O'REILLY®

A Whirlwind Tour of Python

Jake VanderPlas

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

A Whirlwind Tour of Python

Jake VanderPlas

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

A Whirlwind Tour of Python
by Jake VanderPlas

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Dawn Schanafelt
Production Editor: Kristen Brown
Copyeditor: Jasmine Kwityn
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2016: First Edition

Revision History for the First Edition
2016-08-10: First Release

978-1-491-96465-1

[LSI]

Table of Contents

A Whirlwind Tour of Python

Introduction

Conceived in the late 1980s as a teaching and scripting language, Python has since become an essential tool for many programmers, engineers, researchers, and data scientists across academia and industry. As an astronomer focused on building and promoting the free open tools for data-intensive science, I've found Python to be a near-perfect fit for the types of problems I face day to day, whether it's extracting meaning from large astronomical datasets, scraping and munging data sources from the Web, or automating day-to-day research tasks.

The appeal of Python is in its simplicity and beauty, as well as the convenience of the large ecosystem of domain-specific tools that have been built on top of it. For example, most of the Python code in scientific computing and data science is built around a group of mature and useful packages:

- NumPy provides efficient storage and computation for multidimensional data arrays.
- SciPy contains a wide array of numerical tools such as numerical integration and interpolation.
- Pandas provides a DataFrame object along with a powerful set of methods to manipulate, filter, group, and transform data.
- Matplotlib provides a useful interface for creation of publication-quality plots and figures.
- Scikit-Learn provides a uniform toolkit for applying common machine learning algorithms to data.

- IPython/Jupyter provides an enhanced terminal and an interactive notebook environment that is useful for exploratory analysis, as well as creation of interactive, executable documents. For example, the manuscript for this report was composed entirely in Jupyter notebooks.

No less important are the numerous other tools and packages which accompany these: if there is a scientific or data analysis task you want to perform, chances are someone has written a package that will do it for you.

To tap into the power of this data science ecosystem, however, first requires familiarity with the Python language itself. I often encounter students and colleagues who have (sometimes extensive) backgrounds in computing in some language—MATLAB, IDL, R, Java, C++, etc.—and are looking for a brief but comprehensive tour of the Python language that respects their level of knowledge rather than starting from ground zero. This report seeks to fill that niche.

As such, this report in no way aims to be a comprehensive introduction to programming, or a full introduction to the Python language itself; if that is what you are looking for, you might check out one of the recommended references listed in "Resources for Further Learning" on page 90. Instead, this will provide a whirlwind tour of some of Python's essential syntax and semantics, built-in data types and structures, function definitions, control flow statements, and other aspects of the language. My aim is that readers will walk away with a solid foundation from which to explore the data science stack just outlined.

Using Code Examples

Supplemental material (code examples, IPython notebooks, etc.) is available for download at *https://github.com/jakevdp/WhirlwindTourOfPython/*.

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: "*A Whirlwind Tour of Python* by Jake VanderPlas (O'Reilly). Copyright 2016 O'Reilly Media, Inc., 978-1-491-96465-1."

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 and Practical Considerations

Installing Python and the suite of libraries that enable scientific computing is straightforward whether you use Windows, Linux, or Mac OS X. This section will outline some of the considerations when setting up your computer.

Python 2 versus Python 3

This report uses the syntax of Python 3, which contains language enhancements that are not compatible with the 2.*x* series of Python. Though Python 3.0 was first released in 2008, adoption has been relatively slow, particularly in the scientific and web development communities. This is primarily because it took some time for many of the essential packages and toolkits to be made compatible with the new language internals. Since early 2014, however, stable releases of the most important tools in the data science ecosystem have been fully compatible with both Python 2 and 3, and so this report will use the newer Python 3 syntax. Even though that is the case, the vast majority of code snippets in this report will also work without modification in Python 2: in cases where a Py2-incompatible syntax is used, I will make every effort to note it explicitly.

Installation with conda

Though there are various ways to install Python, the one I would suggest—particularly if you wish to eventually use the data science tools mentioned earlier—is via the cross-platform Anaconda distribution. There are two flavors of the Anaconda distribution:

- 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.

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 IPython notebook package:

```
[~]$ conda install ipython-notebook
```

For more information on `conda`, including information about creating and using `conda` environments, refer to the Miniconda package documentation linked at the above page.

The Zen of Python

Python aficionados are often quick to point out how "intuitive", "beautiful", or "fun" Python is. While I tend to agree, I also recognize that beauty, intuition, and fun often go hand in hand with familiarity, and so for those familiar with other languages such florid sentiments can come across as a bit smug. Nevertheless, I hope that if you give Python a chance, you'll see where such impressions might come from. And if you *really* want to dig into the programming philosophy that drives much of the coding practice of Python power users, a nice little Easter egg exists in the Python interpreter—simply close your eyes, meditate for a few minutes, and run `import this`:

```
In [1]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

```
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one--and preferably only one--obvious way
    to do it.
Although that way may not be obvious at first unless
    you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea--let's do more of those!
```

With that, let's start our tour of the Python language.

How to Run Python Code

Python is a flexible language, and there are several ways to use it depending on your particular task. One thing that distinguishes Python from other programming languages is that it is *interpreted* rather than *compiled*. This means that it is executed line by line, which allows programming to be interactive in a way that is not directly possible with compiled languages like Fortran, C, or Java. This section will describe four primary ways you can run Python code: the *Python interpreter*, the *IPython interpreter*, via *self-contained scripts*, or in the *Jupyter notebook*.

The Python interpreter

The most basic way to execute Python code is line by line within the *Python interpreter*. The Python interpreter can be started by installing the Python language (see the previous section) and typing `python` at the command prompt (look for the Terminal on Mac OS X and Unix/Linux systems, or the Command Prompt application in Windows):

```
$ python
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec  7...
Type "help", "copyright", "credits" or "license" for more...
>>>
```

With the interpreter running, you can begin to type and execute code snippets. Here we'll use the interpreter as a simple calculator, performing calculations and assigning values to variables:

```
>>> 1 + 1
2
>>> x = 5
>>> x * 3
15
```

The interpreter makes it very convenient to try out small snippets of Python code and to experiment with short sequences of operations.

The IPython interpreter

If you spend much time with the basic Python interpreter, you'll find that it lacks many of the features of a full-fledged interactive development environment. An alternative interpreter called *IPython* (for Interactive Python) is bundled with the Anaconda distribution, and includes a host of convenient enhancements to the basic Python interpreter. It can be started by typing `ipython` at the command prompt:

```
$ ipython
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec  7...
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra...

In [1]:
```

The main aesthetic difference between the Python interpreter and the enhanced IPython interpreter lies in the command prompt: Python uses `>>>` by default, while IPython uses numbered commands (e.g., `In [1]:`). Regardless, we can execute code line by line just as we did before:

```
In [1]: 1 + 1
Out[1]: 2

In [2]: x = 5

In [3]: x * 3
Out[3]: 15
```

Note that just as the input is numbered, the output of each command is numbered as well. IPython makes available a wide array of useful features; for some suggestions on where to read more, see "Resources for Further Learning" on page 90.

Self-contained Python scripts

Running Python snippets line by line is useful in some cases, but for more complicated programs it is more convenient to save code to file, and execute it all at once. By convention, Python scripts are saved in files with a *.py* extension. For example, let's create a script called *test.py* that contains the following:

```
# file: test.py
print("Running test.py")
x = 5
print("Result is", 3 * x)
```

To run this file, we make sure it is in the current directory and type `python` *`filename`* at the command prompt:

```
$ python test.py
Running test.py
Result is 15
```

For more complicated programs, creating self-contained scripts like this one is a must.

The Jupyter notebook

A useful hybrid of the interactive terminal and the self-contained script is the *Jupyter notebook*, a document format that allows executable code, formatted text, graphics, and even interactive features to be combined into a single document. Though the notebook began as a Python-only format, it has since been made compatible with a large number of programming languages, and is now an essential part of the *Jupyter Project*. The notebook is useful both as a development environment and as a means of sharing work via rich computational and data-driven narratives that mix together code, figures, data, and text.

A Quick Tour of Python Language Syntax

Python was originally developed as a teaching language, but its ease of use and clean syntax have led it to be embraced by beginners and experts alike. The cleanliness of Python's syntax has led some to call

it "executable pseudocode", and indeed my own experience has been that it is often much easier to read and understand a Python script than to read a similar script written in, say, C. Here we'll begin to discuss the main features of Python's syntax.

Syntax refers to the structure of the language (i.e., what constitutes a correctly formed program). For the time being, we won't focus on the semantics—the meaning of the words and symbols within the syntax—but will return to this at a later point.

Consider the following code example:

```
In [1]: # set the midpoint
        midpoint = 5

        # make two empty lists
        lower = []; upper = []

        # split the numbers into lower and upper
        for i in range(10):
            if (i < midpoint):
                lower.append(i)
            else:
                upper.append(i)

        print("lower:", lower)
        print("upper:", upper)
```

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

This script is a bit silly, but it compactly illustrates several of the important aspects of Python syntax. Let's walk through it and discuss some of the syntactical features of Python.

Comments Are Marked by

The script starts with a comment:

```
# set the midpoint
```

Comments in Python are indicated by a pound sign (#), and anything on the line following the pound sign is ignored by the interpreter. This means, for example, that you can have standalone comments like the one just shown, as well as inline comments that follow a statement. For example:

```
x += 2  # shorthand for x = x + 2
```

Python does not have any syntax for multiline comments, such as the `/* ... */` syntax used in C and C++, though multiline strings are often used as a replacement for multiline comments (more on this in "String Manipulation and Regular Expressions" on page 69).

End-of-Line Terminates a Statement

The next line in the script is

```
midpoint = 5
```

This is an assignment operation, where we've created a variable named `midpoint` and assigned it the value `5`. Notice that the end of this statement is simply marked by the end of the line. This is in contrast to languages like C and C++, where every statement must end with a semicolon (`;`).

In Python, if you'd like a statement to continue to the next line, it is possible to use the `\` marker to indicate this:

```
In [2]: x = 1 + 2 + 3 + 4 +\
        5 + 6 + 7 + 8
```

It is also possible to continue expressions on the next line within parentheses, without using the `\` marker:

```
In [3]: x = (1 + 2 + 3 + 4 +
        5 + 6 + 7 + 8)
```

Most Python style guides recommend the second version of line continuation (within parentheses) to the first (use of the `\` marker).

Semicolon Can Optionally Terminate a Statement

Sometimes it can be useful to put multiple statements on a single line. The next portion of the script is:

```
lower = []; upper = []
```

This shows the example of how the semicolon (`;`) familiar in C can be used optionally in Python to put two statements on a single line. Functionally, this is entirely equivalent to writing:

```
lower = []
upper = []
```

Using a semicolon to put multiple statements on a single line is generally discouraged by most Python style guides, though occasionally it proves convenient.

Indentation: Whitespace Matters!

Next, we get to the main block of code:

```
for i in range(10):
    if i < midpoint:
        lower.append(i)
    else:
        upper.append(i)
```

This is a compound control-flow statement including a loop and a conditional—we'll look at these types of statements in a moment. For now, consider that this demonstrates what is perhaps the most controversial feature of Python's syntax: whitespace is meaningful!

In programming languages, a *block* of code is a set of statements that should be treated as a unit. In C, for example, code blocks are denoted by curly braces:

```
// C code
for(int i=0; i<100; i++)
   {
      // curly braces indicate code block
      total += i;
   }
```

In Python, code blocks are denoted by *indentation*:

```
for i in range(100):
    # indentation indicates code block
    total += i
```

In Python, indented code blocks are always preceded by a colon (`:`) on the previous line.

The use of indentation helps to enforce the uniform, readable style that many find appealing in Python code. But it might be confusing to the uninitiated; for example, the following two snippets will produce different results:

```
>>> if x < 4:
...     y = x * 2
...     print(x)
```

```
>>> if x < 4:
...     y = x * 2
... print(x)
```

In the snippet on the left, `print(x)` is in the indented block, and will be executed only if `x` is less than 4. In the snippet on the right, `print(x)` is outside the block, and will be executed regardless of the value of `x`!

Python's use of meaningful whitespace often is surprising to programmers who are accustomed to other languages, but in practice it can lead to much more consistent and readable code than languages that do not enforce indentation of code blocks. If you find Python's use of whitespace disagreeable, I'd encourage you to give it a try: as I did, you may find that you come to appreciate it.

Finally, you should be aware that the *amount* of whitespace used for indenting code blocks is up to the user, as long as it is consistent throughout the script. By convention, most style guides recommend to indent code blocks by four spaces, and that is the convention we will follow in this report. Note that many text editors like Emacs and Vim contain Python modes that do four-space indentation automatically.

Whitespace *Within* Lines Does Not Matter

While the mantra of *meaningful whitespace* holds true for whitespace *before* lines (which indicate a code block), whitespace *within* lines of Python code does not matter. For example, all three of these expressions are equivalent:

```
In [4]: x=1+2
        x = 1 + 2
        x             =         1    +                  2
```

Abusing this flexibility can lead to issues with code readability—in fact, abusing whitespace is often one of the primary means of intentionally obfuscating code (which some people do for sport). Using whitespace effectively can lead to much more readable code, especially in cases where operators follow each other—compare the following two expressions for exponentiating by a negative number:

```
x=10**-2
```

to

```
x = 10 ** -2
```

I find the second version with spaces much more easily readable at a single glance. Most Python style guides recommend using a single space around binary operators, and no space around unary operators. We'll discuss Python's operators further in "Basic Python Semantics: Variables and Objects" on page 13.

Parentheses Are for Grouping or Calling

In the following code snippet, we see two uses of parentheses. First, they can be used in the typical way to group statements or mathematical operations:

```
In [5]: 2 * (3 + 4)

Out [5]: 14
```

They can also be used to indicate that a *function* is being called. In the next snippet, the `print()` function is used to display the contents of a variable (see the sidebar that follows). The function call is indicated by a pair of opening and closing parentheses, with the *arguments* to the function contained within:

```
In [6]: print('first value:', 1)

first value: 1

In [7]: print('second value:', 2)

second value: 2
```

Some functions can be called with no arguments at all, in which case the opening and closing parentheses still must be used to indicate a function evaluation. An example of this is the `sort` method of lists:

```
In [8]: L = [4,2,3,1]
        L.sort()
        print(L)

        [1, 2, 3, 4]
```

The `()` after `sort` indicates that the function should be executed, and is required even if no arguments are necessary.

A Note on the print() Function

The `print()` function is one piece that has changed between Python 2.*x* and Python 3.*x*. In Python 2, `print` behaved as a statement—that is, you could write:

```
# Python 2 only!
>> print "first value:", 1
first value: 1
```

For various reasons, the language maintainers decided that in Python 3 `print()` should become a function, so we now write:

```
# Python 3 only!
>>> print("first value:", 1)
first value: 1
```

This is one of the many backward-incompatible constructs between Python 2 and 3. As of the writing of this report, it is common to find examples written in both versions of Python, and the presence of the `print` statement rather than the `print()` function is often one of the first signs that you're looking at Python 2 code.

Finishing Up and Learning More

This has been a very brief exploration of the essential features of Python syntax; its purpose is to give you a good frame of reference for when you're reading the code in later sections. Several times we've mentioned Python "style guides," which can help teams to write code in a consistent style. The most widely used style guide in Python is known as PEP8, and can be found at *https://www.python.org/dev/peps/pep-0008/*. As you begin to write more Python code, it would be useful to read through this! The style suggestions contain the wisdom of many Python gurus, and most suggestions go beyond simple pedantry: they are experience-based recommendations that can help avoid subtle mistakes and bugs in your code.

Basic Python Semantics: Variables and Objects

This section will begin to cover the basic semantics of the Python language. As opposed to the *syntax* covered in the previous section, the *semantics* of a language involve the meaning of the statements. As with our discussion of syntax, here we'll preview a few of the essential semantic constructions in Python to give you a better frame of reference for understanding the code in the following sections.

This section will cover the semantics of *variables* and *objects*, which are the main ways you store, reference, and operate on data within a Python script.

Python Variables Are Pointers

Assigning variables in Python is as easy as putting a variable name to the left of the equals sign (=):

```
# assign 4 to the variable x
x = 4
```

This may seem straightforward, but if you have the wrong mental model of what this operation does, the way Python works may seem confusing. We'll briefly dig into that here.

In many programming languages, variables are best thought of as containers or buckets into which you put data. So in C, for example, when you write

```
// C code
int x = 4;
```

you are essentially defining a "memory bucket" named `x`, and putting the value `4` into it. In Python, by contrast, variables are best thought of not as containers but as pointers. So in Python, when you write

```
x = 4
```

you are essentially defining a *pointer* named `x` that points to some other bucket containing the value `4`. Note one consequence of this: because Python variables just point to various objects, there is no need to "declare" the variable, or even require the variable to always point to information of the same type! This is the sense in which people say Python is *dynamically typed*: variable names can point to objects of any type. So in Python, you can do things like this:

```
In [1]: x = 1         # x is an integer
        x = 'hello'   # now x is a string
        x = [1, 2, 3] # now x is a list
```

While users of statically typed languages might miss the type-safety that comes with declarations like those found in C,

```
int x = 4;
```

this dynamic typing is one of the pieces that makes Python so quick to write and easy to read.

There is a consequence of this "variable as pointer" approach that you need to be aware of. If we have two variable names pointing to the same *mutable* object, then changing one will change the other as well! For example, let's create and modify a list:

```
In [2]: x = [1, 2, 3]
        y = x
```

We've created two variables `x` and `y` that both point to the same object. Because of this, if we modify the list via one of its names, we'll see that the "other" list will be modified as well:

```
In [3]: print(y)

[1, 2, 3]

In [4]: x.append(4) # append 4 to the list pointed to by x
        print(y) # y's list is modified as well!

[1, 2, 3, 4]
```

This behavior might seem confusing if you're wrongly thinking of variables as buckets that contain data. But if you're correctly thinking of variables as pointers to objects, then this behavior makes sense.

Note also that if we use `=` to assign another value to `x`, this will not affect the value of `y`—assignment is simply a change of what object the variable points to:

```
In [5]: x = 'something else'
        print(y)  # y is unchanged

[1, 2, 3, 4]
```

Again, this makes perfect sense if you think of `x` and `y` as pointers, and the `=` operator as an operation that changes what the name points to.

You might wonder whether this pointer idea makes arithmetic operations in Python difficult to track, but Python is set up so that this is not an issue. Numbers, strings, and other *simple types* are immutable: you can't change their value—you can only change what values the variables point to. So, for example, it's perfectly safe to do operations like the following:

```
In [6]: x = 10
        y = x
        x += 5  # add 5 to x's value, and assign it to x
        print("x =", x)
        print("y =", y)

x = 15
y = 10
```

When we call `x += 5`, we are not modifying the value of the `5` object pointed to by `x`, but rather we are changing the object to which `x`

points. For this reason, the value of `y` is not affected by the operation.

Everything Is an Object

Python is an object-oriented programming language, and in Python everything is an object.

Let's flesh out what this means. Earlier we saw that variables are simply pointers, and the variable names themselves have no attached type information. This leads some to claim erroneously that Python is a type-free language. But this is not the case! Consider the following:

```
In [7]:  x = 4
         type(x)

Out [7]: int

In [8]:  x = 'hello'
         type(x)

Out [8]: str

In [9]:  x = 3.14159
         type(x)

Out [9]: float
```

Python has types; however, the types are linked not to the variable names but *to the objects themselves*.

In object-oriented programming languages like Python, an *object* is an entity that contains data along with associated metadata and/or functionality. In Python, everything is an object, which means every entity has some metadata (called *attributes*) and associated functionality (called *methods*). These attributes and methods are accessed via the dot syntax.

For example, before we saw that lists have an `append` method, which adds an item to the list, and is accessed via the dot syntax (`.`):

```
In [10]: L = [1, 2, 3]
         L.append(100)
         print(L)

[1, 2, 3, 100]
```

While it might be expected for compound objects like lists to have attributes and methods, what is sometimes unexpected is that in Python even simple types have attached attributes and methods. For

example, numerical types have a `real` and `imag` attribute that return the real and imaginary part of the value, if viewed as a complex number:

```
In [11]: x = 4.5
         print(x.real, "+", x.imag, 'i')

4.5 + 0.0 i
```

Methods are like attributes, except they are functions that you can call using a pair of opening and closing parentheses. For example, floating-point numbers have a method called `is_integer` that checks whether the value is an integer:

```
In [12]:  x = 4.5
          x.is_integer()

Out [12]: False

In [13]:  x = 4.0
          x.is_integer()

Out [13]: True
```

When we say that everything in Python is an object, we really mean that *everything* is an object—even the attributes and methods of objects are themselves objects with their own `type` information:

```
In [14]:  type(x.is_integer)

Out [14]: builtin_function_or_method
```

We'll find that the everything-is-object design choice of Python allows for some very convenient language constructs.

Basic Python Semantics: Operators

In the previous section, we began to look at the semantics of Python variables and objects; here we'll dig into the semantics of the various *operators* included in the language. By the end of this section, you'll have the basic tools to begin comparing and operating on data in Python.

Arithmetic Operations

Python implements seven basic binary arithmetic operators, two of which can double as unary operators. They are summarized in the following table:

Operator	Name	Description
`a + b`	Addition	Sum of a and b
`a - b`	Subtraction	Difference of a and b
`a * b`	Multiplication	Product of a and b
`a / b`	True division	Quotient of a and b
`a // b`	Floor division	Quotient of a and b, removing fractional parts
`a % b`	Modulus	Remainder after division of a by b
`a ** b`	Exponentiation	a raised to the power of b
`-a`	Negation	The negative of a
`+a`	Unary plus	a unchanged (rarely used)

These operators can be used and combined in intuitive ways, using standard parentheses to group operations. For example:

```
In [1]:  # addition, subtraction, multiplication
         (4 + 8) * (6.5 - 3)

Out [1]: 42.0
```

Floor division is true division with fractional parts truncated:

```
In [2]: # True division
        print(11 / 2)

5.5

In [3]: # Floor division
        print(11 // 2)

5
```

The floor division operator was added in Python 3; you should be aware if working in Python 2 that the standard division operator (`/`) acts like floor division for integers and like true division for floating-point numbers.

Finally, I'll mention that an eighth arithmetic operator was added in Python 3.5: the `a @ b` operator, which is meant to indicate the *matrix product* of `a` and `b`, for use in various linear algebra packages.

Bitwise Operations

In addition to the standard numerical operations, Python includes operators to perform bitwise logical operations on integers. These are much less commonly used than the standard arithmetic opera-

tions, but it's useful to know that they exist. The six bitwise operators are summarized in the following table:

Operator	Name	Description
`a & b`	Bitwise AND	Bits defined in both `a` and `b`
`a | b`	Bitwise OR	Bits defined in `a` or `b` or both
`a ^ b`	Bitwise XOR	Bits defined in `a` or `b` but not both
`a << b`	Bit shift left	Shift bits of `a` left by `b` units
`a >> b`	Bit shift right	Shift bits of `a` right by `b` units
`~a`	Bitwise NOT	Bitwise negation of `a`

These bitwise operators only make sense in terms of the binary representation of numbers, which you can see using the built-in `bin` function:

```
In [4]:  bin(10)

Out [4]: '0b1010'
```

The result is prefixed with `0b`, which indicates a binary representation. The rest of the digits indicate that the number 10 is expressed as the sum:

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

Similarly, we can write:

```
In [5]:  bin(4)

Out [5]: '0b100'
```

Now, using bitwise OR, we can find the number which combines the bits of 4 and 10:

```
In [6]:  4 | 10

Out [6]: 14

In [7]:  bin(4 | 10)

Out [7]: '0b1110'
```

These bitwise operators are not as immediately useful as the standard arithmetic operators, but it's helpful to see them at least once to understand what class of operation they perform. In particular, users from other languages are sometimes tempted to use XOR (i.e., `a ^ b`) when they really mean exponentiation (i.e., `a ** b`).

Assignment Operations

We've seen that variables can be assigned with the `=` operator, and the values stored for later use. For example:

```
In [8]: a = 24
        print(a)
```

```
24
```

We can use these variables in expressions with any of the operators mentioned earlier. For example, to add 2 to `a` we write:

```
In [9]:  a + 2
```

```
Out [9]: 26
```

We might want to update the variable `a` with this new value; in this case, we could combine the addition and the assignment and write `a = a + 2`. Because this type of combined operation and assignment is so common, Python includes built-in update operators for all of the arithmetic operations:

```
In [10]: a += 2  # equivalent to a = a + 2
         print(a)
```

```
26
```

There is an augmented assignment operator corresponding to each of the binary operators listed earlier; in brief, they are:

```
a += b   a -= b  a *= b   a /= b
a //= b  a %= b  a **= b  a &= b
a |= b   a ^= b  a <<= b  a >>= b
```

Each one is equivalent to the corresponding operation followed by assignment: that is, for any operator `#`, the expression `a #= b` is equivalent to `a = a # b`, with a slight catch. For mutable objects like lists, arrays, or DataFrames, these augmented assignment operations are actually subtly different than their more verbose counterparts: they modify the contents of the original object rather than creating a new object to store the result.

Comparison Operations

Another type of operation that can be very useful is comparison of different values. For this, Python implements standard comparison

operators, which return Boolean values `True` and `False`. The comparison operations are listed in the following table:

Operation	Description
`a == b`	a equal to b
`a != b`	a not equal to b
`a < b`	a less than b
`a > b`	a greater than b
`a <= b`	a less than or equal to b
`a >= b`	a greater than or equal to b

These comparison operators can be combined with the arithmetic and bitwise operators to express a virtually limitless range of tests for the numbers. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

```
In [11]:  # 25 is odd
          25 % 2 == 1

Out [11]: True

In [12]:  # 66 is odd
          66 % 2 == 1

Out [12]: False
```

We can string together multiple comparisons to check more complicated relationships:

```
In [13]:  # check if a is between 15 and 30
          a = 25
          15 < a < 30

Out [13]: True
```

And, just to make your head hurt a bit, take a look at this comparison:

```
In [14]:  -1 == ~0

Out [14]: True
```

Recall that ~ is the bit-flip operator, and evidently when you flip all the bits of zero you end up with –1. If you're curious as to why this is, look up the *two's complement* integer encoding scheme, which is what Python uses to encode signed integers, and think about happens when you start flipping all the bits of integers encoded this way.

Boolean Operations

When working with Boolean values, Python provides operators to combine the values using the standard concepts of "and", "or", and "not". Predictably, these operators are expressed using the words `and`, `or`, and `not`:

```
In [15]:  x = 4
          (x < 6) and (x > 2)

Out [15]: True

In [16]:  (x > 10) or (x % 2 == 0)

Out [16]: True

In [17]:  not (x < 6)

Out [17]: False
```

Boolean algebra aficionados might notice that the XOR operator is not included; this can of course be constructed in several ways from a compound statement of the other operators. Otherwise, a clever trick you can use for XOR of Boolean values is the following:

```
In [18]:  # (x > 1) xor (x < 10)
          (x > 1) != (x < 10)

Out [18]: False
```

These sorts of Boolean operations will become extremely useful when we begin discussing *control flow statements* such as conditionals and loops.

One sometimes confusing thing about the language is when to use Boolean operators (`and`, `or`, `not`), and when to use bitwise operations (`&`, `|`, `~`). The answer lies in their names: Boolean operators should be used when you want to compute Boolean values (i.e., truth or falsehood) of entire statements. Bitwise operations should be used when you want to operate on individual bits or components of the objects in question.

Identity and Membership Operators

Like `and`, `or`, and `not`, Python also contains prose-like operators to check for identity and membership. They are the following:

Operator	Description
`a is b`	True if a and b are identical objects
`a is not b`	True if a and b are not identical objects
`a in b`	True if a is a member of b
`a not in b`	True if a is not a member of b

Identity operators: is and is not

The identity operators, `is` and `is not`, check for *object identity*. Object identity is different than equality, as we can see here:

```
In [19]:  a = [1, 2, 3]
          b = [1, 2, 3]

In [20]:  a == b

Out [20]: True

In [21]:  a is b

Out [21]: False

In [22]:  a is not b

Out [22]: True
```

What do identical objects look like? Here is an example:

```
In [23]:  a = [1, 2, 3]
          b = a
          a is b

Out [23]: True
```

The difference between the two cases here is that in the first, `a` and `b` point to *different objects*, while in the second they point to the *same object*. As we saw in the previous section, Python variables are pointers. The `is` operator checks whether the two variables are pointing to the same container (object), rather than referring to what the container contains. With this in mind, in most cases that a beginner is tempted to use `is`, what they really mean is `==`.

Membership operators

Membership operators check for membership within compound objects. So, for example, we can write:

```
In [24]:  1 in [1, 2, 3]

Out [24]: True
```

```
In [25]:  2 not in [1, 2, 3]

Out [25]: False
```

These membership operations are an example of what makes Python so easy to use compared to lower-level languages such as C. In C, membership would generally be determined by manually constructing a loop over the list and checking for equality of each value. In Python, you just type what you want to know, in a manner reminiscent of straightforward English prose.

Built-In Types: Simple Values

When discussing Python variables and objects, we mentioned the fact that all Python objects have type information attached. Here we'll briefly walk through the built-in simple types offered by Python. We say "simple types" to contrast with several compound types, which will be discussed in the following section.

Python's simple types are summarized in Table 1-1.

Table 1-1. Python scalar types

Type	Example	Description
`int`	`x = 1`	Integers (i.e., whole numbers)
`float`	`x = 1.0`	Floating-point numbers (i.e., real numbers)
`complex`	`x = 1 + 2j`	Complex numbers (i.e., numbers with a real and imaginary part)
`bool`	`x = True`	Boolean: True/False values
`str`	`x = 'abc'`	String: characters or text
`NoneType`	`x = None`	Special object indicating nulls

We'll take a quick look at each of these in turn.

Integers

The most basic numerical type is the integer. Any number without a decimal point is an integer:

```
In [1]:  x = 1
         type(x)

Out [1]: int
```

Python integers are actually quite a bit more sophisticated than integers in languages like `C`. C integers are fixed-precision, and usually

overflow at some value (often near 2^{31} or 2^{63}, depending on your system). Python integers are variable-precision, so you can do computations that would overflow in other languages:

```
In [2]:  2 ** 200

Out [2]:
1606938044258990275541962092341162602522202993782792835301376
```

Another convenient feature of Python integers is that by default, division upcasts to floating-point type:

```
In [3]:  5 / 2

Out [3]: 2.5
```

Note that this upcasting is a feature of Python 3; in Python 2, like in many statically typed languages such as C, integer division truncates any decimal and always returns an integer:

```
# Python 2 behavior
>>> 5 / 2
2
```

To recover this behavior in Python 3, you can use the floor-division operator:

```
In [4]:  5 // 2

Out [4]: 2
```

Finally, note that although Python 2.*x* had both an `int` and `long` type, Python 3 combines the behavior of these two into a single `int` type.

Floating-Point Numbers

The floating-point type can store fractional numbers. They can be defined either in standard decimal notation, or in exponential notation:

```
In [5]: x = 0.000005
        y = 5e-6
        print(x == y)

True

In [6]: x = 1400000.00
        y = 1.4e6
        print(x == y)

True
```

In the exponential notation, the `e` or `E` can be read "…times ten to the…", so that `1.4e6` is interpreted as 1.4×10^6.

An integer can be explicitly converted to a float with the `float` constructor:

```
In [7]:  float(1)
```

```
Out [7]: 1.0
```

Floating-point precision

One thing to be aware of with floating-point arithmetic is that its precision is limited, which can cause equality tests to be unstable. For example:

```
In [8]:  0.1 + 0.2 == 0.3
```

```
Out [8]: False
```

Why is this the case? It turns out that it is not a behavior unique to Python, but is due to the fixed-precision format of the binary floating-point storage used by most, if not all, scientific computing platforms. All programming languages using floating-point numbers store them in a fixed number of bits, and this leads some numbers to be represented only approximately. We can see this by printing the three values to high precision:

```
In [9]: print("0.1 = {0:.17f}".format(0.1))
        print("0.2 = {0:.17f}".format(0.2))
        print("0.3 = {0:.17f}".format(0.3))
```

```
0.1 = 0.10000000000000001
0.2 = 0.20000000000000001
0.3 = 0.29999999999999999
```

We're accustomed to thinking of numbers in decimal (base-10) notation, so that each fraction must be expressed as a sum of powers of 10:

$$1/8 = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

In the familiar base-10 representation, we represent this in the familiar decimal expression: 0.125.

Computers usually store values in binary notation, so that each number is expressed as a sum of powers of 2:

$$1/8 = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

In a base-2 representation, we can write this 0.001_2, where the subscript 2 indicates binary notation. The value $0.125 = 0.001_2$ happens to be one number which both binary and decimal notation can represent in a finite number of digits.

In the familiar base-10 representation of numbers, you are probably familiar with numbers that can't be expressed in a finite number of digits. For example, dividing 1 by 3 gives, in standard decimal notation:

$$1/3 = 0.333333333...$$

The 3s go on forever: that is, to truly represent this quotient, the number of required digits is infinite!

Similarly, there are numbers for which binary representations require an infinite number of digits. For example:

$$1/10 = 0.00011001100110011..._2$$

Just as decimal notation requires an infinite number of digits to perfectly represent 1/3, binary notation requires an infinite number of digits to represent 1/10. Python internally truncates these representations at 52 bits beyond the first nonzero bit on most systems.

This rounding error for floating-point values is a necessary evil of working with floating-point numbers. The best way to deal with it is to always keep in mind that floating-point arithmetic is approximate, and *never* rely on exact equality tests with floating-point values.

Complex Numbers

Complex numbers are numbers with real and imaginary (floating-point) parts. We've seen integers and real numbers before; we can use these to construct a complex number:

```
In [10]:  complex(1, 2)

Out [10]: (1+2j)
```

Alternatively, we can use the `j` suffix in expressions to indicate the imaginary part:

```
In [11]:  1 + 2j

Out [11]: (1+2j)
```

Complex numbers have a variety of interesting attributes and methods, which we'll briefly demonstrate here:

```
In [12]:  c = 3 + 4j

In [13]:  c.real  # real part

Out [13]: 3.0

In [14]:  c.imag  # imaginary part

Out [14]: 4.0

In [15]:  c.conjugate()  # complex conjugate

Out [15]: (3-4j)

In [16]:
abs(c)  # magnitude--that is, sqrt(c.real ** 2 + c.imag ** 2)

Out [16]: 5.0
```

String Type

Strings in Python are created with single or double quotes:

```
In [17]: message = "what do you like?"
         response = 'spam'
```

Python has many extremely useful string functions and methods; here are a few of them:

```
In [18]:  # length of string
          len(response)

Out [18]: 4

In [19]:  # Make uppercase. See also str.lower()
          response.upper()

Out [19]: 'SPAM'

In [20]:  # Capitalize. See also str.title()
          message.capitalize()

Out [20]: 'What do you like?'

In [21]:  # concatenation with +
          message + response

Out [21]: 'what do you like?spam'

In [22]:  # multiplication is multiple concatenation
          5 * response

Out [22]: 'spamspamspamspamspam'
```

```
In [23]:  # Access individual characters (zero-based indexing)
          message[0]

Out [23]: 'w'
```

For more discussion of indexing in Python, see "Lists" on page 31.

None Type

Python includes a special type, the `NoneType`, which has only a single possible value: `None`. For example:

```
In [24]:  type(None)

Out [24]: NoneType
```

You'll see `None` used in many places, but perhaps most commonly it is used as the default return value of a function. For example, the `print()` function in Python 3 does not return anything, but we can still catch its value:

```
In [25]: return_value = print('abc')

abc

In [26]: print(return_value)

None
```

Likewise, any function in Python with no return value is, in reality, returning `None`.

Boolean Type

The Boolean type is a simple type with two possible values: `True` and `False`, and is returned by comparison operators discussed previously:

```
In [27]:  result = (4 < 5)
          result

Out [27]: True

In [28]:  type(result)

Out [28]: bool
```

Keep in mind that the Boolean values are case-sensitive: unlike some other languages, `True` and `False` must be capitalized!

```
In [29]: print(True, False)

True False
```

Booleans can also be constructed using the `bool()` object constructor: values of any other type can be converted to Boolean via predictable rules. For example, any numeric type is False if equal to zero, and True otherwise:

```
In [30]:  bool(2014)
Out [30]: True
In [31]:  bool(0)
Out [31]: False
In [32]:  bool(3.1415)
Out [32]: True
```

The Boolean conversion of `None` is always False:

```
In [33]:  bool(None)
Out [33]: False
```

For strings, `bool(s)` is False for empty strings and True otherwise:

```
In [34]:  bool("")
Out [34]: False
In [35]:  bool("abc")
Out [35]: True
```

For sequences, which we'll see in the next section, the Boolean representation is False for empty sequences and True for any other sequences:

```
In [36]:  bool([1, 2, 3])
Out [36]: True
In [37]:  bool([])
Out [37]: False
```

Built-In Data Structures

We have seen Python's simple types: `int`, `float`, `complex`, `bool`, `str`, and so on. Python also has several built-in compound types, which act as containers for other types. These compound types are:

Type Name	Example	Description
`list`	`[1, 2, 3]`	Ordered collection
`tuple`	`(1, 2, 3)`	Immutable ordered collection

Type Name	Example	Description
`dict`	`{'a':1, 'b':2, 'c':3}`	Unordered (key,value) mapping
`set`	`{1, 2, 3}`	Unordered collection of unique values

As you can see, round, square, and curly brackets have distinct meanings when it comes to the type of collection produced. We'll take a quick tour of these data structures here.

Lists

Lists are the basic *ordered* and *mutable* data collection type in Python. They can be defined with comma-separated values between square brackets; here is a list of the first several prime numbers:

```
In [1]: L = [2, 3, 5, 7]
```

Lists have a number of useful properties and methods available to them. Here we'll take a quick look at some of the more common and useful ones:

```
In [2]:  # Length of a list
         len(L)

Out [2]: 4

In [3]:  # Append a value to the end
         L.append(11)
         L

Out [3]: [2, 3, 5, 7, 11]

In [4]:  # Addition concatenates lists
         L + [13, 17, 19]

Out [4]: [2, 3, 5, 7, 11, 13, 17, 19]

In [5]:  # sort() method sorts in-place
         L = [2, 5, 1, 6, 3, 4]
         L.sort()
         L

Out [5]: [1, 2, 3, 4, 5, 6]
```

In addition, there are many more built-in list methods; they are well-covered in Python's online documentation.

While we've been demonstrating lists containing values of a single type, one of the powerful features of Python's compound objects is that they can contain objects of *any* type, or even a mix of types. For example:

```
In [6]: L = [1, 'two', 3.14, [0, 3, 5]]
```

This flexibility is a consequence of Python's dynamic type system. Creating such a mixed sequence in a statically typed language like C can be much more of a headache! We see that lists can even contain other lists as elements. Such type flexibility is an essential piece of what makes Python code relatively quick and easy to write.

So far we've been considering manipulations of lists as a whole; another essential piece is the accessing of individual elements. This is done in Python via *indexing* and *slicing*, which we'll explore next.

List indexing and slicing

Python provides access to elements in compound types through *indexing* for single elements, and *slicing* for multiple elements. As we'll see, both are indicated by a square-bracket syntax. Suppose we return to our list of the first several primes:

```
In [7]: L = [2, 3, 5, 7, 11]
```

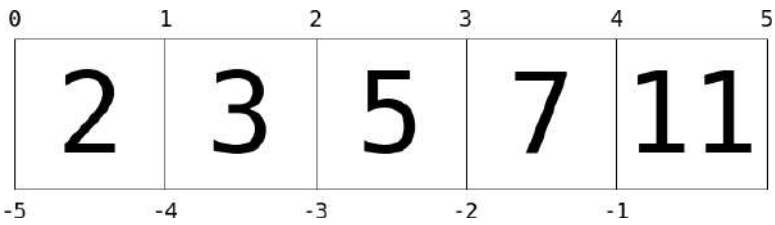
Python uses *zero-based* indexing, so we can access the first and second element in using the following syntax:

```
In [8]:  L[0]
Out [8]: 2
In [9]:  L[1]
Out [9]: 3
```

Elements at the end of the list can be accessed with negative numbers, starting from -1:

```
In [10]:  L[-1]
Out [10]: 11
In [12]:  L[-2]
Out [12]: 7
```

You can visualize this indexing scheme this way:



Here values in the list are represented by large numbers in the squares; list indices are represented by small numbers above and below. In this case, `L[2]` returns `5`, because that is the next value at index `2`.

Where *indexing* is a means of fetching a single value from the list, *slicing* is a means of accessing multiple values in sublists. It uses a colon to indicate the start point (inclusive) and end point (non-inclusive) of the subarray. For example, to get the first three elements of the list, we can write it as follows:

```
In [12]:  L[0:3]
```

```
Out [12]: [2, 3, 5]
```

Notice where `0` and `3` lie in the preceding diagram, and how the slice takes just the values between the indices. If we leave out the first index, `0` is assumed, so we can equivalently write the following:

```
In [13]:  L[:3]
```

```
Out [13]: [2, 3, 5]
```

Similarly, if we leave out the last index, it defaults to the length of the list. Thus, the last three elements can be accessed as follows:

```
In [14]:  L[-3:]
```

```
Out [14]: [5, 7, 11]
```

Finally, it is possible to specify a third integer that represents the step size; for example, to select every second element of the list, we can write:

```
In [15]:  L[::2]  # equivalent to L[0:len(L):2]
```

```
Out [15]: [2, 5, 11]
```

A particularly useful version of this is to specify a negative step, which will reverse the array:

```
In [16]:  L[::-1]
```

```
Out [16]: [11, 7, 5, 3, 2]
```

Both indexing and slicing can be used to set elements as well as access them. The syntax is as you would expect:

```
In [17]: L[0] = 100
         print(L)
```

```
[100, 3, 5, 7, 11]
```

```
In [18]: L[1:3] = [55, 56]
         print(L)
```

```
[100, 55, 56, 7, 11]
```

A very similar slicing syntax is also used in many data science–oriented packages, including NumPy and Pandas (mentioned in the introduction).

Now that we have seen Python lists and how to access elements in ordered compound types, let's take a look at the other three standard compound data types mentioned earlier.

Tuples

Tuples are in many ways similar to lists, but they are defined with parentheses rather than square brackets:

```
In [19]: t = (1, 2, 3)
```

They can also be defined without any brackets at all:

```
In [20]: t = 1, 2, 3
         print(t)
```

```
(1, 2, 3)
```

Like the lists discussed before, tuples have a length, and individual elements can be extracted using square-bracket indexing:

```
In [21]:  len(t)
```

```
Out [21]: 3
```

```
In [22]:  t[0]
```

```
Out [22]: 1
```

The main distinguishing feature of tuples is that they are *immutable*: this means that once they are created, their size and contents cannot be changed:

```
In [23]: t[1] = 4
```

```
---------------------------------------------------------

TypeError                 Traceback (most recent call last)

<ipython-input-23-141c76cb54a2> in <module>()
----> 1 t[1] = 4


TypeError: 'tuple' object does not support item assignment
```

```
In [24]: t.append(4)

------------------------------------------------------------

AttributeError                Traceback (most recent call last)

<ipython-input-24-e8bd1632f9dd> in <module>()
----> 1 t.append(4)


AttributeError: 'tuple' object has no attribute 'append'
```

Tuples are often used in a Python program; a particularly common case is in functions that have multiple return values. For example, the `as_integer_ratio()` method of floating-point objects returns a numerator and a denominator; this dual return value comes in the form of a tuple:

```
In [25]:  x = 0.125
          x.as_integer_ratio()

Out [25]: (1, 8)
```

These multiple return values can be individually assigned as follows:

```
In [26]: numerator, denominator = x.as_integer_ratio()
         print(numerator / denominator)

0.125
```

The indexing and slicing logic covered earlier for lists works for tuples as well, along with a host of other methods. Refer to the Data Structures documentation for a more complete list of these.

Dictionaries

Dictionaries are extremely flexible mappings of keys to values, and form the basis of much of Python's internal implementation. They can be created via a comma-separated list of `key:value` pairs within curly braces:

```
In [27]: numbers = {'one':1, 'two':2, 'three':3}
```

Items are accessed and set via the indexing syntax used for lists and tuples, except here the index is not a zero-based order but valid key in the dictionary:

```
In [28]:  # Access a value via the key
          numbers['two']

Out [28]: 2
```

New items can be added to the dictionary using indexing as well:

```
In [29]: # Set a new key/value pair
         numbers['ninety'] = 90
         print(numbers)

