A Whirlwind Tour of Python for STAT/CS 287

James Bagrow, August 2021

This mini-text serves as introductory reading for my course, Data Science I, held at the University of Vermont. It has been lightly adapted and specialized from the original "A Whirlwind Tour of Python" by Jake VanderPlas, and he deserves most of the credit for this work.

A Whirlwind Tour of Python is a fast-paced introduction to essential components of the Python language for researchers and developers who are already familiar with programming in another language.

The material is particularly aimed at those who wish to use Python for data science and/or scientific programming, and in this capacity serves as an introduction to Jake VanderPlas' book, The Python Data Science Handbook.

- This material is also available online at bagrow.com/ds1.

Contents

License and Citation

1 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 a computational and data-focused scientist, I've found Python to be a near-perfect fit for the types of problems I face, whether it's extracting meaning from large social network 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 multi-dimensional 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. 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.

1.1 About the authors

This material was created by Jake VanderPlas. I (Jim Bagrow) have made minor revisions, deletions and insertions only, with the goal of specializing this text to University of Vermont's Data Science Course (STAT/CS 287).

Here is the licensing details from the original text:

> 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."

I would appreciate, but do not require, attribution alongside Jake VanderPlas if any derivations come from my specialization of the original text.

1.2 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.

1.2.1 Python 2 vs 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 was relatively slow at first, particularly in the scientific and web development communities. This was 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, In fact, many major projects are now deprecating Python 2, and so this course will use Python 3 syntax. Even though that is the case, the vast majority of code snippets here will also work with little if any modification in Python 2.

1.2.2 Installation with conda

Though there are various ways to install Python, the one I would **strongly suggest** — particularly if you wish to eventually use the data science tools mentioned above – is via the cross-platform Anaconda distribution:

- Anaconda gives you Python and the Python standard library, a command-line tool called `conda` which allows you to easily install many third-party packages and libraries, and additionally bundles a suite of other pre-installed third-party packages geared toward scientific computing.

To get started, download and install the Anaconda package – make sure to choose a version with Python 3. When finished, run the `HW01_required_packages.py` script to ensure all packages for the course are available.

- If that script terminates with an `ImportError` message, use conda to install the missing package.

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

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

2 How to Run and Explore 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*.

2.1 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 2015, 11:24:55)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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.

2.2 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 2015, 11:24:55)
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 details.

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.

- Notice that `In [2]` does not have a corresponding `Out [2]`. This is because IPython **automatically prints** to screen the last unassigned object created by a command. Here, `In [1]` and `In [3]` generate unassigned objects, but `In [2]` merely assigns the integer `5` to the variable `x`.

IPython makes available a wide array of useful features; for some suggestions on where to read more, see Resources for Further Learning.

2.3 Exploring Python interactively

IPython provides a number of useful functions that make learning Python easier.

2.3.1 Docstrings

Well written Python code contains *docstrings* (short for documentation strings). These strings offer help and tutorials for what the code does. IPython makes it very easy to access these strings during an interactive session, simply append `?` to the name of the object, function, method…:

```
In [1]: x = "foo"

In [2]: x.join?
```

```
Signature: x.join(iterable, /)
Docstring:
Concatenate any number of strings.

The string whose method is called is inserted in between each given string.
The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
Type:      builtin_function_or_method
```

But how do you even know that a string has a method called `join`?

2.3.2 Tab completion

IPython allows you to search the methods associated with an object using tab completion. For example, entering `x.` and then immediately hitting the tab key will show a menu that you can page through of all the methods and attributes associated with `x`.

If you type the beginning of a the method or attribute, then hit tab, the menu will display only the matching options. For example, typing `x.j` and then hitting tab will show everything associated with `x` that begins with j.

2.3.3 Current workspace

The command `%whos` in IPython will display a table showing all currently defined variables. This is an example of a "magic command". IPython defines many helpful magic commands. These are distinguished from regular Python code with a leading percent sign `%` and will only function within IPython sessions.

Another helpful magic command is `%paste` which will take the contents of your clipboard and run it within the IPython session. This is great for running snippets of Python code, and will automatically deal with indentation issues.

2.3.4 Timing and profiling code

IPython provides many tools for identifying slow portions of code, helpful for optimizing code in an intelligent manor.

2.4 Self-contained Python scripts

Running Python snippets line by line is useful in some cases, especially for iteratively learning long code, 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* which 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
```

Here the `$` represents your command line prompt.

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

3 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'll not 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:

[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 **illustrates several important aspects** of Python syntax. Let's walk through it and discuss some of the syntactical features of Python.

3.1 Comments Are Marked by

The script starts with a code comment:

```
# set the midpoint
```

Comments in Python are indicated by a pound or hash sign (#), and anything on the line following the pound sign is ignored by the interpreter. This means, for example, that you can have stand-alone 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 multi-line comments, such as the `/* ... */` syntax used in C and C-style languages, though multi-line strings are often used as a replacement for multi-line comments (more on this in String Manipulation).

3.2 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:

[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:

[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).

3.3 Semicolons 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.

3.4 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
```

It is the indentation level that defines what is in the code block; the block ends when the indentation does.

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

In the snippet on the left, `print(x)` is within 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 surprises progammers 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 initially find Python's use of whitespace **disagreeable**, I'd encourage you to give it an open-minded 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.

- Configuring your text editor to insert four literal spaces when you press tab is known as a **hard tab**. In contrast, if your text editor instead inserts a "tab" character (another whitespace character), that is a **soft tab**. Generally, hard tabs are preferred for python code because the width of a soft tab may look different in different views or on different computers. For example, it may appear as the same size as 4 spaces, but it may appear as 8 or 2 spaces. Different programmers may configure their soft tabs differently, making their code look different on your computer, and vice versa

3.5 Whitespace *Within* Lines Does Not Matter

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

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

Abusing this flexibility can lead to issues with code readibility – in fact, abusing white space is often one of the primary means of intentionally **obfuscating** code (which some people do for sport).

