CHAPTER 2
A Crash Course in Python

> *People are still crazy about Python after twenty-five years, which I find hard to believe.*
>
> —Michael Palin

All new employees at DataSciencester are required to go through new employee orientation, the most interesting part of which is a crash course in Python.

This is not a comprehensive Python tutorial but instead is intended to highlight the parts of the language that will be most important to us (some of which are often not the focus of Python tutorials).

The Basics

Getting Python

You can download Python from python.org. But if you don't already have Python, I recommend instead installing the Anaconda distribution, which already includes most of the libraries that you need to do data science.

As I write this, the latest version of Python is 3.4. At DataSciencester, however, we use old, reliable Python 2.7. Python 3 is not backward-compatible with Python 2, and many important libraries only work well with 2.7. The data science community is still firmly stuck on 2.7, which means we will be, too. Make sure to get that version.

If you don't get Anaconda, make sure to install pip, which is a Python package manager that allows you to easily install third-party packages (some of which we'll need). It's also worth getting IPython, which is a much nicer Python shell to work with.

(If you installed Anaconda then it should have come with pip and IPython.)

Just run:

```
pip install ipython
```

and then search the Internet for solutions to whatever cryptic error messages that causes.

The Zen of Python

Python has a somewhat Zen description of its design principles, which you can also find inside the Python interpreter itself by typing `import this`.

One of the most discussed of these is:

> There should be one—and preferably only one—obvious way to do it.

Code written in accordance with this "obvious" way (which may not be obvious at all to a newcomer) is often described as "Pythonic." Although this is not a book about Python, we will occasionally contrast Pythonic and non-Pythonic ways of accomplishing the same things, and we will generally favor Pythonic solutions to our problems.

Whitespace Formatting

Many languages use curly braces to delimit blocks of code. Python uses indentation:

```
for i in [1, 2, 3, 4, 5]:
    print i                     # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print j                 # first line in "for j" block
        print i + j             # last line in "for j" block
    print i                     # last line in "for i" block
print "done looping"
```

This makes Python code very readable, but it also means that you have to be very careful with your formatting. Whitespace is ignored inside parentheses and brackets, which can be helpful for long-winded computations:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

and for making code easier to read:

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

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

You can also use a backslash to indicate that a statement continues onto the next line, although we'll rarely do this:

```
two_plus_three = 2 + \
                 3
```

One consequence of whitespace formatting is that it can be hard to copy and paste code into the Python shell. For example, if you tried to paste the code:

```
for i in [1, 2, 3, 4, 5]:

    # notice the blank line
    print i
```

into the ordinary Python shell, you would get a:

```
IndentationError: expected an indented block
```

because the interpreter thinks the blank line signals the end of the `for` loop's block.

IPython has a magic function `%paste`, which correctly pastes whatever is on your clipboard, whitespace and all. This alone is a good reason to use IPython.

Modules

Certain features of Python are not loaded by default. These include both features included as part of the language as well as third-party features that you download yourself. In order to use these features, you'll need to `import` the modules that contain them.

One approach is to simply import the module itself:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Here `re` is the module containing functions and constants for working with regular expressions. After this type of `import` you can only access those functions by prefixing them with `re.`.

If you already had a different `re` in your code you could use an alias:

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

You might also do this if your module has an unwieldy name or if you're going to be typing it a lot. For example, when visualizing data with `matplotlib`, a standard convention is:

```
import matplotlib.pyplot as plt
```

If you need a few specific values from a module, you can import them explicitly and use them without qualification:

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

If you were a bad person, you could import the entire contents of a module into your namespace, which might inadvertently overwrite variables you've already defined:

```
match = 10
from re import *    # uh oh, re has a match function
print match         # "<function re.match>"
```

However, since you are not a bad person, you won't ever do this.

Arithmetic

Python 2.7 uses integer division by default, so that `5 / 2` equals `2`. Almost always this is not what we want, so we will always start our files with:

```
from __future__ import division
```

after which `5 / 2` equals `2.5`. Every code example in this book uses this new-style division. In the handful of cases where we need integer division, we can get it with a double slash: `5 // 2`.

