PythonBasics

January 14, 2015

1 The Basics of Python Programming

In this notebook we will cover the very basics of `Python` programming. A typical way to introduce a new programming languauge is to write what is called a "Hello World" program. This is a simple program that prints the message "Hello World".

In `Python` this looks like the following:

```
In [109]: print("Hello, World!")
```

```
Hello, World!
```

I know, pretty boring. We can add a little bit of code surrounding this single line of `Python` to give us a start writing more complex programs.

```
In [110]: def main(message):
              print(message)

          if __name__ == "__main__":
              main("Hello, World")
```

```
Hello, World
```

This is a standard format that we will be using in this class to write our programs. It consists of a `main` function to control our program. We will learn about functions in a while.

1.1 Introducing Python

`Python` is an easy-to-use yet powerful programming language developed by Guido van Rossum, first released in 1991. In `Python` you can write quick and dirty throw-away small programs. Buy `Python` can also scale up to mission-critical and highly performant applications.

1.1.1 Python Is Easy to Use

The purpose of any programming language is encode the thoughts of a programmer into computer code that can be executed by a computer. There are different categories of languages such as high-level and low-level languages. Examples of high-level languages are `C#`, `Java`, and `Visual Basic`. The distinguishing feature of a high-level language is that it is closer to human language than machine language. `Python` is a high-level language, and among these it is considered one of the simplest and closest to English. This ease of use translates to programmer productivity. `Python` programs are shortern and take less time to write than other programs in other languages.

Examples of lower-level languages are `C`, `C++`, and the newcomer `D` language.

1.1.2 Python in Powerful

`Python` has all the power you might want from a modern programming language. `Python` is powerful enough to be used by companies such as Google, IBM, Industrial Light + Magic, Microsoft, NASA, as well as Goldman Sachs. `Python` is used in the finance industry by asset management firms, investment banks, hedge funds, and high-frequency traders.

1.1.3 Python in Object-Oriented

Object-oriented programming (OOP) is a modern approach to solving problems with computers. OOP is characterized by its use of objects, which are abstract data structures representing objects in the real world. In the OOP paradigm data and the methods that operate on them are coupled together in one concept.

Languages like `C#` and `Java` are also object-oriented, but they force the OOP paradigm on programmers. For small programs this can be quite burdensome. In `Python` OOP is optional. It is not mandatory for small programs, or for situations that the programmer might not find it necessary. But it is there for large programs and when it is needed.

1.1.4 Python Is a "Glue" Language

`Python` can be used together with other programming languages, such as `C/C++`, `Java`, and `Fortran`. This means that a programmer can take advantage of work already done in another language while using `Python`. It also means that he or she can leverage the strengths of other languages, such as the speed that `C` or `C++` might offer, while still enjoying the benefits of working in `Python`. This is how I use `Python` in my own research: I write performance critical pieces in `C++` and glue them together with `Python`. I also prototype new models and algorithms in `Python` and then re-write in `C++` once proven.

Many of the modules that we will make heavy use of are actually written in `Fortran`, `C` or `C++`, although you can use them in `Python` without even being aware of it.

1.1.5 Python in Heavily Used in Scientific Computing

Third-party modules
There is a vast body of Python modules created by the `Python` community. These include utilities for database connectivity, mathematics, statistics, and charting/plotting. Some notable ones that we will be using include:

- ***IPython***: An enhanced Python shell, designed to increase the efficiency and usability of coding, testing and debugging Python. It includes both a Qt-based console and an interactive HTML notebook interface, both of which feature multiline editing, interactive plotting and syntax highlighting.
- ***NumPy***: Numerical Python (NumPy) is a set of extensions that provides the ability to specify and manipulate array data structures. It provides array manipulation and computational capabilities similar to those found in Matlab or Octave.
- ***SciPy***: An open source library of scientific tools for Python, SciPy supplements the NumPy module. SciPy gathering a variety of high level science and engineering modules together as a single package. SciPy includes modules for graphics and plotting, optimization, integration, special functions, signal and image processing, genetic algorithms, ODE solvers, and others.
- ***Matplotlib***: Matplotlib is a python 2D plotting library which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms. Its syntax is very similar to Matlab.
- ***Pandas***: A module that provides high-performance, easy-to-use data structures and data analysis tools. In particular, the `DataFrame` class is useful for spreadsheet-like representation and mannipulation of data. Also includes high-level plotting functionality.