3.6 Parentheses Are for Grouping or Calling

There are two uses of parentheses in Python. First, they can be used in the typical way to group statements or mathematical operations:

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

[5]:
```
14
```

Second, 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. The function call is indicated by a pair of opening and closing parentheses, with the *arguments* to the function contained within:

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

```
first value: 1
```

[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:

[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.

3.7 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**.

4 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.

4.1 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 static "memory bucket" named `x`, and putting the value `4` into it. In Python, by contrast, variables are best thought of not as **containers for data** but as **pointers to data**. 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:

[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:

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

We've created two variables `x` and `y` **which 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:

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

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

[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: `x` and `y` both **point** to the same list!

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:

[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 designed 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:

[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.

4.2 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:

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

[7]:
```
int
```

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

[8]:
```
str
```

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

[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:

[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 returns the real and imaginary part of the value, if viewed as a complex number:

[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 opening and closing parentheses. For example, floating point numbers have a method called `is_integer` that checks whether the value is an integer:

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

[12]:
```
False
```

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

[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:

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

[14]:
```
builtin_function_or_method
```

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

5 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.

5.1 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	Integer 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:

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

[1]: 42.0

Floor division is true division with fractional parts truncated:

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

```
5.5
```

[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. This is a holdover from early C language standards.

5.2 Assignment Operations

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

[4]:
```
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:

[5]:
```
a + 2
```

[5]: 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:

[6]:
```
a += 2  # equivalent to a = a + 2
print(a)
```

```
26
```

There are **augmented assignment operators** corresponding to each of the binary operators listed earlier; For example: `a += b` is equivalent to `a = a + b`. These allow for more compact code for common operations

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.

5.3 Comparison Operations

Another type of operation which 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	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 other 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:

[7]:
```
# is 25 odd?
25 % 2 == 1
```

[7]:
```
True
```

[8]:
```
# is 66 odd?
66 % 2 == 1
```

[8]:
```
False
```

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

[9]:
```
# check if a is between 15 and 30
a = 25
15 < a < 30
```

[9]:
```
True
```

5.4 Boolean Operations

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

[10]:
```
x = 4
(x < 6) and (x > 2)
```

[10]:
```
True
```

[11]:
```
(x > 10) or (x % 2 == 0)
```

[11]:
```
True
```

[12]:
```
not (x < 6)
```

[12]:
```
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:

[13]:
```
# (x > 1) xor (x < 10)
(x > 1) != (x < 10)
```

[13]:
```
False
```

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

5.5 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`

5.5.1 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:

[14]:
```
a = [1, 2, 3]
b = [1, 2, 3]
```

[15]:
```
a == b
```

[15]:
```
True
```

[16]:
```
a is b
```

[16]:
```
False
```

[17]:
```
a is not b
```

[17]:
```
True
```

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

[18]:
```
a = [1, 2, 3]
b = a
a is b
```

[18]:
```
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 when a beginner is tempted to use "`is`" what they really mean is ==.

5.5.2 Membership operators

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

[19]:
```
1 in [1, 2, 3]
```

[19]:
```
True
```

[20]:
```
2 not in [1, 2, 3]
```

[20]:
```
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.

6 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 the following table:

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
`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.

6.1 Integers

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

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

[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:

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

[2]:
```
1606938044258990275541962092341162602522202993782792835301376
```

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

[3]:
```
5 / 2
```

[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 "double slash" operator:

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

[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.

6.2 Floating-Point Numbers

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

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

```
True
```

[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:

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

[7]:
```
1.0
```

6.2.1 Aside: 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:

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

[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:

[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
```

By printing so many decimal places, we see that these floats are actually **truncated**.

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:

$$1/3 = 0.333333333\cdots$$

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:

$$1/10 = 0.00011001100110011\cdots$$

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.

6.3 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:

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

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

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

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

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

Complex numbers have a variety of interesting attributes and methods, but they will be seldom used in this course.

6.4 Strings

Strings in Python are created with single or double quotes:

[12]:
```
message = "what do you like?"
response = 'spam'
```

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

[13]:
```
# length of string
len(response)
```

[13]:
```
4
```

[14]:
```
# Make upper-case. See also str.lower()
response.upper()
```

[14]:
```
'SPAM'
```

[15]:
```
# Capitalize. See also str.title()
message.capitalize()
```

[15]:
```
'What do you like?'
```

[16]:
```
# concatenation with +
message + response
```

[16]:
```
'what do you like?spam'
```

[17]:
```
# multiplication is multiple concatenation!!!
5 * response
```

[17]:
```
'spamspamspamspamspam'
```

[18]:
```
# Access individual characters (zero-based indexing)
print(message[0])
print(message[-1]) # get last character without knowing how long the str is
```

```
w
?
```

For more discussion of indexing in Python, see Lists.

6.5 None Type

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

[19]:
```
type(None)
```

[19]:
```
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:

[20]:
```
return_value = print('abc')
```

```
abc
```

[21]:
```
print(return_value)
```

```
None
```

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

6.6 Boolean Type

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

[22]:
```
result = (4 < 5)
print(result)
```

```
True
```

[23]:
```
type(result)
```

[23]:
```
bool
```

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

[24]:
```
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:

[25]:
```
bool(2014)
```

[25]:
```
True
```

[26]:
```
bool(0)
```

[26]:
```
False
```

[27]:
```
bool(3.1415)
```

[27]:
```
True
```

The Boolean conversion of `None` is always False:

[28]:
```
bool(None)
```

[28]:
```
False
```

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

[29]:
```
bool("")
```

[29]:
```
False
```

[30]:
```
bool("abc")
```

[30]:
```
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

[31]:
```
bool([1, 2, 3])
```

[31]:
```
True
```

[32]:
```
bool([])
```

[32]: False

This holds regardless of what is inside the sequence:

[33]:
```
list1 = [False]
list2 = [0]
list3 = [None]

print(bool(list1))
print(bool(list2))
print(bool(list3))

print(bool(list3[0]))
```

```
True
True
True
False
```

7 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 group together variables of other types in different ways. These compound types are:

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

7.1 Lists

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

[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:

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

[2]: 4

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

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

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

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

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

[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, Python's compound objects can contain objects of *any* type, or even a mix of types. For example:

[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.