{'three': 3, 'ninety': 90, 'two': 2, 'one': 1}
```

Keep in mind that dictionaries do not maintain any sense of order for the input parameters; this is by design. This lack of ordering allows dictionaries to be implemented very efficiently, so that random element access is very fast, regardless of the size of the dictionary (if you're curious how this works, read about the concept of a *hash table*). The Python documentation has a complete list of the methods available for dictionaries.

Sets

The fourth basic collection is the set, which contains unordered collections of unique items. They are defined much like lists and tuples, except they use the curly brackets of dictionaries:

```
In [30]: primes = {2, 3, 5, 7}
         odds = {1, 3, 5, 7, 9}
```

If you're familiar with the mathematics of sets, you'll be familiar with operations like the union, intersection, difference, symmetric difference, and others. Python's sets have all of these operations built in via methods or operators. For each, we'll show the two equivalent methods:

```
In [31]:  # union: items appearing in either
          primes | odds      # with an operator
          primes.union(odds) # equivalently with a method

Out [31]: {1, 2, 3, 5, 7, 9}

In [32]:  # intersection: items appearing in both
          primes & odds             # with an operator
          primes.intersection(odds) # equivalently with a method

Out [32]: {3, 5, 7}

In [33]:  # difference: items in primes but not in odds
          primes - odds           # with an operator
          primes.difference(odds) # equivalently with a method

Out [33]: {2}
```

```
In [34]:
# symmetric difference: items appearing in only one set
primes ^ odds                    # with an operator
primes.symmetric_difference(odds) # equivalently with a method
```

```
Out [34]: {1, 2, 9}
```

Many more set methods and operations are available. You've probably already guessed what I'll say next: refer to Python's online documentation for a complete reference.

More Specialized Data Structures

Python contains several other data structures that you might find useful; these can generally be found in the built-in `collections` module. The `collections` module is fully documented in Python's online documentation, and you can read more about the various objects available there.

In particular, I've found the following very useful on occasion:

`collections.namedtuple`
: Like a tuple, but each value has a name

`collections.defaultdict`
: Like a dictionary, but unspecified keys have a user-specified default value

`collections.OrderedDict`
: Like a dictionary, but the order of keys is maintained

Once you've seen the standard built-in collection types, the use of these extended functionalities is very intuitive, and I'd suggest reading about their use.

Control Flow

Control flow is where the rubber really meets the road in programming. Without it, a program is simply a list of statements that are sequentially executed. With control flow, you can execute certain code blocks conditionally and/or repeatedly: these basic building blocks can be combined to create surprisingly sophisticated programs!

Here we'll cover conditional statements (including `if`, `elif`, and `else`) and loop statements (including `for` and `while`, and the accompanying `break`, `continue`, and `pass`).

Conditional Statements: if, elif, and else

Conditional statements, often referred to as *if-then* statements, allow the programmer to execute certain pieces of code depending on some Boolean condition. A basic example of a Python conditional statement is this:

```
In [1]: x = -15

        if x == 0:
            print(x, "is zero")
        elif x > 0:
            print(x, "is positive")
        elif x < 0:
            print(x, "is negative")
        else:
            print(x, "is unlike anything I've ever seen...")

-15 is negative
```

Note especially the use of colons (`:`) and whitespace to denote separate blocks of code.

Python adopts the `if` and `else` often used in other languages; its more unique keyword is `elif`, a contraction of "else if". In these conditional clauses, `elif` and `else` blocks are optional; additionally, you can optionally include as few or as many `elif` statements as you would like.

for loops

Loops in Python are a way to repeatedly execute some code statement. So, for example, if we'd like to print each of the items in a list, we can use a `for` loop:

```
In [2]: for N in [2, 3, 5, 7]:
            print(N, end=' ') # print all on same line

2 3 5 7
```

Notice the simplicity of the `for` loop: we specify the variable we want to use, the sequence we want to loop over, and use the `in` operator to link them together in an intuitive and readable way. More precisely, the object to the right of the `in` can be any Python *iterator*.

An iterator can be thought of as a generalized sequence, and we'll discuss them in "Iterators" on page 52.

For example, one of the most commonly used iterators in Python is the `range` object, which generates a sequence of numbers:

```
In [3]: for i in range(10):
            print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

Note that the range starts at zero by default, and that by convention the top of the range is not included in the output. Range objects can also have more complicated values:

```
In [4]:  # range from 5 to 10
         list(range(5, 10))

Out [4]: [5, 6, 7, 8, 9]

In [5]:  # range from 0 to 10 by 2
         list(range(0, 10, 2))

Out [5]: [0, 2, 4, 6, 8]
```

You might notice that the meaning of `range` arguments is very similar to the slicing syntax that we covered in "Lists" on page 31.

Note that the behavior of `range()` is one of the differences between Python 2 and Python 3: in Python 2, `range()` produces a list, while in Python 3, `range()` produces an iterable object.

while loops

The other type of loop in Python is a `while` loop, which iterates until some condition is met:

```
In [6]: i = 0
        while i < 10:
            print(i, end=' ')
            i += 1

0 1 2 3 4 5 6 7 8 9
```

The argument of the `while` loop is evaluated as a Boolean statement, and the loop is executed until the statement evaluates to False.

break and continue: Fine-Tuning Your Loops

There are two useful statements that can be used within loops to fine-tune how they are executed:

- The `break` statement breaks out of the loop entirely
- The `continue` statement skips the remainder of the current loop, and goes to the next iteration

These can be used in both `for` and `while` loops.

Here is an example of using `continue` to print a string of even numbers. In this case, the result could be accomplished just as well with an `if-else` statement, but sometimes the `continue` statement can be a more convenient way to express the idea you have in mind:

```
In [7]: for n in range(20):
            # check if n is even
            if n % 2 == 0:
                continue
            print(n, end=' ')
```

```
1 3 5 7 9 11 13 15 17 19
```

Here is an example of a `break` statement used for a less trivial task. This loop will fill a list with all Fibonacci numbers up to a certain value:

```
In [8]: a, b = 0, 1
        amax = 100
        L = []

        while True:
            (a, b) = (b, a + b)
            if a > amax:
                break
            L.append(a)

        print(L)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Notice that we use a `while True` loop, which will loop forever unless we have a break statement!

Loops with an else Block

One rarely used pattern available in Python is the `else` statement as part of a `for` or `while` loop. We discussed the `else` block earlier: it executes if all the `if` and `elif` statements evaluate to `False`. The loop-`else` is perhaps one of the more confusingly named statements in Python; I prefer to think of it as a `nobreak` statement: that is, the

`else` block is executed only if the loop ends naturally, without encountering a `break` statement.

As an example of where this might be useful, consider the following (non-optimized) implementation of the *Sieve of Eratosthenes*, a well-known algorithm for finding prime numbers:

```
In [9]: L = []
        nmax = 30

        for n in range(2, nmax):
            for factor in L:
                if n % factor == 0:
                    break
            else: # no break
                L.append(n)
        print(L)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The `else` statement only executes if none of the factors divide the given number. The `else` statement works similarly with the `while` loop.

Defining and Using Functions

So far, our scripts have been simple, single-use code blocks. One way to organize our Python code and to make it more readable and reusable is to factor-out useful pieces into reusable *functions*. Here we'll cover two ways of creating functions: the `def` statement, useful for any type of function, and the `lambda` statement, useful for creating short anonymous functions.

Using Functions

Functions are groups of code that have a name and can be called using parentheses. We've seen functions before. For example, `print` in Python 3 is a function:

```
In [1]: print('abc')
```

```
abc
```

Here `print` is the function name, and `'abc'` is the function's *argument*.

In addition to arguments, there are *keyword arguments* that are specified by name. One available keyword argument for the `print()`

function (in Python 3) is `sep`, which tells what character or characters should be used to separate multiple items:

```
In [2]: print(1, 2, 3)

1 2 3

In [3]: print(1, 2, 3, sep='--')

1--2--3
```

When non-keyword arguments are used together with keyword arguments, the keyword arguments must come at the end.

Defining Functions

Functions become even more useful when we begin to define our own, organizing functionality to be used in multiple places. In Python, functions are defined with the `def` statement. For example, we can encapsulate a version of our Fibonacci sequence code from the previous section as follows:

```
In [4]: def fibonacci(N):
            L = []
            a, b = 0, 1
            while len(L) < N:
                a, b = b, a + b
                L.append(a)
            return L
```

Now we have a function named `fibonacci` which takes a single argument `N`, does something with this argument, and `returns` a value; in this case, a list of the first `N` Fibonacci numbers:

```
In [5]:  fibonacci(10)

Out [5]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

If you're familiar with strongly typed languages like `C`, you'll immediately notice that there is no type information associated with the function inputs or outputs. Python functions can return any Python object, simple or compound, which means constructs that may be difficult in other languages are straightforward in Python.

For example, multiple return values are simply put in a tuple, which is indicated by commas:

```
In [6]: def real_imag_conj(val):
            return val.real, val.imag, val.conjugate()
```

```
r, i, c = real_imag_conj(3 + 4j)
print(r, i, c)
```

```
3.0 4.0 (3-4j)
```

Default Argument Values

Often when defining a function, there are certain values that we want the function to use *most* of the time, but we'd also like to give the user some flexibility. In this case, we can use *default values* for arguments. Consider the `fibonacci` function from before. What if we would like the user to be able to play with the starting values? We could do that as follows:

```
In [7]: def fibonacci(N, a=0, b=1):
            L = []
            while len(L) < N:
                a, b = b, a + b
                L.append(a)
            return L
```

With a single argument, the result of the function call is identical to before:

```
In [8]:  fibonacci(10)
```

```
Out [8]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

But now we can use the function to explore new things, such as the effect of new starting values:

```
In [9]:  fibonacci(10, 0, 2)
```

```
Out [9]: [2, 2, 4, 6, 10, 16, 26, 42, 68, 110]
```

The values can also be specified by name if desired, in which case the order of the named values does not matter:

```
In [10]:  fibonacci(10, b=3, a=1)
```

```
Out [10]: [3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

*args and **kwargs: Flexible Arguments

Sometimes you might wish to write a function in which you don't initially know how many arguments the user will pass. In this case, you can use the special form `*args` and `**kwargs` to catch all arguments that are passed. Here is an example:

```
In [11]: def catch_all(*args, **kwargs):
             print("args =", args)
             print("kwargs = ", kwargs)
```

```
In [12]: catch_all(1, 2, 3, a=4, b=5)

         args = (1, 2, 3)
         kwargs =  {'a': 4, 'b': 5}

In [13]: catch_all('a', keyword=2)

         args = ('a',)
         kwargs =  {'keyword': 2}
```

Here it is not the names `args` and `kwargs` that are important, but the `*` characters preceding them. `args` and `kwargs` are just the variable names often used by convention, short for "arguments" and "keyword arguments". The operative difference is the asterisk characters: a single `*` before a variable means "expand this as a sequence", while a double `**` before a variable means "expand this as a dictionary". In fact, this syntax can be used not only with the function definition, but with the function call as well!

```
In [14]: inputs = (1, 2, 3)
         keywords = {'pi': 3.14}

         catch_all(*inputs, **keywords)

         args = (1, 2, 3)
         kwargs =  {'pi': 3.14}
```

Anonymous (lambda) Functions

Earlier we quickly covered the most common way of defining functions, the `def` statement. You'll likely come across another way of defining short, one-off functions with the `lambda` statement. It looks something like this:

```
In [15]:  add = lambda x, y: x + y
          add(1, 2)

Out [15]: 3
```

This lambda function is roughly equivalent to

```
In [16]: def add(x, y):
             return x + y
```

So why would you ever want to use such a thing? Primarily, it comes down to the fact that *everything is an object* in Python, even functions themselves! That means that functions can be passed as arguments to functions.

As an example of this, suppose we have some data stored in a list of dictionaries:

```
In [17]:
data = [{'first':'Guido', 'last':'Van Rossum', 'YOB':1956},
        {'first':'Grace', 'last':'Hopper',     'YOB':1906},
        {'first':'Alan',  'last':'Turing',     'YOB':1912}]
```

Now suppose we want to sort this data. Python has a `sorted` function that does this:

```
In [18]:  sorted([2,4,3,5,1,6])

Out [18]: [1, 2, 3, 4, 5, 6]
```

But dictionaries are not orderable: we need a way to tell the function *how* to sort our data. We can do this by specifying the `key` function, a function which given an item returns the sorting key for that item:

```
In [19]:  # sort alphabetically by first name
          sorted(data, key=lambda item: item['first'])

Out [19]:
[{'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},
 {'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},
 {'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]

In [20]:  # sort by year of birth
          sorted(data, key=lambda item: item['YOB'])

Out [20]:
[{'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},
 {'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},
 {'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]
```

While these key functions could certainly be created by the normal, `def` syntax, the `lambda` syntax is convenient for such short one-off functions like these.

Errors and Exceptions

No matter your skill as a programmer, you will eventually make a coding mistake. Such mistakes come in three basic flavors:

Syntax errors
: Errors where the code is not valid Python (generally easy to fix)

Runtime errors
: Errors where syntactically valid code fails to execute, perhaps due to invalid user input (sometimes easy to fix)

Semantic errors
: Errors in logic: code executes without a problem, but the result is not what you expect (often very difficult to identify and fix)

Here we're going to focus on how to deal cleanly with runtime errors. As we'll see, Python handles runtime errors via its *exception handling* framework.

Runtime Errors

If you've done any coding in Python, you've likely come across runtime errors. They can happen in a lot of ways.

For example, if you try to reference an undefined variable:

```
In [1]: print(Q)
---------------------------------------------------------
NameError               Traceback (most recent call last)

<ipython-input-3-e796bdcf24ff> in <module>()
----> 1 print(Q)


NameError: name 'Q' is not defined
```

Or if you try an operation that's not defined:

```
In [2]: 1 + 'abc'
---------------------------------------------------------
TypeError               Traceback (most recent call last)

<ipython-input-4-aab9e8ede4f7> in <module>()
----> 1 1 + 'abc'


TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Or you might be trying to compute a mathematically ill-defined result:

```
In [3]: 2 / 0
---------------------------------------------------------
ZeroDivisionError               Traceback (most recent call last)

<ipython-input-5-ae0c5d243292> in <module>()
----> 1 2 / 0


ZeroDivisionError: division by zero
```

Or maybe you're trying to access a sequence element that doesn't exist:

```
In [4]: L = [1, 2, 3]
        L[1000]

---------------------------------------------------------

IndexError                Traceback (most recent call last)

<ipython-input-6-06b6eb1b8957> in <module>()
      1 L = [1, 2, 3]
----> 2 L[1000]


IndexError: list index out of range
```

Note that in each case, Python is kind enough to not simply indicate that an error happened, but to spit out a *meaningful* exception that includes information about what exactly went wrong, along with the exact line of code where the error happened. Having access to meaningful errors like this is immensely useful when trying to trace the root of problems in your code.