1.1.6 Python Runs Everywhere

`Python` runs on everything from a smartphone to a supercomputer. It is also platform independent. That means that programs you write on your Windows computer will also run on a Linux computer unaltered.

1.1.7 The Python Interpreter

`Python` is an *interpreted* language. The interpreter runs a program by executing one statement at a time. The standard `Python` interpreter can be invoked on the command line with the `python` command:

```
$ python
Python 2.7.9 |Anaconda 2.1.0 (x86_64)| (default, Dec 15 2014, 10:37:34)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>>
```

The >>> you see is the *prompt* where you type expressions. To exit the `Python` interpreter and return to the command line, you can either type `exit()` or press `CTRL-D`.

Running `Python` programs is as simple as calling `python` with a `.py` file as its first argument. Suppose we had created the script **hello.py** with these contents:

```
print("Hello, World!")
```

We can run from the terminal as:

```
$ python hello.py
Hello, World!
```

While a lot of `Python` programmers work this way, a lot programmers - especially in the *scientific* programming world - make use of IPython, an enhanced interactive `Python` interpreter. We will be taking a deeper dive on IPython in a few lectures. For now, you can work in IPython with the following:

```
$ ipython
Python 2.7.9 |Anaconda 2.1.0 (x86_64)| (default, Dec 15 2014, 10:37:34)
Type "copyright", "credits" or "license" for more information.

IPython 2.2.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?         -> 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]: %run hello.py
Hello, World!

In [2]:
```

1.2 Python Language Essentials

1.2.1 Variables and Types

`Python` has a small set of built-in types for handling numerical data, strings of characters, boolean (`True` or `False`) values, as well as dates and times. The following table lists the main types of interest:

Type	Description
`None`	The `Python` "null" value (only instance of the `None` object exist)
`str`	String type. ASCII-valued only in `Python 2.x` and Unicode in `Python 3`

Type	Description
`unicode`	Unicode string type
`float`	Double-precision (64-bit) floating point number. Note there is no separate `double` type
`bool`	A `True` or `False` value
`int`	Signed integer with maximum value determined by the platform
`long`	Arbitrary precision signed integer. Large `int` values are automatically converted to `long`

Numeric Types

The primary `Python` types for numbers are `int` and `float`. The size of the integer which can be stored as an `int` is dependent on your platform (32 or 64-bit), but `Python` will transparently convert a very large integer to `long`, which can store arbitrarily long integers.

```
In [4]: ival = 17239871

In [5]: ival ** 6

Out[5]: 26254519291092456596965462913230729701102721L
```

Floating point numbers are represented with the `Python float` type. Under the hood each one is a double-precision (64-bits) value. They can also be expressed using scientific notation:

```
In [6]: fval = 7.243

In [7]: fval2 = 6.78e-5
```

Strings

Many people use `Python` for its powerful and flexible built-in string processing capabilities. You can write *string literal* using either single quotes ' or double quotes":

```
In [8]: a = 'one way of writing a string'

In [9]: b = "another way"
```

For multiline strings with line breaks, you can use triple quotes, either ''' or """:

```
In [11]: c = """
       This is a longer string that
       spans multiple lines
       """
```

`Python` strings are immutable; you cannot modify a string without creating a new string:

```
In [12]: a = 'this is a string'

In [13]: a[10] = 'f'


    ---------------------------------------------------------------------------
 TypeError                                 Traceback (most recent call last)

    <ipython-input-13-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'


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

```
In [14]: b = a.replace('string', 'longer string')

In [15]: b

Out[15]: 'this is a longer string'
```

Many `Python` objects can be converted to a string using the `str` function:

```
In [16]: a = 5.6

In [17]: s = str(a)