7.1.1 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:

[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:

[8]:
```
L[0]
```

[8]: 2

[9]:
```
L[1]
```

[9]:
```
3
```

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

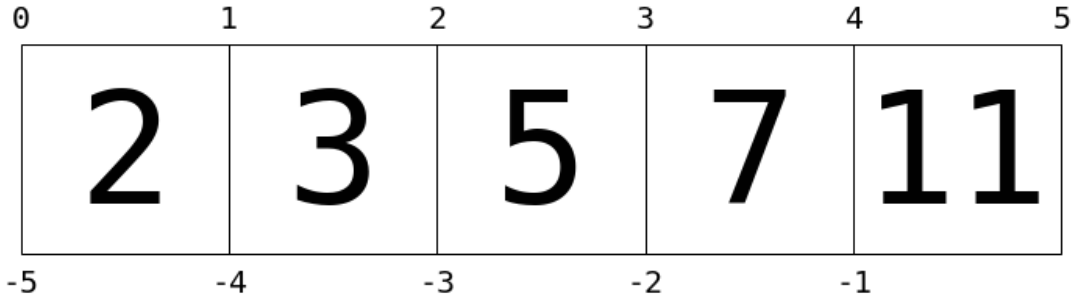
[10]:
```
L[-1]
```

[10]:
```
11
```

[11]:
```
L[-2]
```

[11]:
```
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 sub-lists. It uses a colon to indicate the start point (inclusive) and end point (non-inclusive) of the sub-array. For example, to get the first three elements of the list, we can write:

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

[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:

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

[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:

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

[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:

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

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

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

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

[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:

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

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

[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 Python 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.

7.2 Tuples

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

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

They can actually also be defined without any brackets at all:

[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:

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

[21]:
```
3
```

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

[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:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-87b0f225887f> in <module>
----> 1 t[1] = 4

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

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

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-24-ada7ed8a579e> 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:

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

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

These multiple return values can be individually assigned as follows:

[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 online Python documentation for a more complete list of these.

7.3 Dictionaries

Dictionaries, sometimes called hashes or associative arrays in other languages, are extremely flexible mappings of keys to values. Dictionaries are one of the **most powerful** and effective aspects of Python! Mastering Python includes mastering dictionaries.

They can be created via a comma-separated list of `key:value` pairs within curly braces:

[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 integer value but a valid key in the dictionary:

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

[28]:
```
2
```

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

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

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

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*).

Despite being unordered, **looping** over a dictionary is very possible and pragmatic, as long as you realize you can only expect the keys and values to be paired properly, but not ordered!

[30]:
```
for key in numbers:
    print(key, "-->", numbers[key])

print("---")

for key, value in numbers.items(): # .items is a handy method
    print(key, "maps to", value)
```

```
one --> 1
two --> 2
three --> 3
ninety --> 90
---
one maps to 1
two maps to 2
three maps to 3
ninety maps to 90
```

The Python documentation has a complete list of the methods available for dictionaries.

7.4 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:

[31]:
```
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:

[32]:
```
# union: items appearing in either
primes | odds       # with an operator
primes.union(odds) # equivalently with a method
```

[32]:
```
{1, 2, 3, 5, 7, 9}
```

[33]:
```
# intersection: items appearing in both
primes & odds              # with an operator
primes.intersection(odds) # equivalently with a method
```

[33]:
```
{3, 5, 7}
```

[34]:
```
# difference: items in primes but not in odds
primes - odds            # with an operator
primes.difference(odds) # equivalently with a method
```

[34]:
```
{2}
```

[35]:
```
# symmetric difference: items appearing in only one set
primes ^ odds                      # with an operator
primes.symmetric_difference(odds) # equivalently with a method
```

[35]:
```
{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.

8 String Manipulation

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. Formatting and manipulating strings is one of the most common tasks a data scientist performs.

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

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

[1]:
```
True
```

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

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

Since quotes are used to mark the beginning and ending of strings, if you wish to use those characters inside a string you need to "escape" them with a backslash (\). This tells Python not to interpret these quotes as the end of the string.

[5]:
```
print("Jon said, \"Hello World\".")
print('Andrea replied, "That\'s nice!"')
```

```
Jon said, "Hello World".
Andrea replied, "That's nice!"
```

While single- and double-quoted strings are functionally equivalent, what needs to be escaped depends on what quote character was used to define the string itself.

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

8.1 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

8.1.1 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:

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

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

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

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

[5]:
```
fox.lower()
```

[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:

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

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

[7]:
```
fox.capitalize()
```

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

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

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

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

8.1.2 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:

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

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

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

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

[10]:
```
'          this is the content'
```

[11]:
```
line.lstrip()
```

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

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

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

[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:

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

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

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

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

[14]:
```
'this is the content           '
```

[15]:
```
line.rjust(30)
```

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

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

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

[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:

[17]:
```
'435'.zfill(10)
```

[17]:
```
'0000000435'
```

8.1.3 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:

[18]:
```
line = 'the quick brown fox jumped over a lazy dog'
line.find('fox')
```

[18]:
```
16
```

[19]:
```
line.index('fox')
```

[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`:

[20]:
```
line.find('bear')
```

[20]:
```
-1
```

[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:

[22]:
```
line.rfind('a')
```

[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:

[23]:
```
line.endswith('dog')
```

[23]:
```
True
```

[24]:
```
line.startswith('fox')
```

[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'`:

[25]:
```
line.replace('brown', 'red')
```

[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:

[26]:
```
line.replace('o', '--')
```

[26]:
```
'the quick br--wn f--x jumped --ver a lazy d--g'
```

8.1.4 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:

[27]:
```
line.partition('fox')
```

[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:

[28]:
```
line.split()
```

[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:

[29]:
```
haiku = """matsushima-ya
aah matsushima-ya
matsushima-ya"""

