent of `%lprun`). The `%memit` function can be used rather simply:

```
%memit sum_of_lists(1000000)
```

```
peak memory: 141.70 MiB, increment: 75.65 MiB
```

We see that this function uses about 140 MB of memory.

For a line-by-line description of memory use, we can use the `%mprun` magic. Unfortunately, this magic works only for functions defined in separate modules rather than the notebook itself, so we'll start by using the `%%file` magic to create a simple module called `mprun_demo.py`, which contains our `sum_of_lists` function, with one addition that will make our memory profiling results more clear:

```
%%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 # remove reference to L
    return total
```

```
Overwriting mprun_demo.py
```

We can now import the new version of this function and run the memory line profiler:

```
from mprun_demo import sum_of_lists
%mprun -f sum_of_lists sum_of_lists(1000000)
```

```
Filename:
/Users/jakevdp/github/jakevdp/PythonDataScienceHandbook/notebooks
_v2/m
 > prun_demo.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     1     66.7 MiB     66.7 MiB           1   def
sum_of_lists(N):
     2     66.7 MiB      0.0 MiB           1       total = 0
     3     75.1 MiB      8.4 MiB           6       for i in
range(5):
     4    105.9 MiB     30.8 MiB     5000015           L = [j ^
(j >> i) for j
 > in range(N)]
     5    109.8 MiB      3.8 MiB           5           total +=
sum(L)
     6     75.1 MiB    -34.6 MiB           5           del L #
remove reference
 > to L
     7     66.9 MiB     -8.2 MiB           1       return total
```

Here the `Increment` column tells us how much each line affects the total memory budget: observe that when we create and delete the list `L`, we are adding about 30 MB of memory usage. This is on top of the background memory usage from the Python interpreter itself.

For more information on `%memit` and `%mprun`, as well as their available options, use the IPython help functionality (i.e., type `%memit?` at the IPython prompt).

More IPython Resources

In this chapter, we've just scratched the surface of using IPython to enable data science tasks. Much more information is available both in print and on the Web, and here we'll list some other resources that you may find helpful.

Web Resources

- The IPython website: The IPython website links to documentation, examples, tutorials, and a variety of other resources.

- The nbviewer website: This site shows static renderings of any IPython notebook available on the internet. The front page features some example notebooks that you can browse to see what other folks are using IPython for!

- A curated collectioin of Jupyter Notebooks: This ever-growing list of notebooks, powered by nbviewer, shows the depth and breadth of numerical analysis you can do with IPython. It includes everything from short examples and tutorials to full-blown courses and books composed in the notebook format!

- Video Tutorials: searching the Internet, you will find many video-recorded tutorials on IPython. I'd especially recommend seeking tutorials from the PyCon, SciPy, and PyData conferenes by Fernando Perez and Brian Granger, two of the primary creators and maintainers of IPython and Jupyter.

Books

- *Python for Data Analysis*: Wes McKinney's book includes a chapter that covers using IPython as a data scientist. Although much of the material overlaps what we've discussed here, another perspective is always helpful.

- *Learning IPython for Interactive Computing and Data Visualization*: This short book by Cyrille Rossant offers a good introduction to using IPython for data analysis.

- *IPython Interactive Computing and Visualization Cookbook*: Also by Cyrille Rossant, this book is a longer and more advanced treatment of using IPython for data science. Despite its name, it's not just about IPython–it also goes into some depth on a broad range of data science topics.

Finally, a reminder that you can find help on your own: IPython's `?`-based help functionality (discussed in "Help and Documentation in IPython") can be useful if you use it well and use it often. As you go through the examples here and elsewhere, this can be used to familiarize yourself with all the tools that IPython has to offer.

Chapter 2. Introduction to NumPy

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. You can find preliminary code and notebook files on GitHub.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

This chapter, along with chapter 3, outlines techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including be collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images–particularly digital images–can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data are, the first step in making it analyzable will be to transform them into arrays of numbers. (We will discuss some specific examples of this process later in [Link to Come])

For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. We'll now take a look at the specialized tools that Python has for handling such numerical arrays: the NumPy package, and the Pandas package (discussed in Chapter 3).

This chapter will cover NumPy in detail. NumPy (short for *Numerical Python*) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice outlined in the Preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you're more the do-it-yourself type, you can go to *http://www.numpy.org/* and follow the installation instructions found there. Once you do, you can import NumPy and double-check the version:

```
import numpy
numpy.__version__
```

```
'1.21.2'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import NumPy using `np` as an alias:

```
import numpy as np
```

Throughout this chapter, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

Reminder about Built-In Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the `?` character – Refer back to "Help and Documentation in IPython").

For example, to display all the contents of the numpy namespace, you can type this:

```
In [3]: np.<TAB>
```

And to display NumPy's built-in documentation, you can use this:

```
In [4]: np?
```

More detailed documentation, along with tutorials and other resources, can be found at *http://www.numpy.org*.

Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn-in by its ease of use, one piece of which is dynamic typing. While a statically-typed language like C or Java requires each variable to be explicitly declared, a dynamically-typed language like Python skips this specification. For example, in C you might specify a particular operation as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

Notice one main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This means, for example, that we can assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an integer to a string. The same thing in C would lead (depending on compiler settings) to a compilation error or other unintented consequences:

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

This sort of flexibility is one piece that makes Python and other dynamically-typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type-flexibility also points to is the fact that Python variables are more than just their value; they also contain extra information about the type of the value. We'll explore this more in the sections that follow.

A Python Integer Is More Than Just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but other information as well. For example, when we define an integer in Python, such as x = 10000, x is not just a “raw” integer. It’s actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.10 source code, we find that the integer (long) type definition effectively looks like this (once the C macros are expanded):

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

A single integer in Python 3.10 actually contains four pieces:

- ob_refcnt, a reference count that helps Python silently handle memory allocation and deallocation
- ob_type, which encodes the type of the variable
- ob_size, which specifies the size of the following data members
- ob_digit, which contains the actual integer value that we expect the Python variable to represent.

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C, as illustrated in Figure 2-1.

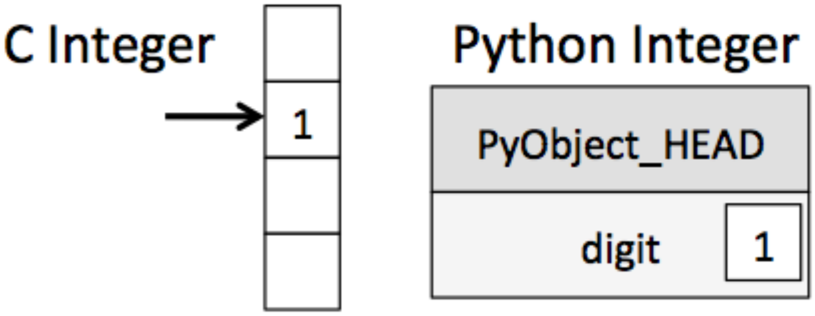


Figure 2-1. The difference between C and Python integers

Here `PyObject_HEAD` is the part of the structure containing the reference count, type code, and other pieces mentioned before.

Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value. This extra information in the Python integer structure is what allows Python to be coded so freely and dynamically. All this additional information in Python types comes at a cost, however, which becomes especially apparent in structures that combine many of these objects.

A Python List Is More Than Just a List

Let’s consider now what happens when we use a Python data structure that holds many Python objects. The standard mutable multi-element container in Python is the list. We can create a list of integers as follows:

```
L = list(range(10))
L
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
type(L[0])
```

```
int
```

Or, similarly, a list of strings:

```
L2 = [str(c) for c in L]
L2
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

```
type(L2[0])
```

```
str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
```

```
[bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information–that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in Figure 2-2.

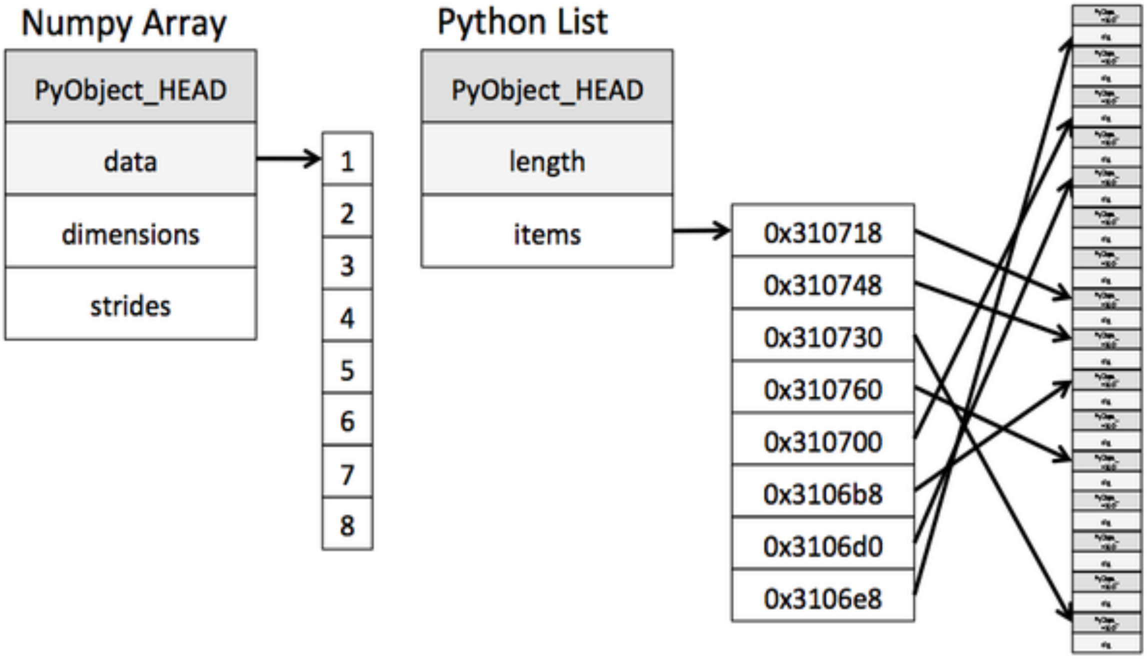


Figure 2-2. The difference between C and Python lists

At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
import array
L = list(range(10))
A = array.array('i', L)
A
```

```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here `'i'` is a type code indicating the contents are integers.

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a NumPy array.

We'll start with the standard NumPy import, under the alias `np`:

```
import numpy as np
```

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
# integer array:
np.array([1, 4, 2, 5, 3])
```

```
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
np.array([3.14, 4, 2, 3])
```

```
array([3.14, 4.  , 2.  , 3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

```
np.array([1, 2, 3, 4], dtype=np.float32)
```

```
array([1., 2., 3., 4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

```
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

```
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

```
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

```
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
# Create a 3x3 array of uniformly distributed
# pseudo-random values between 0 and 1
np.random.random((3, 3))
```

```
array([[0.09610171, 0.88193001, 0.70548015],
       [0.35885395, 0.91670468, 0.8721031 ],
       [0.73237865, 0.09708562, 0.52506779]])
```

```
# Create a 3x3 array of normally distributed pseudo-random
# values with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

```
array([[-0.46652655, -0.59158776, -1.05392451],
       [-1.72634268,  0.03194069, -0.51048869],
       [ 1.41240208,  1.77734462, -0.43820037]])
```

```
# Create a 3x3 array of pseudo-random integers in the interval
[0, 10)
np.random.randint(0, 10, (3, 3))
```

```
array([[4, 3, 8],
       [6, 5, 0],
       [1, 1, 4]])
```

```
# Create a 3x3 identity matrix
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
# Create an uninitialized array of three integers
# The values will be whatever happens to already exist at that
memory location
np.empty(3)
```

```
array([1., 1., 1.])
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.

`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

More advanced type specification is possible, such as specifying big or little endian numbers; for more information, refer to the NumPy documentation. NumPy also supports compound data types, which will be covered in "Structured Data: NumPy's Structured Arrays".

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter 3) are built around the NumPy array. This section will present several examples of using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book. Get to know them well!

We'll cover a few categories of basic array manipulations here:

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining random arrays of one, two, and three dimensions. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

```
import numpy as np
rng = np.random.default_rng(seed=1701)  # seed for
reproducibility

x1 = rng.integers(10, size=6)  # One-dimensional array
x2 = rng.integers(10, size=(3, 4))  # Two-dimensional array
x3 = rng.integers(10, size=(3, 4, 5))  # Three-dimensional array
```

Each array has attributes including `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array), and `dtype` (the type of each element);

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:   ", x3.dtype)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
dtype:    int64
```

For more discussion of `dtype`, see "Understanding Data Types in Python"):

Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

```
x1
```

```
array([9, 4, 0, 3, 8, 6])
```

```
x1[0]
```

```
9
```

```
x1[4]
```

```
8
```

To index from the end of the array, you can use negative indices:

```
x1[-1]
```

```
6
```

```
x1[-2]
```

```
8
```

In a multi-dimensional array, items can be accessed using a comma-separated `(row, column)` tuple:

```
x2
```

```
array([[3, 1, 3, 7],
       [4, 0, 2, 3],
       [0, 0, 6, 9]])
```

```
x2[0, 0]
```

```
3
```

```
x2[2, 0]
```

```
0
```

```
x2[2, -1]
```

```
9
```

Values can also be modified using any of the above index notation:

```
x2[0, 0] = 12
x2
```

```
array([[12,  1,  3,  7],
       [ 4,  0,  2,  3],
       [ 0,  0,  6,  9]])
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

```
x1[0] = 3.14159  # this will be truncated!
x1
```

```
array([3, 4, 0, 3, 8, 6])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values start=0, stop=`__`size of dimension`__`, `step=1. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

One-dimensional subarrays

```
x1
```

```
array([3, 4, 0, 3, 8, 6])
```

```
x1[:3]  # first three elements
```

```
array([3, 4, 0])
```

```
x1[3:]  # elements after index 3
```

```
array([3, 8, 6])
```

```
x1[1:4]  # middle sub-array
```

```
array([4, 0, 3])
```

```
x1[::2]  # every other element
```

```
array([3, 0, 8])
```

```
x1[1::2]  # every other element, starting at index 1
```

```
array([4, 3, 6])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
x1[::-1]  # all elements, reversed
```

```
array([6, 8, 3, 0, 4, 3])
```

```
x1[4::-2]  # reversed every other from index 4
```

```
array([8, 0, 3])
```

Multi-dimensional subarrays

Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

```
x2
```

```
array([[12,  1,  3,  7],
       [ 4,  0,  2,  3],
       [ 0,  0,  6,  9]])
```

```
x2[:2, :3]  # first two rows & three columns
```

```
array([[12,  1,  3],
       [ 4,  0,  2]])
```

```
x2[:3, ::2]  # three rows, every other column
```

```
array([[12,  3],
       [ 4,  2],
       [ 0,  6]])
```

```
x2[::-1, ::-1]  # all rows & columns, reversed
```

```
array([[ 9,  6,  0,  0],
       [ 3,  2,  0,  4],
       [ 7,  3,  1, 12]])
```

Accessing array rows and columns

One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (`:`):

```
x2[:, 0]  # first column of x2
```

```
array([12,  4,  0])
```

```
x2[0, :]  # first row of x2
```

```
array([12,  1,  3,  7])
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

```
x2[0]  # equivalent to x2[0, :]
```

```
array([12,  1,  3,  7])
```

Subarrays as no-copy views

Unlike Python list slices, NumPy array slices are returned as *views* rather than *copies* of the array data. Consider our two-dimensional array from before:

```
print(x2)
```

```
[[12  1  3  7]
 [ 4  0  2  3]
 [ 0  0  6  9]]
```

Let’s extract a 2×2 subarray from this:

```
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  1]
 [ 4  0]]
```

Now if we modify this subarray, we’ll see that the original array is changed! Observe:

```
x2_sub[0, 0] = 99
print(x2_sub)
```

```
[[99  1]
 [ 4  0]]
```

```
print(x2)
```

```
[[99  1  3  7]
 [ 4  0  2  3]
 [ 0  0  6  9]]
```

Some users may find this surprising, but it can be advantageous: for example, when working with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[99  1]
 [ 4  0]]
```

If we now modify this subarray, the original array is not touched:

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  1]
 [ 4  0]]
```

```
print(x2)
```

```
[[99  1  3  7]
 [ 4  0  2  3]
 [ 0  0  6  9]]
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays, which can be done with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
grid = np.arange(1, 10).reshape(3, 3)
print(grid)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array, and in most cases the `reshape` method will return a no-copy view of the initial array.

A common reshaping operation is converting a one-dimensional array into a two-dimensional row or column matrix:

```
x = np.array([1, 2, 3])
x.reshape((1, 3))  # row vector via reshape
```

```
array([[1, 2, 3]])
```

```
x.reshape((3, 1))  # column vector via reshape
```

```
array([[1],
       [2],
       [3]])
```

A convenient shorthand for this is to use `np.newaxis` within a slicing syntax:

```
x[np.newaxis, :]  # row vector via newaxis
```

```
array([[1, 2, 3]])
```

```
x[:, np.newaxis]  # column vector via newaxis
```

```
array([[1],
       [2],
       [3]])
```

This is a pattern that we will utilize often through the remainder of the book.

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. NumPy also provides tools to combine multiple arrays into one, and to conversely split a single array into multiple arrays.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

```
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
z = np.array([99, 99, 99])
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])

# concatenate along the first axis
np.concatenate([grid, grid])
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

```
# vertically stack the arrays
np.vstack([x, grid])
```

```
array([[1, 2, 3],
       [1, 2, 3],
       [4, 5, 6]])
```

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

```
array([[ 1,  2,  3, 99],
       [ 4,  5,  6, 99]])
```

Similary, for higher-dimensional arrays, `np.dstack` will stack arrays along the third axis.

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

Notice that *N* split-points, leads to *N* + *1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
grid = np.arange(16).reshape((4, 4))
grid
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Similarly, for higher-dimensional arrays, `np.dsplit` will split arrays along the third axis.

Computation on NumPy Arrays: Universal Functions

Up until now, we have been discussing some of the basic nuts and bolts of NumPy; in the next few sections, we will dive into the reasons that NumPy is so important in the Python data science world. Namely, it provides an easy and flexible interface to optimized computation with arrays of data.

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *universal functions* (ufuncs). This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more efficient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to the dynamic, interpreted nature of the language: the fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran. Recently there have been various attempts to address this weakness: well-known examples are the PyPy project, a just-in-time compiled implementation of Python; the Cython project, which converts Python code to compilable C code; and the Numba project, which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated – for instance looping over arrays to operate on each element. For example, imagine we have an array of values and we'd like to compute the reciprocal of each. A straightforward approach might look like this:

```
import numpy as np
rng = np.random.default_rng(seed=1701)

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

```
values = rng.integers(1, 10, size=5)
compute_reciprocals(values)
```

```
array([0.11111111, 0.25      , 1.        , 0.33333333, 0.125
])
```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! We'll benchmark this with IPython's `%timeit` magic (discussed in "Profiling and Timing Code"):

```
big_array = rng.integers(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

```
2.61 s ± 192 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

It takes several seconds to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e., billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the type-checking and function dispatches that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a *vectorized* operation. For simple operations like the element-wise division here, vectorization is as simple as using Python arithmetic operators directly on the array object. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

Compare the results of the following two:

```
print(compute_reciprocals(values))
print(1.0 / values)
```

```
[0.11111111 0.25       1.         0.33333333 0.125     ]
[0.11111111 0.25       1.         0.33333333 0.125     ]
```

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

```
%timeit (1.0 / big_array)
```

```
2.54 ms ± 383 µs per loop (mean ± std. dev. of 7 runs, 100 loops
each)
```

Vectorized operations in NumPy are implemented via *ufuncs*, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. Ufuncs are extremely flexible – before we saw an operation between a scalar and an array, but we can also operate between two arrays:

```
np.arange(5) / np.arange(1, 6)
```

```
array([0.        , 0.5       , 0.66666667, 0.75      , 0.8
])
```

And ufunc operations are not limited to one-dimensional arrays–they can also act on multi-dimensional arrays as well:

```
x = np.arange(9).reshape((3, 3))
2 ** x
```

```
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]])
```

Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented using Python loops, especially as the arrays grow in size. Any time you see such a loop in a NumPy script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We'll see examples of both these types of functions here.

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
x = np.arange(4)
print("x      =", x)
print("x + 5  =", x + 5)
print("x - 5  =", x - 5)
print("x * 2  =", x * 2)
print("x / 2  =", x / 2)
print("x // 2 =", x // 2)  # floor division
```

```
x      = [0 1 2 3]
x + 5  = [5 6 7 8]
x - 5  = [-5 -4 -3 -2]
x * 2  = [0 2 4 6]
x / 2  = [0.  0.5 1.  1.5]
x // 2 = [0 0 1 1]
```

There is also a unary ufunc for negation, and a `**` operator for exponentiation, and a `%` operator for modulus:

```
print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
```

```
-x      =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

In addition, these can be strung together however you wish, and the standard order of operations is respected:

```
-(0.5*x + 1) ** 2
```

```
array([-1.  , -2.25, -4.  , -6.25])
```

Each of these arithmetic operations are simply convenient wrappers around specific ufuncs built into NumPy; for example, the + operator is a wrapper for the `add` ufunc:

```
np.add(x, 2)
```

```
array([2, 3, 4, 5])
```

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Additionally there are Boolean/bitwise operators; we will explore these in "Comparisons, Masks, and Boolean Logic".

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function:

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

```
array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is `np.absolute`, which is also available under the alias `np.abs`:

```
np.absolute(x)
```

```
array([2, 1, 0, 1, 2])
```

```
np.abs(x)
```

```
array([2, 1, 0, 1, 2])
```

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

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

```
array([5., 5., 2., 1.])
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

```
theta      =  [0.         1.57079633 3.14159265]
sin(theta) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

```
x = [-1, 0, 1]
print("x         = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))
```

```
x         =  [-1, 0, 1]
arcsin(x) =  [-1.57079633  0.          1.57079633]
arccos(x) =  [3.14159265 1.57079633 0.        ]
arctan(x) =  [-0.78539816  0.          0.78539816]
```

Exponents and logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

```
x = [1, 2, 3]
print("x   =", x)
print("e^x =", np.exp(x))
print("2^x =", np.exp2(x))
print("3^x =", np.power(3., x))
```

```
x   = [1, 2, 3]
e^x = [ 2.71828183  7.3890561  20.08553692]
2^x = [2. 4. 8.]
3^x = [ 3.  9. 27.]
```

The inverse of the exponentials, the logarithms, are also available. The basic `np.log` gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

```
x = [1, 2, 4, 10]
print("x        =", x)
print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
x        = [1, 2, 4, 10]
ln(x)    = [0.         0.69314718 1.38629436 2.30258509]
log2(x)  = [0.         1.         2.         3.32192809]
log10(x) = [0.         0.30103    0.60205999 1.        ]
```

There are also some specialized versions that are useful for maintaining precision with very small input:

```
x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =", np.expm1(x))
print("log(1 + x) =", np.log1p(x))

exp(x) - 1 = [0.         0.0010005  0.01005017 0.10517092]
log(1 + x) = [0.         0.0009995  0.00995033 0.09531018]
```

When `x` is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were to be used.

Specialized ufuncs

NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the submodule `scipy.special`. If you want to compute some obscure mathematical function on your data, chances are it is implemented in `scipy.special`. There are far too many functions to list them all, but the following snippet shows a couple that might come up in a statistics context:

```
from scipy import special

# Gamma functions (generalized factorials) and related functions
x = [1, 5, 10]
print("gamma(x)     =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2)   =", special.beta(x, 2))

gamma(x)     = [1.0000e+00 2.4000e+01 3.6288e+05]
ln|gamma(x)| = [ 0.          3.17805383 12.80182748]
beta(x, 2)   = [0.5        0.03333333 0.00909091]
```

```
# Error function (integral of Gaussian)
# its complement, and its inverse
x = np.array([0, 0.3, 0.7, 1.0])
print("erf(x)  =", special.erf(x))
print("erfc(x) =", special.erfc(x))
print("erfinv(x) =", special.erfinv(x))
```

```
erf(x)  = [0.         0.32862676 0.67780119 0.84270079]
erfc(x) = [1.         0.67137324 0.32219881 0.15729921]
erfinv(x) = [0.         0.27246271 0.73286908        inf]
```

There are many, many more ufuncs available in both NumPy and `scipy.special`. Because the documentation of these packages is available online, a web search along the lines of “gamma function python” will generally find the relevant information.

Advanced Ufunc Features

Many NumPy users make use of ufuncs without ever learning their full set of features. We’ll outline a few specialized features of ufuncs here.

Specifying output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. Rather than creating a temporary array, this can be used to write computation results directly to the memory location where you’d like them to be. For all ufuncs, this can be done using the `out` argument of the function:

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```

```
[ 0. 10. 20. 30. 40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
```

```
[ 1.   0.   2.   0.   4.   0.   8.   0. 16.   0.]
```

If we had instead written `y[::2] = 2 ** x`, this would have resulted in the creation of a temporary array to hold the results of `2 ** x`, followed by a second operation copying those values into the `y` array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the `out` argument can be significant.

Aggregations

For binary ufuncs, there are some interesting aggregations that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```
x = np.arange(1, 6)
np.add.reduce(x)
```

```
15
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

```
np.multiply.reduce(x)
```

```
120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

```
np.add.accumulate(x)
```

```
array([ 1,  3,  6, 10, 15])
```

```
np.multiply.accumulate(x)
```

```
array([  1,   2,   6,  24, 120])
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), which we'll explore in "Aggregations: Min, Max, and Everything In Between".

Outer products

Finally, any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:

```
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

The `ufunc.at` and `ufunc.reduceat` methods are useful as well, and we will explore them in "Fancy Indexing".

We will also encounter the ability of ufuncs to operate between arrays of different shapes and sizes, a set of operations known as *broadcasting*. This subject is important enough that we will devote a whole section to it (see "Computation on Arrays: Broadcasting").

Ufuncs: Learning More

More information on universal functions (including the full list of available functions) can be found on the NumPy and SciPy documentation websites.