Functions

A function is a rule for taking zero or more inputs and returning a corresponding output. In Python, we typically define functions using `def`:

```
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

Python functions are *first-class*, which means that we can assign them to variables and pass them into functions just like any other arguments:

```
def apply_to_one(f):
    """calls the function f with 1 as its argument"""
    return f(1)

my_double = double              # refers to the previously defined function
x = apply_to_one(my_double)     # equals 2
```

It is also easy to create short anonymous functions, or lambdas:

```
y = apply_to_one(lambda x: x + 4)       # equals 5
```

You can assign lambdas to variables, although most people will tell you that you should just use `def` instead:

```
another_double = lambda x: 2 * x        # don't do this
def another_double(x): return 2 * x     # do this instead
```

Function parameters can also be given default arguments, which only need to be specified when you want a value other than the default:

```
def my_print(message="my default message"):
    print message
```

```
my_print("hello")   # prints 'hello'
my_print()          # prints 'my default message'
```

It is sometimes useful to specify arguments by name:

```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # returns 5
subtract(0, 5)  # returns -5
subtract(b=5)   # same as previous
```

We will be creating many, many functions.

Strings

Strings can be delimited by single or double quotation marks (but the quotes have to match):

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

Python uses backslashes to encode special characters. For example:

```
tab_string = "\t"       # represents the tab character
len(tab_string)         # is 1
```

If you want backslashes as backslashes (which you might in Windows directory names or in regular expressions), you can create *raw* strings using `r""`:

```
not_tab_string = r"\t"  # represents the characters '\' and 't'
len(not_tab_string)     # is 2
```

You can create multiline strings using triple-[double-]-quotes:

```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

Exceptions

When something goes wrong, Python raises an *exception*. Unhandled, these will cause your program to crash. You can handle them using `try` and `except`:

```
try:
    print 0 / 0
except ZeroDivisionError:
    print "cannot divide by zero"
```

Although in many languages exceptions are considered bad, in Python there is no shame in using them to make your code cleaner, and we will occasionally do so.

Lists

Probably the most fundamental data structure in Python is the `list`. A list is simply an ordered collection. (It is similar to what in other languages might be called an array, but with some added functionality.)

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)     # equals 3
list_sum    = sum(integer_list)     # equals 6
```

You can get or set the *n*th element of a list with square brackets:

```
x = range(10)   # is the list [0, 1, ..., 9]
zero = x[0]     # equals 0, lists are 0-indexed
one = x[1]      # equals 1
nine = x[-1]    # equals 9, 'Pythonic' for last element
eight = x[-2]   # equals 8, 'Pythonic' for next-to-last element
x[0] = -1       # now x is [-1, 1, 2, 3, ..., 9]
```

You can also use square brackets to "slice" lists:

```
first_three   = x[:3]               # [-1, 1, 2]
three_to_end = x[3:]                # [3, 4, ..., 9]
one_to_four = x[1:5]                # [1, 2, 3, 4]
last_three = x[-3:]                 # [7, 8, 9]
without_first_and_last = x[1:-1]    # [1, 2, ..., 8]
copy_of_x = x[:]                    # [-1, 1, 2, ..., 9]
```

Python has an `in` operator to check for list membership:

```
1 in [1, 2, 3]    # True
0 in [1, 2, 3]    # False
```

This check involves examining the elements of the list one at a time, which means that you probably shouldn't use it unless you know your list is pretty small (or unless you don't care how long the check takes).

It is easy to concatenate lists together:

```
x = [1, 2, 3]
x.extend([4, 5, 6])     # x is now [1,2,3,4,5,6]
```

If you don't want to modify x you can use list addition:

```
x = [1, 2, 3]
y = x + [4, 5, 6]       # y is [1, 2, 3, 4, 5, 6]; x is unchanged
```

More frequently we will append to lists one item at a time:

```
x = [1, 2, 3]
x.append(0)      # x is now [1, 2, 3, 0]
```

```
y = x[-1]        # equals 0
z = len(x)       # equals 4
```

It is often convenient to *unpack* lists if you know how many elements they contain:

```
x, y = [1, 2]    # now x is 1, y is 2
```

although you will get a `ValueError` if you don't have the same numbers of elements on both sides.

It's common to use an underscore for a value you're going to throw away:

