A Crash Course in Python for Data Science

Python is a high-level, general-purpose, dynamic and interpreted scripting language. It has widespread applications in developing software, building web applications, scientific computing and data analysis. A lot of people starting out in Data Science often spend too much time trying to learn Python as a scripting language. I wouldn't recommend going down that road.

In my opinion, it is not necessary to become a proficient Python developer to do be productive as a data scientist. This chapter is therefore intended to serve as a guide for you to learn the most pertinent and helpful features of Python that will enable you to process and analyze structured and unstructured data rapidly and effectively. After all, for a data scientist, a programming language is just another tool to express their ideas and analyses in. It thus becomes critical here to define the scope of work that data science comprises of.

Hilary Mason, Data Scientist at bit.ly, when asked by a journalist *But what does a data scientist really do?* replied with a clever acronym: *OSEMN things*, she said (pronounced 'awesome'.)

Here's what those characters stand for

- O is for **O**btaining Data
- S is for **S**crubbing
- E is for **E**xploratory Analysis
- M is for **M**odel Building
- N is for i**N**terpretation and Deployment

Though this list doesn't dive into too much detail and omits a few nuances, it's a great summary of the data science process.

We will now take a look at Python data types, data structures (and their methods) and other important building blocks that help us in accomplishing the *awesome* things listed above. We will focus largely on the elements of Python that you need to know to be able to *read code* and *understand what it is doing* in order to to move smoothly through the following chapters.

Getting Python and the Jupyter Notebook

Over the past few years, it has become increasingly simpler to set up the Python Data Science stack on different operating systems with the advent of meta-packages like Continuum Analytics' *Anaconda* and Enthought's *Canopy*. This section will guide you through the process of setting your machine up to start punching Python code as quickly as possible.

But first - Python 2 or Python 3?

As of the writing of this book, the latest official release of Python is 3.5, but we will be using Python 2.7 for its reliability and superior support. Many packages for data science work well with Python 2.7 and most of the community uses this version. So we will too.

Option 1: `Anaconda` and `conda` (*recommended*)

The quickest and hassle-free way to get up and running with the Python Data Science stack is by using Continuum Analytics' bundled distribution called *Anaconda*. It is free for personal and enterprise use and consists of 150+ libraries and a very capable package manager called *conda*.

You may download the latest version from *https://www.continuum.io/downloads* (Make sure that you choose Python 2.7)
Follow the instruction given on that page and execute the steps for your particular OS.

Once installed, open up the Terminal/Command Prompt and run `$ conda update conda` to receive the most recently updated list of conda's packages.

Then, to install a new library with conda, just run: `$ conda install <package>`

When you're ready, run `$ jupyter notebook` and the notebook will launch on your default browser.

Option 2: Base `Python` and `pip`

If you choose not to use Anaconda, you can download Python from the *www.python.org/downloads*. Ensure that you have `pip` installed which is the package manager for Python and will enable you to easily install 3rd party packages that you'll need to perform data science tasks.

> Note that `pip` is automatically installed if you're using Python 2.7.9 or later.

To install packages with `pip`, the process is fairly simple, just write `pip install <package(s)>` on the command-line interface of your operating system.

Here's an example that will install the most important core data science libraries for you that we will use throughout this text.

```
$ pip install ipython ipython-notebook numpy scipy pandas matplotlib scikit-learn
```

That's it. Now you and your system are ready to follow along the chapters in this book.

Bon voyage!

The Jupyter Notebook

Note that Python is an **interpreted** language, which means that the code is executed *line-by-line*. This allows for an interactive programming, which is critical for exploratory data science. Depending on the task at hand, Python allows you the flexibility to use it in one or more of several ways.

- The **Python Interpreter** is a REPL (Read Eval Print Loop) that presents the user with a `>>>` prompt, making it very easy to run and test small snippets of code.
- The **IPython Shell** is richer than the basic interpreter as it provides development enhancements like command history, auto-complete suggestions and code/object introspection.
- **Scripts** allow you to bundle complex logic in files saved with the `.py` extension, which can be run all at once from the CLI using `$ python my-script.py`
- The **Jupyter Notebook** is web-based interface to use IPython that allows authors to create engaging documents that combine live code, narrative text, LaTeX equations, HTML objects (images, sound and videos) and even

interactive widgets. These notebooks can then be shared in static HTML or dynamic versions with collaborators over email, cloud-storage or the excellend NBviewer service.

Advantages

1. Support for 40+ languages (including Julia, Python, R, Scala and Spark) through kernels
2. In-line visualizations and interactive widgets
3. Support for distributed computing
4. Code annotations with Markdown
5. Create slideshows directly from a Notebook (no more PowerPoint!)

An instance of the Jupyter Notebook can be started by typing

```
$ jupyter notebook
```

on the CLI.

This will start two things:

A ***backend* kernel** process that will handle the execution of your code.

- `IPython` by default, but could be one of many options (R, Scala, Julia etc)

A ***frontend* web application** (with your default browser) that allows you to edit code.