Recall that you can also access information directly from within IPython by importing the packages and using IPython's tab-completion and help (`?`) functionality, as described in "Help and Documentation in IPython".

Aggregations: Min, Max, and Everything In Between

A first step in exploring any dataset is often to compute various summary statistics. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset, but other aggregations are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function:

```
import numpy as np
rng = np.random.default_rng()
```

```
L = rng.random(100)
sum(L)
```

```
52.76825337322368
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same in the simplest case:

```
np.sum(L)
```

```
52.76825337322366
```

However, because it executes the operation in compiled code, NumPy's version of the operation is computed much more quickly:

```
big_array = rng.random(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
89.9 ms ± 233 µs per loop (mean ± std. dev. of 7 runs, 10 loops
each)
521 µs ± 8.37 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings, and `np.sum` is aware of multiple array dimensions, as we will see in the following section.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

```
min(big_array), max(big_array)
```

```
(2.0114398036064074e-07, 0.9999997912802653)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

```
np.min(big_array), np.max(big_array)
```

```
(2.0114398036064074e-07, 0.9999997912802653)
```

```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
72 ms ± 177 µs per loop (mean ± std. dev. of 7 runs, 10 loops
each)
564 µs ± 3.11 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
2.0114398036064074e-07 0.9999997912802653 499854.0273321711
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

Multi dimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

```
M = rng.integers(0, 10, (3, 4))
print(M)
```

```
[[0 3 1 2]
 [1 9 7 0]
 [4 8 3 7]]
```

Numpy aggregations will apply across all elements of a multi-dimensional array:

```
M.sum()
```

```
45
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

```
M.min(axis=0)
```

```
array([0, 3, 1, 0])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

```
M.max(axis=1)
```

```
array([3, 9, 8])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the first axis will be collapsed: for two-dimensional arrays, values within each column will be aggregated.

Other aggregation functions

NumPy provides several other aggregation functions with a similar API, and additionally most have a `NaN`-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point `NaN` value (see "Handling Missing Data").

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the book.

Example: What is the Average Height of US Presidents?

Aggregates available in NumPy can act as summary statistics for a set of values. As a simple example, let’s consider the heights of all US presidents.

This data is available in the file *president_heights.csv*, which is a simple comma-separated list of labels and values:

```
!head -4 data/president_heights.csv
```

```
order,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas Jefferson,189
```

We’ll use the Pandas package, which we’ll explore more fully in Chapter 3, to read the file and extract this information (note that the heights are measured in centimeters).

```
import pandas as pd
data = pd.read_csv('data/president_heights.csv')
heights = np.array(data['height(cm)'])
print(heights)
```

```
[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193
178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193
182 183
 177 185 188 188 182 185 191 182]
```

Now that we have this data array, we can compute a variety of summary statistics:

```
print("Mean height:       ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:    ", heights.min())
print("Maximum height:    ", heights.max())
```

```
Mean height:        180.04545454545453
Standard deviation: 6.983599441335736
Minimum height:     163
Maximum height:     193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

```
print("25th percentile:   ", np.percentile(heights, 25))
print("Median:            ", np.median(heights))
print("75th percentile:   ", np.percentile(heights, 75))
```

```
25th percentile:    174.75
Median:             182.0
75th percentile:    183.5
```

We see that the median height of US presidents is 182 cm, or just shy of six feet.

Of course, sometimes it’s more useful to see a visual representation of this data, which we can accomplish using tools in Matplotlib (we’ll discuss Matplotlib more fully in [Link to Come]). For example, this code generates Figure 2-3.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')

plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```

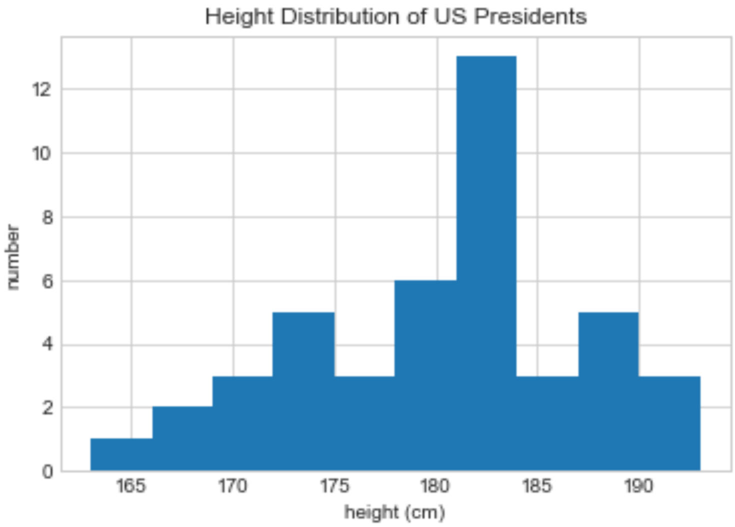


Figure 2-3. Histogram of presidential heights

Computation on Arrays: Broadcasting

We saw in a previous section how NumPy's universal functions can be used to *vectorize* operations and thereby remove slow Python loops. This section discusses *broadcasting*: a set of rules by which NumPy lets you apply binary operations (e.g., addition, subtraction, multiplication, etc.) between arrays of different sizes and shapes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

```
import numpy as np

a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b

array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different sizes–for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

```
a + 5
```

```
array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value `5` into the array `[5, 5, 5]`, and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

We can similarly extend this idea to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

```
M = np.ones((3, 3))
M
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
M + a
```

```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

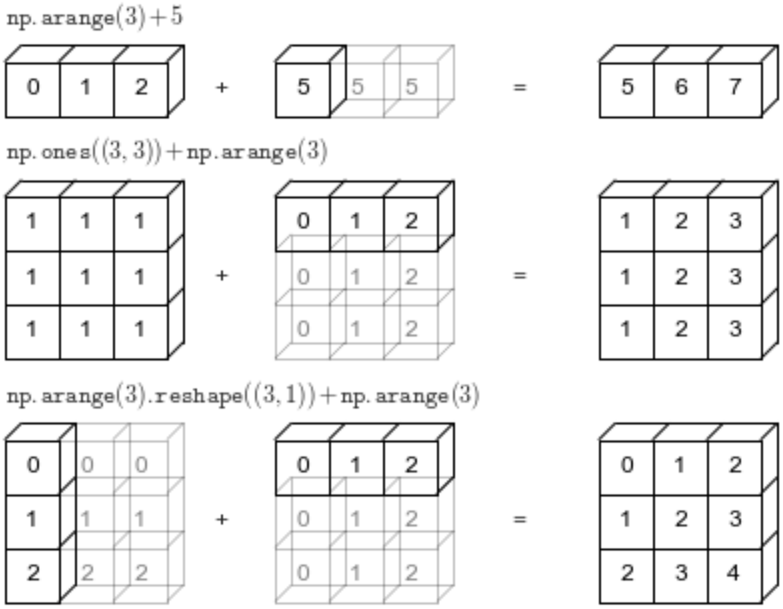
print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

```
a + b
```

```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of these examples is visualized in Figure 2-4 (Code to produce this plot can be found in the online [Link to Come], and is adapted from source published in the astroML documentation. Used by permission).



$\mathrm{np.arange}(3)+5$

$\mathrm{np.ones}((3,3))+\mathrm{np.arange}(3)$

$\mathrm{np.arange}(3).\mathrm{reshape}((3,1))+\mathrm{np.arange}(3)$

Figure 2-4. Visualization of NumPy broadcasting

The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

```
M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:

```
M + a
```

```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Broadcasting example 2

Let’s take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we’ll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`
- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Because the result matches, these shapes are compatible. We can see this here:

```
a + b
```

```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Broadcasting example 3

Now let’s take a look at an example in which the two arrays are not compatible:

```
M = np.ones((3, 2))
a = np.arange(3)
```

This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
M + a
```

```
 ValueError: operands could not be broadcast together with shapes
(3,2) (3,)
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding `a`'s shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword introduced in "The Basics of NumPy Arrays"):

```
a[:, np.newaxis].shape
```

```
(3, 1)
```

```
M + a[:, np.newaxis]
```

```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

Also notice that while we've been focusing on the + operator here, these broadcasting rules apply to *any* binary `ufunc`. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach:

```
np.logaddexp(M, a[:, np.newaxis])
```

```
array([[1.31326169, 1.31326169],
       [1.69314718, 1.69314718],
       [2.31326169, 2.31326169]])
```

For more information on the many available universal functions, refer to "Computation on NumPy Arrays: Universal Functions".

Broadcasting in Practice

Broadcasting operations form the core of many examples we'll see throughout this book. We'll now take a look at a couple simple examples of where they can be useful.

Centering an array

In the previous section, we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One commonly seen example is when centering an array of data. Imagine you have an array of 10 observations, each of which consists of 3 values. Using the standard convention (see [Link to Come]), we'll store this in a 10×3 array:

```
rng = np.random.default_rng(seed=1701)
X = rng.random((10, 3))
```

We can compute the mean of each feature using the `mean` aggregate across the first dimension:

```
Xmean = X.mean(0)
Xmean
```

```
array([0.38503638, 0.36991443, 0.63896043])
```

And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

```
X_centered = X - Xmean
```

To double-check that we've done this correctly, we can check that the centered array has near zero mean:

```
X_centered.mean(0)
```

```
array([ 4.99600361e-17, -4.44089210e-17,  0.00000000e+00])
```

To within machine precision, the mean is now zero.

Plotting a two-dimensional function

One place that broadcasting comes in handy is in displaying images based on two-dimensional functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

```
# x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

We’ll use Matplotlib to plot this two-dimensional array (these tools will be discussed in full in [Link to Come]):

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.imshow(z, origin='lower', extent=[0, 5, 0, 5])
plt.colorbar();
```

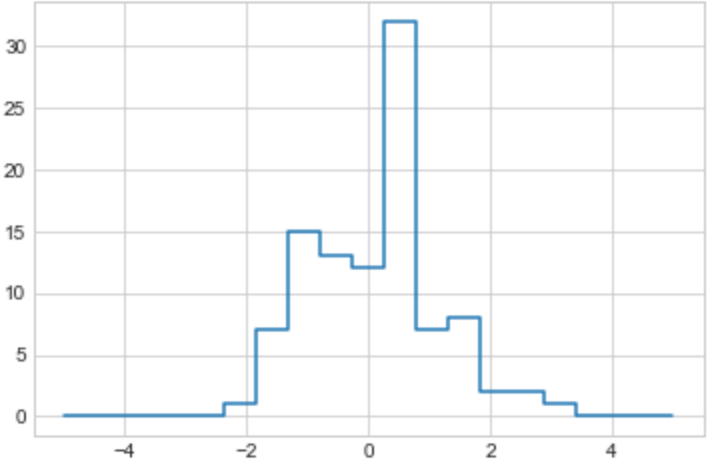


Figure 2-5. Visualization of a 2D array

The result is a compelling visualization of the two-dimensional function.

Comparisons, Masks, and Boolean Logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2015, using Pandas (see Chapter 3):

```
import numpy as np
from vega_datasets import data

# use dataframe operations to extract rainfall as a NumPy array
rainfall_mm = np.array(
    data.seattle_weather().set_index('date')['precipitation']
['2015'])
len(rainfall_mm)
```

```
365
```

The array contains 365 values, giving daily rainfall in millimeters from January 1 to December 31, 2015.

As a first quick visualization, let's look at the histogram of rainy days, which was generated using Matplotlib (we will explore this tool more fully in [Link to Come]):

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
```

```
plt.hist(rainfall_mm, 40);
```

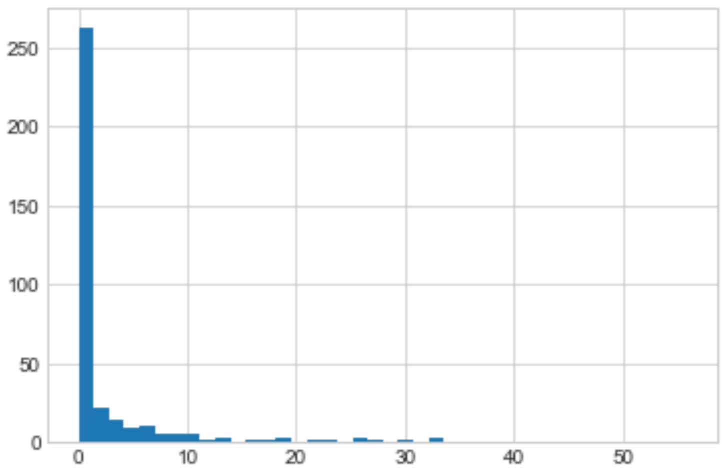


Figure 2-6. Histogram of 2014 rainfall in Seattle

This histogram gives us a general idea of what the data looks like: despite the city's rainy reputation, the vast majority of days in Seattle saw near zero measured rainfall in 2015. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What is the average precipitation on those rainy days? How many days were there with more than 10mm of rainfall?

Digging into the data

One approach to this would be to answer these questions by hand: loop through the data, incrementing a counter each time we see values in some desired range. For reasons discussed throughout this chapter, such an approach is very inefficient, both from the standpoint of time writing code and time computing the result. We saw in "Computation on NumPy Arrays: Universal Functions" that NumPy's ufuncs can be used in place of loops to do fast element-wise arithmetic operations on arrays; in the same way, we can use other ufuncs to do element-wise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for right now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

Comparison Operators as ufuncs

In “Computation on NumPy Arrays: Universal Functions” we introduced ufuncs, and focused in particular on arithmetic operators. We saw that using `+`, `-`, `*`, `/`, and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as < (less than) and > (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:

```
x = np.array([1, 2, 3, 4, 5])

x < 3  # less than

array([ True,  True, False, False, False])

x > 3  # greater than

array([False, False, False,  True,  True])

x <= 3  # less than or equal

array([ True,  True,  True, False, False])

x >= 3  # greater than or equal

array([False, False,  True,  True,  True])

x != 3  # not equal

array([ True,  True, False,  True,  True])

x == 3  # equal

array([False, False,  True, False, False])
```

It is also possible to do an element-wise comparison of two arrays, and to include compound expressions:

```
(2 * x) == (x ** 2)
```

```
array([False,  True, False, False, False])
```

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufunc is shown here:

```
 Operator | Equivalent ufunc || Operator | Equivalent ufunc |
```

|————-|——————-||————|—————| |== |np.equal ||!= |np.not_equal | |< |np.less ||<= |np.less_equal | |> |np.greater ||>= |np.greater_equal|

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```
rng = np.random.default_rng(seed=1701)
x = rng.integers(10, size=(3, 4))
x
```

```
array([[9, 4, 0, 3],
       [8, 6, 3, 1],
       [3, 7, 4, 0]])
```

```
x < 6
```

```
array([[False,  True,  True,  True],
       [False, False,  True,  True],
       [ True, False,  True,  True]])
```

In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

Working with Boolean Arrays

Given a Boolean array, there are a host of useful operations you can do. We’ll work with `x`, the two-dimensional array we created earlier.

```
print(x)
```

```
[[9 4 0 3]
 [8 6 3 1]
 [3 7 4 0]]
```

Counting entries

To count the number of `True` entries in a Boolean array, `np.count_nonzero` is useful:

```
# how many values less than 6?
np.count_nonzero(x < 6)
```

```
8
```

We see that there are eight array entries that are less than 6. Another way to get at this information is to use `np.sum`; in this case, `False` is interpreted as `0`, and `True` is interpreted as `1`:

```
np.sum(x < 6)
```

```
8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
# how many values less than 6 in each row?
np.sum(x < 6, axis=1)
```

```
array([3, 2, 3])
```

This counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any` or `np.all`:

```
# are there any values greater than 8?
np.any(x > 8)
```

```
True
```

```
# are there any values less than zero?
np.any(x < 0)
```

```
False
```

```
# are all values less than 10?
np.all(x < 10)
```

```
True
```

```
# are all values equal to 6?
np.all(x == 6)
```

```
False
```

`np.all` and `np.any` can be used along particular axes as well. For example:

```
# are all values in each row less than 8?
np.all(x < 8, axis=1)
```

```
array([False, False,  True])
```

Here all the elements in the third row is less than 8, while this is not the case for others.

Finally, a quick warning: as mentioned in "Aggregations: Min, Max, and Everything In Between", Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multidimensional arrays. Be sure that you are using `np.sum()`, `np.any()`, and `np.all()` for these examples!

Boolean operators

We've already seen how we might count, say, all days with rain less than 20mm, or all days with rain greater than 10mm. But what if we want to know about all days with rain less than 20mm **and** greater than 10mm? This is accomplished through Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`. Like with the standard arithmetic operators, NumPy overloads these as ufuncs which work element-wise on (usually Boolean) arrays.

For example, we can address this sort of compound question as follows:

```
np.sum((rainfall_mm > 10) & (rainfall_mm < 20))
```

```
16
```

So we see that there are 16 days with rainfall between 10 and 20 millimeters.

The parentheses here are important–because of operator precedence rules, with parentheses removed this expression would be evaluated as follows, which results in an error:

```
rainfall_mm > (10 & rainfall_mm) < 20
```

Let's demonstrate a more complicated expression: using De Morgan's laws, we can compute the same result in a different manner:

```
np.sum(~( (rainfall_mm <= 10) | (rainfall_mm >= 20) ))
```

```
16
```

Combining comparison operators and Boolean operators on arrays can lead to a wide range of efficient logical operations.

The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:

```
 Operator | Equivalent ufunc || Operator | Equivalent ufunc |
```

|————-|———————-||————-|———————-| |& |`np.bitwise_and` ||| |`np.bitwise_or` | |^ |`np.bitwise_xor` ||~

|np.bitwise_not|

Using these tools, we might start to answer the types of questions we have about our weather data. Here are some examples of results we can compute when combining masking with aggregations:

```
print("Number days without rain:  ", np.sum(rainfall_mm == 0))
print("Number days with rain:     ", np.sum(rainfall_mm != 0))
print("Days with more than 10 mm: ", np.sum(rainfall_mm > 10))
print("Rainy days with < 5 mm:    ", np.sum((rainfall_mm > 0) &
                                            (rainfall_mm < 5)))
```

```
Number days without rain:   221
Number days with rain:      144
Days with more than 10 mm:  34
Rainy days with < 5 mm:     83
```

Boolean Arrays as Masks

In the preceding section we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from before, suppose we want an array of all values in the array that are less than, say, 5:

```
x
```

```
array([[9, 4, 0, 3],
       [8, 6, 3, 1],
       [3, 7, 4, 0]])
```

We can obtain a Boolean array for this condition easily, as we've already seen:

```
x < 5
```

```
array([[False,  True,  True,  True],
       [False, False,  True,  True],
       [ True, False,  True,  True]])
```

Now to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

```
x[x < 5]
```

```
array([4, 0, 3, 3, 1, 3, 4, 0])
```

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is `True`.

We are then free to operate on these values as we wish. For example, we can compute some relevant statistics on our Seattle rain data:

```
# construct a mask of all rainy days
rainy = (rainfall_mm > 0)

# construct a mask of all summer days (June 21st is the 172nd
day)
days = np.arange(365)
summer = (days > 172) & (days < 262)

print("Median precip on rainy days in 2015 (mm):   ",
      np.median(rainfall_mm[rainy]))
print("Median precip on summer days in 2015 (mm):  ",
      np.median(rainfall_mm[summer]))
print("Maximum precip on summer days in 2015 (mm): ",
      np.max(rainfall_mm[summer]))
print("Median precip on non-summer rainy days (mm):",
      np.median(rainfall_mm[rainy & ~summer]))
```

```
Median precip on rainy days in 2015 (mm):    3.8
Median precip on summer days in 2015 (mm):   0.0
Maximum precip on summer days in 2015 (mm):  32.5
Median precip on non-summer rainy days (mm): 4.1
```

By combining Boolean operations, masking operations, and aggregates, we can very quickly answer these sorts of questions for our dataset.

Aside: Using the Keywords and/or Versus the Operators &/|

One common point of confusion is the difference between the keywords `and` and `or` on one hand, and the operators `&` and `|` on the other hand. When would you use one versus the other?

The difference is this: `and` and `or` gauge the truth or falsehood of *entire object*, while `&` and `|` refer to *bits within each object*.

When you use `and` or `or`, it is equivalent to asking Python to treat the object as a single Boolean entity. In Python, all nonzero integers will evaluate as True. Thus:

```
bool(42), bool(0)
```

```
(True, False)
```

```
bool(42 and 0)
```

```
False
```

```
bool(42 or 0)
```

```
True
```

When you use `&` and `|` on integers, the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
bin(42)
```

```
'0b101010'
```

```
bin(59)
```

```
'0b111011'
```

```
bin(42 & 59)
```

```
'0b101010'
```

```
bin(42 | 59)
```

```
'0b111011'
```

Notice that the corresponding bits of the binary representation are compared in order to yield the result.

When you have an array of Boolean values in NumPy, this can be thought of as a string of bits where `1 = True` and `0 = False`, and the result of `&` and `|` operates similarly to above:

```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
```

```
array([ True,  True,  True, False,  True,  True])
```

Using `or` on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
A or B
```

```
 ValueError: The truth value of an array with more than one
element is
 > ambiguous.
 a.any() or a.all()
```

Similarly, when doing a Boolean expression on a given array, you should use `|` or `&` rather than `or` or `and`:

```
x = np.arange(10)
(x > 4) & (x < 8)
```

```
array([False, False, False, False, False,  True,  True,  True,
False,
       False])
```

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError` we saw previously:

```
(x > 4) and (x < 8)
```

```
 ValueError: The truth value of an array with more than one
element is
 > ambiguous.
 a.any() or a.all()
```

So remember this: `and` and `or` perform a single Boolean evaluation on an entire object, while `&` and `|` perform multiple Boolean evaluations on the content (the individual bits or bytes) of an object. For Boolean NumPy arrays, the latter is nearly always the desired operation.

Fancy Indexing

In the previous sections, we saw how to access and modify portions of arrays using simple indices (e.g., `arr[0]`), slices (e.g., `arr[:5]`), and Boolean masks (e.g., `arr[arr > 0]`). In this section, we'll look at another style of array indexing, known as *fancy* or *vectorized* indexing, in which we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once. For example, consider the following array:

```
import numpy as np
rng = np.random.default_rng(seed=1701)

x = rng.integers(100, size=10)
print(x)
```

```
[90 40  9 30 80 67 39 15 33 79]
```

Suppose we want to access three different elements. We could do it like this:

```
[x[3], x[7], x[2]]
```

```
[30, 15, 9]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
ind = [3, 7, 4]
x[ind]
```

```
array([30, 15, 80])
```

When using arrays of indices, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

```
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
```

```
array([[30, 15],
       [80, 67]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
X = np.arange(12).reshape((3, 4))
X
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

```
array([ 2,  5, 11])
```

Notice that the first value in the result is `X[0, 2]`, the second is `X[1, 1]`, and the third is `X[2, 3]`. The pairing of indices in fancy indexing follows all the broadcasting rules that were mentioned in "Computation on Arrays: Broadcasting". So, for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
X[row[:, np.newaxis], col]
```

```
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Here, each row value is matched with each column vector, exactly as we saw in broadcasting of arithmetic operations. For example:

```
row[:, np.newaxis] * col
```

```
array([[0, 0, 0],
       [2, 1, 3],
       [4, 2, 6]])
```

It is always important to remember with fancy indexing that the return value reflects the *broadcasted shape of the indices*, rather than the shape of the array being indexed.

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen:

```
print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
X[2, [2, 0, 1]]
```

```
array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
X[1:, [2, 0, 1]]
```

```
array([[ 6,  4,  5],
       [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
mask = np.array([True, False, True, False])
X[row[:, np.newaxis], mask]
```

```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for efficiently accessing and modifying array values.

Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. For example, we might have an N by D matrix representing N points in D dimensions, such as the following points drawn from a two-dimensional normal distribution:

```
mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
```

```
X = rng.multivariate_normal(mean, cov, 100)
X.shape
```

```
(100, 2)
```

Using the plotting tools we will discuss in [Link to Come], we can visualize these points as a scatter-plot:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')

plt.scatter(X[:, 0], X[:, 1]);
```

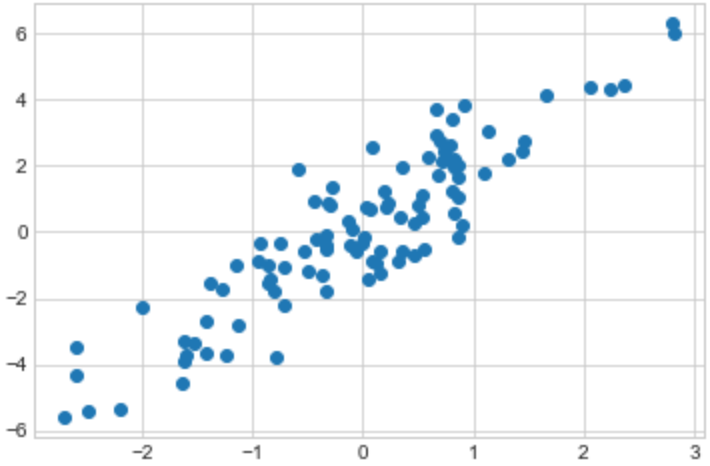


Figure 2-7. Normally distributed points

Let’s use fancy indexing to select 20 random points. We’ll do this by first choosing 20 random indices with no repeats, and use these indices to select a portion of the original array:

```
indices = np.random.choice(X.shape[0], 20, replace=False)
indices
```

```
array([82, 84, 10, 55, 14, 33,  4, 16, 34, 92, 99, 64,  8, 76,
68, 18, 59,
       80, 87, 90])
```

```
selection = X[indices]  # fancy indexing here
selection.shape
```

```
(20, 2)
```

Now to see which points were selected, let’s over-plot large circles at the locations of the selected points:

```
plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='none', edgecolor='black', s=200);
```

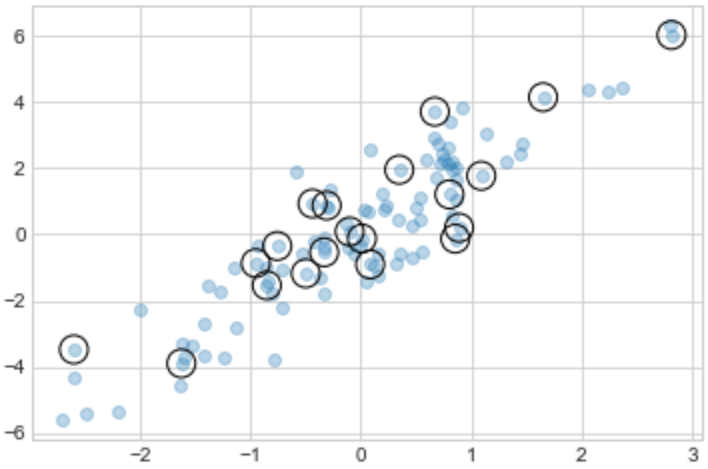


Figure 2-8. Random selection among points

This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models (see [Link to Come]), and in sampling approaches to answering statistical questions.

Modifying Values with Fancy Indexing

Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array. For example, imagine we have an array of indices and we’d like to set the corresponding items in an array to some value:

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
```

We can use any assignment-type operator for this. For example:

```
x[i] -= 10
print(x)
```

```
[ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
x = np.zeros(10)
x[[0, 0]] = [4, 6]
print(x)
```

```
[6. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Where did the 4 go? The result of this operation is to first assign `x[0] = 4`, followed by `x[0] = 6`. The result, of course, is that `x[0]` contains the value 6.

Fair enough, but consider this operation:

```
i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x
```

```
array([6., 0., 1., 1., 1., 0., 0., 0., 0., 0.])
```

You might expect that `x[3]` would contain the value 2, and `x[4]` would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because `x[i] += 1` is meant as a shorthand of `x[i] = x[i] + 1`. `x[i] + 1` is evaluated, and then the result is assigned to the indices in x. With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather nonintuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the `at()` method of ufuncs and do the following:

```
x = np.zeros(10)
np.add.at(x, i, 1)
print(x)
```

```
[0. 0. 1. 2. 3. 0. 0. 0. 0. 0.]
```

The `at()` method does an in-place application of the given operator at the specified indices (here, `i`) with the specified value (here, 1). Another method that is similar in spirit is the `reduceat()` method of ufuncs, which you can read about in the NumPy documentation.

Example: Binning Data

You can use these ideas to efficiently bin data to create a histogram by hand. For example, imagine we have 1,000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
rng = np.random.default_rng(seed=1701)
x = rng.normal(size=100)

# compute a histogram by hand
bins = np.linspace(-5, 5, 20)
counts = np.zeros_like(bins)

# find the appropriate bin for each x
i = np.searchsorted(bins, x)

# add 1 to each of these bins
np.add.at(counts, i, 1)
```

The counts now reflect the number of points within each bin–in other words, a histogram:

```
# plot the results
plt.plot(bins, counts, drawstyle='steps');
```

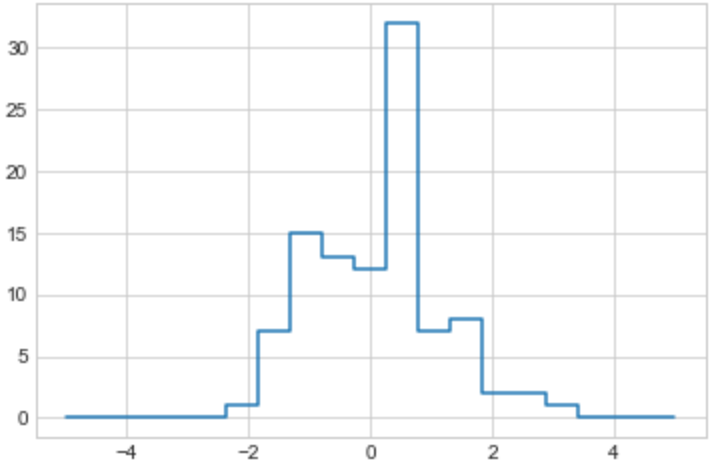


Figure 2-9. A histogram computed by hand

Of course, it would be silly to have to do this each time you want to plot a histogram. This is why Matplotlib provides the `plt.hist()` routine, which does the same in a single line:

```
plt.hist(x, bins, histtype='step');
```

This function will create a nearly identical plot to the one seen here. To compute the binning, `matplotlib` uses the `np.histogram` function, which does a very similar computation to what we did before. Let's compare the two here:

```
print(f"NumPy histogram ({len(x)} points):")
%timeit counts, edges = np.histogram(x, bins)

print(f"Custom histogram ({len(x)} points):")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy histogram (100 points):
33.8 µs ± 311 ns per loop (mean ± std. dev. of 7 runs, 10000
loops each)
Custom histogram (100 points):
17.6 µs ± 113 ns per loop (mean ± std. dev. of 7 runs, 100000
loops each)
```

Our own one-line algorithm is twice the speed of the optimized algorithm in NumPy! How can this be? If you dig into the `np.histogram` source code (you can do this in IPython by typing `np.histogram??`), you’ll see that it’s quite a bit more involved than the simple search-and-count that we’ve done; this is because NumPy’s algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

```
x = rng.normal(size=1000000)
print(f"NumPy histogram ({len(x)} points):")
%timeit counts, edges = np.histogram(x, bins)

print(f"Custom histogram ({len(x)} points):")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy histogram (1000000 points):
84.4 ms ± 2.82 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
Custom histogram (1000000 points):
128 ms ± 2.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa (see 02.08-Sorting.ipynb#Aside:-Big-O-Notation[Big-O Notation]). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. A key to efficiently using Python in data-intensive applications is knowing about general convenience routines like `np.histogram` and when they’re appropriate, but also knowing how to make use of lower-level functionality when you need more pointed behavior.

Sorting Arrays

Up to this point we have been concerned mainly with tools to access and operate on array data with NumPy. This section covers algorithms related to sorting values in NumPy arrays. These algorithms are a favorite topic in introductory computer science courses: if you've ever taken one, you probably have had dreams (or, depending on your temperament, nightmares) about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more. All are means of accomplishing a similar task: sorting the values in a list or array.

For example, a simple *selection sort* repeatedly finds the minimum value from a list, and makes swaps until the list is sorted. We can code this in just a few lines of Python:

```python
import numpy as np

def selection_sort(x):
    for i in range(len(x)):
        swap = i + np.argmin(x[i:])
        (x[i], x[swap]) = (x[swap], x[i])
    return x


x = np.array([2, 1, 4, 3, 5])
selection_sort(x)
```

```
array([1, 2, 3, 4, 5])
```

As any first-year computer science major will tell you, the selection sort is useful for its simplicity, but is much too slow to be useful for larger arrays. For a list of N values, it requires N loops, each of which does on order $\sim N$ comparisons to find the swap value. In terms of the "big-O" notation often used to characterize these algorithms (see #Aside:-Big-O-Notation[Big-O Notation]), selection sort averages $\mathcal{O}\left[N^2\right]$: if you double the number of items in the list, the execution time will go up by about a factor of four.

Even selection sort, though, is much better than my all-time favorite sorting algorithms, the *bogosort*:

```python
def bogosort(x):
    while np.any(x[:-1] > x[1:]):
        np.random.shuffle(x)
    return x
```

```
x = np.array([2, 1, 4, 3, 5])
bogosort(x)
```

```
array([1, 2, 3, 4, 5])
```

This silly sorting method relies on pure chance: it repeatedly applies a random shuffling of the array until the result happens to be sorted. With an average scaling of $\mathcal{O}[N \times N!]$, (that's N times N factorial) this should–quite obviously–never be used for any real computation.

Fortunately, Python contains built-in sorting algorithms that are *much* more efficient than either of the simplistic algorithms just shown. We'll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and optimized for NumPy arrays.

Fast Sorting in NumPy: `np.sort` and `np.argsort`

Although Python has built-in `sort` and `sorted` functions to work with lists, we won't discuss them here because NumPy's `np.sort` function turns out to be much more efficient and useful for our purposes. By default `np.sort` uses an $\mathcal{O}[N \log N]$, *quicksort* algorithm, though *mergesort* and *heapsort* are also available. For most applications, the default quicksort is more than sufficient.

To return a sorted version of the array without modifying the input, you can use `np.sort`:

```
x = np.array([2, 1, 4, 3, 5])
np.sort(x)
```

```
array([1, 2, 3, 4, 5])
```

If you prefer to sort the array in-place, you can instead use the `sort` method of arrays:

```
x.sort()
print(x)
```

```
[1 2 3 4 5]
```

A related function is `argsort`, which instead returns the *indices* of the sorted elements:

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

```
[1 0 3 2 4]
```

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, and so on. These indices can then be used (via fancy indexing) to construct the sorted array if desired:

```
x[i]
```

```
array([1, 2, 3, 4, 5])
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the `axis` argument. For example:

```
rng = np.random.default_rng(seed=42)
X = rng.integers(0, 10, (4, 6))
print(X)
```

```
[[0 7 6 4 4 8]
 [0 6 2 0 5 9]
 [7 7 7 7 5 1]
 [8 4 5 3 1 9]]
```

```
# sort each column of X
np.sort(X, axis=0)
```

```
array([[0, 4, 2, 0, 1, 1],
       [0, 6, 5, 3, 4, 8],
       [7, 7, 6, 4, 5, 9],
       [8, 7, 7, 7, 5, 9]])
```

```
# sort each row of X
np.sort(X, axis=1)
```

```
array([[0, 4, 4, 6, 7, 8],
       [0, 0, 2, 5, 6, 9],
       [1, 5, 7, 7, 7, 7],
       [1, 3, 4, 5, 8, 9]])
```

Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the *k* smallest values in the array. NumPy provides this in the `np.partition` function. `np.partition` takes an array and a number *K*; the result is a new array with the smallest *K* values to the left of the partition, and the remaining values to the right, in arbitrary order:

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)
```

```
array([2, 1, 3, 4, 6, 5, 7])
```

Notice that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

```
np.partition(X, 2, axis=1)
```

```
array([[0, 4, 4, 7, 6, 8],
       [0, 0, 2, 6, 5, 9],
       [1, 5, 7, 7, 7, 7],
       [1, 3, 4, 5, 8, 9]])
```

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Finally, just as there is a `np.argsort` that computes indices of the sort, there is a `np.argpartition` that computes indices of the partition. We'll see this in action in the following section.

Example: k-Nearest Neighbors

Let's quickly see how we might use this `argsort` function along multiple axes to find the nearest neighbors of each point in a set. We'll start by creating a random set of 10 points on a two-dimensional plane. Using the standard convention, we'll arrange these in a 10 × 2 array:

```
X = rng.random((10, 2))
```

To get an idea of how these points look, let's quickly scatter plot them:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
plt.scatter(X[:, 0], X[:, 1], s=100);
```

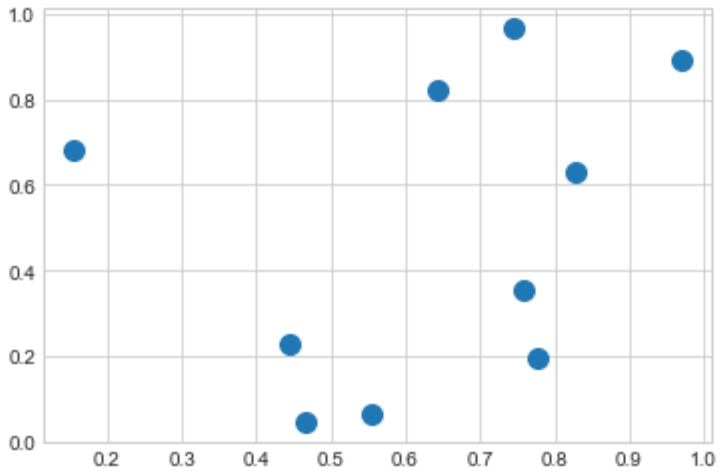


Figure 2-10. Visualization of points in the k-neighbors example

Now we'll compute the distance between each pair of points. Recall that the squared-distance between two points is the sum of the squared differences in each dimension; using the efficient broadcasting ("Computation on Arrays: Broadcasting") and aggregation ("Aggregations: Min, Max, and Everything In Between") routines provided by NumPy we can compute the matrix of square distances in a single line of code:

```
dist_sq = np.sum((X[:, np.newaxis] - X[np.newaxis, :]) ** 2,
axis=-1)
```

This operation has a lot packed into it, and it might be a bit confusing if you're unfamiliar with NumPy's broadcasting rules. When you come across code like this, it can be useful to break it down into its component steps:

```
# for each pair of points, compute differences in their
coordinates
differences = X[:, np.newaxis] - X[np.newaxis, :]
differences.shape
```

```
(10, 10, 2)
```

```
# square the coordinate differences
sq_differences = differences ** 2
sq_differences.shape
```

```
(10, 10, 2)
```

```
# sum the coordinate differences to get the squared distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape
```

```
(10, 10)
```

As a quick check of our logic, we should see that the diagonal of this matrix (i.e., the set of distances between each point and itself) is all zero:

```
dist_sq.diagonal()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

With the pairwise square-distances converted, we can now use `np.argsort` to sort along each row. The leftmost columns will then give the indices of the nearest neighbors:

```
nearest = np.argsort(dist_sq, axis=1)
print(nearest)
```

```
[[0 9 3 5 4 8 1 6 2 7]
 [1 7 2 6 4 8 3 0 9 5]
 [2 7 1 6 4 3 8 0 9 5]
 [3 0 4 5 9 6 1 2 8 7]
 [4 6 3 1 2 7 0 5 9 8]
 [5 9 3 0 4 6 8 1 2 7]
 [6 4 2 1 7 3 0 5 9 8]
 [7 2 1 6 4 3 8 0 9 5]
 [8 0 1 9 3 4 7 2 6 5]
 [9 0 5 3 4 8 6 1 2 7]]
```

Notice that the first column gives the numbers 0 through 9 in order: this is due to the fact that each point's closest neighbor is itself, as we would expect.

By using a full sort here, we've actually done more work than we need to in this case. If we're simply interested in the nearest k neighbors, all we need is to partition each row so that the smallest $k + 1$ squared distances come first, with larger distances filling the remaining positions of the array. We can do this with the `np.argpartition` function:

```
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

In order to visualize this network of neighbors, let's quickly plot the points along with lines representing the connections from each point to its two nearest neighbors:

```
plt.scatter(X[:, 0], X[:, 1], s=100)

# draw lines from each point to its two nearest neighbors
K = 2

for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
        # plot a line from X[i] to X[j]
        # use some zip magic to make it happen:
        plt.plot(*zip(X[j], X[i]), color='black')
```

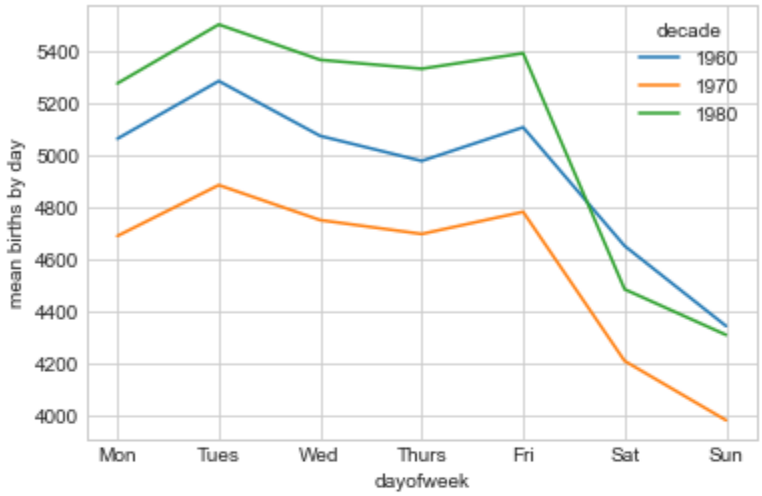


Figure 2-11. Visualization of the neighbors of each point

Each point in the plot has lines drawn to its two nearest neighbors. At first glance, it might seem strange that some of the points have more than two lines coming out of them: this is due to the fact that if point A is one of the two nearest neighbors of point B, this does not necessarily imply that point B is one of the two nearest neighbors of point A.

Although the broadcasting and row-wise sorting of this approach might seem less straightforward than writing a loop, it turns out to be a very efficient way of operating on this data in Python. You might be tempted to do the same type of operation by manually looping through the data and sorting each set of neighbors individually, but this would almost certainly lead to a slower algorithm than the vectorized version we used. The beauty of this approach is that it's written in a way that's agnostic to the size of the input data: we could just as easily compute the neighbors among 100 or 1,000,000 points in any number of dimensions, and the code would look the same.

Finally, I'll note that when doing very large nearest neighbor searches, there are tree-based and/or approximate algorithms that can scale as $\mathcal{O}[N \log N]$ or better rather than the $\mathcal{O}\left[N^2\right]$ of the brute-force algorithm. One example of this is the KD-Tree, implemented in Scikit-learn.

Aside: Big-O Notation

Big-O notation is a means of describing how the number of operations required for an algorithm scales as the input grows in size. To use it correctly is to dive deeply into the realm of computer science theory, and to carefully distinguish it from the related small-o notation, big-θ notation, big-Ω notation, and probably many mutant hybrids thereof. While these distinctions add precision to statements about algorithmic scaling, outside computer science theory exams and the remarks of pedantic blog commenters, you'll rarely see such distinctions made in practice. Far more common in the data science world is a less rigid use of big-O notation: as a general (if imprecise) description of the scaling of an algorithm. With apologies to theorists and pedants, this is the interpretation we'll use throughout this book.

Big-O notation, in this loose sense, tells you how much time your algorithm will take as you increase the amount of data. If you have an $\mathcal{O}[N]$ (read "order N") algorithm that takes 1 second to operate on a list of length N=1,000, then you should expect it to take roughly 5 seconds for a list of length N=5,000. If you have an $\mathcal{O}\left[N^2\right]$ (read "order N squared") algorithm that takes 1 second for N=1000, then you should expect it to take about 25 seconds for N=5000.

For our purposes, the N will usually indicate some aspect of the size of the dataset (the number of points, the number of dimensions, etc.). When trying to analyze billions or trillions of samples, the difference between $\mathcal{O}[N]$ and $\mathcal{O}\left[N^2\right]$ can make or break your analysis!

Notice that the big-O notation by itself tells you nothing about the actual wall-clock time of a computation, but only about its scaling as you change N. Generally, for example, an $\mathcal{O}[N]$ algorithm is considered to have better scaling than an $\mathcal{O}\left[N^2\right]$ algorithm, and for good reason. But for small datasets in particular, the algorithm with better scaling might not be faster. For example, in a given problem an $\mathcal{O}\left[N^2\right]$ algorithm might take 0.01 seconds, while a “better” $\mathcal{O}[N]$ algorithm might take 1 second. Scale up N by a factor of 1,000, though, and the $\mathcal{O}[N]$ algorithm will win out.

Structured Data: NumPy’s Structured Arrays

While often our data can be well represented by a homogeneous array of values, sometimes this is not the case. This section demonstrates the use of NumPy’s *structured arrays* and *record arrays*, which provide efficient storage for compound, heterogeneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas *Dataames*, which we’ll explore in Chapter 3.

```
import numpy as np
```

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we’d like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit clumsy. There’s nothing here that tells us that the three arrays are related; NumPy’s structured arrays allow us to do this more naturally by using a single structure to store all of this data.

Recall that previously we created a simple array using an expression like this:

```
x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

```
# Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
print(data.dtype)
```

```
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

Here `'U10'` translates to “Unicode string of maximum length 10,” `'i4'` translates to “4-byte (i.e., 32 bit) integer,” and `'f8'` translates to “8-byte (i.e., 64 bit) float.” We’ll discuss other options for these type codes in the following section.

Now that we’ve created an empty container array, we can fill the array with our lists of values:

```
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
```

```
[('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
 ('Doug', 19, 61.5)]
```

As we had hoped, the data is now conveniently arranged in one structured array.

The handy thing with structured arrays is that you can now refer to values either by index or by name:

```
# Get all names
data['name']
```

```
array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')
```

```
# Get first row of data
data[0]
```

```
('Alice', 25, 55.)
```

```
# Get the name from the last row
data[-1]['name']
```

```
'Doug'
```

Using Boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

```
# Get names where age is under 30
data[data['age'] < 30]['name']
```

```
array(['Alice', 'Doug'], dtype='<U10')
```

If you'd like to do any operations that are any more complicated than these, you should probably consider the Pandas package, covered in the next chapter. As we'll see, Pandas provides a `Dataframe` object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what we've shown here, as well as much, much more.

Exploring Structured Array Creation

Structured array data types can be specified in a number of ways. Earlier, we saw the dictionary method:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':('U10', 'i4', 'f8')})
```

```
dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

For clarity, numerical types can be specified using Python types or NumPy *dtype*s instead:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':((np.str_, 10), int, np.float32)})
```

```
dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

A compound type can also be specified as a list of tuples:

```
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

```
dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

```
np.dtype('S10,i4,f8')
```

```
dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

The shortened string format codes may not be immediately intuitive, but they are built on simple principles. The first (optional) character is < or >, means “little endian” or “big endian,” respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, and so on (see the table below). The last character or characters represents the size of the object in bytes.

Character	Description	Example
`'b'`	Byte	`np.dtype('b')`
`'i'`	Signed integer	`np.dtype('i4') == np.int32`
`'u'`	Unsigned integer	`np.dtype('u1') == np.uint8`
`'f'`	Floating point	`np.dtype('f8') == np.int64`
`'c'`	Complex floating point	`np.dtype('c16') == np.complex128`
`'S'`, `'a'`	String	`np.dtype('S5')`
`'U'`	Unicode string	`np.dtype('U') == np.str_`
`'V'`	Raw data (void)	`np.dtype('V') == np.void`

More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we'll create a data type with a `mat` component consisting of a 3×3 floating-point matrix:

```
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
```

```
(0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Now each element in the X array consists of an id and a 3×3 matrix. Why would you use this rather than a simple multidimensional array, or perhaps a Python dictionary? One reason is that this NumPy dtype directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. If you find yourself writing a Python interface to a legacy C or Fortran library that manipulates structured data, structured arrays can provide a powerful interface.

RecordArrays: Structured Arrays with a Twist

NumPy also provides the np.recarray class, which is almost identical to the structured arrays just described, but with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the ages by writing:

```
data['age']
```

```
array([25, 45, 37, 19], dtype=int32)
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

```
data_rec = data.view(np.recarray)
data_rec.age
```

```
array([25, 45, 37, 19], dtype=int32)
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax:

```
%timeit data['age']
%timeit data_rec['age']
%timeit data_rec.age
```

```
121 ns ± 1.4 ns per loop (mean ± std. dev. of 7 runs, 1000000
loops each)
```

```
2.41 µs ± 15.7 ns per loop (mean ± std. dev. of 7 runs, 100000
loops each)
3.98 µs ± 20.5 ns per loop (mean ± std. dev. of 7 runs, 100000
loops each)
```

Whether the more convenient notation is worth the (slight) overhead will depend on your own application.

On to Pandas

This section on structured and record arrays is purposely at the end of this chapter, because it leads so well into the next package we will cover: the Pandas package. Structured arrays like the ones discussed here are good to know about for certain situations, especially in case you’re using NumPy arrays to map onto binary data formats in C, Fortran, or another language. For day-to-day use of structured data, the Pandas package is a much better choice, and we’ll dive into a full discussion of it in the chapter that follows.

Chapter 3. Data Manipulation with Pandas

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. You can find preliminary code and notebook files on GitHub.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

In the previous chapter, we dove into detail on NumPy and its `ndarray` object, which provides efficient storage and manipulation of dense typed arrays in Python. Here we'll build on this knowledge by looking in detail at the data structures provided by the Pandas library. Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a `DataFrame`. *DataFrames* are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

As we saw, NumPy's `ndarray` data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available in many forms in the world around us. Pandas, and in particular its `Series` and `DataFrame` objects, builds on the NumPy array structure and provides efficient access to these sorts of "data munging" tasks that occupy much of a data scientist's time.

In this chapter, we will focus on the mechanics of using `Series`, `DataFrame`, and related structures effectively. We will use examples drawn from real datasets where appropriate, but these examples are not necessarily the focus.

Installing and Using Pandas

Installation of Pandas on your system requires NumPy to be installed, and if building the library from source, requires the appropriate tools to compile the C and Cython sources on which Pandas is built. Details on this installation can be found in the Pandas documentation. If you followed the advice outlined in the Preface and used the Anaconda stack, you already have Pandas installed.

Once Pandas is installed, you can import it and check the version:

```
import pandas
pandas.__version__
```

```
'1.3.5'
```

Just as we generally import NumPy under the alias `np`, we will import Pandas under the alias `pd`:

```
import pandas as pd
```

This import convention will be used throughout the remainder of this book.

Reminder about Built-In Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature) as well as the documentation of various functions (using the `?` character). (Refer back to "Help and Documentation in IPython" if you need a refresher on this.)

For example, to display all the contents of the pandas namespace, you can type

```
In [3]: pd.<TAB>
```

And to display Pandas's built-in documentation, you can use this:

```
In [4]: pd?
```

More detailed documentation, along with tutorials and other resources, can be found at *http://pandas.pydata.org/*.

Introducing Pandas Objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see during the course of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are. Thus, before we go any further, let's introduce these three fundamental Pandas data structures: the `Series`, `DataFrame`, and `Index`.

We will start our code sessions with the standard NumPy and Pandas imports:

```
import numpy as np
import pandas as pd
```

The Pandas Series Object

A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

The `Series` combines a sequence of values with an explicit sequence of indices, which we can access with the `values` and `index` attributes. The `values` are simply a familiar NumPy array:

```
data.values
```

```
array([0.25, 0.5 , 0.75, 1.  ])
```

The `index` is an array-like object of type `pd.Index`, which we'll discuss in more detail momentarily.

```
data.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
data[1]
```

```
0.5
```

```
data[1:3]
```

```
1    0.50
2    0.75
dtype: float64
```

As we will see, though, the Pandas `Series` is much more general and flexible than the one-dimensional NumPy array that it emulates.