```
_, y = [1, 2]    # now y == 2, didn't care about the first element
```

Tuples

Tuples are lists' immutable cousins. Pretty much anything you can do to a list that doesn't involve modifying it, you can do to a tuple. You specify a tuple by using parentheses (or nothing) instead of square brackets:

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3      # my_list is now [1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print "cannot modify a tuple"
```

Tuples are a convenient way to return multiple values from functions:

```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)    # equals (5, 6)
s, p = sum_and_product(5, 10) # s is 15, p is 50
```

Tuples (and lists) can also be used for *multiple assignment*:

```
x, y = 1, 2     # now x is 1, y is 2
x, y = y, x     # Pythonic way to swap variables; now x is 2, y is 1
```

Dictionaries

Another fundamental data structure is a dictionary, which associates *values* with *keys* and allows you to quickly retrieve the value corresponding to a given key:

```
empty_dict = {}                         # Pythonic
empty_dict2 = dict()                    # less Pythonic
grades = { "Joel" : 80, "Tim" : 95 }    # dictionary literal
```

You can look up the value for a key using square brackets:

```
joels_grade = grades["Joel"]            # equals 80
```

But you'll get a KeyError if you ask for a key that's not in the dictionary:

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print "no grade for Kate!"
```

You can check for the existence of a key using in:

```
joel_has_grade = "Joel" in grades     # True
kate_has_grade = "Kate" in grades     # False
```

Dictionaries have a get method that returns a default value (instead of raising an exception) when you look up a key that's not in the dictionary:

```
joels_grade = grades.get("Joel", 0)   # equals 80
kates_grade = grades.get("Kate", 0)   # equals 0
no_ones_grade = grades.get("No One")  # default default is None
```

You assign key-value pairs using the same square brackets:

```
grades["Tim"] = 99                    # replaces the old value
grades["Kate"] = 100                  # adds a third entry
num_students = len(grades)            # equals 3
```

We will frequently use dictionaries as a simple way to represent structured data:

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

Besides looking for specific keys we can look at all of them:

```
tweet_keys   = tweet.keys()     # list of keys
tweet_values = tweet.values()   # list of values
tweet_items  = tweet.items()    # list of (key, value) tuples

"user" in tweet_keys            # True, but uses a slow list in
"user" in tweet                 # more Pythonic, uses faster dict in
"joelgrus" in tweet_values      # True
```

Dictionary keys must be immutable; in particular, you cannot use lists as keys. If you need a multipart key, you should use a tuple or figure out a way to turn the key into a string.

defaultdict

Imagine that you're trying to count the words in a document. An obvious approach is to create a dictionary in which the keys are words and the values are counts. As you check each word, you can increment its count if it's already in the dictionary and add it to the dictionary if it's not:

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

You could also use the "forgiveness is better than permission" approach and just handle the exception from trying to look up a missing key:

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

A third approach is to use `get`, which behaves gracefully for missing keys:

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

Every one of these is slightly unwieldy, which is why `defaultdict` is useful. A `defaultdict` is like a regular dictionary, except that when you try to look up a key it doesn't contain, it first adds a value for it using a zero-argument function you provided when you created it. In order to use `defaultdicts`, you have to import them from `collections`:

```
from collections import defaultdict

word_counts = defaultdict(int)          # int() produces 0
for word in document:
    word_counts[word] += 1
```

They can also be useful with `list` or `dict` or even your own functions:

```
dd_list = defaultdict(list)             # list() produces an empty list
dd_list[2].append(1)                    # now dd_list contains {2: [1]}

dd_dict = defaultdict(dict)             # dict() produces an empty dict
dd_dict["Joel"]["City"] = "Seattle"     # { "Joel" : { "City" : Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # now dd_pair contains {2: [0,1]}
```

These will be useful when we're using dictionaries to "collect" results by some key and don't want to have to check every time to see if the key exists yet.

Counter

A `Counter` turns a sequence of values into a `defaultdict(int)`-like object mapping keys to counts. We will primarily use it to create histograms:

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c is (basically) { 0 : 2, 1 : 1, 2 : 1 }
```

This gives us a very simple way to solve our `word_counts` problem:

```
word_counts = Counter(document)
```