haiku.splitlines()
```

[29]:
```
['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']
```

8.1.5 Joining strings

Note that if you would like to undo a `split()`, you can use the `join()` method, which returns a string built from a splitpoint and an iterable:

[30]:
```
'--'.join(['1', '2', '3'])
```

[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:

[31]:
```
print("\n".join(['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']))
```

```
matsushima-ya
aah matsushima-ya
matsushima-ya
```

8.2 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:

[32]:
```
pi = 3.14159
str(pi)
```

[32]:
```
'3.14159'
```

For more complicated formats, you might be tempted to use string arithmetic as outlined in Basic Python Semantics: Operators:

[33]:
```
"The value of pi is " + str(pi)
```

[33]:
```
'The value of pi is 3.14159'
```

However, 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:

[34]:
```
"The value of pi is {}".format(pi)
```

[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:

[35]:
```
"""First letter: {0}. Last letter: {1}.""".format('A', 'Z')
```

[35]:
```
'First letter: A. Last letter: Z.'
```

If you include a string, it will refer to the key of any keyword argument:

[36]:
```
"""First letter: {first}. Last letter: {last}.""".format(last='Z', first='A')
```

[36]:
```
'First letter: A. Last letter: Z.'
```

Finally, for numerical inputs, you can include format codes which 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:

[37]:
```
"pi = {0:.3f}".format(pi)
```

[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.

8.3 Special characters

Above we saw a special character, the newline, represented in Python strings as `\n`. While displayed in source code with two characters, `\` and `n`, when paired they are interpreted as a single character representing what would be emitted when pressing the enter or return key.

[1]:
```
print(len("\n"))
```

```
1
```

Another special character often encountered is the tab (`\t`).

9 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`"), *loop statements* (including "`for`" and "`while`" and the accompanying "`break`", "`continue`", and "`pass`").

9.1 Conditional Statements: `if-elif-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:

[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 optinally include as few or as many `elif` statements as you would like.

9.2 `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:

[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. We do not need to deal with a looping index variable (unless we want to!).

- The Python for loop is considered a "foreach" loop in other languages. You can read the above as "for each N in [2,3,5,7]".

The object to the right of the "`in`" can be any Python *iterator*, not just a list. An iterator can be thought of as a generalized sequence, and we'll discuss them in Iterators and List Comprehensions.

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

[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:

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

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

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

[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.

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.

9.3 `while` loops

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

[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.

9.4 `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:

[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:

[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!

See also Iterators and List Comprehensions for more advanced looping mechanisms in Python.

10 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.

10.1 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:

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

```
abc
```

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

In addition to arguments, functions can be designed to accept *keyword arguments* that are optional and 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:

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

```
1 2 3
```

[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.

10.2 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 the Fibonacci sequence code from the previous section as follows:

[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:

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

[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. Of course, if a function is designed around an integer argument, it is likely an error will occur if it receives a string.

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:

[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)
```

10.3 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:

[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:

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

[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:

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

[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:

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

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

10.4 `*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:

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

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

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

[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!

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

catch_all(*inputs, **keywords)
```

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

It is common in many large python packages, especially matplotlib, to have many functions that take `**kwargs`.

10.5 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:

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

[15]: 3

This lambda function is roughly equivalent to

[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:

[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:

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

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

But dictionaries are not orderable: we need a way to tell the `sorted()` 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:

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

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

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

[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 compact `lambda` syntax is convenient for such short one-off functions like these.

11 Classes and Methods

As mentioned before, everything in Python is an object. That means Python code can be organized into classes, allowing objects to neatly bundle up data and associated functionality. Classes allow for more modular and reusable code. Functions attached to objects are referred to as "methods".

Even if you are not writing classes yourself, since everything is an object, it is important to understand the basics if you encounter errors related to classes or class functionality.

11.1 Defining classes and methods

Here is a simple example of a class with an initialization function (all Python initialization functions are named `__init__`) and several methods

[1]:
```
class Dog():
    def __init__(self, name, age,weight):
        self.name = name
        self.age = age
        self.weight = weight

        self.location = (0,0)
        self.trajectory = [self.location]

    def howl(self):
        print("I am {}. Hear me roar!!!".format(self.name))

    def walk(self, step=0.1):
        """To update the position of the dog..."""
        x,y = self.location
        self.location = x+step, y+step
        self.trajectory.append(self.location)

    def save(self, prefix):
        filename = "{}_n{name}_a{age}_w{weight}_traj.txt".
↪format(prefix,name=self.name, age=self.age, weight=self.weight)
        print(filename)
        # write to disk in some way
```

Here we defined a class called `Dog` and gave it several attributes (`name`, `age`, ...) and methods (`howl`, `walk`, ...). The syntax is very simple, methods are defined just like functions except they are indented within the `class` block.

Each method of the class should begin with an argument called `self`. You can think of this argument as a placeholder for the object that will be created.

[2]:
```
fido = Dog("fido", 2.5, 20)

fido.howl()
```

```
I am fido. Hear me roar!!!
```

Here we create a `Dog` object giving it a name (`fido`), age (`2.5`), and weight (`20`) which are set up as attributes automatically inside the `__init__` method. We then call the `howl` method. Note that the definition of the method has an argument called `self` but this is only a placeholder for the object. Now that the object (`fido`) exists, `fido.howl()` will automatically resolve to `Dog.howl(fido)` and we do not need to worry about `self`.

11.2 Applications of objects

Object-oriented programming is helpful for **maintaining state**. For example, a *chatbot* can switch itself between different modes of functionality and use an attribute to remember what state it is in.

Likewise, attributes and methods make it easier to pass many arguments between functions. Consider this code:

```
result, time, date = computation(input1, input2)

experiment['result'] = result
experiment['time'] = time
experiment['date'] = date
```

Here the user has to call a function that takes two inputs in a specific order and returns three outputs. She must remember the order of these outputs, which can be easy to forget if she did not (recently) write `computation`. And she must deal with updating a dictionary called `experiment`.

Using an object, much of this bookkeeping can be **hidden away** from the user:

```
class Experiment()
    ...

    def update_computation(self):
        self.result = self.get_new_result(self.input1)
        self.time = self.get_new_time(self.input2)
        self.date = self.get_new_date(self.input1)
    ...
```

Now the end user does not need to put nearly as much thought into the internal processes, and won't need to remember the order of inputs and outputs:

```
experiment = Experiment()
experiment.update_computation()
```

While not always optimal, structuring code in this way is often effective.

See also: the official Python tutorial on classes.

12 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 track-down 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.

12.0.1 Tracebacks

Python reports error messages called "tracebacks" (sometimes known as "stack traces"). These messages highlight the portion of code where the error occurred and any code that may have called the erroneous code. This "tracing" backwards through the sequence of code allows you to pinpoint the location of the error and helps guide you to identifying the root cause of the error.

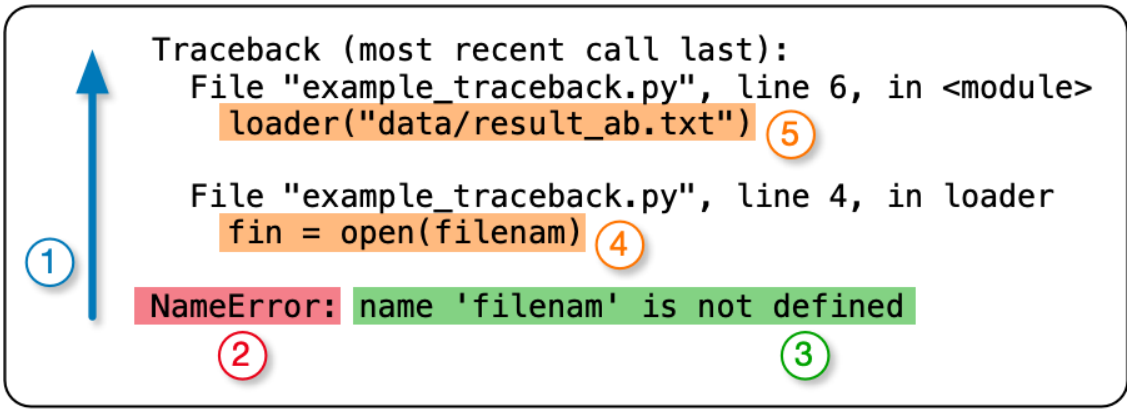
This example (`example_traceback.py`) contains an error:

```
# example_traceback.py

def loader(filename):
    fin = open(filenam)

loader("data/result_ab.txt")
```

Here is the traceback that is emitted when running this code. I have added color-coding and annotations to help show how to read tracebacks.



1. Read from bottom to top
2. Name of exception
3. Error message
4, 5. Line(s) of code leading backwards from error

Googling for parts of tracebacks, such as error messages (after removing any information such as filenames that are unique to your code), are a great way to solve problems.

12.1 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:

[1]:
```
print(Q)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-e796bdcf24ff> in <module>()
----> 1 print(Q)

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

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

[2]:
```
1 + 'abc'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-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:

[3]:
```
2 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-ae0c5d243292> in <module>()
----> 1 2 / 0

ZeroDivisionError: division by zero
```

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

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-06b6eb1b8957> 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. Although it may not always be as clear as these examples, having access to meanngful errors like this is immensely useful when trying to trace the root of problems in your code.

12.2 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:

[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:

[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}:

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

[8]:
```
safe_divide(1, 2)
```

[8]:
```
0.5
```

[9]:
```
safe_divide(2, 0)
```

[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:

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

[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*:

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

[12]:
```
safe_divide(1, 0)
```

[12]:
```
1e+100
```

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-2331af6a0acf> in <module>()
----> 1 safe_divide(1, '2')

<ipython-input-11-10b5f0163af8> 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.

Try-except blocks are powerful and efficient in Python, and being explicit about the errors you catch helps ensure that other errors are not being skipped over without your knowledge.

13 Iterators and List Comprehensions

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 a 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:

[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.

13.1 Iterating over lists

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

[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:

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

[3]:
```
<list_iterator at 0x7fb430aaabe0>
```

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 there **is** a next object, which can be seen with the built-in function `next`:

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

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

```
2
```

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

```
4
```

[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*.

13.2 `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:

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

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

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

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

[9]:
```
<range_iterator at 0x7fb430a8e300>
```

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

[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!):

[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:

[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`).

13.3 Useful Iterators

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

13.3.1 `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:

[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:

[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.

13.3.2 `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:

[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`.

13.3.3 `map` and `filter`

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

[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:

[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).

13.3.4 Iterators as function arguments

We saw in *args and **kwargs: Flexible Arguments. 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:

[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:

[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 which 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:

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

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

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

[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!

13.4 Specialized Iterators: `itertools`

We briefly looked at the infinite `range` iterator, `itertools.count`. 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:

[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:

[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:

[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.

13.5 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:

[26]:
```
[i for i in range(20) if i % 3 > 0]
```

[26]:
```
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

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

13.5.1 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:

[27]:
```
L = []
for n in range(12):
    L.append(n ** 2)
L
```

[27]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

The list comprehension equivalent of this is the following:

[28]:
```
[n ** 2 for n in range(12)]
```

[28]:
```
[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` from zero 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.

13.5.2 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:

[29]:
```
[(i, j) for i in range(2) for j in range(3)]
```

[29]:
```
[(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 readibility will suffer!