Catching Exceptions: try and except

The main tool Python gives you for handling runtime exceptions is the `try`…`except` clause. Its basic structure is this:

```
In [5]:
try:
    print("this gets executed first")
except:
    print("this gets executed only if there is an error")
```

```
this gets executed first
```

Note that the second block here did not get executed: this is because the first block did not return an error. Let's put a problematic statement in the `try` block and see what happens:

```
In [6]: try:
            print("let's try something:")
            x = 1 / 0 # ZeroDivisionError
        except:
            print("something bad happened!")
```

```
        let's try something:
        something bad happened!
```

Here we see that when the error was raised in the `try` statement (in this case, a `ZeroDivisionError`), the error was caught, and the `except` statement was executed.

One way this is often used is to check user input within a function or another piece of code. For example, we might wish to have a function that catches zero-division and returns some other value, perhaps a suitably large number like 10^{100}:

```
In [7]:  def safe_divide(a, b):
             try:
                 return a / b
             except:
                 return 1E100

In [8]:  safe_divide(1, 2)

Out [8]: 0.5

In [9]:  safe_divide(2, 0)

Out [9]: 1e+100
```

There is a subtle problem with this code, though: what happens when another type of exception comes up? For example, this is probably not what we intended:

```
In [10]:  safe_divide (1, '2')

Out [10]: 1e+100
```

Dividing an integer and a string raises a `TypeError`, which our over-zealous code caught and assumed was a `ZeroDivisionError`! For this reason, it's nearly always a better idea to catch exceptions *explicitly*:

```
In [11]:  def safe_divide(a, b):
              try:
                  return a / b
              except ZeroDivisionError:
                  return 1E100

In [12]:  safe_divide(1, 0)

Out [12]: 1e+100

In [13]:  safe_divide(1, '2')

---------------------------------------------------------

TypeError                 Traceback (most recent call last)

<ipython-input-15-2331af6a0acf> in <module>()
----> 1 safe_divide(1, '2')
```

```
<ipython-input-13-10b5f0163af8> in safe_divide(a, b)
      1 def safe_divide(a, b):
      2     try:
----> 3         return a / b
      4     except ZeroDivisionError:
      5         return 1E100


TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

We're now catching zero-division errors only, and letting all other errors pass through unmodified.

Raising Exceptions: raise

We've seen how valuable it is to have informative exceptions when using parts of the Python language. It's equally valuable to make use of informative exceptions within the code you write, so that users of your code (foremost yourself!) can figure out what caused their errors.

The way you raise your own exceptions is with the `raise` statement. For example:

```
In [14]: raise RuntimeError("my error message")

---------------------------------------------------------

RuntimeError                 Traceback (most recent call last)

<ipython-input-16-c6a4c1ed2f34> in <module>()
----> 1 raise RuntimeError("my error message")


RuntimeError: my error message
```

As an example of where this might be useful, let's return to the `fibonacci` function that we defined previously:

```
In [15]: def fibonacci(N):
             L = []
             a, b = 0, 1
             while len(L) < N:
                 a, b = b, a + b
                 L.append(a)
             return L
```

One potential problem here is that the input value could be negative. This will not currently cause any error in our function, but we might want to let the user know that a negative `N` is not supported. Errors stemming from invalid parameter values, by convention, lead to a `ValueError` being raised:

```
In [16]:  def fibonacci(N):
              if N < 0:
                  raise ValueError("N must be non-negative")
              L = []
              a, b = 0, 1
              while len(L) < N:
                  a, b = b, a + b
                  L.append(a)
              return L

In [17]:  fibonacci(10)

Out [17]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

In [18]:  fibonacci(-10)

---------------------------------------------------------

RuntimeError              Traceback (most recent call last)

<ipython-input-20-3d291499cfa7> in <module>()
----> 1 fibonacci(-10)


<ipython-input-18-01d0cf168d63> in fibonacci(N)
      1 def fibonacci(N):
      2     if N < 0:
----> 3         raise ValueError("N must be non-negative")
      4     L = []
      5     a, b = 0, 1


ValueError: N must be non-negative
```

Now the user knows exactly why the input is invalid, and could even use a `try…except` block to handle it!

```
In [19]: N = -10
         try:
             print("trying this...")
             print(fibonacci(N))
         except ValueError:
             print("Bad value: need to do something else")

         trying this...
         Bad value: need to do something else
```

Diving Deeper into Exceptions

Briefly, I want to mention here some other concepts you might run into. I'll not go into detail on these concepts and how and why to use them, but instead simply show you the syntax so you can explore more on your own.

Accessing the error message

Sometimes in a `try…except` statement, you would like to be able to work with the error message itself. This can be done with the `as` keyword:

```
In [20]: try:
             x = 1 / 0
         except ZeroDivisionError as err:
             print("Error class is:  ", type(err))
             print("Error message is:", err)

         Error class is:   <class 'ZeroDivisionError'>
         Error message is: division by zero
```

With this pattern, you can further customize the exception handling of your function.

Defining custom exceptions

In addition to built-in exceptions, it is possible to define custom exceptions through *class inheritance*. For instance, if you want a special kind of `ValueError`, you can do this:

```
In [21]: class MySpecialError(ValueError):
             pass

         raise MySpecialError("here's the message")

---------------------------------------------------------

MySpecialError              Traceback (most recent call last)

<ipython-input-23-92c36e04a9d0> in <module>()
      2     pass
      3
----> 4 raise MySpecialError("here's the message")


MySpecialError: here's the message
```

This would allow you to use a `try…except` block that only catches this type of error:

```
In [22]:
try:
    print("do something")
    raise MySpecialError("[informative error message here]")
except MySpecialError:
    print("do something else")
```

```
do something
do something else
```

You might find this useful as you develop more customized code.

try…except…else…finally

In addition to `try` and `except`, you can use the `else` and `finally` keywords to further tune your code's handling of exceptions. The basic structure is this:

```
In [23]: try:
             print("try something here")
         except:
             print("this happens only if it fails")
         else:
             print("this happens only if it succeeds")
         finally:
             print("this happens no matter what")
```

```
try something here
this happens only if it succeeds
this happens no matter what
```

The utility of `else` here is clear, but what's the point of `finally`? Well, the `finally` clause really is executed *no matter what*: I usually see it used to do some sort of cleanup after an operation completes.

Iterators

Often an important piece of data analysis is repeating a similar calculation, over and over, in an automated fashion. For example, you may have a table of names that you'd like to split into first and last, or perhaps of dates that you'd like to convert to some standard format. One of Python's answers to this is the *iterator* syntax. We've seen this already with the `range` iterator:

```
In [1]: for i in range(10):
            print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

Here we're going to dig a bit deeper. It turns out that in Python 3, `range` is not a list, but is something called an *iterator*, and learning how it works is key to understanding a wide class of very useful Python functionality.

Iterating over lists

Iterators are perhaps most easily understood in the concrete case of iterating through a list. Consider the following:

```
In [2]: for value in [2, 4, 6, 8, 10]:
            # do some operation
            print(value + 1, end=' ')

3 5 7 9 11
```

The familiar `for x in y` syntax allows us to repeat some operation for each value in the list. The fact that the syntax of the code is so close to its English description (*for [each] value in [the] list*) is just one of the syntactic choices that makes Python such an intuitive language to learn and use.

But the face-value behavior is not what's *really* happening. When you write something like `for val in L`, the Python interpreter checks whether it has an *iterator* interface, which you can check yourself with the built-in `iter` function:

```
In [3]:  iter([2, 4, 6, 8, 10])

Out [3]: <list_iterator at 0x104722400>
```

It is this iterator object that provides the functionality required by the `for` loop. The `iter` object is a container that gives you access to the next object for as long as it's valid, which can be seen with the built-in function `next`:

```
In [4]: I = iter([2, 4, 6, 8, 10])

In [5]: print(next(I))

2

In [6]: print(next(I))

4

In [7]: print(next(I))

6
```

What is the purpose of this level of indirection? Well, it turns out this is incredibly useful, because it allows Python to treat things as lists that are *not actually lists*.

range(): A List Is Not Always a List

Perhaps the most common example of this indirect iteration is the `range()` function in Python 3 (named `xrange()` in Python 2), which returns not a list, but a special `range()` object:

```
In [8]:  range(10)

Out [8]: range(0, 10)
```

`range`, like a list, exposes an iterator:

```
In [9]:  iter(range(10))

Out [9]: <range_iterator at 0x1045a1810>
```

So Python knows to treat it *as if* it's a list:

```
In [10]: for i in range(10):
             print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

The benefit of the iterator indirection is that *the full list is never explicitly created!* We can see this by doing a range calculation that would overwhelm our system memory if we actually instantiated it (note that in Python 2, `range` creates a list, so running the following will not lead to good things!):

```
In [11]: N = 10 ** 12
         for i in range(N):
             if i >= 10: break
             print(i, end=', ')

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

If `range` were to actually create that list of one trillion values, it would occupy tens of terabytes of machine memory: a waste, given the fact that we're ignoring all but the first 10 values!

In fact, there's no reason that iterators ever have to end at all! Python's `itertools` library contains a `count` function that acts as an infinite range:

```
In [12]: from itertools import count

         for i in count():
             if i >= 10:
                 break
             print(i, end=', ')
```

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

Had we not thrown in a loop break here, it would go on happily counting until the process is manually interrupted or killed (using, for example, `ctrl-C`).

Useful Iterators

This iterator syntax is used nearly universally in Python built-in types as well as the more data science–specific object we'll explore in later sections. Here we'll cover some of the more useful iterators in the Python language.

enumerate

Often you need to iterate not only the values in an array, but also keep track of the index. You might be tempted to do things this way:

```
In [13]: L = [2, 4, 6, 8, 10]
         for i in range(len(L)):
             print(i, L[i])
```

```
0 2
1 4
2 6
3 8
4 10
```

Although this does work, Python provides a cleaner syntax using the `enumerate` iterator:

```
In [14]: for i, val in enumerate(L):
             print(i, val)
```

```
0 2
1 4
2 6
3 8
4 10
```

This is the more "Pythonic" way to enumerate the indices and values in a list.

zip

Other times, you may have multiple lists that you want to iterate over simultaneously. You could certainly iterate over the index as in the non-Pythonic example we looked at previously, but it is better to use the `zip` iterator, which zips together iterables:

```
In [15]: L = [2, 4, 6, 8, 10]
         R = [3, 6, 9, 12, 15]
         for lval, rval in zip(L, R):
             print(lval, rval)
```

```
2 3
4 6
6 9
8 12
10 15
```

Any number of iterables can be zipped together, and if they are different lengths, the shortest will determine the length of the `zip`.

map and filter

The `map` iterator takes a function and applies it to the values in an iterator:

```
In [16]: # find the first 10 square numbers
         square = lambda x: x ** 2
         for val in map(square, range(10)):
             print(val, end=' ')
```

```
0 1 4 9 16 25 36 49 64 81
```

The `filter` iterator looks similar, except it only passes through values for which the filter function evaluates to True:

```
In [17]: # find values up to 10 for which x % 2 is zero
         is_even = lambda x: x % 2 == 0
         for val in filter(is_even, range(10)):
             print(val, end=' ')
```

```
0 2 4 6 8
```

The `map` and `filter` functions, along with the `reduce` function (which lives in Python's `functools` module) are fundamental components of the *functional programming* style, which, while not a dominant programming style in the Python world, has its outspoken proponents (see, for example, the pytoolz library).

Iterators as function arguments

We saw in "*args and **kwargs: Flexible Arguments" on page 43 that `*args` and `**kwargs` can be used to pass sequences and dictionaries to functions. It turns out that the `*args` syntax works not just with sequences, but with any iterator:

```
In [18]: print(*range(10))

0 1 2 3 4 5 6 7 8 9
```

So, for example, we can get tricky and compress the `map` example from before into the following:

```
In [19]: print(*map(lambda x: x ** 2, range(10)))

0 1 4 9 16 25 36 49 64 81
```

Using this trick lets us answer the age-old question that comes up in Python learners' forums: why is there no `unzip()` function that does the opposite of `zip()`? If you lock yourself in a dark closet and think about it for a while, you might realize that the opposite of `zip()` is… `zip()`! The key is that `zip()` can zip together any number of iterators or sequences. Observe:

```
In [20]: L1 = (1, 2, 3, 4)
         L2 = ('a', 'b', 'c', 'd')

In [21]: z = zip(L1, L2)
         print(*z)

(1, 'a') (2, 'b') (3, 'c') (4, 'd')

In [22]: z = zip(L1, L2)
         new_L1, new_L2 = zip(*z)
         print(new_L1, new_L2)

(1, 2, 3, 4) ('a', 'b', 'c', 'd')
```

Ponder this for a while. If you understand why it works, you'll have come a long way in understanding Python iterators!

Specialized Iterators: itertools

We briefly looked at the infinite `range` iterator, `itertools.count`, earlier. The `itertools` module contains a whole host of useful iterators; it's well worth your while to explore the module to see what's available. As an example, consider the `itertools.permutations` function, which iterates over all permutations of a sequence:

```
In [23]: from itertools import permutations
         p = permutations(range(3))
         print(*p)
```

```
(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1, 0)
```

Similarly, the `itertools.combinations` function iterates over all unique combinations of `N` values within a list:

```
In [24]: from itertools import combinations
         c = combinations(range(4), 2)
         print(*c)
```

```
(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
```

Somewhat related is the `product` iterator, which iterates over all sets of pairs between two or more iterables:

```
In [25]: from itertools import product
         p = product('ab', range(3))
         print(*p)
```

```
('a', 0) ('a', 1) ('a', 2) ('b', 0) ('b', 1) ('b', 2)
```

Many more useful iterators exist in `itertools`: the full list can be found, along with some examples, in Python's online documentation.

List Comprehensions

If you read enough Python code, you'll eventually come across the terse and efficient construction known as a *list comprehension*. This is one feature of Python I expect you will fall in love with if you've not used it before; it looks something like this:

```
In [1]:  [i for i in range(20) if i % 3 > 0]
```

```
Out [1]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

The result of this is a list of numbers that excludes multiples of 3. While this example may seem a bit confusing at first, as familiarity with Python grows, reading and writing list comprehensions will become second nature.

Basic List Comprehensions

List comprehensions are simply a way to compress a list-building `for` loop into a single short, readable line. For example, here is a loop that constructs a list of the first 12 square integers:

```
In [2]:  L = []
         for n in range(12):
             L.append(n ** 2)
         L

Out [2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

The list comprehension equivalent of this is the following:

```
In [3]:  [n ** 2 for n in range(12)]

Out [3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

As with many Python statements, you can almost read off the meaning of this statement in plain English: "construct a list consisting of the square of `n` for each `n` up to 12".

This basic syntax, then, is `[`*`expr`* `for` *`var`* `in` *`iterable`*`]`, where *`expr`* is any valid expression, *`var`* is a variable name, and *`iterable`* is any iterable Python object.

Multiple Iteration

Sometimes you want to build a list not just from one value, but from two. To do this, simply add another `for` expression in the comprehension:

```
In [4]:  [(i, j) for i in range(2) for j in range(3)]

Out [4]: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Notice that the second `for` expression acts as the interior index, varying the fastest in the resulting list. This type of construction can be extended to three, four, or more iterators within the comprehension, though at some point code readability will suffer!