Series as generalized NumPy array

From what we've seen so far, the `Series` object may appear to be basically interchangeable with a one-dimensional NumPy array. The essential difference is that while the Numpy Array has an *implicitly defined* integer index used to access the values, the Pandas `Series` has an *explicitly defined* index associated with the values.

This explicit index definition gives the `Series` object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

And the item access works as expected:

```
data['b']
```

```
0.5
```

We can even use non-contiguous or non-sequential indices:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

```
data[5]
```

```
0.5
```

Series as specialized dictionary

In this way, you can think of a Pandas `Series` a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a `Series` is a structure which maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas `Series` makes it more efficient than Python dictionaries for certain operations.

The `Series`-as-dictionary analogy can be made even more clear by constructing a `Series` object directly from a Python dictionary, here the five most populous US states according to the 2020 census:

```
population_dict = {'California': 39538223, 'Texas': 29145505,
                   'Florida': 21538187, 'New York': 20201249,
                   'Pennsylvania': 13002700}
population = pd.Series(population_dict)
population
```

```
California      39538223
Texas           29145505
Florida         21538187
New York        20201249
Pennsylvania    13002700
dtype: int64
```

From here, typical dictionary-style item access can be performed:

```
population['California']
```

```
39538223
```

Unlike a dictionary, though, the `Series` also supports array-style operations such as slicing:

```
population['California':'Florida']
```

```
California    39538223
Texas         29145505
Florida       21538187
dtype: int64
```

We’ll discuss some of the quirks of Pandas indexing and slicing in “Data Indexing and Selection”.

Constructing Series objects

We’ve already seen a few ways of constructing a Pandas `Series` from scratch; all of them are some version of the following:

```
pd.Series(data, index=index)
```

where `index` is an optional argument, and `data` can be one of many entities.

For example, `data` can be a list or NumPy array, in which case `index` defaults to an integer sequence:

```
pd.Series([2, 4, 6])
```

```
0    2
1    4
2    6
dtype: int64
```

`data` can be a scalar, which is repeated to fill the specified index:

```
pd.Series(5, index=[100, 200, 300])
```

```
100    5
200    5
300    5
dtype: int64
```

`data` can be a dictionary, in which `index` defaults to the dictionary keys:

```
pd.Series({2:'a', 1:'b', 3:'c'})
```

```
2    a
1    b
3    c
dtype: object
```

In each case, the index can be explicitly set to control the order or the subset of keys used:

```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[1, 2])
```

```
1    b
2    a
dtype: object
```

The Pandas DataFrame Object

The next fundamental structure in Pandas is the `DataFrame`. Like the `Series` object discussed in the previous section, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

DataFrame as a generalized NumPy array

If a `Series` is an analog of a one-dimensional array with explicit indices, a `DataFrame` is an analog of a two-dimensional array with explicit row and column indices. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a `DataFrame` as a sequence of aligned `Series` objects. Here, by "aligned" we mean that they share the same index.

To demonstrate this, let's first construct a new `Series` listing the area of each of the five states discussed in the previous section (in square kilometers):

```
area_dict = {'California': 423967, 'Texas': 695662, 'Florida': 170312,
             'New York': 141297, 'Pennsylvania': 119280}
area = pd.Series(area_dict)
area
```

```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
dtype: int64
```

Now that we have this along with the `population` Series from before, we can use a dictionary to construct a single two-dimensional object containing this information:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

	population	area
California	39538223	423967
Texas	29145505	695662
Florida	21538187	170312
New York	20201249	141297
Pennsylvania	13002700	119280

Like the `Series` object, the `DataFrame` has an `index` attribute that gives access to the index labels:

```
states.index
```

```
Index(['California', 'Texas', 'Florida', 'New York', 'Pennsylvania'],
 > dtype='object')
```

Additionally, the `DataFrame` has a `columns` attribute, which is an `Index` object holding the column labels:

```
states.columns
```

```
Index(['population', 'area'], dtype='object')
```

Thus the `DataFrame` can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

DataFrame as specialized dictionary

Similarly, we can also think of a `DataFrame` as a specialization of a dictionary. Where a dictionary maps a key to a value, a `DataFrame` maps a column name to a `Series` of column data. For example, asking for the `'area'` attribute returns the `Series` object containing the areas we saw earlier:

```
states['area']
```

```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimesnional NumPy array, `data[0]` will return the first *row*. For a `DataFrame`, `data['col0']` will return the first *column*. Because of this, it is probably better to think about $DataFrame$s as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful. We'll explore more flexible means of indexing $DataFrame$s in "Data Indexing and Selection".

Constructing DataFrame objects

A Pandas `DataFrame` can be constructed in a variety of ways. Here we'll give several examples.

From a single Series object

A DataFrame is a collection of Series objects, and a single-column DataFrame can be constructed from a single Series:

```
pd.DataFrame(population, columns=['population'])
```

	population
California	39538223
Texas	29145505
Florida	21538187
New York	20201249
Pennsylvania	13002700

From a list of dicts

Any list of dictionaries can be made into a DataFrame. We’ll use a simple list comprehension to create some data:

```
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
```

	a	b
0	0	0
1	1	2
2	2	4

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., “not a number”) values:

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

From a dictionary of Series objects

As we saw before, a DataFrame can be constructed from a dictionary of Series objects as well:

```
pd.DataFrame({'population': population,
              'area': area})
```

	population	area
California	39538223	423967
Texas	29145505	695662
Florida	21538187	170312
New York	20201249	141297
Pennsylvania	13002700	119280

From a two-dimensional NumPy array

Given a two-dimensional array of data, we can create a `DataFrame` with any specified column and index names. If omitted, an integer index will be used for each:

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

	foo	bar
a	0.471098	0.317396
b	0.614766	0.305971
c	0.533596	0.512377

From a NumPy structured array

We covered structured arrays in "Structured Data: NumPy's Structured Arrays". A Pandas `DataFrame` operates much like a structured array, and can be created directly from one:

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
```

```
array([(0, 0.), (0, 0.), (0, 0.)], dtype=[('A', '<i8'), ('B', '<f8')])
```

```
pd.DataFrame(A)
```

	A	B
0	0	0.0
1	0	0.0
2	0	0.0

The Pandas Index Object

We have seen here that both the `Series` and `DataFrame` objects contain an explicit *index* that lets you reference and modify data. This `Index` object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set* (technically a multi-set, as `Index` objects may contain repeated values). Those views have some interesting consequences in the operations available on `Index` objects. As a simple example, let's construct an `Index` from a list of integers:

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as immutable array

The `Index` in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

```
ind[1]
```

```
3
```

```
ind[::2]
```

```
Int64Index([2, 5, 11], dtype='int64')
```

`Index` objects also have many of the attributes familiar from NumPy arrays:

```
print(ind.size, ind.shape, ind.ndim, ind.dtype)
```

```
5 (5,) 1 int64
```

One difference between `Index` objects and NumPy arrays is that indices are immutable–that is, they cannot be modified via the normal means:

```
ind[1] = 0
```

```
 TypeError: Index does not support mutable operations
```

This immutability makes it safer to share indices between multiple *DataFrame*s and arrays, without the potential for side effects from inadvertent index modification.

Index as ordered set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The `Index` object follows many of the conventions used by Python's built-in `set` data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

```
indA.intersection(indB)
```

```
Int64Index([3, 5, 7], dtype='int64')
```

```
indA.union(indB)
```

```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

```
indA.symmetric_difference(indB)
```

```
Int64Index([1, 2, 9, 11], dtype='int64')
```

Data Indexing and Selection

In Chapter 2, we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included indexing (e.g., `arr[2, 1]`), slicing (e.g., `arr[:, 1:5]`), masking (e.g., `arr[arr > 0]`), fancy indexing (e.g., `arr[0, [1, 5]]`), and combinations thereof (e.g., `arr[:, [1, 5]]`). Here we'll look at similar means of accessing and modifying values in Pandas `Series` and `DataFrame` objects. If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

We'll start with the simple case of the one-dimensional `Series` object, and then move on to the more complicated two-dimesnional `DataFrame` object.

Data Selection in Series

As we saw in the previous section, a `Series` object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

```
data['b']
```

```
0.5
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
'a' in data
```

```
True
```

```
data.keys()
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
list(data.items())
```

```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`Series` objects can also be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a `Series` by assigning to a new index value:

```
data['e'] = 1.25
data
```

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

Series as one-dimensional array

A `Series` builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, *slices*, *masking*, and *fancy indexing*. Examples of these are as follows:

```
# slicing by explicit index
data['a':'c']
```

```
a    0.25
b    0.50
c    0.75
dtype: float64
```

```
# slicing by implicit integer index
data[0:2]
```

```
a    0.25
b    0.50
dtype: float64
```

```
# masking
data[(data > 0.3) & (data < 0.8)]
```

```
b    0.50
c    0.75
dtype: float64
```

```
# fancy indexing
data[['a', 'e']]
```

```
a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., `data['a':'c']`), the final index is *included* in the slice, while when slicing with an implicit index (i.e., `data[0:2]`), the final index is *excluded* from the slice.

Indexers: loc and iloc

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

```
1    a
3    b
5    c
dtype: object
```

```
# explicit index when indexing
data[1]
```

```
'a'
```

```
# implicit index when slicing
data[1:3]
```

```
3    b
5    c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the `Series`.

First, the `loc` attribute allows indexing and slicing that always references the explicit index:

```
data.loc[1]
```

```
'a'
```

```
data.loc[1:3]
```

```
1    a
3    b
dtype: object
```

The `iloc` attribute allows indexing and slicing that always references the implicit Python-style index:

```
data.iloc[1]
```

```
'b'
```

```
data.iloc[1:3]
```

```
3    b
5    c
dtype: object
```

One guiding principle of Python code is that “explicit is better than implicit.” The explicit nature of `loc` and `iloc` make them very in maintaining clean and readable code; especially in the case of integer indexes, using them consistently can prevent subtle bugs due to the mixed indexing/slicing convention.

Data Selection in DataFrame

Recall that a `DataFrame` acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of `Series` structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

DataFrame as a dictionary

The first analogy we will consider is the `DataFrame` as a dictionary of related `Series` objects. Let's return to our example of areas and populations of states:

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'Florida': 170312, 'New York': 141297,
                  'Pennsylvania': 119280})
pop = pd.Series({'California': 39538223, 'Texas': 29145505,
                 'Florida': 21538187, 'New York': 20201249,
                 'Pennsylvania': 13002700})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

	area	pop
California	423967	39538223
Texas	695662	29145505
Florida	170312	21538187
New York	141297	20201249
Pennsylvania	119280	13002700

The individual `Series` that make up the columns of the `DataFrame` can be accessed via dictionary-style indexing of the column name:

```
data['area']
```

```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

```
data.area
```

```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
Name: area, dtype: int64
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the `DataFrame`, this attribute-style access is not possible. For example, the `DataFrame` has a `pop()` method, so `data.pop` will point to this rather than the `"pop"` column:

```
data.pop is data["pop"]
```

```
False
```

In particular, you should avoid the temptation to try column assignment via attribute (i.e., use `data['pop'] = z` rather than `data.pop = z`).

Like with the `Series` objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:

```
data['density'] = data['pop'] / data['area']
data
```

	area	pop	density
California	423967	39538223	93.257784
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

This shows a preview of the straightforward syntax of element-by-element arithmetic between `Series` objects; we'll dig into this further in "Operating on Data in Pandas".

DataFrame as two-dimensional array

As mentioned previously, we can also view the `DataFrame` as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

```
data.values
```

```
array([[4.23967000e+05, 3.95382230e+07, 9.32577842e+01],
       [6.95662000e+05, 2.91455050e+07, 4.18960717e+01],
       [1.70312000e+05, 2.15381870e+07, 1.26463121e+02],
       [1.41297000e+05, 2.02012490e+07, 1.42970120e+02],
       [1.19280000e+05, 1.30027000e+07, 1.09009893e+02]])
```

With this picture in mind, many familiar array-like observations can be done on the `DataFrame` itself. For example, we can transpose the full `DataFrame` to swap rows and columns:

```
data.T
```

	California	Texas	Florida	New York	Pennsylvania
area	4.239670e+05	6.956620e+05	1.703120e+05	1.412970e+05	1.192800e+05
pop	3.953822e+07	2.914550e+07	2.153819e+07	2.020125e+07	1.300270e+07
density	9.325778e+01	4.189607e+01	1.264631e+02	1.429701e+02	1.090099e+02

When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

```
data.values[0]
```

```
array([4.23967000e+05, 3.95382230e+07, 9.32577842e+01])
```

and passing a single “index” to a `DataFrame` accesses a column:

```
data['area']
```

```
California      423967
Texas           695662
Florida         170312
New York        141297
Pennsylvania    119280
Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc` and `iloc` indexers mentioned earlier. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

```
data.iloc[:3, :2]
```

	area	pop
California	423967	39538223
Texas	695662	29145505
Florida	170312	21538187

Similarly, using the `loc` indexer we can index the underlying data in an array-like style but using the explicit index and column names:

```
data.loc[:'Florida', :'pop']
```

	area	pop
California	423967	39538223
Texas	695662	29145505
Florida	170312	21538187

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

```
data.loc[data.density > 120, ['pop', 'density']]
```

	pop	density
Florida	21538187	126.463121
New York	20201249	142.970120

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

```
data.iloc[0, 2] = 90
data
```

	area	pop	density
California	423967	39538223	90.000000
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120
Pennsylvania	119280	13002700	109.009893

To build up your fluency in Pandas data manipulation, I suggest spending some time with a simple `DataFrame` and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

Additional indexing conventions

There are a couple extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be useful in practice. First, while *indexing* refers to columns, *slicing* refers to rows:

```
data['Florida':'New York']
```

	area	pop	density
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120

Such slices can also refer to rows by number rather than by index:

```
data[1:3]
```

	area	pop	density
Texas	695662	29145505	41.896072
Florida	170312	21538187	126.463121

Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

```
data[data.density > 120]
```

	area	pop	density
Florida	170312	21538187	126.463121
New York	141297	20201249	142.970120

These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the Pandas conventions, they are included due to their practical utility.

Operating on Data in Pandas

One of the strengths of NumPy is the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the ufuncs that we introduced in "Computation on NumPy Arrays: Universal Functions" are key to this.

Pandas includes a couple useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in the output, and for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc. This means that keeping the context of data and combining data from different sources–both potentially error-prone tasks with raw NumPy arrays–become essentially foolproof ones with Pandas. We will additionally see that there are well-defined operations between one-dimensional `Series` structures and two-dimensional `DataFrame` structures.

Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas `Series` and `DataFrame` objects. Let's start by defining a simple `Series` and `DataFrame` on which to demonstrate this:

```
import pandas as pd
import numpy as np

rng = np.random.default_rng(42)
ser = pd.Series(rng.integers(0, 10, 4))
ser
```

```
0    0
1    7
2    6
3    4
dtype: int64
```

```
df = pd.DataFrame(rng.integers(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	A	B	C	D
0	4	8	0	6
1	2	0	5	9
2	7	7	7	7

If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object *with the indices preserved:*

```
np.exp(ser)
```

```
0       1.000000
1    1096.633158
2     403.428793
3      54.598150
dtype: float64
```

This is true also for more involved sequences of operations:

```
np.sin(df * np.pi / 4)
```

	A	B	C	D
0	1.224647e-16	-2.449294e-16	0.000000	-1.000000
1	1.000000e+00	0.000000e+00	-0.707107	0.707107
2	-7.071068e-01	-7.071068e-01	-0.707107	-0.707107

Any of the ufuncs discussed in “Computation on NumPy Arrays: Universal Functions” can be used in a similar manner.

UFuncs: Index Alignment

For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data, as we’ll see in some of the examples that follow.

Index alignment in Series

As an example, suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 39538223, 'Texas': 29145505,
                        'Florida': 21538187}, name='population')
```

Let’s see what happens when we divide these to compute the population density:

```
population / area
```

```
Alaska              NaN
California    93.257784
Florida             NaN
Texas         41.896072
dtype: float64
```

The resulting array contains the *union* of indices of the two input arrays, which could be determined directly from these indices:

```
area.index.union(population.index)
```

```
Index(['Alaska', 'California', 'Florida', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked with NaN, or “Not a Number,” which is how Pandas marks missing data (see further discussion of missing data in “Handling Missing Data”). This index matching is implemented this way for any of Python’s built-in arithmetic expressions; any missing values are marked by NaN:

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
```

```
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
```

If using NaN values is not the desired behavior, the fill value can be modified using appropriate object methods in place of the operators. For example, calling A.add(B) is equivalent to calling A + B, but allows optional explicit specification of the fill value for any elements in A or B that might be missing:

```
A.add(B, fill_value=0)
```

```
0    2.0
1    5.0
```

```
2    9.0
3    5.0
dtype: float64
```

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on `DataFrame` objects:

```
A = pd.DataFrame(rng.integers(0, 20, (2, 2)),
                 columns=['a', 'b'])
A
```

	a	b
0	10	2
1	16	9

```
B = pd.DataFrame(rng.integers(0, 10, (3, 3)),
                 columns=['b', 'a', 'c'])
B
```

	b	a	c
0	5	3	1
1	9	7	6
2	4	8	5

```
A + B
```

	a	b	c
0	13.0	7.0	NaN
1	23.0	18.0	NaN
2	NaN	NaN	NaN

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with `Series`, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries. Here we'll fill with the mean of all values in `A`:

```
A.add(B, fill_value=A.values.mean())
```

	a	b	c
0	13.00	7.00	10.25
1	23.00	18.00	15.25
2	17.25	13.25	14.25

The following table lists Python operators and their equivalent Pandas object methods:

Python Operator	Pandas Method(s)
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Ufuncs: Operations Between DataFrame and Series

When performing operations between a `DataFrame` and a `Series`, the index and column alignment is similarly maintained, and the result is similar to operations between a two-dimensional and one-dimensional NumPy array. Consider one common operation, where we find the difference of a two-dimensional array and one of its rows:

```
A = rng.integers(10, size=(3, 4))
A
```

```
array([[4, 4, 2, 0],
       [5, 8, 0, 8],
       [8, 2, 6, 1]])
```

```
A - A[0]
```

```
array([[ 0,  0,  0,  0],
       [ 1,  4, -2,  8],
       [ 4, -2,  4,  1]])
```

According to NumPy's broadcasting rules (see "Computation on Arrays: Broadcasting"), subtraction between a two-dimensional array and one of its rows is applied row-wise.

In Pandas, the convention similarly operates row-wise by default:

```
df = pd.DataFrame(A, columns=['Q', 'R', 'S', 'T'])
df - df.iloc[0]
```

	Q	R	S	T
0	0	0	0	0
1	1	4	-2	8
2	4	-2	4	1

If you would instead like to operate column-wise, you can use the object methods mentioned earlier, while specifying the `axis` keyword:

```
df.subtract(df['R'], axis=0)
```

	Q	R	S	T
0	0	0	-2	-4
1	-3	0	-8	0
2	6	0	4	-1

Note that these `DataFrame`/`Series` operations, like the operations discussed above, will automatically align indices between the two elements:

```
halfrow = df.iloc[0, ::2]
halfrow
```

```
Q    4
S    2
Name: 0, dtype: int64
```

```
df - halfrow
```

	Q	R	S	T
0	0.0	NaN	0.0	NaN
1	1.0	NaN	-2.0	NaN
2	4.0	NaN	4.0	NaN

This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the common errors that might arise when working with heterogeneous and/or misaligned data in raw NumPy arrays.

Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

In this section, we will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here and throughout the book, we'll refer to missing data in general as *null*, *NaN*, or *NA* values.

Trade-Offs in Missing Data Conventions

A number of approaches have been developed to track the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a *mask* that globally indicates missing values, or choosing a *sentinel value* that indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with -9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

None of these approaches is without trade-offs: use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be represented, and may require extra (often non-optimized) logic in CPU and GPU arithmetic. Common special values like NaN are not available for all data types.

As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the R language uses reserved bit patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell which indicates a NA state.

Missing Data in Pandas

The way in which Pandas handles missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point data types.

Perhaps pandas could have followed R's lead in specifying bit patterns for each individual data type to indicate nullness, but this approach turns out to be rather unwieldy. While R contains four basic data types, NumPy supports *far* more than this: for example, while R has a single integer type, NumPy supports *fourteen* basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, likely even requiring a new fork of the NumPy package. Further, for the smaller data types (such as 8-bit integers), sacrificing a bit to use as a mask will significantly reduce the range of values it can represent.

Because of these constraints and tradeoffs, Pandas has two "modes" of storing and manipulating null values:

- the default mode is to use a sentinel-based missing data scheme, with sentinel values `NaN` or `None` depending on the type of the data.
- alternatively, you can opt-in to using pandas' *nullable dtypes*, which results in the creation an accompanying mask array to track missing entries. These missing entries are then presented to the user as the special `pd.NA` value.

In either case, the data operations and manipulations provided by the pandas API will handle and propagate those missing entries in a predictable manner. But to develop some intuition into *why* these choices are made, let's dive quickly into the tradeoffs inherent in `None`, `NaN`, and `NA`.

```
import numpy as np
import pandas as pd
```

None as a sentinel value

For some data types, pandas uses `None` as a sentinel value. `None` is a Python object, which means that any array containing `None` must have `dtype=object`, that is, it must be a sequence of Python objects.

For example, observe what happens if you pass `None` to a numpy array:

```
vals1 = np.array([1, None, 2, 3])
vals1
```

```
array([1, None, 2, 3], dtype=object)
```

This `dtype=object` means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. The downside of using `None` in this way is that operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types:

```
%timeit np.arange(1E6, dtype=int).sum()
```

```
2.73 ms ± 288 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit np.arange(1E6, dtype=object).sum()
```

```
92.1 ms ± 3.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Further, because Python does not support arithmetic operations with `None`, then aggregations like `sum()` or `min()` will generally lead to an error:

```
vals1.sum()
```

```
 TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

For this reason, pandas does not use `None` as a sentinel in its numerical arrays.

NaN: Missing numerical data

The other missing data sentinel, `NaN` (acronym for *Not a Number*), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

```
vals2 = np.array([1, np.nan, 3, 4])
vals2
```

```
array([ 1., nan,  3.,  4.])
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. Keep in mind that `NaN` is a bit like a data virus–it infects any other object it touches. Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`:

```
1 + np.nan
```

```
nan
```

```
0 * np.nan
```

```
nan
```

This means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

```
vals2.sum(), vals2.min(), vals2.max()
```

```
(nan, nan, nan)
```

that said, NumPy does provide NaN-aware versions of aggregations that will ignore these missing values:

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

```
(8.0, 1.0, 4.0)
```

The main downside of `NaN` is that it is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

NaN and None in Pandas

`NaN` and `None` both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

```
pd.Series([1, np.nan, 2, None])
```

```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be upcast to a floating-point type to accommodate the NA:

```
x = pd.Series(range(2), dtype=int)
x
```

```
0    0
1    1
dtype: int64
```

```
x[0] = None
x
```

```
0    NaN
1    1.0
dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value. (Be aware that there is a proposal to add a native integer NA to Pandas in the future; as of this writing, it has not been included).

While this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/casting approach works quite well in practice and in my experience only rarely causes issues.

The following table lists the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

Pandas Nullable Dtypes

In early versions of Pandas, `NaN` and `None` as sentinel values were the only missing data representation available. The primary difficulty of this is the implicit type casting: for example there was no way to represent a true integer array with missing data.

To address this difficulty, Pandas later added *nullable dtypes*, which are distinguished from regular dtypes by capitalization of their name (e.g. `pd.Int32` vs `np.int32`. For backward compatibility, these nullable dtypes are only used if specifically requested.

For example, here is a `Series` of integers with missing data, created from a list containing all three available markers of missing data:

```
pd.Series([1, np.nan, 2, None, pd.NA], dtype='Int32')
```

```
0       1
1    <NA>
2       2
3    <NA>
4    <NA>
dtype: Int32
```

This representation can be used interchangeably with the others in all the operations explored through the rest of this section.

Operating on Null Values

As we have seen, Pandas treats `None`, `NaN`, and `NA` as essentially interchangeable for indicating missing or null values. To facilitate this convention, pandas provides several methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()`: Generate a boolean mask indicating missing values
- `notnull()`: Opposite of `isnull()`
- `dropna()`: Return a filtered version of the data
- `fillna()`: Return a copy of the data with missing values filled or imputed

We will conclude this section with a brief exploration and demonstration of these routines.

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a Boolean mask over the data. For example:

```
data = pd.Series([1, np.nan, 'hello', None])
```

```
data.isnull()
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in "Data Indexing and Selection", Boolean masks can be used directly as a `Series` or `DataFrame` index:

```
data[data.notnull()]
```

```
0        1
2    hello
dtype: object
```

The `isnull()` and `notnull()` methods produce similar Boolean results for `DataFrame` objects.

Dropping null values

In addition to these masking methods, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). For a `Series`, the result is straightforward:

```
data.dropna()
```

```
0        1
2    hello
dtype: object
```

For a `DataFrame`, there are more options. Consider the following `DataFrame`:

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

	0	**1**	**2**
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

We cannot drop single values from a `DataFrame`; we can only drop entire rows or columns. Depending on the application, you might want one or the other, so `dropna()` includes a number of options for a `DataFrame`.

By default, `dropna()` will drop all rows in which *any* null value is present:

```
df.dropna()
```

	0	1	2
1	2.0	3.0	5

Alternatively, you can drop NA values along a different axis; `axis=1` or `axis='columns'` drops all columns containing a null value:

```
df.dropna(axis='columns')
```

	2
0	2
1	5
2	6

But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

The default is `how='any'`, such that any row or column containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns that are *all* null values:

```
df[3] = np.nan
df
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
df.dropna(axis='columns', how='all')
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
df.dropna(axis='rows', thresh=3)
```

	0	1	2	3
1	2.0	3.0	5	NaN

Here the first and last row have been dropped, because they contain only two non-null values.

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

Consider the following `Series`:

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'), dtype='Int32')
data
```

```
a       1
b    <NA>
c       2
d    <NA>
e       3
dtype: Int32
```

We can fill NA entries with a single value, such as zero:

```
data.fillna(0)
```

```
a    1
b    0
c    2
d    0
e    3
dtype: Int32
```

We can specify a forward-fill to propagate the previous value forward:

```
# forward-fill
data.fillna(method='ffill')
```

```
a    1
b    1
c    2
d    2
e    3
dtype: Int32
```

Or we can specify a back-fill to propagate the next values backward:

```
# back-fill
data.fillna(method='bfill')
```

```
a    1
b    2
c    2
d    3
e    3
dtype: Int32
```

In the case of a `DataFrame`, the options are similar, but we can also specify an `axis` along which the fills take place:

```
df
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
df.fillna(method='ffill', axis=1)
```

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

Notice that if a previous value is not available during a forward fill, the NA value remains.