13.5.3 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:

[30]:
```
[val for val in range(20) if val % 3 > 0]
```

[30]:
```
[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:

[31]:
```
L = []
for val in range(20):
    if val % 3:
        L.append(val)
L
```

[31]:
```
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

13.5.4 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:

[32]:
```
val = -10
val if val >= 0 else -val
```

[32]:
```
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:

[33]:
```
[val if val % 2 else -val
 for val in range(20) if val % 3]
```

[33]:
```
[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 readibility. Look this over: what we're doing is constructing a list, leaving out multiples of 3, and negating all mutliples 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*:

[34]:
```
{n**2 for n in range(12)}
```

[34]:
```
{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:

[35]:
```
{a % 3 for a in range(1000)}
```

[35]:
```
{0, 1, 2}
```

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

[36]:
```
{n:n**2 for n in range(6)}
```

[36]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

13.5.5 Generators

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

[37]:
```
(n**2 for n in range(12))
```

[37]:
```
<generator object <genexpr> at 0x7fb4210f9a98>
```

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.

14 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 and tools for installing third-party modules.

14.1 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.

14.1.1 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:

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

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

14.1.2 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`:

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

[2]:
```
-1.0
```

14.1.3 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:

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

[3]:
```
-1.0
```

14.1.4 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:

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

[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:

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

```
Help on built-in function sum in module builtins:

sum(iterable, start=0, /)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
```

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

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

[6]:
```
9
```

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

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

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

[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.

14.2 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 seen 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/.

14.3 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/.

14.3.1 Anaconda Python

For Anaconda python, another package manager is available called conda. When working with Anaconda Python, it is generally better to try installing first with `conda` and then falling back to `pip` if needed.

15 Working with Files and Folders

15.1 Reading files

Python provides functionality for accessing files and folders (directories). The `open` function is a builtin Python function that can be used to read and write files.

```
fin = open("f.txt")
```

Here we have opened the text file `f.txt` for reading (the default open "mode", equivalent to calling `open("f.txt", 'r')`). This creates an object (or "handle") `fin` that provides access to the file. For example,

```
fin.read()
```

or

```
fin.readline()
```

or

```
fin.readlines()
```

allow us different ways to read the contents of the file:

- `read` reads the file into a single string
- `readline` reads from the current position in the file until the first newline is encountered
- `readlines` reads the file into a list of strings, one list element for each line.

File handles keep track of their current position in the file, so for example if you call `fin.readline()` and then call it again, the second call will begin reading where the previous call stopped. This is handy for incrementally reading files.

File handles work well with loops:

```
for line in open("f.txt"):
    print(line.strip())
```

Here the for loop automatically retrieves each line of the text file and assigns it to the looping variable `line`. The `line.strip()` call removes the trailing newline character that is kept with each `line`.

15.2 Writing files

The `open` function can also be used to write contents to a file. Here you need to make sure to open the file in "writing mode"

```
fout = open("g.txt", 'w')
fout.write("Saving data...\n")
```

Note that using `.write` requires dealing with newlines.

The builtin `print` function interacts nicely with file handles:

```
print("Hello!", file=fout)
```

and by default it will even add the newlines for you!

It is important when you are done writing to a file to **close** it:

```
fout.close()
```

File handles can also be opened in "append" mode (`a`) and in binary read/write modes (`rb`, `wb`). These are less common than read mode (`r`) and write mode (`w`).

15.3 File names, paths, and working directories

A file **path** is used to represent the name and location of a file. For example, a file named `f.txt` stored in Alice Smith's Documents folder on a Mac may have the following path:

```
/Users/alicesmith/Documents/f.txt
```

Here the folder `Documents` is inside the `alicesmith` folder which is inside the `Users` folder. The forward slash character (`/`) is used as a **delimiter** to separate the elements of the path. Some operating systems use different symbols for this purpose, and Python provides an `os` and `os.path` module to handle this automatically.

15.3.1 Working directories

When Python is running it has a default directory for file operations. This is known as the **working directory** or **current working directory**. For example, when opening a file just using the filename, Python assumes the file is in the current working directory.

The `os` module provides a function for printing the full path of the current working directory (cwd):

```
>>> import os
>>> cwd = os.getcwd()
>>> print(cwd)
/Users/alicesmith
```

This example shows the current working directory is Alice's home folder.

A **relative path** is a file path that starts from the current working directory, whereas an **absolute path** starts from the top of the file system. For example, the path `statistics/f.txt` assumes there is a folder inside the current working directory called `statistics` that contains the file `f.txt`.

The `os.path` modules allows you to convert from relative to absolute paths:

```
>>> os.path.abspath("statistics/f.txt")
'/Users/alicesmith/statistics/f.txt'
```

Changing the working directory is a common action and is often needed to make code that is portable across different computers. By keeping file paths relative to a working directory, machine-specific information (such as the name of user's home folder) is not needed.

16 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.

16.1 NumPy: Numerical Python

NumPy provides an efficient way to store and manipulate multi-dimensional 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()`):

[1]:
```
import numpy as np
x = np.arange(1, 10)
x
```

[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:

[2]:
```
x ** 2
```

[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:

[5]:
```
[val ** 2 for val in range(1, 10)]
```

[5]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Unlike Python lists (which are limited to one dimension), NumPy arrays can be multi-dimensional. For example, here we will reshape our `x` array into a 3x3 array:

[6]:
```
M = x.reshape((3, 3))
M
```

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

NumPy arrays have a `shape` attribute that is helpful to check the size of the array:

[7]:
```
print(x.shape)
print(M.shape)
```

```
(9,)
(3, 3)
```

Here `x` is a 1-dimensional array of length 9, indicated by the shape tuple containing a single element (the trailing comma is used to distinguish a tuple of one element from a number inside parentheses), whereas `M` is a 2-dimensional array that has three rows and three columns.

- Note: If you are used to dealing with row- and column-vectors, please be aware that NumPy's array representation is somewhat different. A typical N-dimensional row vector, like you may encounter in a linear algebra course, would have shape `(1,N)` and likewise an N-dimensional column vector would have shape `(N,1)`. Yet a numpy array of shape `(N,)` behaves somewhat differently than an array of shape `(N,1)`.

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`:

[8]:
```
M.T
```

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

or a matrix-vector product using `np.dot`:

[9]:
```
np.dot(M, [5, 6, 7])
```

[9]:
```
array([ 38,  92, 146])
```

and even more sophisticated operations like eigenvalue decomposition:

[10]:
```
np.linalg.eigvals(M)
```

[10]:
```
array([ 1.61168440e+01, -1.11684397e+00, -9.75918483e-16])
```

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.

16.2 Pandas: Labeled Column-oriented Data

Pandas provides is a labeled interface to multi-dimensional 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:

[11]:
```
import pandas as pd
df = pd.DataFrame({'label': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'value': [1, 2, 3, 4, 5, 6]})
df
```

[11]:

	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:

[12]:
```
df['label']
```

```
[12]: 0    A
      1    B
      2    C
      3    A
      4    B
      5    C
      Name: label, dtype: object
```

Apply string operations across string entries:

```
[13]: df['label'].str.lower()
```

```
[13]: 0    a
      1    b
      2    c
      3    a
      4    b
      5    c
      Name: label, dtype: object
```

Apply aggregates across numerical entries:

```
[14]: df['value'].sum()
```

```
[14]: 21
```

And, perhaps most importantly, do efficient database-style joins and groupings:

```
[15]: df.groupby('label').sum()
```

```
[15]:        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 Resources for Further Learning.

16.3 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.

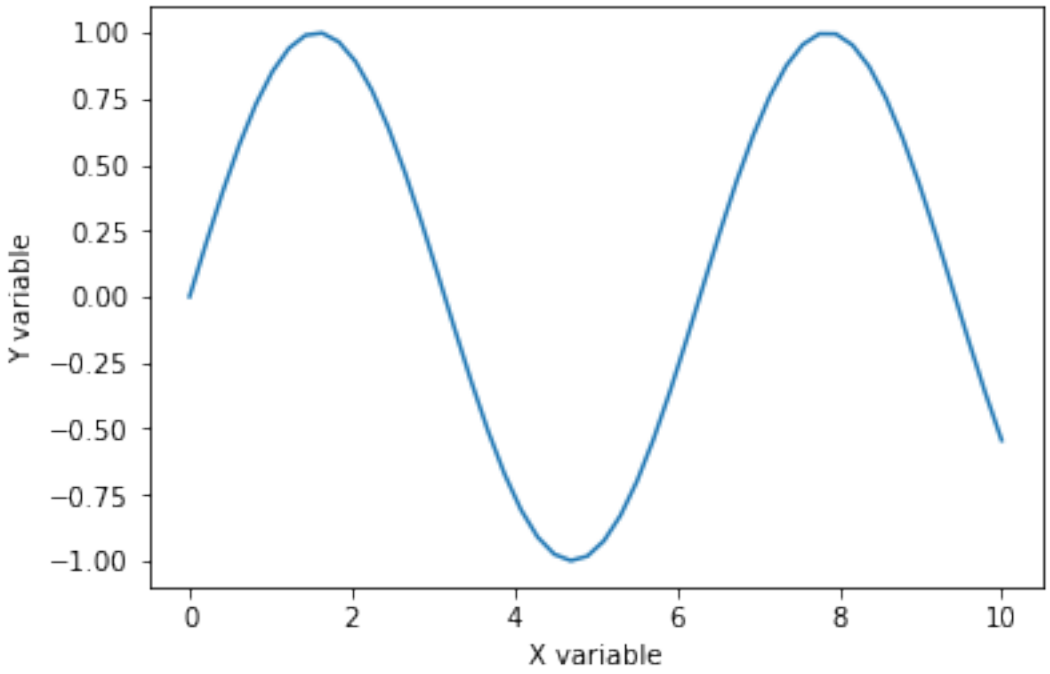
To use Matplotlib, we can start by enabling the notebook mode (for use in the Jupyter notebook) and then importing the package as `plt`"

[1]:
```
# run this only if using a Jupyter notebook
%matplotlib inline
```

[2]:
```
import matplotlib.pyplot as plt
```

Now let's create some data (as NumPy arrays, of course) and plot the results:

[3]:
```
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
plt.xlabel("X variable")
plt.ylabel("Y variable");
```



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.