A `Counter` instance has a `most_common` method that is frequently useful:

```
# print the 10 most common words and their counts
for word, count in word_counts.most_common(10):
    print word, count
```

Sets

Another data structure is `set`, which represents a collection of *distinct* elements:

```
s = set()
s.add(1)       # s is now { 1 }
s.add(2)       # s is now { 1, 2 }
s.add(2)       # s is still { 1, 2 }
x = len(s)     # equals 2
y = 2 in s     # equals True
z = 3 in s     # equals False
```

We'll use sets for two main reasons. The first is that `in` is a very fast operation on sets. If we have a large collection of items that we want to use for a membership test, a set is more appropriate than a list:

```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]

"zip" in stopwords_list     # False, but have to check every element

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # very fast to check
```

The second reason is to find the *distinct* items in a collection:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)                # 6
item_set = set(item_list)                 # {1, 2, 3}
num_distinct_items = len(item_set)        # 3
distinct_item_list = list(item_set)       # [1, 2, 3]
```

We'll use sets much less frequently than `dicts` and `lists`.

Control Flow

As in most programming languages, you can perform an action conditionally using `if`:

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

You can also write a *ternary* if-then-else on one line, which we will do occasionally:

```
parity = "even" if x % 2 == 0 else "odd"
```

Python has a `while` loop:

```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```

although more often we'll use `for` and `in`:

```
for x in range(10):
    print x, "is less than 10"
```

If you need more-complex logic, you can use `continue` and `break`:

```
for x in range(10):
    if x == 3:
        continue  # go immediately to the next iteration
    if x == 5:
        break     # quit the loop entirely
    print x
```

This will print `0`, `1`, `2`, and `4`.

Truthiness

Booleans in Python work as in most other languages, except that they're capitalized:

```
one_is_less_than_two = 1 < 2          # equals True
true_equals_false = True == False     # equals False
```

Python uses the value `None` to indicate a nonexistent value. It is similar to other languages' `null`:

```
x = None
print x == None     # prints True, but is not Pythonic
print x is None     # prints True, and is Pythonic
```

Python lets you use any value where it expects a Boolean. The following are all "Falsy":

- `False`
- `None`
- `[]` (an empty `list`)
- `{}` (an empty `dict`)
- `""`
- `set()`
- `0`
- `0.0`

Pretty much anything else gets treated as `True`. This allows you to easily use `if` statements to test for empty lists or empty strings or empty dictionaries or so on. It also sometimes causes tricky bugs if you're not expecting this behavior:

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

A simpler way of doing the same is:

```
first_char = s and s[0]
```

since `and` returns its second value when the first is "truthy," the first value when it's not. Similarly, if `x` is either a number or possibly `None`:

```
safe_x = x or 0
```

is definitely a number.

Python has an `all` function, which takes a list and returns `True` precisely when every element is truthy, and an `any` function, which returns `True` when at least one element is truthy:

```
all([True, 1, { 3 }])   # True
all([True, 1, {}])      # False, {} is falsy
any([True, 1, {}])      # True, True is truthy
all([])                 # True, no falsy elements in the list
any([])                 # False, no truthy elements in the list
```

The Not-So-Basics

Here we'll look at some more-advanced Python features that we'll find useful for working with data.

Sorting

Every Python list has a `sort` method that sorts it in place. If you don't want to mess up your list, you can use the `sorted` function, which returns a new list:

```
x = [4,1,2,3]
y = sorted(x)     # is [1,2,3,4], x is unchanged
x.sort()          # now x is [1,2,3,4]
```

By default, `sort` (and `sorted`) sort a list from smallest to largest based on naively comparing the elements to one another.