In [18]: s

Out[18]: '5.6'
```

Booleans

1.2.2 Control Flow

if, elif, and else

```
if x < 0:
    print("It's negative")
```

An `if` statement can be optionally followed by one or more `elif` blocks and a catch-all `else` block if all of the conditions are `False`:

```
if x < 0:
    print("It's negative")
elif x == 0:
    print("Equal to zero")
elif 0 < x < 5:
    print("Positive but smaller than 5")
else:
    print("Positive and larger than or equal to 5")
```

If any of the conditions is `True`, no further `elif` or `else` blocks will be reached. With a compound condition using `and` or `or`, conditions are evaluated left-to-right and will short circuit:

```
In [19]: a = 5; b = 7

In [20]: c = 8; d = 4

In [21]: if a < b or c > d:
             print("Made it")

Made it
```

In this example the `c > d` never gets evaluated because the first comparison was `True`.

for loops

`for` loops are for iterating over a collection or an interator. The standard syntax for a `for` loop is:

```
for value in collection:
    # do something with value
```

An example:

```
In [22]: for i in range(10):
             print(i)
```

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

A `for` loop can be advanced to the next iteration, skipping the remainder of the block,using the `continue` keyword. Consider this code which sums up integers in a list and skips `None` values:

```
In [25]: sequence = [1, 2, None, 4, None, 5]
         total = 0
         for value in sequence:
             if value is None:
                 continue
             total += value

         print(total)
```

```
12
```

A `for` loop can exited altogether using the `break` keyword. This code sums elements of the list until 5 is reached:

```
In [27]: sequence = [1, 2, 0, 4, 6, 5, 2, 1]
         total_until_5 = 0
         for value in sequence:
             if value == 5:
                 break
             total_until_5 += value
```

As we will see in more detail, if the elements in the collection or iterator are sequences (tuples or lists, say), they can be conveniently *unpacked* into variables in the `for` loop statement:

```
for a, b, c in iterator:
    # do something
```

while loops

A `while` loop specifies a condition and a block of code that is to be executed until the condition evaluates to `False` or the loop is explicitly ended with `break`:

```
In [28]: x = 256
         total = 0
         while x > 0:
             if total > 500:
                 break
             total += x
             x = x // 2
```

pass

pass is the "no-op" statement in Python. It can be used in blocks where no action is to be taken; it is only required because Python uses whitespace to delimit code blocks:

```
if x < 0:
    print("negative!")
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print("positive!"
```

It's common to use pass as a place-holder in code while working on a new piece of functionality:

```
def f(x, y, z):
    # TODO: implement this function!
    pass
```

Exception Handling

Handling Python errors or *exceptions* gracefully is an important part of building robust programs. In data analysis applications, many functions only work on certain kinds of input. For example, Python's float function is capable of casting a string to a floating point number, but fails with ValueError on improper inputs:

```
In [29]: float('1.2345')
```

```
Out[29]: 1.2345
```

```
In [30]: float('something')
```

```
        ---------------------------------------------------------------------------
    ValueError                                Traceback (most recent call last)

        <ipython-input-30-439904410854> in <module>()
    ----> 1 float('something')


        ValueError: could not convert string to float: something
```

Imagine that we want a version of the float function that fails gracefully, returning the input argument. We can do this by writing a function that encloses the call to float in a try/except block:

```
In [31]: def attempt_float(x):
             try:
                 return float(x)
             except:
                 return x
```

The code in the except part will only execute if float(x) raises an exception:

```
In [32]: attempt_float('1.2345')
```

```
Out[32]: 1.2345
```

```
In [33]: attempt_float('something')
```

```
Out[33]: 'something'
```

Notice that `float` can raise exceptions other than `ValueError`:

```
In [34]: float((1,2))
```

```
    ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-34-a7437c28cf3f> in <module>()
----> 1 float((1,2))


    TypeError: float() argument must be a string or a number
```

range

The `range` function produces a list of evenly-spaced integers:

```
In [35]: range(10)
```

```
Out[35]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Both a start, end, and step can be given:

```
In [36]: range(0, 20, 2)
```

```
Out[36]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

As you can see, `range` produces integers up to but not including the endpoint. A common use of `range` is for iterating through sequences by index:

```
In [2]: seq = [1, 2, 3, 4]
        for i in range(len(seq)):
            val = seq[i]
            print(val)
```

```
1
2
3
4
```

Ternary Expressions

A *ternary expression* in `Python` allows you to put an `if-else` that produces a value to be expressed in a single line or expression. The syntax in `Python` for this is:

```
value = true-expr if condition else
false-expr
```

Here `true-expr` and `false-expr` can be any `Python` expressions. It has the identical effect as the more verbose:

```
if condition:
    value = true-expr
else:
    value = false-expr
```

This is a more concrete example:

```
In [3]: x = 5

In [5]: 'Non-negative' if x >= 0 else 'Negative'

Out[5]: 'Non-negative'
```

Just as with `if-else` blocks, only one of the expressions will be evaluated. While it may be tempting to always use ternary expressions to condense your code, realize that you may sacrifice readability if the condition as well as the true and false expressions are very comples.

1.3 Data Structures and Sequences

`Python` has some simple, but powerful data structures. To become a proficient `Python` programmer one must master their use.

1.3.1 Tuple

A *tuple* is a one-dimensional, fixed-length, *immutable* sequence of `Python` objects. The easiest way to create one is with a comma-separated sequence of values:

```
In [38]: tup = 4, 5, 6

In [39]: tup

Out[39]: (4, 5, 6)

In [40]: nested_tup = (4,5,6), (7,8)

In [41]: nested_tup

Out[41]: ((4, 5, 6), (7, 8))

In [42]: tuple([4, 0, 2])

Out[42]: (4, 0, 2)

In [43]: tup = tuple('string')

In [44]: tup

Out[44]: ('s', 't', 'r', 'i', 'n', 'g')
```

Elements can be accessed with squared brackets [] as with most other sequence types. Like `C++`, and many other languages, sequences are 0-indexed in `Python`:

```
In [45]: tup[0]

Out[45]: 's'
```

Tuples are immutable in `Python`:

```
In [46]: tup[0] = 'S'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-46-d856c5743148> in <module>()
----> 1 tup[0] = 'S'


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

Tuples can be concatenated using the + operator to produce longer tupels:

```
In [47]: (4, None, 'foo') + (6, 0) + ('bar',)

Out[47]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple.

```
In [48]: ('foo', 'bar') * 4

Out[48]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Unpacking tuples

If you try to `assign` to a tuple-like expression of variables, `Python` will attempt to *unpack* the value on the right-hand side of the equals sign:

```
In [49]: tup = (4,5,6)

In [50]: a, b, c = tup

In [51]: b

Out[51]: 5

In [52]: d

Out[52]: 4
```

Even nested tuples can be unpacked:

```
In [53]: tup = 4, 5, (6,7)

In [54]: a, b, (c, d) = tup

In [55]: d

Out[55]: 7
```

One of the most common uses of variable unpacking when iterating over sequences of tuples or lists:

```
In [57]: seq = [(1,2,3), (4,5,6), (7,8,9)]
         for a, b, c in seq:
             pass
```

Another common use is for returning multiple values from a function. More on this later.

Tuple methods

Since the size and contents of a tuple cannot be modified, it is very light on instance methods. One particularly useful one (also available on lists) is `count`, which counts the number of occurrences of a value:

```
In [58]: a = (1,2,2,2,3,4,2)

In [59]: a.count(2)

Out[59]: 4
```

1.3.2 List

In contrast with tuples, lists are variable-length and their contents can be modified. They can be defined using square brackets [] or using the `list` type function:

```
In [60]: a_list = [2, 3, 7, None]

In [66]: tup = ('foo', 'bar', 'baz')

In [67]: b_list = list(tup)

In [68]: b_list

Out[68]: ['foo', 'bar', 'baz']

In [69]: b_list[1] = 'peekaboo'

In [70]: b_list

Out[70]: ['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar as one-dimensional sequences of objects and thus can be used interchangably in many functions.

Adding and removing elements

Elements can be appended to the end of the list with the `append` method:

```
In [71]: b_list.append('dwarf')

In [72]: b_list

Out[72]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Using `insert` you can insert an element at a specific location in the list:

```
In [74]: b_list.insert(1, 'red')

In [76]: b_list

Out[76]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

The inverse operation to `insert` is `pop`, which removes and returns an element at a particular index:

```
In [77]: b_list.pop(2)

Out[77]: 'peekaboo'

In [79]: b_list

Out[79]: ['foo', 'red', 'baz', 'dwarf']
```

Elements can be removed by value using `remove`, which locates the first instance of the value and removes it from the list:

```
In [80]: b_list.append('foo')

In [81]: b_list.remove('foo')

In [82]: b_list

Out[82]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using `append` and `remove`, a `Python` list can be used as a perfectly suitable "multi-set" data structure.

You can check if a list contains a value using the `in` keyword:

```
In [83]: 'dwarf' in b_list

Out[83]: True
```

Concatenating and combining lists

Just as with tuples, adding two lists together with `+` concatenates them:

```
In [84]: [4, None, 'foo'] + [7, 8, (2,3)]

Out[84]: [4, None, 'foo', 7, 8, (2, 3)]
```

If you have a list already defined, you can append multiple elements to it using the extend method:

```
In [85]: x = [4, None, 'foo']

In [86]: x.extend([7, 8, (2,3)])

In [87]: x

Out[87]: [4, None, 'foo', 7, 8, (2, 3)]
```

Note that list concatenation is a comparatively expensive operation since a new list must be created and the objects copied over. Using `extend` to append elements to an existing list, especially if you are building up a large list, is usually preferable. Thus,

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

is faster than

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

sorting

A list can be sorted in-place (without creating a new object) by calling its `sort` function:

```
In [91]: a = [7, 2, 5, 1, 3]

In [92]: a.sort()

In [93]: a

Out[93]: [1, 2, 3, 5, 7]
```

`sort` has a few options that will occassionally come in handy. One is the ability to pass a secondary `sort key`, i.e. a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
In [94]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [95]: b.sort(key=len)

In [96]: b
```

```
Out[96]: ['He', 'saw', 'six', 'small', 'foxes']
```

Binary search and maintaining a sorted list

The built-in `bisect` module implements binary-search and insertion into a sorted list. `bisect.bisect` finds the location where an element should be inserted to keep it sorted, while `bisect.insort` actually inserts the element into that location:

```
In [6]: import bisect

In [7]: c = [1, 2, 2, 2, 3, 4, 7]

In [8]: bisect.bisect(c, 2)

Out[8]: 4

In [9]: bisect.bisect(c, 5)

Out[9]: 6

In [10]: bisect.insort(c, 6)

In [11]: c

Out[11]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Slicing

You can select sections of list-like types (arrays, types, NumPy arrays) by slice notation, which in its basic form consists of `start:stop` passed to the indexing operator `[]`:

```
In [21]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [22]: seq[1:5]

Out[22]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [23]: seq[3:4] = [6, 3]

In [15]: seq

Out[15]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While the element at the `start` is included, the `stop` element is not. The number of elements in the result is `stop - start`.

Either `start` or `stop` can be omitted in which case they default to the start of the sequence and the end of the sequence, respectively

```
In [24]: seq[:5]

Out[24]: [7, 2, 3, 6, 3]

In [25]: seq[3:]

Out[25]: [6, 3, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [26]: seq[-4:]
```

```
Out[26]: [5, 6, 0, 1]
```

```
In [28]: seq[-6:-2]
```

```
Out[28]: [6, 3, 5, 6]
```

A `step` can be used after a second colon to, for example, take every other element:

```
In [29]: seq[::2]
```

```
Out[29]: [7, 3, 3, 6, 1]
```

A tricky way to use this is to pass `-1` which has the effect of reversing a list or tuple:

```
In [30]: seq[::-1]
```

```
Out[30]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Built-in Sequence Functions There are many built-in sequence methods that you want to become very familiar with.