Conditionals on the Iterator

You can further control the iteration by adding a conditional to the end of the expression. In the first example of the section, we iterated over all numbers from 1 to 20, but left out multiples of 3. Look at this again, and notice the construction:

```
In [5]:  [val for val in range(20) if val % 3 > 0]

Out [5]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

The expression `(i % 3 > 0)` evaluates to `True` unless `val` is divisible by 3. Again, the English language meaning can be immediately read off: "Construct a list of values for each value up to 20, but only if the

value is not divisible by 3". Once you are comfortable with it, this is much easier to write—and to understand at a glance—than the equivalent loop syntax:

```
In [6]:  L = []
         for val in range(20):
             if val % 3:
                 L.append(val)
         L

Out [6]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Conditionals on the Value

If you've programmed in C, you might be familiar with the single-line conditional enabled by the `?` operator:

```
int absval = (val < 0) ? -val : val
```

Python has something very similar to this, which is most often used within list comprehensions, `lambda` functions, and other places where a simple expression is desired:

```
In [7]:  val = -10
         val if val >= 0 else -val

Out [7]: 10
```

We see that this simply duplicates the functionality of the built-in `abs()` function, but the construction lets you do some really interesting things within list comprehensions. This is getting pretty complicated now, but you could do something like this:

```
In [8]:  [val if val % 2 else -val
          for val in range(20) if val % 3]

Out [8]: [1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

Note the line break within the list comprehension before the `for` expression: this is valid in Python, and is often a nice way to break-up long list comprehensions for greater readability. Look this over: what we're doing is constructing a list, leaving out multiples of 3, and negating all multiples of 2.

Once you understand the dynamics of list comprehensions, it's straightforward to move on to other types of comprehensions. The syntax is largely the same; the only difference is the type of bracket you use.

For example, with curly braces you can create a `set` with a *set comprehension*:

```
In [9]:  {n**2 for n in range(12)}

Out [9]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121}
```

Recall that a `set` is a collection that contains no duplicates. The set comprehension respects this rule, and eliminates any duplicate entries:

```
In [10]:  {a % 3 for a in range(1000)}

Out [10]: {0, 1, 2}
```

With a slight tweak, you can add a colon (`:`) to create a *dict comprehension*:

```
In [11]:  {n:n**2 for n in range(6)}

Out [11]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Finally, if you use parentheses rather than square brackets, you get what's called a *generator expression*:

```
In [12]:  (n**2 for n in range(12))

Out [12]: <generator object <genexpr> at 0x1027a5a50>
```

A generator expression is essentially a list comprehension in which elements are generated as needed rather than all at once, and the simplicity here belies the power of this language feature: we'll explore this more next.

Generators

Here we'll take a deeper dive into Python generators, including *generator expressions* and *generator functions*.

Generator Expressions

The difference between list comprehensions and generator expressions is sometimes confusing; here we'll quickly outline the differences between them.

List comprehensions use square brackets, while generator expressions use parentheses

This is a representative list comprehension:

```
In [1]:  [n ** 2 for n in range(12)]
```

```
Out [1]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

While this is a representative generator expression:

```
In [2]:  (n ** 2 for n in range(12))

Out [2]: <generator object <genexpr> at 0x104a60518>
```

Notice that printing the generator expression does not print the contents; one way to print the contents of a generator expression is to pass it to the `list` constructor:

```
In [3]:  G = (n ** 2 for n in range(12))
         list(G)

Out [3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

A list is a collection of values, while a generator is a recipe for producing values

When you create a list, you are actually building a collection of values, and there is some memory cost associated with that. When you create a generator, you are not building a collection of values, but a recipe for producing those values. Both expose the same iterator interface, as we can see here:

```
In [4]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121

In [5]: G = (n ** 2 for n in range(12))
        for val in G:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121
```

The difference is that a generator expression does not actually compute the values until they are needed. This not only leads to memory efficiency, but to computational efficiency as well! This also means that while the size of a list is limited by available memory, the size of a generator expression is unlimited!

An example of an infinite generator expression can be created using the `count` iterator defined in `itertools`:

```
In [6]:  from itertools import count
         count()

Out [6]: count(0)
```

```
In [7]:  for i in count():
             print(i, end=' ')
             if i >= 10: break

0 1 2 3 4 5 6 7 8 9 10
```

The `count` iterator will go on happily counting forever until you tell it to stop; this makes it convenient to create generators that will also go on forever:

```
In [8]:
factors = [2, 3, 5, 7]
G = (i for i in count() if all(i % n > 0 for n in factors))
for val in G:
    print(val, end=' ')
    if val > 40: break

1 11 13 17 19 23 29 31 37 41
```

You might see what we're getting at here: if we were to expand the list of factors appropriately, what we would have the beginnings of is a prime number generator, using the Sieve of Eratosthenes algorithm. We'll explore this more momentarily.

A list can be iterated multiple times; a generator expression is single use

This is one of those potential gotchas of generator expressions. With a list, we can straightforwardly do this:

```
In [9]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')
        print()

        for val in L:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

A generator expression, on the other hand, is used up after one iteration:

```
In [10]:  G = (n ** 2 for n in range(12))
          list(G)

Out [10]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]

In [11]:  list(G)

Out [11]: []
```

This can be very useful because it means iteration can be stopped and started:

```
In [12]: G = (n**2 for n in range(12))
         for n in G:
             print(n, end=' ')
             if n > 30: break

         print("\ndoing something in between")

         for n in G:
             print(n, end=' ')
```

```
0 1 4 9 16 25 36
doing something in between
49 64 81 100 121
```

One place I've found this useful is when working with collections of data files on disk; it means that you can quite easily analyze them in batches, letting the generator keep track of which ones you have yet to see.

Generator Functions: Using yield

We saw in the previous section that list comprehensions are best used to create relatively simple lists, while using a normal `for` loop can be better in more complicated situations. The same is true of generator expressions: we can make more complicated generators using *generator functions*, which make use of the `yield` statement.

Here we have two ways of constructing the same list:

```
In [13]: L1 = [n ** 2 for n in range(12)]

         L2 = []
         for n in range(12):
             L2.append(n ** 2)

         print(L1)
         print(L2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Similarly, here we have two ways of constructing equivalent generators:

```
In [14]: G1 = (n ** 2 for n in range(12))

         def gen():
             for n in range(12):
```

```
            yield n ** 2

        G2 = gen()
        print(*G1)
        print(*G2)
```

```
0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

A generator function is a function that, rather than using `return` to return a value once, uses `yield` to yield a (potentially infinite) sequence of values. Just as in generator expressions, the state of the generator is preserved between partial iterations, but if we want a fresh copy of the generator we can simply call the function again.

Example: Prime Number Generator

Here I'll show my favorite example of a generator function: a function to generate an unbounded series of prime numbers. A classic algorithm for this is the Sieve of Eratosthenes, which works something like this:

```
In [15]: # Generate a list of candidates
         L = [n for n in range(2, 40)]
         print(L)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, \
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,  \
30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
```

```
In [16]: # Remove all multiples of the first value
         L = [n for n in L if n == L[0] or n % L[0] > 0]
         print(L)
```

```
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, \
29, 31, 33, 35, 37, 39]
```

```
In [17]: # Remove all multiples of the second value
         L = [n for n in L if n == L[1] or n % L[1] > 0]
         print(L)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37]
```

```
In [18]: # Remove all multiples of the third value
         L = [n for n in L if n == L[2] or n % L[2] > 0]
         print(L)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

If we repeat this procedure enough times on a large enough list, we can generate as many primes as we wish.

Let's encapsulate this logic in a generator function:

```
In [19]: def gen_primes(N):
             """Generate primes up to N"""
             primes = set()
             for n in range(2, N):
                 if all(n % p > 0 for p in primes):
                     primes.add(n)
                     yield n

         print(*gen_primes(70))
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
```

That's all there is to it! While this is certainly not the most computationally efficient implementation of the Sieve of Eratosthenes, it illustrates how convenient the generator function syntax can be for building more complicated sequences.

Modules and Packages

One feature of Python that makes it useful for a wide range of tasks is the fact that it comes "batteries included"—that is, the Python standard library contains useful tools for a wide range of tasks. On top of this, there is a broad ecosystem of third-party tools and packages that offer more specialized functionality. Here we'll take a look at importing standard library modules, tools for installing third-party modules, and a description of how you can make your own modules.

Loading Modules: the import Statement

For loading built-in and third-party modules, Python provides the `import` statement. There are a few ways to use the statement, which we will mention briefly here, from most recommended to least recommended.

Explicit module import

Explicit import of a module preserves the module's content in a namespace. The namespace is then used to refer to its contents with a `.` between them. For example, here we'll import the built-in `math` module and compute the sine of pi:

```
In [1]:  import math
         math.cos(math.pi)
```

```
Out [1]: -1.0
```

Explicit module import by alias

For longer module names, it's not convenient to use the full module name each time you access some element. For this reason, we'll commonly use the `import ... as ...` pattern to create a shorter alias for the namespace. For example, the NumPy (Numerical Python) package, a popular third-party package useful for data science, is by convention imported under the alias `np`:

```
In [2]: import numpy as np
        np.cos(np.pi)

Out [2]: -1.0
```

Explicit import of module contents

Sometimes rather than importing the module namespace, you would just like to import a few particular items from the module. This can be done with the `from ... import ...` pattern. For example, we can import just the `cos` function and the `pi` constant from the `math` module:

```
In [3]: from math import cos, pi
        cos(pi)

Out [3]: -1.0
```

Implicit import of module contents

Finally, it is sometimes useful to import the entirety of the module contents into the local namespace. This can be done with the `from ... import *` pattern:

```
In [4]: from math import *
        sin(pi) ** 2 + cos(pi) ** 2

Out [4]: 1.0
```

This pattern should be used sparingly, if at all. The problem is that such imports can sometimes overwrite function names that you do not intend to overwrite, and the implicitness of the statement makes it difficult to determine what has changed.

For example, Python has a built-in `sum` function that can be used for various operations:

```
In [5]: help(sum)

        Help on built-in function sum in module builtins:

        sum(...)
```

```
sum(iterable[, start]) -> value

Return the sum of an iterable of numbers
(NOT strings) plus the value of parameter
'start' (which defaults to 0).
When the iterable is empty, return start.
```

We can use this to compute the sum of a sequence, starting with a certain value (here, we'll start with `-1`):

```
In [6]:  sum(range(5), -1)

Out [6]: 9
```

Now observe what happens if we make the *exact same function call* after importing `*` from `numpy`:

```
In [7]:  from numpy import *

In [8]:  sum(range(5), -1)

Out [8]: 10
```

The result is off by one! The reason for this is that the `import *` statement *replaces* the built-in `sum` function with the `numpy.sum` function, which has a different call signature: in the former, we're summing `range(5)` starting at `-1`; in the latter, we're summing `range(5)` along the last axis (indicated by `-1`). This is the type of situation that may arise if care is not taken when using `import *`—for this reason, it is best to avoid this unless you know exactly what you are doing.

Importing from Python's Standard Library

Python's standard library contains many useful built-in modules, which you can read about fully in Python's documentation. Any of these can be imported with the `import` statement, and then explored using the help function discussed in the previous section. Here is an extremely incomplete list of some of the modules you might wish to explore and learn about:

`os` and `sys`	Tools for interfacing with the operating system, including navigating file directory structures and executing shell commands
`math` and `cmath`	Mathematical functions and operations on real and complex numbers
`itertools`	Tools for constructing and interacting with iterators and generators
`functools`	Tools that assist with functional programming
`random`	Tools for generating pseudorandom numbers

`pickle`	Tools for object persistence: saving objects to and loading objects from disk
`json` and `csv`	Tools for reading JSON-formatted and CSV-formatted files
`urllib`	Tools for doing HTTP and other web requests

You can find information on these, and many more, in the Python standard library documentation: *https://docs.python.org/3/library/*.

Importing from Third-Party Modules

One of the things that makes Python useful, especially within the world of data science, is its ecosystem of third-party modules. These can be imported just as the built-in modules, but first the modules must be installed on your system. The standard registry for such modules is the Python Package Index (*PyPI* for short), found on the Web at *http://pypi.python.org/*. For convenience, Python comes with a program called `pip` (a recursive acronym meaning "pip installs packages"), which will automatically fetch packages released and listed on PyPI (if you use Python version 2, `pip` must be installed separately). For example, if you'd like to install the `supersmoother` package that I wrote, all that is required is to type the following at the command line:

```
$ pip install supersmoother
```

The source code for the package will be automatically downloaded from the PyPI repository, and the package installed in the standard Python path (assuming you have permission to do so on the computer you're using).

For more information about PyPI and the `pip` installer, refer to the documentation at *http://pypi.python.org/*.

String Manipulation and Regular Expressions

One place where the Python language really shines is in the manipulation of strings. This section will cover some of Python's built-in string methods and formatting operations, before moving on to a quick guide to the extremely useful subject of *regular expressions*. Such string manipulation pattens come up often in the context of data science work, and is one big perk of Python in this context.

Strings in Python can be defined using either single or double quotations (they are functionally equivalent):

```
In [1]:  x = 'a string'
         y = "a string"
         x == y

Out [1]: True
```

In addition, it is possible to define multiline strings using a triple-quote syntax:

```
In [2]: multiline = """
        one
        two
        three
        """
```

With this, let's take a quick tour of some of Python's string manipulation tools.

Simple String Manipulation in Python

For basic manipulation of strings, Python's built-in string methods can be extremely convenient. If you have a background working in C or another low-level language, you will likely find the simplicity of Python's methods extremely refreshing. We introduced Python's string type and a few of these methods earlier; here we'll dive a bit deeper.

Formatting strings: Adjusting case

Python makes it quite easy to adjust the case of a string. Here we'll look at the `upper()`, `lower()`, `capitalize()`, `title()`, and `swapcase()` methods, using the following messy string as an example:

```
In [3]: fox = "tHe qUICk bROWn fOx."
```

To convert the entire string into uppercase or lowercase, you can use the `upper()` or `lower()` methods respectively:

```
In [4]:  fox.upper()

Out [4]: 'THE QUICK BROWN FOX.'

In [5]:  fox.lower()

Out [5]: 'the quick brown fox.'
```

A common formatting need is to capitalize just the first letter of each word, or perhaps the first letter of each sentence. This can be done with the `title()` and `capitalize()` methods:

```
In [6]:  fox.title()
```

```
Out [6]: 'The Quick Brown Fox.'

In [7]:  fox.capitalize()

Out [7]: 'The quick brown fox.'
```

The cases can be swapped using the `swapcase()` method:

```
In [8]:  fox.swapcase()

Out [8]: 'ThE QuicK BrowN FoX.'
```

Formatting strings: Adding and removing spaces

Another common need is to remove spaces (or other characters) from the beginning or end of the string. The basic method of removing characters is the `strip()` method, which strips whitespace from the beginning and end of the line:

```
In [9]:  line = '         this is the content         '
         line.strip()

Out [9]: 'this is the content'
```

To remove just space to the right or left, use `rstrip()` or `lstrip()`, respectively:

```
In [10]:  line.rstrip()

Out [10]: '         this is the content'

In [11]:  line.lstrip()

Out [11]: 'this is the content         '
```

To remove characters other than spaces, you can pass the desired character to the `strip()` method:

```
In [12]:  num = "000000000000435"
          num.strip('0')

Out [12]: '435'
```

The opposite of this operation, adding spaces or other characters, can be accomplished using the `center()`, `ljust()`, and `rjust()` methods.