If you want elements sorted from largest to smallest, you can specify a `reverse=True` parameter. And instead of comparing the elements themselves, you can compare the results of a function that you specify with `key`:

```
# sort the list by absolute value from largest to smallest
x = sorted([-4,1,-2,3], key=abs, reverse=True)  # is [-4,3,-2,1]

# sort the words and counts from highest count to lowest
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
            reverse=True)
```

List Comprehensions

Frequently, you'll want to transform a list into another list, by choosing only certain elements, or by transforming elements, or both. The Pythonic way of doing this is *list comprehensions*:

```
even_numbers = [x for x in range(5) if x % 2 == 0]  # [0, 2, 4]
squares      = [x * x for x in range(5)]            # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]        # [0, 4, 16]
```

You can similarly turn lists into dictionaries or sets:

```
square_dict = { x : x * x for x in range(5) }  # { 0:0, 1:1, 2:4, 3:9, 4:16 }
square_set  = { x * x for x in [1, -1] }       # { 1 }
```

If you don't need the value from the list, it's conventional to use an underscore as the variable:

```
zeroes = [0 for _ in even_numbers]      # has the same length as even_numbers
```

A list comprehension can include multiple `for`s:

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]   # 100 pairs (0,0) (0,1) ... (9,8), (9,9)
```

and later `for`s can use the results of earlier ones:

```
increasing_pairs = [(x, y)                       # only pairs with x < y,
                    for x in range(10)           # range(lo, hi) equals
                    for y in range(x + 1, 10)]   # [lo, lo + 1, ..., hi - 1]
```

We will use list comprehensions a lot.

Generators and Iterators

A problem with lists is that they can easily grow very big. `range(1000000)` creates an actual list of 1 million elements. If you only need to deal with them one at a time, this can be a huge source of inefficiency (or of running out of memory). If you potentially only need the first few values, then calculating them all is a waste.

A *generator* is something that you can iterate over (for us, usually using `for`) but whose values are produced only as needed (*lazily*).

One way to create generators is with functions and the `yield` operator:

```
def lazy_range(n):
    """a lazy version of range"""
    i = 0
    while i < n:
        yield i
        i += 1
```

The following loop will consume the `yield`ed values one at a time until none are left:

```
for i in lazy_range(10):
    do_something_with(i)
```

(Python actually comes with a `lazy_range` function called `xrange`, and in Python 3, `range` itself is lazy.) This means you could even create an infinite sequence:

```
def natural_numbers():
    """returns 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

although you probably shouldn't iterate over it without using some kind of `break` logic.

The flip side of laziness is that you can only iterate through a generator once. If you need to iterate through something multiple times, you'll need to either recreate the generator each time or use a list.

A second way to create generators is by using `for` comprehensions wrapped in parentheses:

```
lazy_evens_below_20 = (i for i in lazy_range(20) if i % 2 == 0)
```

Recall also that every dict has an items() method that returns a list of its key-value pairs. More frequently we'll use the iteritems() method, which lazily yields the key-value pairs one at a time as we iterate over it.

Randomness

As we learn data science, we will frequently need to generate random numbers, which we can do with the random module:

```
import random

four_uniform_randoms = [random.random() for _ in range(4)]

#  [0.8444218515250481,      # random.random() produces numbers
#   0.7579544029403025,      # uniformly between 0 and 1
#   0.420571580830845,       # it's the random function we'll use
#   0.25891675029296335]     # most often
```

The random module actually produces pseudorandom (that is, deterministic) numbers based on an internal state that you can set with random.seed if you want to get reproducible results:

```
random.seed(10)         # set the seed to 10
print random.random()   # 0.57140259469
random.seed(10)         # reset the seed to 10
print random.random()   # 0.57140259469 again
```

We'll sometimes use random.randrange, which takes either 1 or 2 arguments and returns an element chosen randomly from the corresponding range():

```
random.randrange(10)    # choose randomly from range(10) = [0, 1, ..., 9]
random.randrange(3, 6)  # choose randomly from range(3, 6) = [3, 4, 5]
```

There are a few more methods that we'll sometimes find convenient. random.shuffle randomly reorders the elements of a list:

```
up_to_ten = range(10)
random.shuffle(up_to_ten)
print up_to_ten
# [2, 5, 1, 9, 7, 3, 8, 6, 4, 0]   (your results will probably be different)
```

If you need to randomly pick one element from a list you can use random.choice:

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])     # "Bob" for me
```

And if you need to randomly choose a sample of elements without replacement (i.e., with no duplicates), you can use random.sample:

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)  # [16, 36, 10, 6, 25, 9]
```

To choose a sample of elements *with* replacement (i.e., allowing duplicates), you can just make multiple calls to `random.choice`:

```
four_with_replacement = [random.choice(range(10))
                         for _ in range(4)]
# [9, 4, 4, 2]
```