Enumerate

It's common to want to keep track of an index when iterating over a sequence. You can do this yourself with something like:

```
i = 0
for value in collection:
    # do something with value
    i += 1
```

Since this is so common, `Python` has a built-in function `enumerate` to take care of this, which returns a sequence of `(i, value)` tuples:

```
In [31]: collection = ['First', 'Second', 'Third', 'Fourth', 'Fifth']
         for  i, value in enumerate(collection):
             print(i, value)
```

```
0 First
1 Second
2 Third
3 Fourth
4 Fifth
```

sorted

The `sorted` function returns a new sorted list from the elements of any sequence:

```
In [32]: sorted([7, 1, 2, 6, 0, 3, 2])
```

```
Out[32]: [0, 1, 2, 2, 3, 6, 7]
```

```
In [33]: sorted('horse race')
```

```
Out[33]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

zip

`zip` "pairs" up the elements of a number of lists, tuples, or other sequences, to create a list of tuples:

```
In [34]: seq1 = ['foo', 'bar', 'baz']
```

```
In [35]: seq2 = ['one', 'two', 'three']

In [44]: list(zip(seq1, seq2))

Out[44]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip can take an arbitrary number of sequences, and the number of elements it produces is determined by the *shortest* sequence:

```
In [45]: seq3 = [False, True]

In [47]: list(zip(seq1, seq2, seq3))

Out[47]: [('foo', 'one', False), ('bar', 'two', True)]
```

A common use of zip is for simultaneously iterating over multiple sequences, possibly combined with enumerate:

```
In [48]: for i, (a, b) in enumerate(zip(seq1, seq2)):
             print("%d: %s, %s" % (i, a, b))

0: foo, one
1: bar, two
2: baz, three
```

reversed

reversed iterates over the elements of a sequence in reverse order:

```
In [49]: list(reversed(range(10)))

Out[49]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

You can take a look at other built-in functions here:
https://docs.python.org/3/library/functions.html

1.3.3 Dict

dict is likely the most important built-in Python data structure. A moer common name for it is a *hash map*, *hash table*, or *associative array*. It is a flexibly-sized collection of key-value pairs, where key and value are any Python objects. One way to create one is by using curly braces {} and using colons to separate keys and values:

```
In [51]: empty_dict = {}

In [52]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [53]: d1

Out[53]: {'b': [1, 2, 3, 4], 'a': 'some value'}

In [54]: d1[7] = 'an integer'

In [55]: d1

Out[55]: {'b': [1, 2, 3, 4], 'a': 'some value', 7: 'an integer'}
```

You can check if a given dict contains a particular key:

```
In [56]: 'b' in d1
```

```
Out[56]: True
```

Values can be removed in two ways. First using the `del` keyword:

```
In [57]: d1[5] = 'some value'

In [58]: d1

Out[58]: {'b': [1, 2, 3, 4], 5: 'some value', 'a': 'some value', 7: 'an integer'}

In [59]: del d1[5]

In [60]: d1

Out[60]: {'b': [1, 2, 3, 4], 'a': 'some value', 7: 'an integer'}
```

And also using the `pop` method:

```
In [61]: d1['dummy'] = 'another value'

In [62]: d1

Out[62]: {'b': [1, 2, 3, 4],
          'dummy': 'another value',
          'a': 'some value',
          7: 'an integer'}

In [63]: ret = d1.pop('dummy')

In [64]: ret

Out[64]: 'another value'

In [65]: d1

Out[65]: {'b': [1, 2, 3, 4], 'a': 'some value', 7: 'an integer'}
```

You can use the `keys` and `values` methods:

```
In [69]: d1.keys()

Out[69]: dict_keys(['b', 'a', 7])

In [70]: d1.values()

Out[70]: dict_values([[1, 2, 3, 4], 'some value', 'an integer'])
```

You can merge two dicts with the `update` method:

```
In [71]: d1.update({'b' : 'foo', 'c' : 12})