For example, we can use the `center()` method to center a given string within a given number of spaces:

```
In [13]:  line = "this is the content"
          line.center(30)

Out [13]: '     this is the content      '
```

Similarly, `ljust()` and `rjust()` will left-justify or right-justify the string within spaces of a given length:

```
In [14]:  line.ljust(30)

Out [14]: 'this is the content           '

In [15]:  line.rjust(30)

Out [15]: '           this is the content'
```

All these methods additionally accept any character which will be used to fill the space. For example:

```
In [16]:  '435'.rjust(10, '0')

Out [16]: '0000000435'
```

Because zero-filling is such a common need, Python also provides `zfill()`, which is a special method to right-pad a string with zeros:

```
In [17]:  '435'.zfill(10)

Out [17]: '0000000435'
```

Finding and replacing substrings

If you want to find occurrences of a certain character in a string, the `find()`/`rfind()`, `index()`/`rindex()`, and `replace()` methods are the best built-in methods.

`find()` and `index()` are very similar, in that they search for the first occurrence of a character or substring within a string, and return the index of the substring:

```
In [18]:  line = 'the quick brown fox jumped over a lazy dog'
          line.find('fox')

Out [18]: 16

In [19]:  line.index('fox')

Out [19]: 16
```

The only difference between `find()` and `index()` is their behavior when the search string is not found; `find()` returns `-1`, while `index()` raises a `ValueError`:

```
In [20]:  line.find('bear')

Out [20]: -1

In [21]: line.index('bear')
```

```
------------------------------------------------------------

ValueError                Traceback (most recent call last)

<ipython-input-21-4cbe6ee9b0eb> in <module>()
----> 1 line.index('bear')


ValueError: substring not found
```

The related `rfind()` and `rindex()` work similarly, except they search for the first occurrence from the end rather than the beginning of the string:

```
In [22]:  line.rfind('a')

Out [22]: 35
```

For the special case of checking for a substring at the beginning or end of a string, Python provides the `startswith()` and `endswith()` methods:

```
In [23]:  line.endswith('dog')

Out [23]: True

In [24]:  line.startswith('fox')

Out [24]: False
```

To go one step further and replace a given substring with a new string, you can use the `replace()` method. Here, let's replace `'brown'` with `'red'`:

```
In [25]:  line.replace('brown', 'red')

Out [25]: 'the quick red fox jumped over a lazy dog'
```

The `replace()` function returns a new string, and will replace all occurrences of the input:

```
In [26]:  line.replace('o', '--')

Out [26]: 'the quick br--wn f--x jumped --ver a lazy d--g'
```

For a more flexible approach to this `replace()` functionality, see the discussion of regular expressions in "Flexible Pattern Matching with Regular Expressions" on page 76.

Splitting and partitioning strings

If you would like to find a substring *and then* split the string based on its location, the `partition()` and/or `split()` methods are what you're looking for. Both will return a sequence of substrings.

The `partition()` method returns a tuple with three elements: the substring before the first instance of the split-point, the split-point itself, and the substring after:

```
In [27]:  line.partition('fox')

Out [27]: ('the quick brown ', 'fox', ' jumped over a lazy dog')
```

The `rpartition()` method is similar, but searches from the right of the string.

The `split()` method is perhaps more useful; it finds *all* instances of the split-point and returns the substrings in between. The default is to split on any whitespace, returning a list of the individual words in a string:

```
In [28]:  line.split()

Out [28]: ['the', 'quick', 'brown', 'fox', 'jumped',\
           'over', 'a', 'lazy', 'dog']
```

A related method is `splitlines()`, which splits on newline characters. Let's do this with a haiku popularly attributed to the 17th-century poet Matsuo Bashō:

```
In [29]: haiku = """matsushima-ya
         aah matsushima-ya
         matsushima-ya"""

         haiku.splitlines()

['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']
```

Note that if you would like to undo a `split()`, you can use the `join()` method, which returns a string built from a split-point and an iterable:

```
In [30]:  '--'.join(['1', '2', '3'])

Out [30]: '1--2--3'
```

A common pattern is to use the special character `\n` (newline) to join together lines that have been previously split, and recover the input:

```
In [31]: print("\n".join(['matsushima-ya', 'aah matsushima-ya',
                         'matsushima-ya']))

matsushima-ya
aah matsushima-ya
matsushima-ya
```

Format Strings

In the preceding methods, we have learned how to extract values from strings, and to manipulate strings themselves into desired formats. Another use of string methods is to manipulate string *representations* of values of other types. Of course, string representations can always be found using the `str()` function; for example:

```
In [32]:  pi = 3.14159
          str(pi)

Out [32]: '3.14159'
```

For more complicated formats, you might be tempted to use string arithmetic as outlined in "Basic Python Semantics: Operators" on page 17:

```
In [33]:  "The value of pi is " + str(pi)

Out [33]: 'The value of pi is 3.14159'
```

A more flexible way to do this is to use *format strings*, which are strings with special markers (noted by curly braces) into which string-formatted values will be inserted. Here is a basic example:

```
In [34]:  "The value of pi is {}".format(pi)

Out [34]: 'The value of pi is 3.14159'
```

Inside the {} marker you can also include information on exactly *what* you would like to appear there. If you include a number, it will refer to the index of the argument to insert:

```
In [35]:
"""First letter: {0}. Last letter: {1}.""".format('A', 'Z')

Out [35]: 'First letter: A. Last letter: Z.'
```

If you include a string, it will refer to the key of any keyword argument:

```
In [36]:
"""First: {first}. Last: {last}.""".format(last='Z', first='A')

Out [36]: 'First: A. Last: Z.'
```

Finally, for numerical inputs, you can include format codes that control how the value is converted to a string. For example, to print a number as a floating point with three digits after the decimal point, you can use the following:

```
In [37]:  "pi = {0:.3f}".format(pi)

Out [37]: 'pi = 3.142'
```

As before, here the `0` refers to the index of the value to be inserted. The `:` marks that format codes will follow. The `.3f` encodes the desired precision: three digits beyond the decimal point, floating-point format.

This style of format specification is very flexible, and the examples here barely scratch the surface of the formatting options available. For more information on the syntax of these format strings, see the "Format Specification" section of Python's online documentation.

Flexible Pattern Matching with Regular Expressions

The methods of Python's `str` type give you a powerful set of tools for formatting, splitting, and manipulating string data. But even more powerful tools are available in Python's built-in *regular expression* module. Regular expressions are a huge topic; there are entire books written on the topic (including Jeffrey E.F. Friedl's *Mastering Regular Expressions*, 3rd Edition), so it will be hard to do justice within just a single subsection.

My goal here is to give you an idea of the types of problems that might be addressed using regular expressions, as well as a basic idea of how to use them in Python. I'll suggest some references for learning more in "Resources for Further Learning" on page 90.

Fundamentally, regular expressions are a means of *flexible pattern matching* in strings. If you frequently use the command line, you are probably familiar with this type of flexible matching with the `*` character, which acts as a wildcard. For example, we can list all the IPython notebooks (i.e., files with extension *.ipynb*) with "Python" in their filename by using the `*` wildcard to match any characters in between:

```
In [38]: !ls *Python*.ipynb

01-How-to-Run-Python-Code.ipynb 02-Basic-Python-Syntax.ipynb
```

Regular expressions generalize this "wildcard" idea to a wide range of flexible string-matching syntaxes. The Python interface to regular expressions is contained in the built-in `re` module; as a simple example, let's use it to duplicate the functionality of the string `split()` method:

```
In [39]:  import re
          regex = re.compile('\s+')
          regex.split(line)

Out [39]: ['the', 'quick', 'brown', 'fox', 'jumped', \
          'over', 'a', 'lazy', 'dog']
```

Here we've first *compiled* a regular expression, then used it to *split* a string. Just as Python's `split()` method returns a list of all substrings between whitespace, the regular expression `split()` method returns a list of all substrings between matches to the input pattern.

In this case, the input is `\s+`: `\s` is a special character that matches any whitespace (space, tab, newline, etc.), and the `+` is a character that indicates *one or more* of the entity preceding it. Thus, the regular expression matches any substring consisting of one or more spaces.

The `split()` method here is basically a convenience routine built upon this *pattern matching* behavior; more fundamental is the `match()` method, which will tell you whether the beginning of a string matches the pattern:

```
In [40]: for s in ["     ", "abc  ", "  abc"]:
             if regex.match(s):
                 print(repr(s), "matches")
             else:
                 print(repr(s), "does not match")

'     ' matches
'abc  ' does not match
'  abc' matches
```

Like `split()`, there are similar convenience routines to find the first match (like `str.index()` or `str.find()`) or to find and replace (like `str.replace()`). We'll again use the line from before:

```
In [41]: line = 'the quick brown fox jumped over a lazy dog'
```

With this, we can see that the `regex.search()` method operates a lot like `str.index()` or `str.find()`:

```
In [42]:  line.index('fox')

Out [42]: 16

In [43]:  regex = re.compile('fox')
          match = regex.search(line)
          match.start()

Out [43]: 16
```

Similarly, the `regex.sub()` method operates much like `str.replace()`:

```
In [44]:  line.replace('fox', 'BEAR')

Out [44]: 'the quick brown BEAR jumped over a lazy dog'

In [45]:  regex.sub('BEAR', line)

Out [45]: 'the quick brown BEAR jumped over a lazy dog'
```

With a bit of thought, other native string operations can also be cast as regular expressions.

A more sophisticated example

But, you might ask, why would you want to use the more complicated and verbose syntax of regular expressions rather than the more intuitive and simple string methods? The advantage is that regular expressions offer *far* more flexibility.

Here we'll consider a more complicated example: the common task of matching email addresses. I'll start by simply writing a (somewhat indecipherable) regular expression, and then walk through what is going on. Here it goes:

```
In [46]: email = re.compile('\w+@\w+\.[a-z]{3}')
```

Using this, if we're given a line from a document, we can quickly extract things that look like email addresses:

```
In [47]:  text = "To email Guido, try guido@python.org \
          or the older address guido@google.com."
          email.findall(text)

Out [47]: ['guido@python.org', 'guido@google.com']
```

(Note that these addresses are entirely made up; there are probably better ways to get in touch with Guido).

We can do further operations, like replacing these email addresses with another string, perhaps to hide addresses in the output:

```
In [48]:  email.sub('--@--.--', text)
```

```
Out [48]:
'To email Guido, try --@--.-- or the older address --@--.--.'
```

Finally, note that if you really want to match *any* email address, the preceding regular expression is far too simple. For example, it only allows addresses made of alphanumeric characters that end in one of several common domain suffixes. So, for example, the period used here means that we only find part of the address:

```
In [49]:  email.findall('barack.obama@whitehouse.gov')

Out [49]: ['obama@whitehouse.gov']
```

This goes to show how unforgiving regular expressions can be if you're not careful! If you search around online, you can find some suggestions for regular expressions that will match *all* valid emails, but beware: they are much more involved than the simple expression used here!

Basics of regular expression syntax

The syntax of regular expressions is much too large a topic for this short section. Still, a bit of familiarity can go a long way: I will walk through some of the basic constructs here, and then list some more complete resources from which you can learn more. My hope is that the following quick primer will enable you to use these resources effectively.

Simple strings are matched directly. If you build a regular expression on a simple string of characters or digits, it will match that exact string:

```
In [50]:  regex = re.compile('ion')
          regex.findall('Great Expectations')

Out [50]: ['ion']
```

Some characters have special meanings. While simple letters or numbers are direct matches, there are a handful of characters that have special meanings within regular expressions. They are:

```
. ^ $ * + ? { } [ ] \ | ( )
```

We will discuss the meaning of some of these momentarily. In the meantime, you should know that if you'd like to match any of these characters directly, you can *escape* them with a backslash:

```
In [51]:  regex = re.compile(r'\$')
          regex.findall("the cost is $20")
```

```
Out [51]: ['$']
```

The `r` preface in `r'\$'` indicates a *raw string*; in standard Python strings, the backslash is used to indicate special characters. For example, a tab is indicated by `\t`:

```
In [52]: print('a\tb\tc')

a   b   c
```

Such substitutions are not made in a raw string:

```
In [53]: print(r'a\tb\tc')

a\tb\tc
```

For this reason, whenever you use backslashes in a regular expression, it is good practice to use a raw string.

Special characters can match character groups. Just as the `\` character within regular expressions can escape special characters, turning them into normal characters, it can also be used to give normal characters special meaning. These special characters match specified groups of characters, and we've seen them before. In the email address regexp from before, we used the character `\w`, which is a special marker matching *any alphanumeric character*. Similarly, in the simple `split()` example, we also saw `\s`, a special marker indicating *any whitespace character*.

Putting these together, we can create a regular expression that will match *any two letters/digits with whitespace between them*:

```
In [54]:  regex = re.compile(r'\w\s\w')
          regex.findall('the fox is 9 years old')

Out [54]: ['e f', 'x i', 's 9', 's o']
```

This example begins to hint at the power and flexibility of regular expressions.

The following table lists a few of these characters that are commonly useful:

Character	Description
`\d`	Match any digit
`\D`	Match any non-digit
`\s`	Match any whitespace
`\S`	Match any non-whitespace

Character	Description
`\w`	Match any alphanumeric char
`\W`	Match any non-alphanumeric char

This is *not* a comprehensive list or description; for more details, see Python's regular expression syntax documentation.

Square brackets match custom character groups. If the built-in character groups aren't specific enough for you, you can use square brackets to specify any set of characters you're interested in. For example, the following will match any lowercase vowel:

```
In [55]:  regex = re.compile('[aeiou]')
          regex.split('consequential')

Out [55]: ['c', 'ns', 'q', '', 'nt', '', 'l']
```

Similarly, you can use a dash to specify a range: for example, `[a-z]` will match any lowercase letter, and `[1-3]` will match any of `1`, `2`, or `3`. For instance, you may need to extract from a document specific numerical codes that consist of a capital letter followed by a digit. You could do this as follows:

```
In [56]:  regex = re.compile('[A-Z][0-9]')
          regex.findall('1043879, G2, H6')

Out [56]: ['G2', 'H6']
```

Wildcards match repeated characters. If you would like to match a string with, say, three alphanumeric characters in a row, it is possible to write, for example, `\w\w\w`. Because this is such a common need, there is a specific syntax to match repetitions—curly braces with a number:

```
In [57]:  regex = re.compile(r'\w{3}')
          regex.findall('The quick brown fox')

Out [57]: ['The', 'qui', 'bro', 'fox']
```

There are also markers available to match any number of repetitions—for example, the `+` character will match *one or more* repetitions of what precedes it:

```
In [58]:  regex = re.compile(r'\w+')
          regex.findall('The quick brown fox')

Out [58]: ['The', 'quick', 'brown', 'fox']
```

The following is a table of the repetition markers available for use in regular expressions:

Character	Description	Example
`?`	Match zero or one repetitions of preceding	`ab?` matches `a` or `ab`
`*`	Match zero or more repetitions of preceding	`ab*` matches `a`, `ab`, `abb`, `abbb`…
`+`	match one or more repetitions of preceding	`ab+` matches `ab`, `abb`, `abbb`… but not `a`
`{n}`	Match `n` repetitions of preceding	`ab{2}` matches `abb`
`{m,n}`	Match between `m` and `n` repetitions of preceding	`ab{2,3}` matches `abb` or `abbb`

With these basics in mind, let's return to our email address matcher:

```
In [59]: email = re.compile(r'\w+@\w+\.[a-z]{3}')
```

We can now understand what this means: we want one or more alphanumeric characters (`\w+`) followed by the *at sign* (`@`), followed by one or more alphanumeric characters (`\w+`), followed by a period (`\.`—note the need for a backslash escape), followed by exactly three lowercase letters.