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas `Series` and `DataFrame` objects, respectively. Often it is useful to go beyond this and store higher-dimensional data–that is, data indexed by more than one or two keys. Early Pandas versions provided `Panel` and `Panel4D` objects that could be thought of 3D or 4D analogs to the 2D `DataFrame`, but they were somewhat clunky to use in practice. A a far more common pattern for handling higher-dimensional data is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional `Series` and two-dimensional `DataFrame` objects. (If you're interested in true N-dimensional arrays with pandas-style flexible indices, you can look into the excelleng xarray package).

In this section, we'll explore the direct creation of `MultiIndex` objects, considerations when indexing, slicing, and computing statistics across multiply indexed data, and useful routines for converting between simple and hierarchically indexed representations of your data.

We begin with the standard imports:

```python
import pandas as pd
import numpy as np
```

A Multiply Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional `Series`. For concreteness, we will consider a series of data where each point has a character and numerical key.

The bad way

Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```python
index = [('California', 2010), ('California', 2020),
         ('New York', 2010), ('New York', 2020),
         ('Texas', 2010), ('Texas', 2020)]
populations = [37253956, 39538223,
               19378102, 20201249,
               25145561, 29145505]
pop = pd.Series(populations, index=index)
pop
```

```
(California, 2010)    37253956
(California, 2020)    39538223
(New York, 2010)      19378102
(New York, 2020)      20201249
(Texas, 2010)         25145561
(Texas, 2020)         29145505
dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this tuple index:

```python
pop[('California', 2020):('Texas', 2010)]
```

```
(California, 2020)    39538223
(New York, 2010)      19378102
(New York, 2020)      20201249
(Texas, 2010)         25145561
dtype: int64
```

But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

```python
pop[[i for i in pop.index if i[1] == 2010]]
```

```
(California, 2010)    37253956
(New York, 2010)      19378102
```

```
(Texas, 2010)         25145561
dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

The Better Way: Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas `MultiIndex` type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

```
index = pd.MultiIndex.from_tuples(index)
```

The `MultiIndex` represents multiple *levels* of indexing–in this case, the state names and the years, as well as multiple *labels* for each data point which encode these levels.

If we re-index our series with this `MultiIndex`, we see the hierarchical representation of the data:

```
pop = pop.reindex(index)
pop

California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Here the first two columns of the `Series` representation show the multiple index values, while the third column shows the data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2020, we can use the Pandas slicing notation:

```
pop[:, 2020]

California    39538223
New York      20201249
Texas         29145505
dtype: int64
```

The result is a singly indexed Series with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. We'll now further discuss this sort of indexing operation on hieararchically indexed data.

MultiIndex as extra dimension

You might notice something else here: we could easily have stored the same data using a simple `DataFrame` with index and column labels. In fact, Pandas is built with this equivalence in mind. The `unstack()` method will quickly convert a multiply indexed `Series` into a conventionally indexed `DataFrame`:

```
pop_df = pop.unstack()
pop_df
```

	2010	2020
California	37253956	39538223
New York	19378102	20201249
Texas	25145561	29145505

Naturally, the `stack()` method provides the opposite operation:

```
pop_df.stack()
```

```
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Seeing this, you might wonder why would we would bother with hierarchical indexing at all. The reason is simple: just as we were able to use multi-indexing to manipulate two-dimensional data within a one-dimensional `Series`, we can also use it to manipulate data of three or more dimensions in a `Series` or `DataFrame`. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent. Concretely, we might want to add another column of demographic data for each state at each year (say, population under 18) ; with a `MultiIndex` this is as easy as adding another column to the `DataFrame`:

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9284094, 8898092,
                                   4318033, 4181528,
                                   6879014, 7432474]})
pop_df
```

		total	under18
California	2010	37253956	9284094
	2020	39538223	8898092
New York	2010	19378102	4318033
	2020	20201249	4181528
Texas	2010	25145561	6879014
	2020	29145505	7432474

In addition, all the ufuncs and other functionality discussed in "Operating on Data in Pandas" work with hierarchical indices as well. Here we compute the fraction of people under 18 by year, given the above data:

```
f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

	2010	2020
California	0.249211	0.225050
New York	0.222831	0.206994
Texas	0.273568	0.255013

This allows us to easily and quickly manipulate and explore even high-dimensional data.

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed `Series` or `DataFrame` is to simply pass a list of two or more index arrays to the constructor. For example:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])
df
```

		data1	data2
a	1	0.748464	0.561409
	2	0.379199	0.622461
b	1	0.701679	0.687932
	2	0.436200	0.950664

The work of creating the `MultiIndex` is done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
data = {('California', 2010): 37253956,
        ('California', 2020): 39538223,
        ('New York', 2010): 19378102,
        ('New York', 2020): 20201249,
        ('Texas', 2010): 25145561,
        ('Texas', 2020): 29145505}
pd.Series(data)
```

```
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Nevertheless, it is sometimes useful to explicitly create a `MultiIndex`; we'll see a couple of these methods here.

Explicit MultiIndex constructors

For more flexibility in how the index is constructed, you can instead use the class method constructors available in the `pd.MultiIndex`. For example, as we did before, you can construct the `MultiIndex` from a simple list of arrays giving the index values within each level:

```
pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
```

```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

You can construct it from a list of tuples giving the multiple index values of each point:

```
pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
```

```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

You can even construct it from a Cartesian product of single indices:

```
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
```

```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

Similarly, you can construct the `MultiIndex` directly using its internal encoding by passing `levels` (a list of lists containing available index values for each level) and `codes` (a list of lists that reference these labels):

```
pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
              codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

Any of these objects can be passed as the `index` argument when creating a `Series` or `Dataframe`, or be passed to the `reindex` method of an existing `Series` or `DataFrame`.

MultiIndex level names

Sometimes it is convenient to name the levels of the `MultiIndex`. This can be accomplished by passing the `names` argument to any of the above `MultiIndex` constructors, or by setting the `names` attribute of the index after the fact:

```
pop.index.names = ['state', 'year']
pop
```

```
state       year
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
```

```
Texas       2010    25145561
            2020    29145505
dtype: int64
```

With more involved datasets, this can be a useful way to keep track of the meaning of various index values.

MultiIndex for columns

In a `DataFrame`, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

```python
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

	subject	**Bob**		**Guido**		**Sue**	
	type	**HR**	**Temp**	**HR**	**Temp**	**HR**	**Ter**
year	**visit**						
2013	1	30.0	38.0	56.0	38.3	45.0	35.8
	2	47.0	37.1	27.0	36.0	37.0	36.4
2014	1	51.0	35.9	24.0	36.7	32.0	36.2
	2	49.0	36.3	48.0	39.2	31.0	35.7

This is fundamentally four-dimensional data, where the dimensions are the subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top-level column by the person's name and get a full `DataFrame` containing just that person's information:

```python
health_data['Guido']
```

	type	HR	Temp
year	**visit**		
2013	1	56.0	38.3
	2	27.0	36.0
2014	1	24.0	36.7
	2	48.0	39.2

Indexing and Slicing a MultiIndex

Indexing and slicing on a `MultiIndex` is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply indexed `Series`, and then multiply-indexed `DataFrame` objects.

Multiply indexed Series

Consider the multiply indexed `Series` of state populations we saw earlier:

```
pop
```

```
state       year
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

We can access single elements by indexing with multiple terms:

```
pop['California', 2010]
```

```
37253956
```

The `MultiIndex` also supports *partial indexing*, or indexing just one of the levels in the index. The result is another `Series`, with the lower-level indices maintained:

```
pop['California']
```

```
year
2010    37253956
2020    39538223
dtype: int64
```

Partial slicing is available as well, as long as the `MultiIndex` is sorted (see discussion in <<section-#sorted-and-unsorted-indices[sorted and unsorted indices]):

```
poploc['california':'new york']
```

```
state       year
california  2010    37253956
            2020    39538223
new york    2010    19378102
```

```
              2020    20201249
dtype: int64
```

with sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index:

```
pop[:, 2010]
```

```
state
california    37253956
new york      19378102
texas         25145561
dtype: int64
```

other types of indexing and selection (discussed in link:0302-data-indexing-and-selection>>) work as well; for example, selection based on Boolean masks:

```
pop[pop > 22000000]
```

```
state       year
California  2010    37253956
            2020    39538223
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Selection based on fancy indexing also works:

```
pop[['California', 'Texas']]
```

```
state       year
California  2010    37253956
            2020    39538223
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Multiply indexed DataFrames

A multiply indexed `DataFrame` behaves in a similar manner. Consider our toy medical `DataFrame` from before:

```
health_data
```

	subject	**Bob**		**Guido**		**Sue**	
	type	**HR**	**Temp**	**HR**	**Temp**	**HR**	**Ter**
year	**visit**						
2013	1	30.0	38.0	56.0	38.3	45.0	35.8
	2	47.0	37.1	27.0	36.0	37.0	36.4
2014	1	51.0	35.9	24.0	36.7	32.0	36.2
	2	49.0	36.3	48.0	39.2	31.0	35.7

Remember that columns are primary in a `DataFrame`, and the syntax used for multiply indexed `Series` applies to the columns. For example, we can recover Guido's heart rate data with a simple operation:

```
health_data['Guido', 'HR']
```

```
year  visit
2013  1        56.0
      2        27.0
2014  1        24.0
      2        48.0
Name: (Guido, HR), dtype: float64
```

Also, as with the single-index case, we can use the `loc`, `iloc`, and `ix` indexers introduced in “Data Indexing and Selection”. For example:

```
health_data.iloc[:2, :2]
```

	subject	Bob	
	type	**HR**	**Temp**
year	**visit**		
	1	30.0	38.0
2013	2	47.0	37.1

These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in `loc` or `iloc` can be passed a tuple of multiple indices. For example:

```
health_data.loc[:, ('Bob', 'HR')]
```

```
year  visit
2013  1        30.0
      2        47.0
2014  1        51.0
      2        49.0
Name: (Bob, HR), dtype: float64
```

Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:

```
health_data.loc[(:, 1), (:, 'HR')]
```

```
 SyntaxError: invalid syntax (3311942670.py, line 1)
```

You could get around this by building the desired slice explicitly using Python’s built-in `slice()` function, but a better way in this context is to use an `IndexSlice` object, which Pandas provides for precisely this situation. For example:

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

	subject	Bob	Guido	Sue
	type	HR	HR	HR
year	visit			
2013	1	30.0	56.0	45.0
2014	1	51.0	24.0	32.0

There are so many ways to interact with data in multiply indexed `Series` and *DataFrame*s, and as with many tools in this book the best way to become familiar with them is to try them out!

Rearranging Multi-Indices

One of the keys to working with multiply indexed data is knowing how to effectively transform the data. There are a number of operations that will preserve all the information in the dataset, but rearrange it for the purposes of various computations. We saw a brief example of this in the `stack()` and `unstack()` methods, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

Sorted and unsorted indices

Earlier, we briefly mentioned a caveat, but we should emphasize it more here. *Many of the `MultiIndex` slicing operations will fail if the index is not sorted.* Let's take a look at this here.

We'll start by creating some simple multiply indexed data where the indices are *not lexographically sorted*:

```
index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
data
```

```
char  int
a     1      0.280341
      2      0.097290
c     1      0.206217
      2      0.431771
b     1      0.100183
      2      0.015851
dtype: float64
```

If we try to take a partial slice of this index, it will result in an error:

```
try:
    data['a':'b']
except KeyError as e:
    print("KeyError", e)
```

```
KeyError 'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Although it is not entirely clear from the error message, this is the result of the MultiIndex not being sorted. For various reasons, partial slices and other similar operations require the levels in the `MultiIndex` to be in sorted (i.e., lexographical) order. Pandas provides a number of convenience routines to perform this type of sorting; examples are the `sort_index()` and `sortlevel()` methods of the `DataFrame`. We'll use the simplest, `sort_index()`, here:

```
data = data.sort_index()
data
```

```
char  int
a     1      0.280341
      2      0.097290
b     1      0.100183
      2      0.015851
c     1      0.206217
      2      0.431771
dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
data['a':'b']
```

```
char  int
a     1      0.280341
      2      0.097290
b     1      0.100183
      2      0.015851
dtype: float64
```

Stacking and unstacking indices

As we saw briefly before, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation, optionally specifying the level to use:

```
pop.unstack(level=0)
```

state	California	New York	Texas
year			
2010	37253956	19378102	25145561
2020	39538223	20201249	29145505

```
pop.unstack(level=1)
```

year	2010	2020
state		
California	37253956	39538223
New York	19378102	20201249
Texas	25145561	29145505

The opposite of `unstack()` is `stack()`, which here can be used to recover the original series:

```
pop.unstack().stack()
```

```
state       year
California  2010    37253956
            2020    39538223
New York    2010    19378102
            2020    20201249
Texas       2010    25145561
            2020    29145505
dtype: int64
```

Index setting and resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the `reset_index` method. Calling this on the population dictionary will result in a `DataFrame` with a *state* and *year* column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

```
pop_flat = pop.reset_index(name='population')
pop_flat
```

	state	year	population
0	California	2010	37253956
1	California	2020	39538223
2	New York	2010	19378102
3	New York	2020	20201249
4	Texas	2010	25145561
5	Texas	2020	29145505

Often when working with data in the real world, the raw input data looks like this and it's useful to build a `MultiIndex` from the column values. This can be done with the `set_index` method of the `DataFrame`, which returns a multiply indexed `DataFrame`:

```
pop_flat.set_index(['state', 'year'])
```

		population
state	**year**	
California	2010	37253956
	2020	39538223
New York	2010	19378102
	2020	20201249
Texas	2010	25145561
	2020	29145505

In practice, this type of reindexing to be one of the more useful patterns when exploring real-world datasets.

Combining Datasets: Concat and Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets. `Series` and $DataFrame$s are built with this type of operation in mind, and Pandas includes functions and methods that make this sort of data wrangling fast and straightforward.

Here we'll take a look at simple concatenation of `Series` and ``DataFrame`s with the `pd.concat`` function; later we'll dive into more sophisticated in-memory merges and joins implemented in Pandas.

We begin with the standard imports:

```
import pandas as pd
import numpy as np
```

For convenience, we'll define this function which creates a `DataFrame` of a particular form that will be useful below:

```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

In addition, we'll create a quick class that allows us to display multiple ``DataFrame`s side by side. The code makes use of the special `_repr_html_`` method, which IPython/Jupyter uses to implement its rich object display:

```
class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}{1}
    """
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

The use of this will become clearer as we continue our discussion in the following section.

Recall: Concatenation of NumPy Arrays

Concatenation of `Series` and `DataFrame` objects behaves similarly to concatenation of Numpy arrays, which can be done via the `np.concatenate` function as discussed in "The Basics of NumPy Arrays". Recall that with it, you can combine the contents of two or more arrays into a single array:

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The first argument is a list or tuple of arrays to concatenate. Additionally, in the case of multi-dimensional arrays, it takes an `axis` keyword that allows you to specify the axis along which the result will be concatenated:

```
x = [[1, 2],
     [3, 4]]
np.concatenate([x, x], axis=1)
```

```
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

Simple Concatenation with `pd.concat`

The `pd.concat()` function provides a similar syntax to `np.concatenate` but contains a number of options that we'll discuss momentarily:

```
# Signature in Pandas v1.3.5
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None,
          levels=None, names=None, verify_integrity=False,
          sort=False, copy=True)
```

`pd.concat()` can be used for a simple concatenation of `Series` or `DataFrame` objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

It also works to concatenate higher-dimensional objects, such as *DataFrames*:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1
```

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

It’s default behavior is to concatenate row-wise within the `DataFrame` (i.e., `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis='columns')")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df3
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

We could have equivalently specified `axis=1`; here we’ve used the more intuitive `axis='columns'`.

Duplicate indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation *preserves indices*, even if the result will have duplicate indices! Consider this simple example:

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index  # make indices match
display('x', 'y', 'pd.concat([x, y])')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>x
```

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

Notice the repeated indices in the result. While this is valid within `DataFrame`s, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Treating repeated indices as an error

If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to True, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: Indexes have overlapping values: Int64Index([0, 1], dtype='int64')
```

Ignoring the index

Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting `DataFrame`:

```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>x
```

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

Adding MultiIndex keys

Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>x
```

		A	B
	0	A0	B0
x	1	A1	B1
	0	A2	B2
y	1	A3	B3

The result is a multiply indexed `DataFrame`, and we can use the tools discussed in “Hierarchical Indexing” to transform this data into the representation we’re interested in.

Concatenation with joins

In the simple examples we just looked at, we were mainly concatenating `DataFrame`s with shared column names. In practice, data from different sources might have different sets of column names, and `pd.concat` offers several options in this case. Consider the concatenation of the following two *DataFrame*s, which have some (but not all!) columns in common:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df5
```

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

The default behavior is to fill entries for which no data is available are filled with NA values. To change this, we can adjust the `join` parameter of the concatenate function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'`:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df5
```

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

Another useful pattern is to use the `reindex` method before concatenation for finer control over which columns are dropped

```
pd.concat([df5, df6.reindex(df5.columns, axis=1)])
```

	A	B	C
1	A1	B1	C1
2	A2	B2	C2
3	NaN	B3	C3
4	NaN	B4	C4

The `append()` method

Because direct array concatenation is so common, `Series` and `DataFrame` objects have an `append` method that can accomplish the same thing in fewer keystrokes. For example, in place of `pd.concat([df1, df2])`, you can use `df1.append(df2)`:

```
display('df1', 'df2', 'df1.append(df2)')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1
```

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

Keep in mind that unlike the `append()` and `extend()` methods of Python lists, the `append()` method in Pandas does not modify the original object–instead it creates a new object with the combined data. It also is not a very efficient method, because it involves creation of a new index *and* data buffer. Thus, if you plan to do multiple append operations, it is generally better to build a list of DataFrame`s and pass them all at once to the `concat() function.

In the next section, we'll look at another more powerful approach to combining data from multiple sources, the database-style merges/joins implemented in `pd.merge`. For more information on `concat()`, `append()`, and related functionality, see the "Merge, Join, and Concatenate" section of the Pandas documentation.

Combining Datasets: Merge and Join

One important feature offered by Pandas is its high-performance, in-memory join and merge operations, which you may be familiar with if you have ever worked with databases. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice.

For convenience, we will again define the `display()` functionality from the previous section:

```
import pandas as pd
import numpy as np

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}{1}
    """
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Relational Algebra

The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data that forms the conceptual foundation of operations available in most databases. The strength of the relational algebra approach is that it proposes several fundamental operations, which become the building blocks of more complicated operations on any dataset. With this lexicon of fundamental operations implemented efficiently in a database or other program, a wide range of fairly complicated composite operations can be performed.

Pandas implements several of these fundamental building-blocks in the `pd.merge()` function and the related `join()` method of `Series` and `Dataframe` objects. As we will see, these let you efficiently link data from different sources.

Categories of Joins

The `pd.merge()` function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

One-to-one joins

Perhaps the simplest type of merge expresion is the one-to-one join, which is in many ways very similar to the column-wise concatenation seen in "Combining Datasets: Concat and Append". As a concrete example, consider the following two `DataFrame` objects which contain information on several employees in a company:

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1
```

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

```
df3 = pd.merge(df1, df2)
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

The `pd.merge()` function recognizes that each `DataFrame` has an “employee” column, and automatically joins using this column as a key. The result of the merge is a new `DataFrame` that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the “employee” column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index (see the `left_index` and `right_index` keywords, discussed momentarily).

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting `DataFrame` will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df3
```

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

The resulting `DataFrame` has an aditional column with the "supervisor" information, where the information is repeated in one or more locations as required by the inputs.

Many-to-many joins

Many-to-many joins may be a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a `DataFrame` showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'software', 'math',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1
```

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	software
3	Jake	Engineering	math
4	Lisa	Engineering	software
5	Lisa	Engineering	math
6	Sue	HR	spreadsheets
7	Sue	HR	organization

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as the one we're working with here. In the following section we'll consider some of the options provided by `pd.merge()` that enable you to tune how the join operations work.

Specification of the Merge Key

We’ve already seen the default behavior of `pd.merge()`: it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

```
display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

This option works only if both the left and right $DataFrame$s have the specified column name.

The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as “name” rather than “employee”. In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee", right_on="name")')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1
```

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

The result has a redundant column that we can drop if desired–for example, by using the `DataFrame.drop()` method:

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

The `left_index` and `right_index` keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1a
```

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()`:

```
display('df1a', 'df2a',
        "pd.merge(df1a, df2a, left_index=True, right_index=True)")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1a
```

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

For convenience, Pandas includes the `DataFrame.join()` method, which performs an index-based merge without extra keywords:

```
df1a.join(df2a)
```

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

If you’d like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

```
display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df1a
```

	group	name	salary
0	Accounting	Bob	70000
1	Engineering	Jake	80000
2	Engineering	Lisa	120000
3	HR	Sue	90000

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the “Merge, Join, and Concatenate” section of the Pandas documentation.

Specifying Set Arithmetic for Joins

In all the preceding examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other. Consider this example:

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df6
```

	name	food	drink
0	Mary	bread	wine

Here we have merged two datasets that have only a single “name” entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to `"inner"`:

```
pd.merge(df6, df7, how='inner')
```

	name	food	drink
0	Mary	bread	wine

Other options for the `how` keyword are `'outer'`, `'left'`, and `'right'`. An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df6
```

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example:

```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df6
```

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

The output rows now correspond to the entries in the left input. Using `how='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the preceding join types.

Overlapping Column Names: The `suffixes` Keyword

Finally, you may end up in a case where your two input *DataFrame*s have conflicting column names. Consider this example:

```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df8
```

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

Because the output would have two conflicting column names, the merge function automatically appends a suffix `_x` or `_y` to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

```
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

For more information on these patterns, see "Aggregation and Grouping" where we dive a bit deeper into relational algebra. Also see the Pandas "Merge, Join and Concatenate" documentation for further discussion of these topics.

Example: US States Data

Merge and join operations come up most often when combining data from different sources. Here we will consider an example of some data about US states and their populations. The data files can be found at *http://github.com/jakevdp/data-USstates/:*

```
# Following are commands to download the data
# repo = "https://raw.githubusercontent.com/jakevdp/data-USstates/master"
# !cd data && curl -O {repo}/state-population.csv
# !cd data && curl -O {repo}/state-areas.csv
# !cd data && curl -O {repo}/state-abbrevs.csv
```

Let's take a look at the three datasets, using the Pandas `read_csv()` function:

```
pop = pd.read_csv('data/state-population.csv')
areas = pd.read_csv('data/state-areas.csv')
abbrevs = pd.read_csv('data/state-abbrevs.csv')

display('pop.head()', 'areas.head()', 'abbrevs.head()')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>pop.head()
```

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

Given this information, say we want to compute a relatively straightforward result: rank US states and territories by their 2010 population density. We clearly have the data here to find this result, but we'll have to combine the datasets to find the result.

We'll start with a many-to-one merge that will give us the full state name within the population `DataFrame`. We want to merge based on the `state/region` column of `pop`, and the `abbreviation` column of `abbrevs`. We'll use `how='outer'` to make sure no data is thrown away due to mismatched labels.

```
merged = pd.merge(pop, abbrevs, how='outer',
                  left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', axis=1) # drop duplicate info
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

Let's double-check whether there were any mismatches here, which we can do by looking for rows with nulls:

```
merged.isnull().any()
```

```
state/region    False
ages            False
year            False
population       True
state            True
dtype: bool
```

Some of the `population` info is null; let's figure out which these are!

```
merged[merged['population'].isnull()].head()
```

	state/region	ages	year	population	state
2448	PR	under18	1990	NaN	NaN
2449	PR	total	1990	NaN	NaN
2450	PR	total	1991	NaN	NaN
2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available from the original source.