In [72]: d1

Out[72]: {'b': 'foo', 'c': 12, 'a': 'some value', 7: 'an integer'}
```

Creating dicts from sequences

A common situation is to have two different sequences that you would like to pair up element-wise. A naive approach would be something like:

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

A more pythonic way to do would be:

```
In [73]: mapping = dict(zip(range(5), reversed(range(5))))

In [74]: mapping

Out[74]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Valid dict key types

While values of a dict can be any `Python` object, the keys have to be immutable objectslike scalar types (int, float, string) or tuples (all the objects in the tuple need to immutable, too). The technical term for this is *hashability*. You can check for it:

```
In [75]: hash('string')

Out[75]: 6620633730282082880

In [76]: hash((1, 2, (2, 3)))

Out[76]: 1097636502276347782

In [77]: hash(1, 2, [2, 3])


        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-77-f748ae008c53> in <module>()
    ----> 1 hash(1, 2, [2, 3])


        TypeError: hash() takes exactly one argument (3 given)
```

If for some reason you have a list that you want to use a key, first convert it to a tuple:

```
In [78]: d = {}

In [79]: d[tuple([1, 2, 3])] = 5

In [80]: d

Out[80]: {(1, 2, 3): 5}
```

1.3.4 Set

A `set` is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the `set` function or using a `set literal` with curly braces:

```
In [81]: set([2, 2, 2, 1, 3, 3])

Out[81]: {1, 2, 3}

In [82]: {2, 2, 2, 1, 3, 3}
```

```
Out[82]: {1, 2, 3}
```

Sets support mathematical set operations:

```
In [83]: a = {1, 2, 3, 4, 5}

In [84]: b = {3, 4, 5, 6, 7, 8}

In [85]: a | b # union (or)

Out[85]: {1, 2, 3, 4, 5, 6, 7, 8}

In [86]: a & b # intersection (and)

Out[86]: {3, 4, 5}

In [87]: a - b # difference

Out[87]: {1, 2}

In [88]: a ^ b # symmetric difference

Out[88]: {1, 2, 6, 7, 8}

In [89]: a_set = {1, 2, 3, 4, 5}

In [90]: {1, 2, 3}.issubset(a_set)

Out[90]: True

In [91]: a_set.issuperset({1, 2, 3})

Out[91]: True

In [92]: {1, 2, 3} == {1, 2, 3}

Out[92]: True
```

See page 417 of *Python for Data Analysis* for more set functions.

1.3.5 List, Set, and Dict Comprehensions

List *comprehensions* are one the most beloved `Python` language features. They allow you to concisely form a new list in one expression:

```
[expr for value in collection if condition]
```

This is equivalent to the following `for loop`:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

For example, given a list of strings we could filter out those that are length 2 or less and simultaneously convert them to uppercase as follows:

```
In [93]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [94]: [x.upper() for x in strings if len(x) > 2]

Out[94]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

`set` and `dict` comprehesions are similar:

```
dict_comp = {key-expr : value-expr for value in collection if condition}

set_comp = {expr for value in collection if condition}
```

1.4 Functions

Functions are the primary and most important method of code organization and reuse in `Python`. As you have seen with my example above in the "Hello, World!" program to define a function you use the `def` keyword:

```
In [111]: def add_two(x, y):
              return x + y

In [96]: add_two(15, 23)

Out[96]: 38
```

Each function can have some number of *positional* arguments and some number of *keyword* arguments. Keyword arguments are usually used to specify default values for arguments.

```
In [100]: def add_three(x, y, z=3):
              return x + y + z

In [101]: add_three(1,2)

Out[101]: 6

In [102]: add_three(1, 2, 5)

Out[102]: 8
```

Most of our work with financial modeling will consist of writing functions. A standard thing will be to have a `main` function that controls the calling of other functions:

```
In [113]: def greet(message):
              print(message)

          def main():
              name = input("Please type your name: ")
              msg = "Hello, {}!".format(name)
              greet(msg)

          if __name__ == "__main__":
              main()

Please type your name: Parker
Hello, Parker!

In []:
```