Regular Expressions

Regular expressions provide a way of searching text. They are incredibly useful but also fairly complicated, so much so that there are entire books written about them. We will explain their details the few times we encounter them; here are a few examples of how to use them in Python:

```
import re

print all([                                 # all of these are true, because
    not re.match("a", "cat"),               # * 'cat' doesn't start with 'a'
    re.search("a", "cat"),                  # * 'cat' has an 'a' in it
    not re.search("c", "dog"),              # * 'dog' doesn't have a 'c' in it
    3 == len(re.split("[ab]", "carbs")),    # * split on a or b to ['c','r','s']
    "R-D-" == re.sub("[0-9]", "-", "R2D2") # * replace digits with dashes
    ])  # prints True
```

Object-Oriented Programming

Like many languages, Python allows you to define *classes* that encapsulate data and the functions that operate on them. We'll use them sometimes to make our code cleaner and simpler. It's probably simplest to explain them by constructing a heavily annotated example.

Imagine we didn't have the built-in Python `set`. Then we might want to create our own `Set` class.

What behavior should our class have? Given an instance of `Set`, we'll need to be able to `add` items to it, `remove` items from it, and check whether it `contains` a certain value. We'll create all of these as *member* functions, which means we'll access them with a dot after a `Set` object:

```
# by convention, we give classes PascalCase names
class Set:

    # these are the member functions
    # every one takes a first parameter "self" (another convention)
    # that refers to the particular Set object being used

    def __init__(self, values=None):
        """This is the constructor.
        It gets called when you create a new Set.
        You would use it like
        s1 = Set()          # empty set
        s2 = Set([1,2,2,3]) # initialize with values"""

        self.dict = {} # each instance of Set has its own dict property
                       # which is what we'll use to track memberships
        if values is not None:
            for value in values:
                self.add(value)

    def __repr__(self):
        """this is the string representation of a Set object
        if you type it at the Python prompt or pass it to str()"""
        return "Set: " + str(self.dict.keys())

    # we'll represent membership by being a key in self.dict with value True
    def add(self, value):
        self.dict[value] = True

    # value is in the Set if it's a key in the dictionary
    def contains(self, value):
        return value in self.dict

    def remove(self, value):
        del self.dict[value]
```

Which we could then use like:

```
s = Set([1,2,3])
s.add(4)
print s.contains(4)     # True
s.remove(3)
print s.contains(3)     # False
```

Functional Tools

When passing functions around, sometimes we'll want to partially apply (or *curry*) functions to create new functions. As a simple example, imagine that we have a function of two variables:

```
def exp(base, power):
    return base ** power
```

and we want to use it to create a function of one variable `two_to_the` whose input is a `power` and whose output is the result of `exp(2, power)`.

We can, of course, do this with `def`, but this can sometimes get unwieldy:

```
def two_to_the(power):
    return exp(2, power)
```

A different approach is to use `functools.partial`:

```
from functools import partial
two_to_the = partial(exp, 2)      # is now a function of one variable
print two_to_the(3)               # 8
```

You can also use `partial` to fill in later arguments if you specify their names:

```
square_of = partial(exp, power=2)
print square_of(3)                   # 9
```

It starts to get messy if you curry arguments in the middle of the function, so we'll try to avoid doing that.

We will also occasionally use `map`, `reduce`, and `filter`, which provide functional alternatives to list comprehensions:

```
def double(x):
    return 2 * x

xs = [1, 2, 3, 4]
twice_xs = [double(x) for x in xs]        # [2, 4, 6, 8]
twice_xs = map(double, xs)                # same as above
list_doubler = partial(map, double)       # *function* that doubles a list
twice_xs = list_doubler(xs)               # again [2, 4, 6, 8]
```

You can use `map` with multiple-argument functions if you provide multiple lists:

```
def multiply(x, y): return x * y

products = map(multiply, [1, 2], [4, 5]) # [1 * 4, 2 * 5] = [4, 10]
```