More importantly, we see also that some of the new `state` entries are also null, which means that there was no corresponding entry in the `abbrevs` key! Let's figure out which regions lack this match:

```
merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

```
array(['PR', 'USA'], dtype=object)
```

We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in the state abbreviation key. We can fix these quickly by filling in appropriate entries:

```
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
merged.isnull().any()
```

```
state/region    False
ages            False
year            False
population       True
state           False
dtype: bool
```

No more nulls in the `state` column: we're all set!

Now we can merge the result with the area data using a similar procedure. Examining our results, we will want to join on the `state` column in both:

```
final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Again, let's check for nulls to see if there were any mismatches:

```
final.isnull().any()
```

```
state/region     False
ages             False
year             False
population        True
state            False
area (sq. mi)     True
dtype: bool
```

There are nulls in the `area` column; we can take a look to see which regions were ignored here:

```
final['state'][final['area (sq. mi)'].isnull()].unique()
```

```
array(['United States'], dtype=object)
```

We see that our `areas DataFrame` does not contain the area of the United States as a whole. We could insert the appropriate value (using the sum of all state areas, for instance), but in this case we'll just drop the null values because the population density of the entire United States is not relevant to our current discussion:

```
final.dropna(inplace=True)
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2000, and the total population. We'll use the `query()` function to do this quickly (this requires the `numexpr` package to be installed; see "High-Performance Pandas: eval() and query()"):

```
data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()
```

	state/region	ages	year	population	state	area (sq. mi)
3	AL	total	2010	4785570.0	Alabama	52423.0
91	AK	total	2010	713868.0	Alaska	656425.0
101	AZ	total	2010	6408790.0	Arizona	114006.0
189	AR	total	2010	2922280.0	Arkansas	53182.0
197	CA	total	2010	37333601.0	California	163707.0

Now let's compute the population density and display it in order. We'll start by re-indexing our data on the state, and then compute the result:

```
data2010.set_index('state', inplace=True)
density = data2010['population'] / data2010['area (sq. mi)']

density.sort_values(ascending=False, inplace=True)
density.head()
```

```
state
District of Columbia    8898.897059
Puerto Rico             1058.665149
New Jersey              1009.253268
Rhode Island             681.339159
Connecticut              645.600649
dtype: float64
```

The result is a ranking of US states, plus Washington, DC and Puerto Rico, in order of their 2010 population density, in residents per square mile. We can see that by far the densest region in this dataset is Washington, DC (i.e., the District of Columbia); among states, the densest is New Jersey.

We can also check the end of the list:

```
density.tail()
```

```
state
South Dakota    10.583512
North Dakota     9.537565
Montana          6.736171
Wyoming          5.768079
Alaska           1.087509
dtype: float64
```

We see that the least dense state, by far, is Alaska, averaging slightly over one resident per square mile.

This type of messy data merging is a common task when trying to answer questions using real-world data sources. I hope that this example has given you an idea of the ways you can combine tools we've covered in order to gain insight from your data!

Aggregation and Grouping

An fundamental piece of analysis of large data is efficient summarization: computing aggregations like `sum()`, `mean()`, `median()`, `min()`, and `max()`, in which a single number summarizes aspects of a potentially large dataset. In this section, we'll explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a `groupby`.

For convenience, we'll use the same `display` magic function that we've seen in previous sections:

```
import numpy as np
import pandas as pd

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}{1}
    """
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Planets Data

Here we will use the Planets dataset, available via the Seaborn package (see [Link to Come]). It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short). It can be downloaded with a simple Seaborn command:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

```
(1035, 6)
```

```
planets.head()
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

This has some details on the 1,000+ extrasolar planets discovered up to 2014.

Simple Aggregation in Pandas

Earlier, we explored some of the data aggregations available for NumPy arrays ("Aggregations: Min, Max, and Everything In Between"). As with a one-dimensional NumPy array, for a Pandas `Series` the aggregates return a single value:

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

```
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64
```

```
ser.sum()
```

```
2.811925491708157
```

```
ser.mean()
```

```
0.5623850983416314
```

For a `DataFrame`, by default the aggregates return results within each column:

```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
```

	A	B
0	0.155995	0.020584
1	0.058084	0.969910
2	0.866176	0.832443
3	0.601115	0.212339
4	0.708073	0.181825

```
df.mean()
```

```
A    0.477888
B    0.443420
dtype: float64
```

By specifying the `axis` argument, you can instead aggregate within each row:

```
df.mean(axis='columns')
```

```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Pandas Series and DataFrame`s include all of the common aggregates mentioned in <<section-0204-computation-on-arrays-aggregates>>; in addition, there is a convenience method `describe() that computes several common aggregates for each column and returns the result. Let’s use this on the Planets data, for now dropping rows with missing values:

```
planets.dropna().describe()
```

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

This method helps us understand the overall properties of a dataset. For example, we see in the `year` column that although exoplanets were discovered as far back as 1989, half of all planets in the dataset were not discovered until 2010 or after. This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

The following table summarizes some other built-in Pandas aggregations:

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

These are all methods of `DataFrame` and `Series` objects.

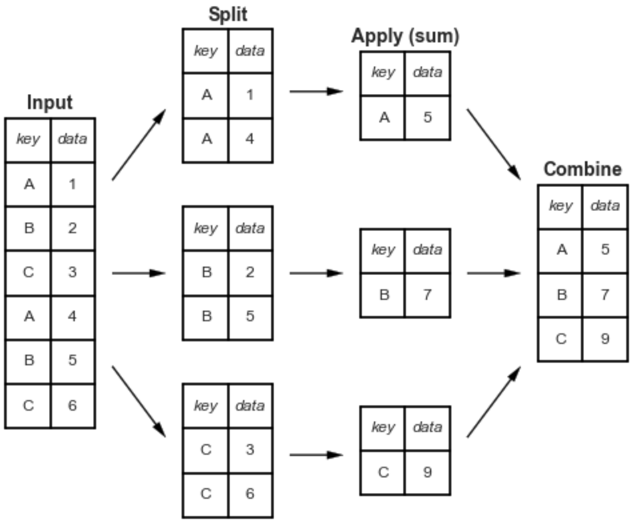
To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the `groupby` operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called `groupby` operation. The name “group by” comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine*.

Split, apply, combine

A canonical example of this split-apply-combine operation, where the “apply” is a summation aggregation, is illustrated in this figure:



[Link to Come]

This illustrates what the `groupby` operation accomplishes:

- The *split* step involves breaking up and grouping a `DataFrame` depending on the value of the specified key.
- The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
- The *combine* step merges the results of these operations into an output array.

While this could certainly be done manually using some combination of the masking, aggregation, and merging commands covered earlier, an important realization is that *the intermediate splits do not need to be explicitly instantiated*. Rather, the `groupby` can (often) do this in a single pass over the data, updating the sum, mean, count, min, or other aggregate for each group along the way. The power of the `groupby` is that it abstracts away these steps: the user need not think about *how* the computation is done under the hood, but rather thinks about the *operation as a whole*.

As a concrete example, let’s take a look at using Pandas for the computation shown in this diagram. We’ll start by creating the input `DataFrame`:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

The most basic split-apply-combine operation can be computed with the `groupby()` method of $DataFrame$s, passing the name of the desired key column:

```
df.groupby('key')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x11d241e20>
```

Notice that what is returned is not a set of `DataFrame` objects, but a `DataFrameGroupBy` object. This object is where the magic is: you can think of it as a special view of the `DataFrame`, which is poised to dig into the groups but does no actual computation until the aggregation is applied. This “lazy evaluation” approach means that common aggregates can be implemented efficiently in a way that is almost transparent to the user.

To produce a result, we can apply an aggregate to this `DataFrameGroupBy` object, which will perform the appropriate apply/combine steps to produce the desired result:

```
df.groupby('key').sum()
```

	data
key	
A	3
B	5
C	7

The `sum()` method is just one possibility here; you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid `DataFrame` operation, as we will see in the following discussion.

The GroupBy object

The `GroupBy` object is a flexible abstraction: in many ways, it can be treated as simply a collection of $DataFrame$s, though it is doing more sophisticated things under the hood. Let’s see some examples using the Planets data.

Perhaps the most important operations made available by a `GroupBy` are *aggregate*, *filter*, *transform*, and *apply*. We’ll discuss each of these more fully in Filter,Transform,Apply below, but before that let’s introduce some of the other functionality that can be used with the basic `GroupBy` operation.

Column indexing

The `GroupBy` object supports column indexing in the same way as the `DataFrame`, and returns a modified `GroupBy` object. For example:

```
planets.groupby('method')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x11d1bc820>
```

```
planets.groupby('method')['orbital_period']
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x11d1bcd60>
```

Here we've selected a particular `Series` group from the original `DataFrame` group by reference to its column name. As with the `GroupBy` object, no computation is done until we call some aggregate on the object:

```
planets.groupby('method')['orbital_period'].median()
```

```
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

This gives an idea of the general scale of orbital periods (in days) that each method is sensitive to.

Iteration over groups

The `GroupBy` object supports direct iteration over the groups, returning each group as a `Series` or `DataFrame`:

```
for (method, group) in planets.groupby('method'):
    print("{0:30s} shape={1}".format(method, group.shape))
```

```
Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

This can be useful for manual inspection of groups for the sake of debugging, but it is often much faster to use the built-in `apply` functionality, which we will discuss momentarily.

Dispatch methods

Through some Python class magic, any method not explicitly implemented by the `GroupBy` object will be passed through and called on the groups, whether they are `DataFrame` or `Series` objects. For example, the `describe()` method is equivalent to calling `describe()` on the `DataFrame` representing each group:

```
planets.groupby('method')['year'].describe().unstack()
```

```
       method
count  Astrometry                          2.0
       Eclipse Timing Variations           9.0
       Imaging                            38.0
       Microlensing                       23.0
       Orbital Brightness Modulation       3.0
                                         ...
max    Pulsar Timing                    2011.0
       Pulsation Timing Variations      2007.0
       Radial Velocity                  2014.0
       Transit                          2014.0
       Transit Timing Variations        2014.0
Length: 80, dtype: float64
```

Looking at this table helps us to better understand the data: for example, the vast majority of planets until 2014 were discovered by the Radial Velocity and Transit methods, though the latter method became common more recently. The newest methods seem to be Transit Timing Variation and Orbital Brightness Modulation, which were not used to discover a new planet until 2011.

Notice that these dispatch methods are applied *to each individual group*, and the results are then combined within `GroupBy` and returned. Again, any valid `DataFrame`/`Series` method can be called in a similar manner on the corresponding `GroupBy` object.

Aggregate, filter, transform, apply

The preceding discussion focused on aggregation for the combine operation, but there are more options available. In particular, `GroupBy` objects have `aggregate()`, `filter()`, `transform()`, and `apply()` methods that efficiently implement a variety of useful operations before combining the grouped data.

For the purpose of the following subsections, we'll use this `DataFrame`:

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                   columns = ['key', 'data1', 'data2'])
df
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation

We're now familiar with `GroupBy` aggregations with `sum()`, `median()`, and the like, but the `aggregate()` method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once. Here is a quick example combining all these:

```
df.groupby('key').aggregate(['min', np.median, max])
```

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Another common pattern is to pass a dictionary mapping column names to operations to be applied on that column:

```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

	data1	data2
key		
A	0	5
B	1	7
C	2	9

Filtering

A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation is larger than some critical value:

```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()",
        "df.groupby('key').filter(filter_func)")
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df
```

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

The filter function should return a Boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the result.

Transformation

While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. A common example is to center the data by subtracting the group-wise mean:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

The apply() method

The `apply()` method lets you apply an arbitrary function to the group results. The function should take a `DataFrame`, and return either a Pandas object (e.g., `DataFrame`, `Series`) or a scalar; the behavior of the *combine* step will be tailored to the type of output returned.

For example, here is an `apply()` that normalizes the first column by the sum of the second:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

df.groupby('key').apply(norm_by_data2)
```

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

`apply()` within a `GroupBy` is flexible: the only criterion is that the function takes a `DataFrame` and returns a Pandas object or scalar; what you do inbetween is up to you!

Specifying the split key

In the simple examples presented before, we split the `DataFrame` on a single column name. This is just one of many options by which the groups can be defined, and we'll go through some other options for group specification here.

A list, array, series, or index providing the grouping keys

The key can be any series or list with a length matching that of the `DataFrame`. For example:

```
L = [0, 1, 0, 1, 2, 0]
df.groupby(L).sum()
```

	data1	data2
0	7	17
1	4	3
2	4	7

Of course, this means there's another, more verbose way of accomplishing the `df.groupby('key')` from before:

```
df.groupby(df['key']).sum()
```

	data1	data2
key		
A	3	8
B	5	7
C	7	12

A dictionary or series mapping index to group

Another method is to provide a dictionary that maps index values to the group keys:

```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
```

```
<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>df2
```

	data1	data2
key		
consonant	12	19
vowel	3	8

Any Python function

Similar to mapping, you can pass any Python function that will input the index value and output the group:

```
df2.groupby(str.lower).mean()
```

	data1	data2
key		
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

A list of valid keys

Further, any of the preceding key choices can be combined to group on a multi-index:

```
df2.groupby([str.lower, mapping]).mean()
```

		data1	data2
key	key		
a	vowel	1.5	4.0
b	consonant	2.5	3.5
c	consonant	3.5	6.0

Grouping example

As an example of this, in a couple lines of Python code we can put all these together and count discovered planets by method and by decade:

```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

decade	1980s	1990s	2000s	2010s
method				
Astrometry	0.0	0.0	0.0	2.0
Eclipse Timing Variations	0.0	0.0	5.0	10.0
Imaging	0.0	0.0	29.0	21.0
Microlensing	0.0	0.0	12.0	15.0
Orbital Brightness Modulation	0.0	0.0	0.0	5.0
Pulsar Timing	0.0	9.0	1.0	1.0
Pulsation Timing Variations	0.0	0.0	1.0	0.0
Radial Velocity	1.0	52.0	475.0	424.0
Transit	0.0	0.0	64.0	712.0
Transit Timing Variations	0.0	0.0	0.0	9.0

This shows the power of combining many of the operations we’ve discussed up to this point when looking at realistic datasets. We quickly gain a coarse understanding of when and how extrasolar planets were detected in the years after the first discovery.

Here I would suggest digging into these few lines of code, and evaluating the individual steps to make sure you understand exactly what they are doing to the result. It’s certainly a somewhat complicated example, but understanding these pieces will give you the means to similarly explore your own data.

Pivot Tables

We have seen how the *groupby* abstraction lets us explore relationships within a dataset. A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data. The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data. The difference between pivot tables and `GroupBy` can sometimes cause confusion; it helps me to think of pivot tables as essentially a *multidimensional* version of `GroupBy` aggregation. That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

Motivating Pivot Tables

For the examples in this section, we'll use the database of passengers on the *Titanic*, available through the Seaborn library (see [Link to Come]):

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')
```

```
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare
0	0	3	male	22.0	1	0	7.25
1	1	1	female	38.0	1	0	71.2
2	1	3	female	26.0	0	0	7.92
3	1	1	female	35.0	1	0	53.1
4	0	3	male	35.0	0	0	8.05

This contains a number of data points on each passenger of that ill-fated voyage, including gender, age, class, fare paid, and much more.

Pivot Tables by Hand

To start learning more about this data, we might begin by grouping according to gender, survival status, or some combination thereof. If you have read the previous section, you might be tempted to apply a *groupby* operation–for example, let's look at survival rate by gender:

```
titanic.groupby('sex')[['survived']].mean()
```

	survived
sex	
female	0.742038
male	0.188908

This gives us some initial insight: overall, three of every four females on board survived, while only one in five males survived!

This is useful, but we might like to go one step deeper and look at survival by both sex and, say, class. Using the vocabulary of groupby, we might proceed using something like this: we *group by* class and gender, *select* survival, *apply* a mean aggregate, *combine* the resulting groups, and then *unstack* the hierarchical index to reveal the hidden multidimensionality. In code:

```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
```

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

This gives us a better idea of how both gender and class affected survival, but the code is starting to look a bit garbled. While each step of this pipeline makes sense in light of the tools we've previously discussed, the long string of code is not particularly easy to read or use. This two-dimensional groupby is common enough that Pandas includes a convenience routine, `pivot_table`, which succinctly handles this type of multi-dimensional aggregation.

Pivot Table Syntax

Here is the equivalent to the preceding operation using the `DataFrame.pivot_table` method.

```
titanic.pivot_table('survived', index='sex', columns='class')
```

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

This is eminently more readable than the manual `groupby` approach, and produces the same result. As you might expect of an early 20th-century transatlantic cruise, the survival gradient favors both women and higher classes. First-class women survived with near certainty (hi, Rose!), while only one in ten third-class men survived (sorry, Jack!).

Multi-level pivot tables

Just as in the groupby, the grouping in pivot tables can be specified with multiple levels, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the `pd.cut` function:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

	class	First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

We can apply the same strategy when working with the columns as well; let's add info on the fare paid using `pd.qcut` to automatically compute quantiles:

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

	fare	(-0.001, 14.454]			(14.454, 512.329]		
	class	First	Second	Third	First	Second	Thi
sex	age						
female	(0, 18]	NaN	1.000000	0.714286	0.909091	1.000000	0.31
	(18, 80]	NaN	0.880000	0.444444	0.972973	0.914286	0.39
male	(0, 18]	NaN	0.000000	0.260870	0.800000	0.818182	0.17
	(18, 80]	0.0	0.098039	0.125000	0.391304	0.030303	0.19

The result is a four-dimensional aggregation with hierarchical indices (see "Hierarchical Indexing"), shown in a grid demonstrating the relationship between the values.

Additional pivot table options

The full call signature of the `DataFrame.pivot_table` method is as follows:

```
# call signature as of Pandas 1.3.5
DataFrame.pivot_table(data, values=None, index=None, columns=None,
                      aggfunc='mean', fill_value=None, margins=False,
                      dropna=True, margins_name='All', observed=False,
                      sort=True)
```

We've already seen examples of the first three arguments; here we'll take a quick look at some of the remaining ones. Two of the options, `fill_value` and `dropna`, have to do with missing data and are fairly straightforward; we will not show examples of them here.

The `aggfunc` keyword controls what type of aggregation is applied, which is a mean by default. As in the GroupBy, the aggregation specification can be a string representing one of several common choices (e.g., `'sum'`, `'mean'`, `'count'`, `'min'`, `'max'`, etc.) or a function that implements an aggregation (e.g., `np.sum()`, `min()`, `sum()`, etc.). Additionally, it can be specified as a dictionary mapping a column to any of the above desired options:

```
titanic.pivot_table(index='sex', columns='class',
                    aggfunc={'survived':sum, 'fare':'mean'})
```

	fare			survived		
class	**First**	**Second**	**Third**	**First**	**Second**	**Third**
sex						
female	106.125798	21.970121	16.118810	91	70	72
male	67.226127	19.741782	12.661633	45	17	47

Notice also here that we've omitted the `values` keyword; when specifying a mapping for `aggfunc`, this is determined automatically.

At times it's useful to compute totals along each grouping. This can be done via the `margins` keyword:

```
titanic.pivot_table('survived', index='sex', columns='class', margins=True)
```

class	First	Second	Third	All
sex				
female	0.968085	0.921053	0.500000	0.742038
male	0.368852	0.157407	0.135447	0.188908
All	0.629630	0.472826	0.242363	0.383838

Here this automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%. The margin label can be specified with the `margins_name` keyword, which defaults to `"All"`.

Example: Birthrate Data

As a more interesting example, let's take a look at the freely available data on births in the United States, provided by the Centers for Disease Control (CDC). This data can be found at *https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv* (this dataset has been analyzed rather extensively by Andrew Gelman and his group; see, for example, this blog post):

```
# shell command to download the data:
# !cd data && curl -O \
#   https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv

births = pd.read_csv('data/births.csv')
```

Taking a look at the data, we see that it's relatively simple–it contains the number of births grouped by date and gender:

```
births.head()
```

	year	month	day	gender	births
0	1969	1	1.0	F	4046
1	1969	1	1.0	M	4440
2	1969	1	2.0	F	4454
3	1969	1	2.0	M	4548
4	1969	1	3.0	F	4548

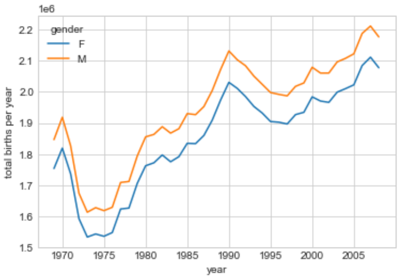
We can start to understand this data a bit more by using a pivot table. Let’s add a decade column, and take a look at male and female births as a function of decade:

```
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')
```

gender	F	M
decade		
1960	1753634	1846572
1970	16263075	17121550
1980	18310351	19243452
1990	19479454	20420553
2000	18229309	19106428

We see that male births outnumber female births in every decade. To see this trend a bit more clearly, we can use the built-in plotting tools in Pandas to visualize the total number of births by year (see [Link to Come] for a discussion of plotting with Matplotlib):

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
births.pivot_table(
    'births', index='year', columns='gender', aggfunc='sum').plot()
plt.ylabel('total births per year');
```



With a simple pivot table and `plot()` method, we can immediately see the annual trend in births by gender. By eye, it appears that over the past 50 years male births have outnumbered female births by around 5%.

Further data exploration

Though this doesn't necessarily relate to the pivot table, there are a few more interesting features we can pull out of this dataset using the Pandas tools covered up to this point. We must start by cleaning the data a bit, removing outliers caused by mistyped dates (e.g., June 31st) or missing values (e.g., June 99th). One easy way to remove these all at once is to cut outliers; we'll do this via a robust sigma-clipping operation:

```
quartiles = np.percentile(births['births'], [25, 50, 75])
mu = quartiles[1]
sig = 0.74 * (quartiles[2] - quartiles[0])
```

This final line is a robust estimate of the sample standard deviation, where the 0.74 comes from the interquartile range of a Gaussian distribution (You can learn more about sigma-clipping operations in a book I coauthored with Željko Ivezić, Andrew J. Connolly, and Alexander Gray: "Statistics, Data Mining, and Machine Learning in Astronomy" (Princeton University Press, 2020)).

With this we can use the `query()` method (discussed further in "High-Performance Pandas: eval() and query()") to filter-out rows with births outside these values:

```
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
```

Next we set the `day` column to integers; previously it had been a string because some columns in the dataset contained the value `'null'`:

```
# set 'day' column to integer; it originally was a string due to nulls
births['day'] = births['day'].astype(int)
```

Finally, we can combine the day, month, and year to create a Date index (see "Working with Time Series"). This allows us to quickly compute the weekday corresponding to each row:

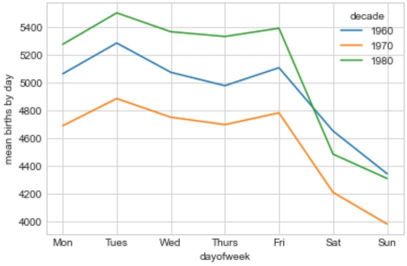
```
# create a datetime index from the year, month, day
births.index = pd.to_datetime(10000 * births.year +
                              100 * births.month +
                              births.day, format='%Y%m%d')

births['dayofweek'] = births.index.dayofweek
```

Using this we can plot births by weekday for several decades:

```
import matplotlib.pyplot as plt
import matplotlib as mpl

births.pivot_table('births', index='dayofweek',
                    columns='decade', aggfunc='mean').plot()
plt.gca().set(xticks=range(7),
              xticklabels=['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'])
plt.ylabel('mean births by day');
```



Apparently births are slightly less common on weekends than on weekdays! Note that the 1990s and 2000s are missing because the CDC data contains only the month of birth starting in 1989.

Another intersting view is to plot the mean number of births by the day of the *year*. Let's first group the data by month and day separately:

```
births_by_date = births.pivot_table('births',
                                    [births.index.month, births.index.day])
births_by_date.head()
```

		births
	1	4009.225
	2	4247.400
	3	4500.900
	4	4571.350
1	5	4603.625

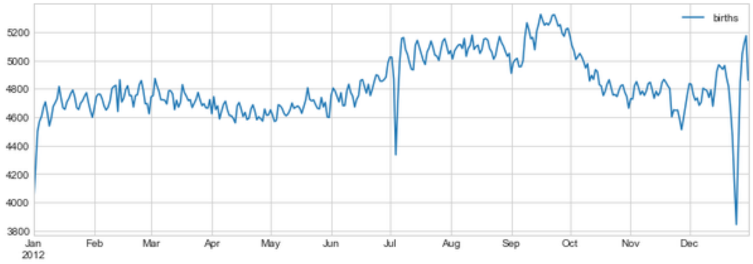
The result is a multi-index over months and days. To make this visualizable, let's turn these months and days into a date by associating them with a dummy year variable (making sure to choose a leap year so February 29th is correctly handled!)

```
from datetime import datetime
births_by_date.index = [datetime(2012, month, day)
                        for (month, day) in births_by_date.index]
births_by_date.head()
```

	births
2012-01-01	4009.225
2012-01-02	4247.400
2012-01-03	4500.900
2012-01-04	4571.350
2012-01-05	4603.625

Focusing on the month and day only, we now have a time series reflecting the average number of births by date of the year. From this, we can use the `plot` method to plot the data. It reveals some interesting trends:

```
# Plot the results
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax);
```



In particular, the striking feature of this graph is the dip in birthrate on US holidays (e.g., Independence Day, Labor Day, Thanksgiving, Christmas, New Year's Day) although this likely reflects trends in scheduled/induced births rather than some deep psychosomatic effect on natural births. For more discussion on this trend, see the analysis and links in Andrew Gelman's blog post on the subject. We'll return to this figure in 04.09-Text-and-Annotation.ipynb#Example:-Effect-of-Holidays-on-US-Births[Example:-Effect-of-Holidays-on-US-Births], where we will use Matplotlib's tools to annotate this plot.

Looking at this short example, you can see that many of the Python and Pandas tools we've seen to this point can be combined and used to gain insight from a variety of datasets. We will see some more sophisticated applications of these data manipulations in future sections!

Vectorized String Operations

One strength of Python is its relative ease in handling and manipulating string data. Pandas builds on this and provides a comprehensive set of *vectorized string operations* that are an important part of the type of munging required when working with (read: cleaning up) real-world data. In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean up a very messy dataset of recipes collected from the Internet.

Introducing Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. For example:

```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2
```

```
array([ 4,  6, 10, 14, 22, 26])
```

This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done. For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax:

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

```
['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with some data, but it will break if there are any missing values, so this approach requires putting in extra checks:

```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
[s if s is None else s.capitalize() for s in data]
```

```
['Peter', 'Paul', None, 'Mary', 'Guido']
```

This kind of manual approach is not only verbose and inconvenient, it can be error-prone.

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the `str` attribute of Pandas Series and Index objects containing strings. So, for example, if we create a Pandas series with this data we can directly call the `str.capitalize()` method, which has missing-value handling built-in:

```
import pandas as pd
names = pd.Series(data)
names.str.capitalize()
```

```
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

Using tab completion on this `str` attribute will list all the vectorized string methods available to Pandas.

Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it’s probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties. The examples in this section use the following `Series` object:

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Methods similar to Python string methods

Nearly all Python’s built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas `str` methods that mirror Python string methods:

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

Notice that these have various return values. Some, like `lower()`, return a series of strings:

```
monte.str.lower()
```

```
0    graham chapman
1       john cleese
2     terry gilliam
3         eric idle
4       terry jones
5     michael palin
dtype: object
```

But some others return numbers:

```
monte.str.len()
```

```
0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

Or Boolean values:

```
monte.str.startswith('T')
```

```
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

Still others return lists or other compound values for each element:

```
monte.str.split()
```

```
0    [Graham, Chapman]
1       [John, Cleese]
2     [Terry, Gilliam]
3         [Eric, Idle]
4       [Terry, Jones]
5     [Michael, Palin]
dtype: object
```

We’ll see further manipulations of this kind of series-of-lists object as we continue our discussion.

Methods using regular expressions

In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python’s built-in `re` module:

Method	Description
`match()`	Call `re.match()` on each element, returning a boolean.
`extract()`	Call `re.match()` on each element, returning matched groups as strings.
`findall()`	Call `re.findall()` on each element
`replace()`	Replace occurrences of pattern with some other string
`contains()`	Call `re.search()` on each element, returning a boolean
`count()`	Count occurrences of pattern
`split()`	Equivalent to `str.split()`, but accepts regexps
`rsplit()`	Equivalent to `str.rsplit()`, but accepts regexps

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

```
monte.str.extract('([A-Za-z]+)', expand=False)
```

```
0     Graham
1       John
2      Terry
3       Eric
4      Terry
5    Michael
dtype: object
```

Or we can do something more complicated, like finding all names that start and end with a consonant, making use of the start-of-string (`^`) and end-of-string (`$`) regular expression characters:

```
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
```

```
0    [Graham Chapman]
1                  []
2     [Terry Gilliam]
3                  []
4       [Terry Jones]
5     [Michael Palin]
dtype: object
```

The ability to concisely apply regular expressions across `Series` or `Dataframe` entries opens up many possibilities for analysis and cleaning of data.

Miscellaneous methods

Finally, there are some miscellaneous methods that enable other convenient operations:

Method	Description
`get()`	Index each element
`slice()`	Slice each element
`slice_replace()`	Replace slice in each element with passed value
`cat()`	Concatenate strings
`repeat()`	Repeat values
`normalize()`	Return Unicode form of string
`pad()`	Add whitespace to left, right, or both sides of strings
`wrap()`	Split long strings into lines with length less than a given width
`join()`	Join strings in each element of the Series with passed separator
`get_dummies()`	extract dummy variables as a dataframe

Vectorized item access and slicing

The `get()` and `slice()` operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using `str.slice(0, 3)`. Note that this behavior is also available through Python's normal indexing syntax–for example, `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

```
monte.str[0:3]
```

```
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

Indexing via `df.str.get(i)` and `df.str[i]` are likewise similar.

These indexing methos also let you access elements of arrays returned by `split()`. For example, to extract the last name of each entry, we can combine `split()` with `str` indexing:

```
monte.str.split().str[-1]
```

```
0    Chapman
1     Cleese
2    Gilliam
3       Idle
4      Jones
5      Palin
dtype: object
```

Indicator variables

Another method that requires a bit of extra explanation is the `get_dummies()` method. This is useful when your data has a column containing some sort of coded indicator. For example, we might have a dataset that contains information in the form of codes, such as A="born in America,” B="born in the United Kingdom,” C="likes cheese,” D="likes spam”:

```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C',
                                    'B|D', 'B|C', 'B|C|D']})
full_monte
```

	name	info
0	Graham Chapman	B|C|D
1	John Cleese	B|D
2	Terry Gilliam	A|C
3	Eric Idle	B|D
4	Terry Jones	B|C
5	Michael Palin	B|C|D

The `get_dummies()` routine lets you split-out these indicator variables into a `DataFrame`:

```
full_monte['info'].str.get_dummies('|')
```

	A	B	C	D
0	0	1	1	1
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	0	1	1	0
5	0	1	1	1

With these operations as building blocks, you can construct an endless range of string processing procedures when cleaning your data.

We won’t dive further into these methods here, but I encourage you to read through Working with Text Data in the Pandas online documentation, or to refer to the resources listed in “Further Resources”.

Example: Recipe Database

These vectorized string operations become most useful in the process of cleaning up messy, real-world data. Here I’ll walk through an example of that, using an open recipe database compiled from various sources on the Web. Our goal will be to parse the recipe data into ingredient lists, so we can quickly find a recipe based on some ingredients we have on hand.

The scripts used to compile this can be found at *https://github.com/fictivekin/openrecipes*, and the link to the most recent version of the database is found there as well.

This database is about 30 MB, and can be downloaded and unzipped with these commands:

```
# repo = "https://raw.githubusercontent.com/jakevdp/open-recipe-data/master"
# !cd data && curl -O {repo}/recipeitems.json.gz
# !gunzip data/recipeitems.json.gz
```

The database is in JSON format, so we will use `pd.read_json` to read it (`lines=True` is required for this dataset because each line of the file is a JSON entry):

```
recipes = pd.read_json('data/recipeitems.json', lines=True)
recipes.shape
```

```
(173278, 17)
```

We see there are nearly 175,000 recipes, and 17 columns. Let's take a look at one row to see what we have:

```
recipes.iloc[0]
```

```
_id                                {'$oid': '5160756b96cc62079cc2db15'}
name                                    Drop Biscuits and Sausage Gravy
ingredients           Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
url                   http://thepioneerwoman.com/cooking/2013/03/dro...
image                 http://static.thepioneerwoman.com/cooking/file...
ts                                             {'$date': 1365276011104}
cookTime                                                          PT30M
source                                                  thepioneerwoman
recipeYield                                                          12
datePublished                                                2013-03-11
prepTime                                                          PT10M
description           Late Saturday afternoon, after Marlboro Man ha...
totalTime                                                           NaN
creator                                                             NaN
recipeCategory                                                      NaN
dateModified                                                        NaN
recipeInstructions                                                  NaN
Name: 0, dtype: object
```

There is a lot of information there, but much of it is in a very messy form, as is typical of data scraped from the Web. In particular, the ingredient list is in string format; we're going to have to carefully extract the information we're interested in. Let's start by taking a closer look at the ingredients:

```
recipes.ingredients.str.len().describe()
```

```
count    173278.000000
mean        244.617926
std         146.705285
min           0.000000
25%         147.000000
50%         221.000000
75%         314.000000
max        9067.000000
Name: ingredients, dtype: float64
```

The ingredient lists average 250 characters long, with a minimum of 0 and a maximum of nearly 10,000 characters!

Just out of curiousity, let's see which recipe has the longest ingredient list:

```
recipes.name[np.argmax(recipes.ingredients.str.len())]
```

```
'Carrot Pineapple Spice &amp; Brownie Layer Cake with Whipped Cream &amp; Cream
 > Cheese Frosting and Marzipan Carrots'
```

We can do other aggregate explorations; for example, let’s see how many of the recipes are for breakfast food (using regular expression syntax to match both lowercase and capital):

```
recipes.description.str.contains('[Bb]reakfast').sum()
```

```
3524
```

Or how many of the recipes list cinnamon as an ingredient:

```
recipes.ingredients.str.contains('[Cc]innamon').sum()
```

```
10526
```

We could even look to see whether any recipes misspell the ingredient as “cinamon”:

```
recipes.ingredients.str.contains('[Cc]inamon').sum()
```

```
11
```

This is the type of data exploration that is possible with Pandas string tools. It is data munging like this that Python really excels at.

A simple recipe recommender

Let’s go a bit further, and start working on a simple recipe recommendation system: given a list of ingredients, find a recipe that uses all those ingredients. While conceptually straightforward, the task is complicated by the heterogeneity of the data: there is no easy operation, for example, to extract a clean list of ingredients from each row. So we will cheat a bit: we’ll start with a list of common ingredients, and simply search to see whether they are in each recipe’s ingredient list. For simplicity, let’s just stick with herbs and spices for the time being:

```
spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
              'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

We can then build a Boolean `DataFrame` consisting of True and False values, indicating whether this ingredient appears in the list:

```
import re
spice_df = pd.DataFrame({
    spice: recipes.ingredients.str.contains(spice, re.IGNORECASE)
    for spice in spice_list})
spice_df.head()
```

	salt	pepper	oregano	sage	parsley	rosemary	tar
0	False	False	False	True	False	False	Fals
1	False	False	False	False	False	False	Fals
2	True	True	False	False	False	False	Fals
3	False	False	False	False	False	False	Fals
4	False	False	False	False	False	False	Fals

Now, as an example, let’s say we’d like to find a recipe that uses parsley, paprika, and tarragon. We can compute this very quickly using the `query()` method of $DataFrame$s, discussed in “High-Performance Pandas: eval() and query()”:

```
selection = spice_df.query('parsley & paprika & tarragon')
len(selection)
```

```
10
```

We find only 10 recipes with this combination; let’s use the index returned by this selection to discover the names of the recipes that have this combination:

```
recipes.name[selection.index]
```

```
2069      All cremat with a Little Gem, dandelion and wa...
74964                         Lobster with Thermidor butter
93768      Burton's Southern Fried Chicken with White Gravy
113926                     Mijo's Slow Cooker Shredded Beef
137686                     Asparagus Soup with Poached Eggs
140530                                 Fried Oyster Po’boys
158475                Lamb shank tagine with herb tabbouleh
158486                 Southern fried chicken in buttermilk
163175            Fried Chicken Sliders with Pickles + Slaw
165243                        Bar Tartine Cauliflower Salad
Name: name, dtype: object
```

Now that we have narrowed down our recipe selection from 175,000 to 10, we are in a position to make a more informed decision about what we’d like to cook for dinner.

Going further with recipes

Hopefully this example has given you a bit of a flavor (heh) for the types of data cleaning operations that are efficiently enabled by Pandas string methods. Of course, building a very robust recipe recommendation system would require a *lot* more work! Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately, the wide variety of formats used makes this a relatively time-consuming process. This points to the truism that in data science, cleaning and munging of real-world data often comprises the majority of the work, and Pandas provides the tools that can help you do this efficiently.

Working with Time Series

Pandas was originally developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data. Date and time data comes in a few flavors, which we will discuss here:

- *Time stamps* reference particular moments in time (e.g., July 4th, 2021 at 7:00am).
- *Time intervals* and *periods* reference a length of time between a particular beginning and end point; for example, the month of June,

    1. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods comprising days).

- *Time deltas* or *durations* reference an exact length of time (e.g., a duration of 22.56 seconds).

In this section, we will introduce how to work with each of these types of date/time data in Pandas. This short section is by no means a complete guide to the time series tools available in Python or Pandas, but instead is intended as a broad overview of how you as a user should approach working with time series. We will start with a brief discussion of tools for dealing with dates and times in Python, before moving more specifically to a discussion of the tools provided by Pandas. After listing some resources that go into more depth, we will review some short examples of working with time series data in Pandas.

Dates and Times in Python

The Python world has a number of available representations of dates, times, deltas, and timespans. While the time series tools provided by Pandas tend to be the most useful for data science applications, it is helpful to see their relationship to other packages used in Python.

Native Python dates and times: `datetime` and `dateutil`

Python's basic objects for working with dates and times reside in the built-in `datetime` module. Along with the third-party `dateutil` module, you can use it to quickly perform a host of useful functionalities on dates and times. For example, you can manually build a date using the `datetime` type:

```
from datetime import datetime
datetime(year=2021, month=7, day=4)
```

```
datetime.datetime(2021, 7, 4, 0, 0)
```

Or, using the `dateutil` module, you can parse dates from a variety of string formats:

```
from dateutil import parser
date = parser.parse("4th of July, 2021")
date
```

```
datetime.datetime(2021, 7, 4, 0, 0)
```

Once you have a `datetime` object, you can do things like printing the day of the week:

```
date.strftime('%A')
```

```
'Sunday'
```

In the final line, we've used one of the standard string format codes for printing dates (`"%A"`), which you can read about in the strftime section of Python's datetime documentation. Documentation of other useful date utilities can be found in dateutil's online documentation. A related package to be aware of is `pytz`, which contains tools for working with the most migrane-inducing piece of time series data: time zones.

The power of `datetime` and `dateutil` lie in their flexibility and easy syntax: you can use these objects and their built-in methods to easily perform nearly any operation you might be interested in. Where they break down is when you wish to work with large arrays of dates and times: just as lists of Python numerical variables are suboptimal compared to NumPy-style typed numerical arrays, lists of Python datetime objects are suboptimal compared to typed arrays of encoded dates.

Typed arrays of times: NumPy's `datetime64`

NumPy's `datetime64` dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented compactly and operated on in an efficient manner. The `datetime64` requires a very specific input format:

```
import numpy as np
date = np.array('2021-07-04', dtype=np.datetime64)
date
```

```
array('2021-07-04', dtype='datetime64[D]')
```

Once we have dates in this form, we can quickly do vectorized operations on it:

```
date + np.arange(12)
```

```
array(['2021-07-04', '2021-07-05', '2021-07-06', '2021-07-07',
       '2021-07-08', '2021-07-09', '2021-07-10', '2021-07-11',
       '2021-07-12', '2021-07-13', '2021-07-14', '2021-07-15'],
      dtype='datetime64[D]')
```

Because of the uniform type in NumPy `datetime64` arrays, this type of operation can be accomplished much more quickly than if we were working directly with Python's `datetime` objects, especially as arrays get large (we introduced this type of vectorization in "Computation on NumPy Arrays: Universal Functions").

One detail of the `datetime64` and related `timedelta64` objects is that they are built on a *fundamental time unit*. Because the `datetime64` object is limited to 64-bit precision, the range of encodable times is 2^{64} times this fundamental unit. In other words, `datetime64` imposes a trade-off between *time resolution* and *maximum time span*.

For example, if you want a time resolution of one nanosecond, you only have enough information to encode a range of 2^{64} nanoseconds, or just under 600 years. NumPy will infer the desired unit from the input; for example, here is a day-based datetime:

```
np.datetime64('2021-07-04')
```

```
numpy.datetime64('2021-07-04')
```

Here is a minute-based datetime:

```
np.datetime64('2021-07-04 12:00')
```

```
numpy.datetime64('2021-07-04T12:00')
```

You can force any desired fundamental unit using one of many format codes; for example, here we'll force a nanosecond-based time:

```
np.datetime64('2021-07-04 12:59:59.50', 'ns')
```

```
numpy.datetime64('2021-07-04T12:59:59.500000000')
```

The following table, drawn from the NumPy datetime64 documentation, lists the available format codes along with the relative and absolute timespans that they can encode:

Code	Meaning	Time span (relative)	Time span (absolute)
`Y`	Year	± 9.2e18 years	[9.2e18 BC, 9.2e18 AD]
`M`	Month	± 7.6e17 years	[7.6e17 BC, 7.6e17 AD]
`W`	Week	± 1.7e17 years	[1.7e17 BC, 1.7e17 AD]
`D`	Day	± 2.5e16 years	[2.5e16 BC, 2.5e16 AD]
`h`	Hour	± 1.0e15 years	[1.0e15 BC, 1.0e15 AD]
`m`	Minute	± 1.7e13 years	[1.7e13 BC, 1.7e13 AD]
`s`	Second	± 2.9e12 years	[2.9e9 BC, 2.9e9 AD]
`ms`	Millisecond	± 2.9e9 years	[2.9e6 BC, 2.9e6 AD]
`us`	Microsecond	± 2.9e6 years	[290301 BC, 294241 AD]
`ns`	Nanosecond	± 292 years	[1678 AD, 2262 AD]
`ps`	Picosecond	± 106 days	[1969 AD, 1970 AD]
`fs`	Femtosecond	± 2.6 hours	[1969 AD, 1970 AD]
`as`	Attosecond	± 9.2 seconds	[1969 AD, 1970 AD]

For the types of data we see in the real world, a useful default is `datetime64[ns]`, as it can encode a useful range of modern dates with a suitably fine precision.

Finally, we will note that while the `datetime64` data type addresses some of the deficiencies of the built-in Python `datetime` type, it lacks many of the convenient methods and functions provided by `datetime` and especially `dateutil`. More information can be found in NumPy's datetime64 documentation.

Dates and times in pandas: best of both worlds

Pandas builds upon all the tools just discussed to provide a `Timestamp` object, which combines the ease-of-use of `datetime` and `dateutil` with the efficient storage and vectorized interface of `numpy.datetime64`. From a group of these `Timestamp` objects, Pandas can construct a `DatetimeIndex` that can be used to index data in a `Series` or `DataFrame`; we'll see many examples of this below.

For example, we can use Pandas tools to repeat the demonstration from above. We can parse a flexibly formatted string date, and use format codes to output the day of the week:

```
import pandas as pd
date = pd.to_datetime("4th of July, 2021")
date
```

```
Timestamp('2021-07-04 00:00:00')
```

```
date.strftime('%A')
```

```
'Sunday'
```

Additionally, we can do NumPy-style vectorized operations directly on this same object:

```
date + pd.to_timedelta(np.arange(12), 'D')
```

```
DatetimeIndex(['2021-07-04', '2021-07-05', '2021-07-06', '2021-07-07',
               '2021-07-08', '2021-07-09', '2021-07-10', '2021-07-11',
               '2021-07-12', '2021-07-13', '2021-07-14', '2021-07-15'],
              dtype='datetime64[ns]', freq=None)
```

In the next section, we will take a closer look at manipulating time series data with the tools provided by Pandas.

Pandas Time Series: Indexing by Time

Where the Pandas time series tools really become useful is when you begin to *index data by timestamps*. For example, we can construct a `Series` object that has time indexed data:

```
index = pd.DatetimeIndex(['2020-07-04', '2020-08-04',
                          '2021-07-04', '2021-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

```
2020-07-04    0
2020-08-04    1
2021-07-04    2
2021-08-04    3
dtype: int64
```

Now that we have this data in a `Series`, we can make use of any of the `Series` indexing patterns we discussed in previous sections, passing values that can be coerced into dates:

```
data['2020-07-04':'2021-07-04']
```

```
2020-07-04    0
2020-08-04    1
2021-07-04    2
dtype: int64
```

There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
data['2021']
```

```
2021-07-04    2
2021-08-04    3
dtype: int64
```

Later, we will see additional examples of the convenience of dates-as-indices. But first, a closer look at the available time series data structures.

Pandas Time Series Data Structures

This section will introduce the fundamental Pandas data structures for working with time series data:

- For *time stamps*, Pandas provides the `Timestamp` type. As mentioned before, it is essentially a replacement for Python's native `datetime`, but is based on the more efficient `numpy.datetime64` data type. The associated Index structure is `DatetimeIndex`.
- For *time Periods*, Pandas provides the `Period` type. This encodes a fixed-frequency interval based on `numpy.datetime64`. The associated index structure is `PeriodIndex`.

- For *time deltas* or *durations*, Pandas provides the `Timedelta` type. `Timedelta` is a more efficient replacement for Python's native `datetime.timedelta` type, and is based on `numpy.timedelta64`. The associated index structure is `TimedeltaIndex`.

The most fundamental of these date/time objects are the `Timestamp` and `DatetimeIndex` objects. While these class objects can be invoked directly, it is more common to use the `pd.to_datetime()` function, which can parse a wide variety of formats. Passing a single date to `pd.to_datetime()` yields a `Timestamp`; passing a series of dates by default yields a `DatetimeIndex`:

```
dates = pd.to_datetime([datetime(2021, 7, 3), '4th of July, 2021',
                        '2021-Jul-6', '07-07-2021', '20210708'])
dates
```

```
DatetimeIndex(['2021-07-03', '2021-07-04', '2021-07-06', '2021-07-07',
               '2021-07-08'],
              dtype='datetime64[ns]', freq=None)
```

Any `DatetimeIndex` can be converted to a `PeriodIndex` with the `to_period()` function with the addition of a frequency code; here we'll use `'D'` to indicate daily frequency:

```
dates.to_period('D')
```

```
PeriodIndex(['2021-07-03', '2021-07-04', '2021-07-06', '2021-07-07',
             '2021-07-08'],
            dtype='period[D]')
```

A `TimedeltaIndex` is created, for example, when a date is subtracted from another:

```
dates - dates[0]
```

```
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
 > dtype='timedelta64[ns]', freq=None)
```

Regular sequences: `pd.date_range()`

To make creation of regular date sequences more convenient, Pandas offers a few functions for this purpose: `pd.date_range()` for timestamps, `pd.period_range()` for periods, and `pd.timedelta_range()` for time deltas. We've seen that Python's `range()` and NumPy's `np.arange()` take a startpoint, endpoint, and optional stepsize and return a sequence. Similarly, `pd.date_range()` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates:

```
pd.date_range('2015-07-03', '2015-07-10')
```

```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
               '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
              dtype='datetime64[ns]', freq='D')
```

Alternatively, the date range can be specified not with a start and endpoint, but with a startpoint and a number of periods:

```
pd.date_range('2015-07-03', periods=8)
```

```
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
               '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
              dtype='datetime64[ns]', freq='D')
```

The spacing can be modified by altering the `freq` argument, which defaults to `D`. For example, here we will construct a range of hourly timestamps:

```
pd.date_range('2015-07-03', periods=8, freq='H')
```

```
DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
               '2015-07-03 02:00:00', '2015-07-03 03:00:00',
               '2015-07-03 04:00:00', '2015-07-03 05:00:00',
               '2015-07-03 06:00:00', '2015-07-03 07:00:00'],
              dtype='datetime64[ns]', freq='H')
```

To create regular sequences of `Period` or `Timedelta` values, the very similar `pd.period_range()` and `pd.timedelta_range()` functions are useful. Here are some monthly periods:

```
pd.period_range('2015-07', periods=8, freq='M')
```

```
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
             '2016-01', '2016-02'],
            dtype='period[M]')
```

And a sequence of durations increasing by an hour:

```
pd.timedelta_range(0, periods=6, freq='H')
```

```
TimedeltaIndex(['0 days 00:00:00', '0 days 01:00:00', '0 days 02:00:00',
                '0 days 03:00:00', '0 days 04:00:00', '0 days 05:00:00'],
               dtype='timedelta64[ns]', freq='H')
```

All of these require an understanding of Pandas frequency codes, which we'll summarize in the next section.

Frequencies and Offsets

Fundamental to these Pandas time series tools is the concept of a frequency or date offset. Just as we saw the `D` (day) and `H` (hour) codes above, we can use such codes to specify any desired frequency spacing. The following table summarizes the main codes available:

Code	Description	Code	Description
`D`	Calendar day	`B`	Business day
`W`	Weekly		
`M`	Month end	`BM`	Business month end
`Q`	Quarter end	`BQ`	Business quarter end
`A`	Year end	`BA`	Business year end
`H`	Hours	`BH`	Business hours
`T`	Minutes		
`S`	Seconds		
`L`	Milliseonds		
`U`	Microseconds		
`N`	nanoseconds		

The monthly, quarterly, and annual frequencies are all marked at the end of the specified period. By adding an S suffix to any of these, they instead will be marked at the beginning:

```
 Code | Description || Code | Description |
```

|——-|—————————||——-|——————————| | MS | Month start ||BMS | Business month start | | QS | Quarter start ||BQS | Business quarter start | | AS | Year start ||BAS | Business year start |

Additionally, you can change the month used to mark any quarterly or annual code by adding a three-letter month code as a suffix:

- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

In the same way, the split-point of the weekly frequency can be modified by adding a three-letter weekday code:

- W-SUN, W-MON, W-TUE, W-WED, etc.

On top of this, codes can be combined with numbers to specify other frequencies. For example, for a frequency of 2 hours 30 minutes, we can combine the hour (H) and minute (T) codes as follows:

```
pd.timedelta_range(0, periods=6, freq="2H30T")
```

```
TimedeltaIndex(['0 days 00:00:00', '0 days 02:30:00', '0 days 05:00:00',
                '0 days 07:30:00', '0 days 10:00:00', '0 days 12:30:00'],
               dtype='timedelta64[ns]', freq='150T')
```

All of these short codes refer to specific instances of Pandas time series offsets, which can be found in the pd.tseries.offsets module. For example, we can create a business day offset directly as follows:

```
from pandas.tseries.offsets import BDay
pd.date_range('2015-07-01', periods=6, freq=BDay())
```

```
DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
               '2015-07-07', '2015-07-08'],
              dtype='datetime64[ns]', freq='B')
```

For more discussion of the use of frequencies and offsets, see the DateOffset section of the Pandas documentation.

Resampling, Shifting, and Windowing

The ability to use dates and times as indices to intuitively organize and access data is an important piece of the Pandas time series tools. The benefits of indexed data in general (automatic alignment during operations, intuitive data slicing and access, etc.) still apply, and Pandas provides several additional time series-specific operations.