If we want to now modify this so that the Obama email address matches, we can do so using the square-bracket notation:

```
In [60]:  email2 = re.compile(r'[\w.]+@\w+\.[a-z]{3}')
          email2.findall('barack.obama@whitehouse.gov')

Out [60]: ['barack.obama@whitehouse.gov']
```

We have changed `\w+` to `[\w.]+`, so we will match any alphanumeric character *or* a period. With this more flexible expression, we can match a wider range of email addresses (though still not all—can you identify other shortcomings of this expression?).

Parentheses indicate *groups* to extract. For compound regular expressions like our email matcher, we often want to extract their components rather than the full match. This can be done using parentheses to *group* the results:

```
In [61]:  email3 = re.compile(r'([\w.]+)@(\w+)\.([a-z]{3})')

In [62]:  text = "To email Guido, try guido@python.org"\
                 "or the older address guido@google.com."
          email3.findall(text)
```

```
Out [62]:
[('guido', 'python', 'org'), ('guido', 'google', 'com')]
```

As we see, this grouping actually extracts a list of the subcomponents of the email address.

We can go a bit further and *name* the extracted components using the `(?P<name> )` syntax, in which case the groups can be extracted as a Python dictionary:

```
In [63]:
email4 = re.compile(r'(?P<user>[\w.]+)@(?P<domain>\w+)'\
                    '\.(?P<suffix>[a-z]{3})')
match = email4.match('guido@python.org')
match.groupdict()
```

```
Out [63]: {'domain': 'python', 'suffix': 'org', 'user': 'guido'}
```

Combining these ideas (as well as some of the powerful regexp syntax that we have not covered here) allows you to flexibly and quickly extract information from strings in Python.

Further Resources on Regular Expressions

The preceding discussion is just a quick (and far from complete) treatment of this large topic. If you'd like to learn more, I recommend the following resources:

Python's `re` package documentation
: I find that I promptly forget how to use regular expressions just about every time I use them. Now that I have the basics down, I've found this page to be an incredibly valuable resource to recall what each specific character or sequence means within a regular expression.

Python's official regular expression HOWTO
: A more narrative approach to regular expressions in Python.

Mastering Regular Expressions (O'Reilly, 2006)
: This is a 500+ page book on the subject. If you want a really complete treatment of this topic, this is the resource for you.

For some examples of string manipulation and regular expressions in action at a larger scale, see "Pandas: Labeled Column-Oriented Data" on page 86, where we look at applying these sorts of expressions across *tables* of string data within the Pandas package.

A Preview of Data Science Tools

If you would like to spring from here and go farther in using Python for scientific computing or data science, there are a few packages that will make your life much easier. This section will introduce and preview several of the more important ones, and give you an idea of the types of applications they are designed for. If you're using the Anaconda or Miniconda environment suggested at the beginning of this report, you can install the relevant packages with the following command:

```
$ conda install numpy scipy pandas matplotlib scikit-learn
```

Let's take a brief look at each of these in turn.

NumPy: Numerical Python

NumPy provides an efficient way to store and manipulate multidimensional dense arrays in Python. The important features of NumPy are:

- It provides an `ndarray` structure, which allows efficient storage and manipulation of vectors, matrices, and higher-dimensional datasets.
- It provides a readable and efficient syntax for operating on this data, from simple element-wise arithmetic to more complicated linear algebraic operations.

In the simplest case, NumPy arrays look a lot like Python lists. For example, here is an array containing the range of numbers 1 to 9 (compare this with Python's built-in `range()`):

```
In [1]:  import numpy as np
         x = np.arange(1, 10)
         x

Out [1]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy's arrays offer both efficient storage of data, as well as efficient element-wise operations on the data. For example, to square each element of the array, we can apply the `**` operator to the array directly:

```
In [2]:  x ** 2

Out [2]: array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Compare this with the much more verbose Python-style list comprehension for the same result:

```
In [3]:  [val ** 2 for val in range(1, 10)]

Out [3]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Unlike Python lists (which are limited to one dimension), NumPy arrays can be multidimensional. For example, here we will reshape our `x` array into a 3x3 array:

```
In [4]:  M = x.reshape((3, 3))
         M

Out [4]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

A two-dimensional array is one representation of a matrix, and NumPy knows how to efficiently do typical matrix operations. For example, you can compute the transpose using `.T`:

```
In [5]:  M.T

Out [5]: array([[1, 4, 7],
                [2, 5, 8],
                [3, 6, 9]])
```

or a matrix-vector product using `np.dot`:

```
In [6]:  np.dot(M, [5, 6, 7])

Out [6]: array([ 38,  92, 146])
```

and even more sophisticated operations like eigenvalue decomposition:

```
In [7]: np.linalg.eigvals(M)

Out [7]:
array([  1.61168440e+01,  -1.11684397e+00,  -1.30367773e-15])
```

Such linear algebraic manipulation underpins much of modern data analysis, particularly when it comes to the fields of machine learning and data mining.

For more information on NumPy, see "Resources for Further Learning" on page 90.

Pandas: Labeled Column-Oriented Data

Pandas is a much newer package than NumPy, and is in fact built on top of it. What Pandas provides is a labeled interface to multidimensional data, in the form of a DataFrame object that will feel very familiar to users of R and related languages. DataFrames in Pandas look something like this:

```
In [8]:
import pandas as pd
df = pd.DataFrame({'label': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'value': [1, 2, 3, 4, 5, 6]})
df
```

```
Out [8]:    label value
         0  A     1
         1  B     2
         2  C     3
         3  A     4
         4  B     5
         5  C     6
```

The Pandas interface allows you to do things like select columns by name:

```
In [9]:  df['label']
```

```
Out [9]: 0    A
         1    B
         2    C
         3    A
         4    B
         5    C
         Name: label, dtype: object
```

Apply string operations across string entries:

```
In [10]:  df['label'].str.lower()
```

```
Out [10]: 0    a
          1    b
          2    c
          3    a
          4    b
          5    c
          Name: label, dtype: object
```

Apply aggregates across numerical entries:

```
In [11]:  df['value'].sum()
```

```
Out [11]: 21
```

And, perhaps most importantly, do efficient database-style joins and groupings:

```
In [12]:  df.groupby('label').sum()

Out [12]:       value
          label
          A     5
          B     7
          C     9
```

Here in one line we have computed the sum of all objects sharing the same label, something that is much more verbose (and much less efficient) using tools provided in NumPy and core Python.

For more information on using Pandas, see the resources listed in "Resources for Further Learning" on page 90.

Matplotlib: MATLAB-style scientific visualization

Matplotlib is currently the most popular scientific visualization packages in Python. Even proponents admit that its interface is sometimes overly verbose, but it is a powerful library for creating a large range of plots.

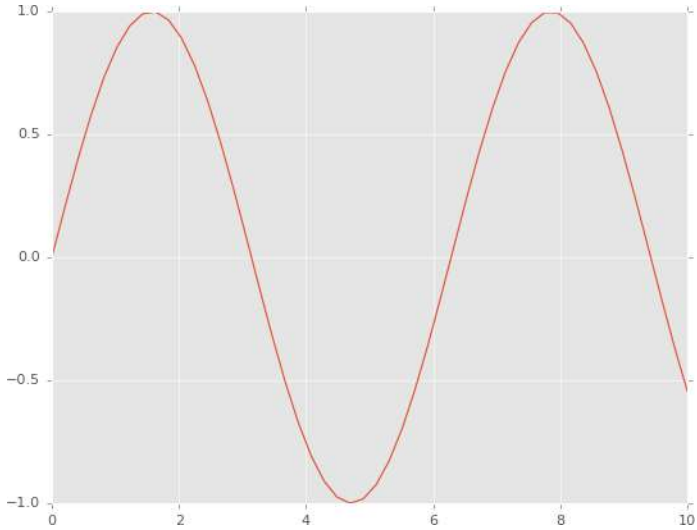
To use Matplotlib, we can start by enabling the notebook mode (for use in the Jupyter notebook) and then importing the package as `plt`:

```
In [13]: # run this if using Jupyter notebook
         %matplotlib notebook

In [14]:
import matplotlib.pyplot as plt
plt.style.use('ggplot')  # make graphs in the style of R's ggplot
```

Now let's create some data (as NumPy arrays, of course) and plot the results:

```
In [15]: x = np.linspace(0, 10)  # range of values from 0 to 10
         y = np.sin(x)           # sine of these values
         plt.plot(x, y);         # plot as a line
```



If you run this code live, you will see an interactive plot that lets you pan, zoom, and scroll to explore the data.

This is the simplest example of a Matplotlib plot; for ideas on the wide range of plot types available, see Matplotlib's online gallery as well as other references listed in "Resources for Further Learning" on page 90.

SciPy: Scientific Python

SciPy is a collection of scientific functionality that is built on NumPy. The package began as a set of Python wrappers to well-known Fortran libraries for numerical computing, and has grown from there. The package is arranged as a set of submodules, each implementing some class of numerical algorithms. Here is an incomplete sample of some of the more important ones for data science:

`scipy.fftpack`	Fast Fourier transforms
`scipy.integrate`	Numerical integration
`scipy.interpolate`	Numerical interpolation
`scipy.linalg`	Linear algebra routines
`scipy.optimize`	Numerical optimization of functions
`scipy.sparse`	Sparse matrix storage and linear algebra
`scipy.stats`	Statistical analysis routines

For example, let's take a look at interpolating a smooth curve between some data:

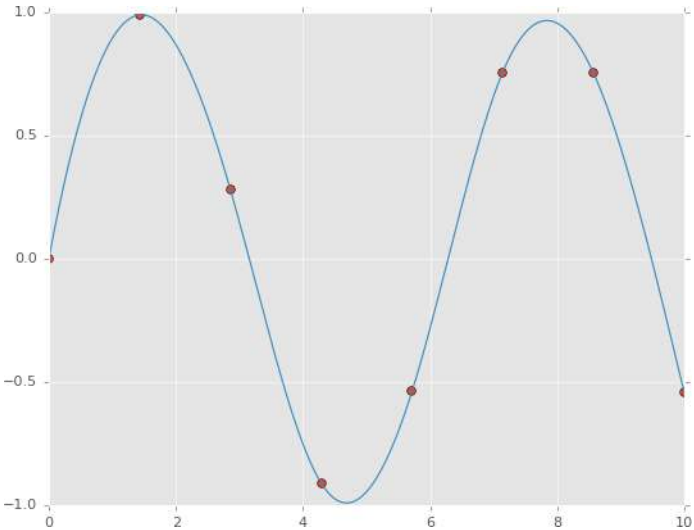
```
In [16]: from scipy import interpolate

         # choose eight points between 0 and 10
         x = np.linspace(0, 10, 8)
         y = np.sin(x)

         # create a cubic interpolation function
         func = interpolate.interp1d(x, y, kind='cubic')

         # interpolate on a grid of 1,000 points
         x_interp = np.linspace(0, 10, 1000)
         y_interp = func(x_interp)

         # plot the results
         plt.figure()  # new figure
         plt.plot(x, y, 'o')
         plt.plot(x_interp, y_interp);
```



What we see is a smooth interpolation between the points.

Other Data Science Packages

Built on top of these tools are a host of other data science packages, including general tools like Scikit-Learn for machine learning, Scikit-Image for image analysis, and StatsModels for statistical modeling, as well as more domain-specific packages like AstroPy for

astronomy and astrophysics, NiPy for neuro-imaging, and many, many more.

No matter what type of scientific, numerical, or statistical problem you are facing, it's likely there is a Python package out there that can help you solve it.

Resources for Further Learning

This concludes our whirlwind tour of the Python language. My hope is that if you read this far, you have an idea of the essential syntax, semantics, operations, and functionality offered by the Python language, as well as some idea of the range of tools and code constructs that you can explore further.

I have tried to cover the pieces and patterns in the Python language that will be most useful to a data scientist using Python, but this has by no means been a complete introduction. If you'd like to go deeper in understanding the Python language itself and how to use it effectively, here are a handful of resources I'd recommend:

Fluent Python by Luciano Ramalho
: This is an excellent O'Reilly book that explores best practices and idioms for Python, including getting the most out of the standard library.

Dive Into Python by Mark Pilgrim
: This is a free online book that provides a ground-up introduction to the Python language.

Learn Python the Hard Way by Zed Shaw
: This book follows a "learn by trying" approach, and deliberately emphasizes developing what may be the most useful skill a programmer can learn: Googling things you don't understand.

Python Essential Reference by David Beazley
: This 700-page monster is well written, and covers virtually everything there is to know about the Python language and its built-in libraries. For a more application-focused Python walkthrough, see Beazley's *Python Cookbook*.

To dig more into Python tools for data science and scientific computing, I recommend the following books:

The Python Data Science Handbook by yours truly
: This book starts precisely where this report leaves off, and provides a comprehensive guide to the essential tools in Python's data science stack, from data munging and manipulation to machine learning.

Effective Computation in Physics by Katie Huff and Anthony Scopatz
: This book is applicable to people far beyond the world of physics research. It is a step-by-step, ground-up introduction to scientific computing, including an excellent introduction to many of the tools mentioned in this report.

Python for Data Analysis by Wes McKinney, creator of the Pandas package
: This book covers the Pandas library in detail, as well as giving useful information on some of the other tools that enable it.

Finally, for an even broader look at what's out there, I recommend the following:

O'Reilly Python Resources
: O'Reilly features a number of excellent books on Python itself and specialized topics in the Python world.

PyCon, SciPy, and PyData
: The PyCon, SciPy, and PyData conferences draw thousands of attendees each year, and archive the bulk of their programs each year as free online videos. These have turned into an incredible set of resources for learning about Python itself, Python packages, and related topics. Search online for videos of both talks and tutorials: the former tend to be shorter, covering new packages or fresh looks at old ones. The tutorials tend to be several hours, covering the use of the tools mentioned here as well as others.

About the Author

Jake VanderPlas is a long-time user and developer of the Python scientific stack. He currently works as an interdisciplinary research director at the University of Washington, conducts his own astronomy research, and spends time advising and consulting with local scientists from a wide range of fields.