16.4 SciPy: Scientific Python

SciPy is a collection of scientific functionality that is built on top of 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

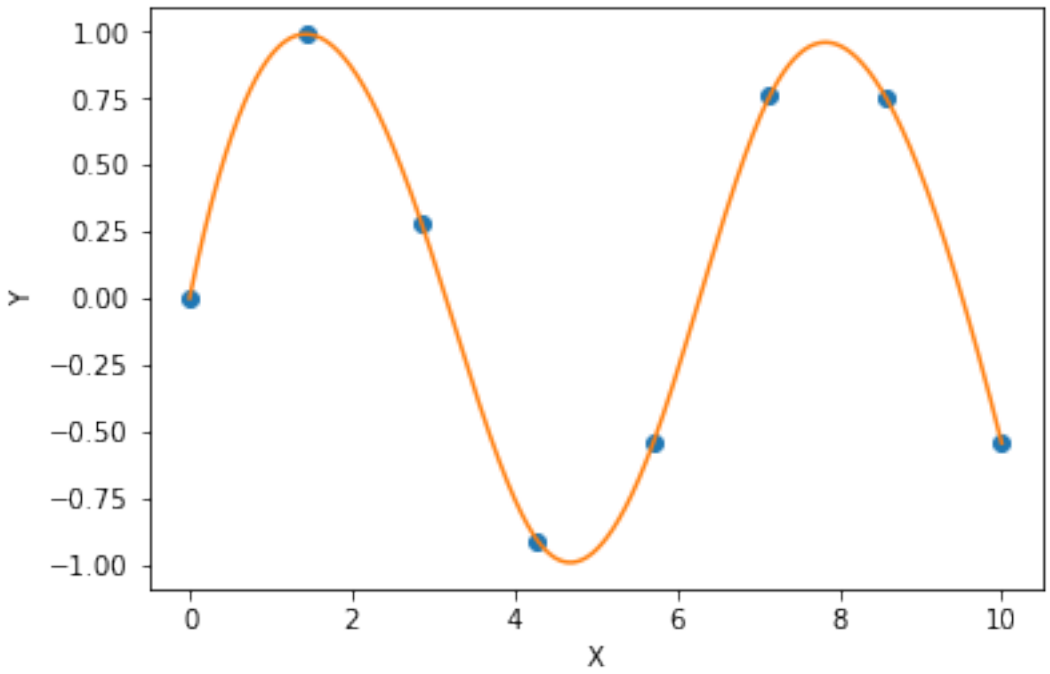
[4]:
```
from scipy import interpolate

# 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)
plt.xlabel("X")
plt.ylabel("Y");
```



What we see is a smooth interpolation between the points.

16.5 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.

17 Profiling and Timing Code

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

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

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

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

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

The magic commands `%timeit` line-magic and `%%timeit` cell-magic can be used within IPython to time the repeated execution of snippets of code:

[1]:
```
%timeit sum(range(100))
```

```
100000 loops, best of 3: 1.54 µs per loop
```

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

[2]:
```
%%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

```
1 loops, best of 3: 407 ms per loop
```

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

[3]:
```
import random
L = [random.random() for i in range(100000)]
%timeit L.sort()
```

```
100 loops, best of 3: 1.9 ms per loop
```

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

[4]:
```
import random
L = [random.random() for i in range(100000)]
print("sorting an unsorted list:")
%time L.sort()
```

```
sorting an unsorted list:
CPU times: user 40.6 ms, sys: 896 µs, total: 41.5 ms
Wall time: 41.5 ms
```

[5]:
```
print("sorting an already sorted list:")
%time L.sort()
```

```
sorting an already sorted list:
CPU times: user 8.18 ms, sys: 10 µs, total: 8.19 ms
Wall time: 8.24 ms
```

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

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

[6]:
```
%%time
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

```
CPU times: user 504 ms, sys: 979 µs, total: 505 ms
Wall time: 505 ms
```

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

17.2 Profiling Full Scripts: `%prun`

A program is made of many single statements, and sometimes timing these statements in context is more important than timing them on their own. Python contains a built-in code profiler (which you can read about in the Python documentation), but IPython offers a much more convenient way to use this profiler, in the form of the magic function `%prun`.

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

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

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

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

In the notebook, the output is printed to the pager, and looks something like this:

```
14 function calls in 0.714 seconds

   Ordered by: internal time

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

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

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

17.3 Line-By-Line Profiling with `%lprun`

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

```
$ pip install line_profiler
```

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

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

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

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

As before, the notebook sends the result to the pager, but it looks something like this:

```
Timer unit: 1e-06 s

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

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1            2      2.0      0.0      total = 0
     3         6            8      1.3      0.1      for i in range(5):
```

```
     4         5         9001   1800.2     95.9          L = [j ^ (j >> i) for j in range(N)]
     5         5          371     74.2      4.0          total += sum(L)
     6         1            0      0.0      0.0      return total
```

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

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

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

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

```
$ pip install memory_profiler
```

Then we can use IPython to load the extension:

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

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

[13]:
```
%memit sum_of_lists(1000000)
```

```
peak memory: 100.08 MiB, increment: 61.36 MiB
```

We see that this function uses about 100 MB of memory.

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

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

```
Overwriting mprun_demo.py
```

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

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

The result, printed to the pager, gives us a summary of the memory use of the function, and looks something like this:

```
Filename: ./mprun_demo.py

Line #    Mem usage    Increment   Line Contents
================================================
     4     71.9 MiB      0.0 MiB           L = [j ^ (j >> i) for j in range(N)]


Filename: ./mprun_demo.py

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

Here the `Increment` column tells us how much each line affects the total memory budget: observe that when we create and delete the list `L`, we are adding about 25 MB of memory usage. This is on top of the background memory usage from the Python interpreter itself.

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

18 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 OReilly 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 walk-through, see Beazley's *Python Cookbook*.

To dig more into Python tools for data science and scientific computing:

- *The Python Data Science Handbook* by Jake VanderPlas. This book starts precisely where this mini-text 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 Kathryn D. Huff and Anthony Scopatz, 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:

- *OReilly 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.