We will take a look at a few of those here, using some stock price data as an example. Because Pandas was developed largely in a finance context, it includes some very specific tools for financial data. For example, the accompanying pandas-datareader package (installable via pip install pandas-datareader), knows how to import data from various online sources. Here we will load part of the Dow Jones Industrial Average price history:

```
from pandas_datareader import data

djia = data.DataReader('DJIA', start='2018', end='2022',
                       data_source='yahoo')
djia.head()
```

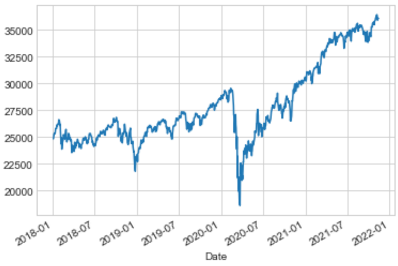
	High	Low	Open	Close	Volume	Adj Close
Date						
2018-01-02	24983.480469	24632.800781	24809.349609	24824.009766	3.367250e+09	24824.009766
2018-01-03	25033.640625	24719.460938	24850.449219	24922.679688	3.538660e+09	24922.679688
2018-01-04	25207.640625	24889.359375	24964.859375	25075.130859	3.695260e+09	25075.130859
2018-01-05	25369.890625	24995.640625	25114.919922	25295.869141	3.236620e+09	25295.869141
2018-01-08	25442.390625	25114.060547	25308.400391	25283.000000	3.242650e+09	25283.000000

For simplicity, we'll use just the closing price:

```
djia = djia['Close']
```

We can visualize this using the `plot()` method, after the normal Matplotlib setup boilerplate (see [Link to Come]):

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
djia.plot();
```

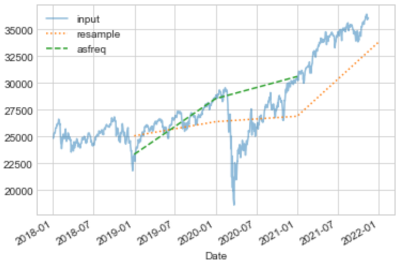


Resampling and converting frequencies

One common need for time series data is resampling at a higher or lower frequency. This can be done using the `resample()` method, or the much simpler `asfreq()` method. The primary difference between the two is that `resample()` is fundamentally a *data aggregation*, while `asfreq()` is fundamentally a *data selection*.

Taking a look at the DJIA closing price, let's compare what the two return when we down-sample the data. Here we will resample the data at the end of business year:

```
djia.plot(alpha=0.5, style='-')
djia.resample('BA').mean().plot(style=':')
djia.asfreq('BA').plot(style='--');
plt.legend(['input', 'resample', 'asfreq'],
           loc='upper left');
```



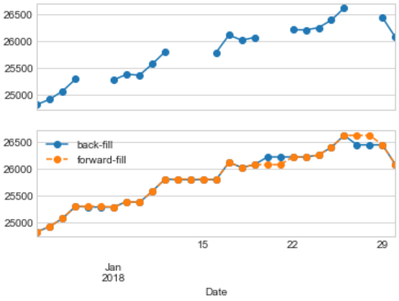
Notice the difference: at each point, `resample` reports the *average of the previous year*, while `asfreq` reports the *value at the end of the year*.

For up-sampling, `resample()` and `asfreq()` are largely equivalent, though resample has many more options available. In this case, the default for both methods is to leave the up-sampled points empty, that is, filled with NA values. Just as with the `pd.fillna()` function discussed previously, `asfreq()` accepts a `method` argument to specify how values are imputed. Here, we will resample the business day data at a daily frequency (i.e., including weekends):

```
fig, ax = plt.subplots(2, sharex=True)
data = djia.iloc[:20]

data.asfreq('D').plot(ax=ax[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=ax[1], style='-o')
data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')
ax[1].legend(["back-fill", "forward-fill"]);
```



Because the DJIA data only exists for business days, the top panel has gaps representing NA values. The bottom panel shows the differences between two strategies for filling the gaps: forward-filling and backward-filling.

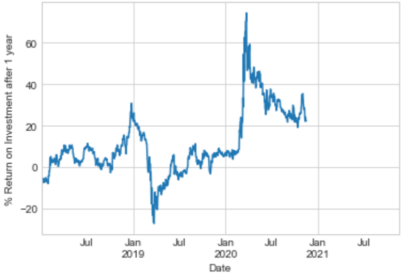
Time-shifts

Another common time series-specific operation is shifting of data in time. For this, pandas provides the `shift()` method, which can be used to shift data by a given number of entries. With time-series data sampled at a regular frequency, this can give us a way to explore trends over time.

For example, here we resample the data to daily values, and shift by 364 to compute the 1-year return-on-investment for the DJIA over time:

```
djia = djia.asfreq('D', method='pad')

ROI = 100 * (djia.shift(-365) - djia) / djia
ROI.plot()
plt.ylabel('% Return on Investment after 1 year');
```



The worst 1-year return was around March 2019, with the coronavirus-related market crash exactly a year later. As you might expect, the best 1-year return was to be found in March 2020, for those with enough foresight or luck to buy low.

This helps us to see the overall trend in Google stock: thus far, the most profitable times to invest in Google have been (unsurprisingly, in retrospect) shortly after its IPO, and in the middle of the 2009 recession.

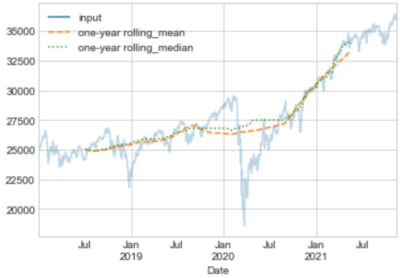
Rolling windows

Rolling statistics are a third type of time series-specific operation implemented by Pandas. These can be accomplished via the `rolling()` attribute of `Series` and `DataFrame` objects, which returns a view similar to what we saw with the `groupby` operation (see "Aggregation and Grouping"). This rolling view makes available a number of aggregation operations by default.

For example, here is the one-year centered rolling mean and standard deviation of the stock prices:

```
rolling = djia.rolling(365, center=True)

data = pd.DataFrame({'input': djia,
                     'one-year rolling_mean': rolling.mean(),
                     'one-year rolling_median': rolling.median()})
ax = data.plot(style=['-', '--', ':'])
ax.lines[0].set_alpha(0.3)
```



As with group-by operations, the `aggregate()` and `apply()` methods can be used for custom rolling computations.

Where to Learn More

This section has provided only a brief summary of some of the most essential features of time series tools provided by Pandas; for a more complete discussion, you can refer to the "Time Series/Date" section of the Pandas online documentation.

Another excellent resource is the textbook Python for Data Analysis by Wes McKinney (OReilly, 2017). Although it is now a few years old, it is an invaluable resource on the use of Pandas. In particular, this book emphasizes time series tools in the context of business and finance, and focuses much more on particular details of business calendars, time zones, and related topics.

As always, you can also use the IPython help functionality to explore and try further options available to the functions and methods discussed here. I find this often is the best way to learn a new Python tool.

Example: Visualizing Seattle Bicycle Counts

As a more involved example of working with some time series data, let's take a look at bicycle counts on Seattle's Fremont Bridge. This data comes from an automated bicycle counter, installed in late 2012, which has inductive sensors on the east and west sidewalks of the bridge. The hourly bicycle counts can be downloaded from *http://data.seattle.gov/;* here is the direct link to the dataset.

As of early 2022, the CSV can be downloaded as follows:

```
# url = ('https://data.seattle.gov/api/views/65db-xm6k/'
#         'rows.csv?accessType=DOWNLOAD')
# !cd data && curl -o FremontBridge.csv {url}
```

Once this dataset is downloaded, we can use Pandas to read the CSV output into a `DataFrame`. We will specify that we want the Date as an index, and we want these dates to be automatically parsed:

```
data = pd.read_csv('data/FremontBridge.csv', index_col='Date', parse_dates=True)
data.head()
```

	Fremont Bridge Total	Fremont Bridge East Sidewalk	Fremont Bridge West Sidewalk
Date			
2019-11-01 00:00:00	12.0	7.0	5.0
2019-11-01 01:00:00	7.0	0.0	7.0
2019-11-01 02:00:00	1.0	0.0	1.0
2019-11-01 03:00:00	6.0	6.0	0.0
2019-11-01 04:00:00	6.0	5.0	1.0

For convenience, we'll shorten the column names:

```
data.columns = ['Total', 'East', 'West']
```

Now let's take a look at the summary statistics for this data:

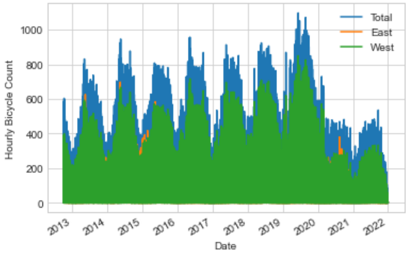
```
data.dropna().describe()
```

	Total	East	West
count	147255.000000	147255.000000	147255.000000
mean	110.341462	50.077763	60.263699
std	140.422051	64.634038	87.252147
min	0.000000	0.000000	0.000000
25%	14.000000	6.000000	7.000000
50%	60.000000	28.000000	30.000000
75%	145.000000	68.000000	74.000000
max	1097.000000	698.000000	850.000000

Visualizing the data

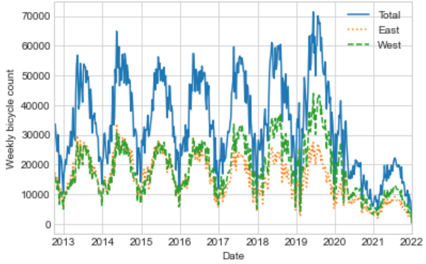
We can gain some insight into the dataset by visualizing it. Let's start by plotting the raw data:

```
data.plot()
plt.ylabel('Hourly Bicycle Count');
```



The ~150,000 hourly samples are far too dense for us to make much sense of. We can gain more insight by resampling the data to a coarser grid. Let's resample by week:

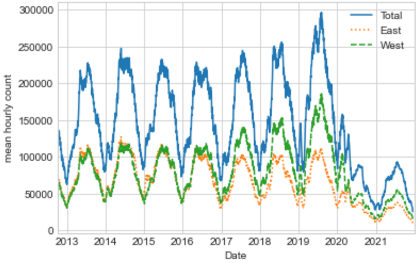
```
weekly = data.resample('W').sum()
weekly.plot(style=['-', ':', '--'])
plt.ylabel('Weekly bicycle count');
```



This shows us some interesting seasonal trends: as you might expect, people bicycle more in the summer than in the winter, and even within a particular season the bicycle use varies from week to week (likely dependent on weather; see [Link to Come] where we explore this further). Further, the effect of the pandemic on commute patterns is quite clear starting in early 2020.

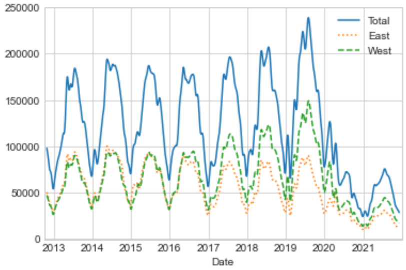
Another way that comes in handy for aggregating the data is to use a rolling mean, utilizing the `pd.rolling_mean()` function. Here we'll do a 30 day rolling mean of our data, making sure to center the window:

```
daily = data.resample('D').sum()
daily.rolling(30, center=True).sum().plot(style=['-', ':', '--'])
plt.ylabel('mean hourly count');
```



The jaggedness of the result is due to the hard cutoff of the window. We can get a smoother version of a rolling mean using a window function–for example, a Gaussian window. The following code specifies both the width of the window (we chose 50 days) and the width of the Gaussian within the window (we chose 10 days):

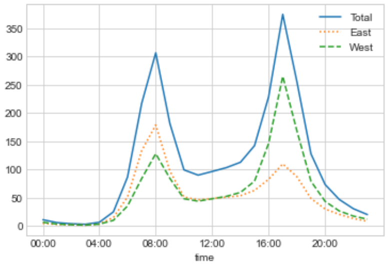
```
daily.rolling(50, center=True,
              win_type='gaussian').sum(std=10).plot(style=['-', ':', '--']);
```



Digging into the data

While these smoothed data views are useful to get an idea of the general trend in the data, they hide much of the interesting structure. For example, we might want to look at the average traffic as a function of the time of day. We can do this using the GroupBy functionality discussed in "Aggregation and Grouping":

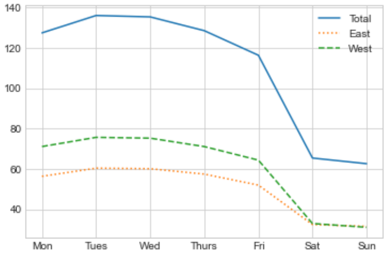
```
by_time = data.groupby(data.index.time).mean()
hourly_ticks = 4 * 60 * 60 * np.arange(6)
by_time.plot(xticks=hourly_ticks, style=['-', ':', '--']);
```



The hourly traffic is a strongly bimodal distribution, with peaks around 8:00 in the morning and 5:00 in the evening. This is likely evidence of a strong component of commuter traffic crossing the bridge. There is a directional component as well: according to the data, the east sidewalk is used more during the AM commute, and the west sidewalk is used more during the PM commute.

We also might be curious about how things change based on the day of the week. Again, we can do this with a simple groupby:

```
by_weekday = data.groupby(data.index.dayofweek).mean()
by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
by_weekday.plot(style=['-', ':', '--']);
```



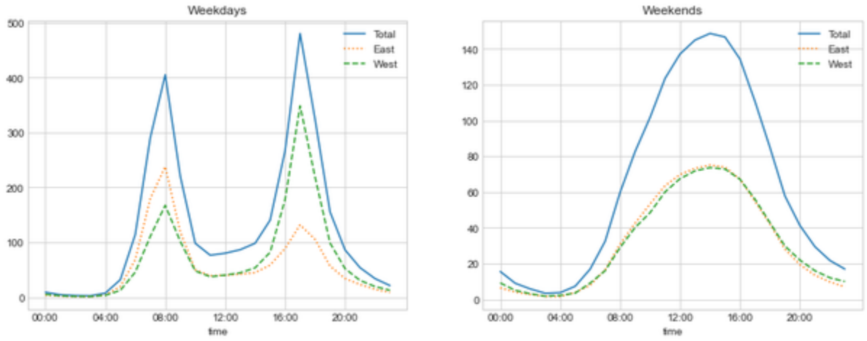
This shows a strong distinction between weekday and weekend totals, with around twice as many average riders crossing the bridge on Monday through Friday than on Saturday and Sunday.

With this in mind, let’s do a compound GroupBy and look at the hourly trend on weekdays versus weekends. We’ll start by grouping by both a flag marking the weekend, and the time of day:

```
weekend = np.where(data.index.weekday < 5, 'Weekday', 'Weekend')
by_time = data.groupby([weekend, data.index.time]).mean()
```

Now we’ll use some of the Matplotlib tools described in [Link to Come] to plot two panels side by side:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 2, figsize=(14, 5))
by_time.loc['Weekday'].plot(ax=ax[0], title='Weekdays',
                            xticks=hourly_ticks, style=['-', ':', '--'])
by_time.loc['Weekend'].plot(ax=ax[1], title='Weekends',
                            xticks=hourly_ticks, style=['-', ':', '--']);
```



The result is very interesting: we see a bimodal commute pattern during the work week, and a unimodal recreational pattern during the weekends. It would be interesting to dig through this data in more detail, and examine the effect of weather, temperature, time of year, and other factors on people’s commuting patterns; for further discussion, see my blog post “Is Seattle Really Seeing an Uptick In Cycling?”, which uses a subset of this data. We will also revisit this dataset in the context of modeling in [Link to Come].

High-Performance Pandas: eval() and query()

As we've already seen in previous sections, the power of the PyData stack is built upon the ability of NumPy and Pandas to push basic operations into C via an intuitive syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effective for many common use cases, they often rely on the creation of temporary intermediate objects, which can cause undue overhead in computational time and memory use.

To address this, pandas includes some methods that allow you to directly access C-speed operations without costly allocation of intermediate arrays. These are the `eval()` and `query()` functions, which rely on the Numexpr package. In this notebook we will walk through their use and give some rules-of-thumb about when you might think about using them.

Motivating `query()` and `eval()`: Compound Expressions

We've seen previously that NumPy and Pandas support fast vectorized operations; for example, when adding the elements of two arrays:

```
import numpy as np
rng = np.random.default_rng(42)
x = rng.random(1000000)
y = rng.random(1000000)
%timeit x + y
```

```
2.21 ms ± 142 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

As discussed in "Computation on NumPy Arrays: Universal Functions", this is much faster than doing the addition via a Python loop or comprehension:

```
%timeit np.fromiter((xi + yi for xi, yi in zip(x, y)),
                    dtype=x.dtype, count=len(x))
```

```
263 ms ± 43.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

But this abstraction can become less efficient when computing compound expressions. For example, consider the following expression:

```
mask = (x > 0.5) & (y < 0.5)
```

Because NumPy evaluates each subexpression, this is roughly equivalent to the following:

```
tmp1 = (x > 0.5)
tmp2 = (y < 0.5)
mask = tmp1 & tmp2
```

In other words, *every intermediate step is explicitly allocated in memory*. If the `x` and `y` arrays are very large, this can lead to significant memory and computational overhead. The Numexpr library gives you the ability to compute this type of compound expression element by element, without the need to allocate full intermediate arrays. The Numexpr documentation has more details, but for the time being it is sufficient to say that the library accepts a *string* giving the NumPy-style expression you'd like to compute:

```
import numexpr
mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
np.allclose(mask, mask_numexpr)
```

```
True
```

The benefit here is that Numexpr evaluates the expression in a way that avoids temporary arrays where possible, and thus can be much more efficient than NumPy, especially for long sequences of computations on large arrays. The Pandas `eval()` and `query()` tools that we will discuss here are conceptually similar, and depend on the Numexpr package.

`pandas.eval()` for Efficient Operations

The `eval()` function in Pandas uses string expressions to efficiently compute operations on `DataFrame` objects. For example, consider the following data:

```
import pandas as pd
nrows, ncols = 100000, 100
df1, df2, df3, df4 = (pd.DataFrame(rng.random((nrows, ncols)))
                      for i in range(4))
```

To compute the sum of all four $DataFrame$s using the typical Pandas approach, we can just write the sum:

```
%timeit df1 + df2 + df3 + df4
```

```
73.2 ms ± 6.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The same result can be computed via `pd.eval` by constructing the expression as a string:

```
%timeit pd.eval('df1 + df2 + df3 + df4')
```

```
34 ms ± 4.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The `eval()` version of this expression is about 50% faster (and uses much less memory), while giving the same result:

```
np.allclose(df1 + df2 + df3 + df4,
            pd.eval('df1 + df2 + df3 + df4'))
```

```
True
```

Operations supported by `pd.eval()`

`pd.eval()` supports a wide range of operations. To demonstrate these, we'll use the following integer data:

```
df1, df2, df3, df4, df5 = (pd.DataFrame(rng.integers(0, 1000, (100, 3)))
                           for i in range(5))
```

Arithmetic operators

`pd.eval()` supports all arithmetic operators. For example:

```
result1 = -df1 * df2 / (df3 + df4) - df5
result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
np.allclose(result1, result2)
```

```
True
```

Comparison operators

`pd.eval()` supports all comparison operators, including chained expressions:

```
result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
result2 = pd.eval('df1 < df2 <= df3 != df4')
np.allclose(result1, result2)
```

```
True
```

Bitwise operators

`pd.eval()` supports the `&` and `|` bitwise operators:

```
result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
np.allclose(result1, result2)
```

```
True
```

In addition, it supports the use of the literal `and` and `or` in Boolean expressions:

```
result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
np.allclose(result1, result3)
```

```
True
```

Object attributes and indices

`pd.eval()` supports access to object attributes via the `obj.attr` syntax, and indexes via the `obj[index]` syntax:

```
result1 = df2.T[0] + df3.iloc[1]
result2 = pd.eval('df2.T[0] + df3.iloc[1]')
np.allclose(result1, result2)
```

```
True
```

Other operations

Other operations such as function calls, conditional statements, loops, and other more involved constructs are currently *not* implemented in `pd.eval()`. If you'd like to execute these more complicated types of expressions, you can use the Numexpr library itself.

`DataFrame.eval()` for Column-Wise Operations

Just as Pandas has a top-level `pd.eval()` function, `DataFrame` objects have an `eval()` method that works in similar ways. The benefit of the `eval()` method is that columns can be referred to *by name*. We'll use this labeled array as an example:

```
df = pd.DataFrame(rng.random((1000, 3)), columns=['A', 'B', 'C'])
df.head()
```

	A	B	C
0	0.850888	0.966709	0.958690
1	0.820126	0.385686	0.061402
2	0.059729	0.831768	0.652259
3	0.244774	0.140322	0.041711
4	0.818205	0.753384	0.578851

Using `pd.eval()` as above, we can compute expressions with the three columns like this:

```
result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
np.allclose(result1, result2)
```

```
True
```

The `DataFrame.eval()` method allows much more succinct evaluation of expressions with the columns:

```
result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
```

```
True
```

Notice here that we treat *column names as variables* within the evaluated expression, and the result is what we would wish.

Assignment in `DataFrame.eval()`

In addition to the options just discussed, `DataFrame.eval()` also allows assignment to any column. Let's use the `DataFrame` from before, which has columns `'A'`, `'B'`, and `'C'`:

```
df.head()
```

	A	**B**	**C**
0	0.850888	0.966709	0.958690
1	0.820126	0.385686	0.061402
2	0.059729	0.831768	0.652259
3	0.244774	0.140322	0.041711
4	0.818205	0.753384	0.578851

We can use `df.eval()` to create a new column `'D'` and assign to it a value computed from the other columns:

```
df.eval('D = (A + B) / C', inplace=True)
df.head()
```

	A	**B**	**C**	**D**
0	0.850888	0.966709	0.958690	1.895916
1	0.820126	0.385686	0.061402	19.638139
2	0.059729	0.831768	0.652259	1.366782
3	0.244774	0.140322	0.041711	9.232370
4	0.818205	0.753384	0.578851	2.715013

In the same way, any existing column can be modified:

```
df.eval('D = (A - B) / C', inplace=True)
df.head()
```

	A	B	C	D
0	0.850888	0.966709	0.958690	-0.120812
1	0.820126	0.385686	0.061402	7.075399
2	0.059729	0.831768	0.652259	-1.183638
3	0.244774	0.140322	0.041711	2.504142
4	0.818205	0.753384	0.578851	0.111982

Local variables in `DataFrame.eval()`

The `DataFrame.eval()` method supports an additional syntax that lets it work with local Python variables. Consider the following:

```
column_mean = df.mean(1)
result1 = df['A'] + column_mean
result2 = df.eval('A + @column_mean')
np.allclose(result1, result2)
```

```
True
```

The `@` character here marks a *variable name* rather than a *column name*, and lets you efficiently evaluate expressions involving the two “namespaces”: the namespace of columns, and the namespace of Python objects. Notice that this `@` character is only supported by the `DataFrame.eval()` *method*, not by the `pandas.eval()` *function*, because the `pandas.eval()` function only has access to the one (Python) namespace.

`DataFrame.query()` Method

The `DataFrame` has another method based on evaluated strings, called the `query()` method. Consider the following:

```
result1 = df[(df.A < 0.5) & (df.B < 0.5)]
result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
np.allclose(result1, result2)
```

```
True
```

As with the example used in our discussion of `DataFrame.eval()`, this is an expression involving columns of the `DataFrame`. It cannot be expressed using the `DataFrame.eval()` syntax, however! Instead, for this type of filtering operation, you can use the `query()` method:

```
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)
```

```
True
```

In addition to being a more efficient computation, compared to the masking expression this is much easier to read and understand. Note that the query() method also accepts the @ flag to mark local variables:

```
Cmean = df['C'].mean()
result1 = df[(df.A < Cmean) & (df.B < Cmean)]
result2 = df.query('A < @Cmean and B < @Cmean')
np.allclose(result1, result2)
```

```
True
```

Performance: When to Use These Functions

When considering whether to use eval() and query(), there are two considerations: *computation time* and *memory use*. Memory use is the most predictable aspect. As already mentioned, every compound expression involving NumPy arrays or Pandas *DataFrame*s will result in implicit creation of temporary arrays: For example, this:

```
x = df[(df.A < 0.5) & (df.B < 0.5)]
```

Is roughly equivalent to this:

```
tmp1 = df.A < 0.5
tmp2 = df.B < 0.5
tmp3 = tmp1 & tmp2
x = df[tmp3]
```

If the size of the temporary DataFrame`s is significant compared to your available system memory (typically several gigabytes) then it's a good idea to use an `eval() or query() expression. You can check the approximate size of your array in bytes using this:

```
df.values.nbytes
```

```
32000
```

On the performance side, eval() can be faster even when you are not maxing-out your system memory. The issue is how your temporary objects compare to the size of the L1 or L2 CPU cache on your system (typically a few megabytes); if they are much bigger, then eval() can avoid some potentially slow movement of values between the different memory caches. In practice, I find that the difference in computation time between the traditional methods and the eval/query method is usually not significant–if anything, the traditional method is faster for smaller arrays! The benefit of eval/query is mainly in the saved memory, and the sometimes cleaner syntax they offer.

We've covered most of the details of eval() and query() here; for more information on these, you can refer to the Pandas documentation. In particular, different parsers and engines can be specified for running these queries; for details on this, see the discussion within the "Enhancing Performance" section.

Further Resources

In this chapter, we've covered many of the basics of using Pandas effectively for data analysis. Still, much has been omitted from our discussion. To learn more about Pandas, I recommend the following resources:

- Pandas online documentation: This is the go-to source for complete documentation of the package. While the examples in the documentation tend to be small generated datasets, the description of the options is complete and generally very useful for understanding the use of various functions.

- *Python for Data Analysis* Written by Wes McKinney (the original creator of Pandas), this book contains much more detail on the Pandas package than we had room for in this chapter. In particular, he takes a deep dive into tools for time series, which were his bread and butter as a financial consultant. The book also has many entertaining examples of applying Pandas to gain insight from real-world datasets.
- Effective Pandas is a short e-book by Pandas developer Tom Augspurger which provides a succinct outline of using the full power of the Pandas library in an effective and idiomatic way.
- Pandas on PyVideo: From PyCon to SciPy to PyData, many conferences have featured tutorials from Pandas developers and power users. The PyCon tutorials in particular tend to be given by very well-vetted presenters.

Using these resources, combined with the walk-through given in this chapter, my hope is that you'll be poised to use Pandas to tackle any data analysis problem you come across!

About the Author

Jake VanderPlas is a software engineer at Google Research, working on tools that support data-intensive research. He maintains a technical blog, *Pythonic Perambulations*, to share tutorials and opinions related to statistics, open software, and scientific computing in Python. He creates and develops Python tools for use in data-intensive science, including packages like Scikit-Learn, SciPy, AstroPy, Altair, JAX, and many others. He participates in the broader data science community, developing and presenting talks and tutorials on scientific computing topics at various conferences in the data science world.