Similarly, `filter` does the work of a list-comprehension `if`:

```
def is_even(x):
    """True if x is even, False if x is odd"""
    return x % 2 == 0

x_evens = [x for x in xs if is_even(x)]    # [2, 4]
x_evens = filter(is_even, xs)              # same as above
list_evener = partial(filter, is_even)     # *function* that filters a list
x_evens = list_evener(xs)                  # again [2, 4]
```

And `reduce` combines the first two elements of a list, then that result with the third, that result with the fourth, and so on, producing a single result:

```
x_product = reduce(multiply, xs)           # = 1 * 2 * 3 * 4 = 24
list_product = partial(reduce, multiply)   # *function* that reduces a list
x_product = list_product(xs)               # again = 24
```

enumerate

Not infrequently, you'll want to iterate over a list and use both its elements and their indexes:

```
# not Pythonic
for i in range(len(documents)):
    document = documents[i]
    do_something(i, document)

# also not Pythonic
i = 0
for document in documents:
    do_something(i, document)
    i += 1
```

The Pythonic solution is `enumerate`, which produces tuples `(index, element)`:

```
for i, document in enumerate(documents):
    do_something(i, document)
```

Similarly, if we just want the indexes:

```
for i in range(len(documents)): do_something(i)      # not Pythonic
for i, _ in enumerate(documents): do_something(i)   # Pythonic
```

We'll use this a lot.

zip and Argument Unpacking

Often we will need to `zip` two or more lists together. `zip` transforms multiple lists into a single list of tuples of corresponding elements:

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)        # is [('a', 1), ('b', 2), ('c', 3)]
```

If the lists are different lengths, `zip` stops as soon as the first list ends.

You can also "unzip" a list using a strange trick:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

The asterisk performs *argument unpacking*, which uses the elements of `pairs` as individual arguments to `zip`. It ends up the same as if you'd called:

```
zip(('a', 1), ('b', 2), ('c', 3))
```

which returns `[('a','b','c'), ('1','2','3')]`.

You can use argument unpacking with any function:

```
def add(a, b): return a + b

add(1, 2)      # returns 3
add([1, 2])    # TypeError!
add(*[1, 2])   # returns 3
```

It is rare that we'll find this useful, but when we do it's a neat trick.

args and kwargs

Let's say we want to create a higher-order function that takes as input some function `f` and returns a new function that for any input returns twice the value of `f`:

```
def doubler(f):
    def g(x):
        return 2 * f(x)
    return g
```

This works in some cases:

```
def f1(x):
    return x + 1

g = doubler(f1)
print g(3)          # 8 (== ( 3 + 1) * 2)
print g(-1)         # 0 (== (-1 + 1) * 2)
```

However, it breaks down with functions that take more than a single argument:

```
def f2(x, y):
    return x + y

g = doubler(f2)
print g(1, 2)    # TypeError: g() takes exactly 1 argument (2 given)
```

What we need is a way to specify a function that takes arbitrary arguments. We can do this with argument unpacking and a little bit of magic:

```
def magic(*args, **kwargs):
    print "unnamed args:", args
    print "keyword args:", kwargs

magic(1, 2, key="word", key2="word2")

# prints
#  unnamed args: (1, 2)
#  keyword args: {'key2': 'word2', 'key': 'word'}
```

That is, when we define a function like this, `args` is a tuple of its unnamed arguments and `kwargs` is a `dict` of its named arguments. It works the other way too, if you want to use a `list` (or `tuple`) and `dict` to *supply* arguments to a function:

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = { "z" : 3 }
print other_way_magic(*x_y_list, **z_dict)   # 6
```

You could do all sorts of strange tricks with this; we will only use it to produce higher-order functions whose inputs can accept arbitrary arguments:

```
def doubler_correct(f):
    """works no matter what kind of inputs f expects"""
    def g(*args, **kwargs):
        """whatever arguments g is supplied, pass them through to f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
print g(1, 2) # 6
```

Welcome to DataSciencester!

This concludes new-employee orientation. Oh, and also, try not to embezzle anything.

For Further Exploration

- There is no shortage of Python tutorials in the world. The official one is not a bad place to start.
- The official IPython tutorial is not quite as good. You might be better off with their videos and presentations. Alternatively, Wes McKinney's *Python for Data Analysis* (O'Reilly) has a really good IPython chapter.