- Allows you to execute CLI commands and write code in other languages using `magics`

Next Steps

> - Open up your browser window, and enter the following URL: `localhost:8888/tree`
> - A tab will load, showing the contents of your root directory (or your current working directory)
> - Navigate to the folder where you want to save your work and data files
> - Hit the `New` button over on the far right top corner
> - Choose `Python2`
> - The notebook will load. As soon as it says 'kernel ready', you can start writing Python code in the cells!

Please visit **http://jupyter.org** to familiarize yourself with the latest and greatest features of Jupyter.

---

3. Syntax, The Zen of Python and the 'Pythonic Way' of doing things

Python's design philosophy emphasizes simplicity, readability, explicitness and brevity of syntax (you'll find yourself writing fewer lines of code for expressing the same ideas than in languages like Java or C++).

Let's look at some sample code:

```
# Declare a list of numbers
some_data = [2, 15, -6, 28, 39, 0, 52]

# Define a function
def adder(x):
    """
    This function takes as input a list of numbers, and prints whether their sum is odd or even.
    It returns the sum.
    """
    sum = 0

    # Loop over the list of numbers
    for i in x:
        sum += i

    if sum % 2 == 0:
        print "The sum is even."
    else:
        print "The sum is odd."

    return sum

some_data_sum = adder(some_data)
print some_data_sum
```

This produces the following output

```
The sum is even.
130
```

The script doesn't do much, but then it's only meant to higlight some important aspects of Python syntax.
Now let's go over the code line-by-line and understand what's happening.

- Comments begin with a `#` sign
- `some_data` is a **variable** name.
- The comma-separated numbers inside square brackets `[ ]` make up a Python `list`
- The `=` creates a binding between the variable name and the `list` of numbers. This is also called an `assignment.`
- Functions are defined with the `def` keyword. The name of this function is `adder`. Functions can optionally `return` things.
- Triple quotes are used for multi-line comments. It is customary to include a `docstring` with Python functions that describes what it does.
- Blocks of code (for-loop, if-else block) are denoted using *indentation* and not curly braces (unlike R, Java, C++)
    - This is critical as it enforces a style of syntax that makes most Python code looking cosmetically similar and hence improves readability.
- A colon `:` denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block
- The `for` loop iterates over a list of numbers
- The `if else` statement is used to check logic. (The `%` operator finds the remainder of a division.)
- Python statements aren't terminated using semi-colons `;`
    - Semi-colons can, however, be used to put multiple statements on one line.

The Zen of Python

Run `import this` on the Python interpreter (or inside a Jupyter Notebook cell).

Go ahead, do it.

You'll find a list of tenets laid down by Tim Peters that describe the philosophy of the creators of Python. One of these, is often the subject of lengthy discussions:

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

Code written in accordance with this “obvious” way is often described as being **Pythonic**.

On community-driven forums like `Stack Overflow`, you will often find questions to the tune of - *Is there a Pythonic way of doing X?* asked by exasperated programmers who feel that the code they've written is ugly and/or too complex. Though this may seem myterious right now, as you read and write more Python code, it will become evident to you and you will begin to leverage features of the language in the fabled *obvious* way.

As in any other language, there are often multiple ways of doing the same thing in Python, and whenever faced with such a choice, we will favor the Pythonic way over others.

Getting help on-the-go

Never fret or despair when you're learning Python, for help is always at hand! The Official Python documentation is built into the Jupyter ecosystem. Whenever you find yourself stuck or unclear on what a function does or whether you're typing the right syntax, just call for help by appending a `?` Question mark at the end of the object, and hit `Shift+Enter`

For example, let's assume you don't know what the `id()` function does. You may simply write

```
id?
```

and the function's docstring will be displayed for you to read through and take note of things such as

- What arguments does the function take?
- What is the data type of the returned object?

If you need even more information on a function or object, try using a double-question mark `??`

Advice for programming newbies

To get started with learning any new OOP language, the very first steps you have to take include learning about:

- variables, values and objects
- data types
- data structures
- arithmetic, control and logical operators

Remember that learning a new programming language is much like learning a new foreign language. You cannot claim to know the new language until you have:

- built a decent vocabulary (conversational, at least)
- can translate/express ideas effectively

The following sections will take you through the core of the language - build your vocabulary (commit certain things to memory) as you go along and try to express concepts that you know (for example, generating numbers from the Fibonacci sequence) using what you learn.

---

4. Built in Data Types and their Methods

Python has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and date/time values. These are the quintessential building blocks for storing and manipulating data in Python.

Python primitives

- `None` - The Python Null Value
- `str`, `unicode` - for strings
- `int` - signed integer whose maximum value is platform dependent.
- `long` - large ints are automatically converted to long
- `float` - 64-bit (double precision) floating point numbers
- `bool` - a True or False value

Checking the data type of an object using `type()` or `isinstance()`

The Python function `isinstance()` takes as input two things - an object and a type. It returns a True if the object belongs to the type specified.

```
In [30]: isinstance(2, int)
Out[30]: True

In [31]: isinstance(2.2, float)
Out[31]: True

In [32]: isinstance('hello', str)
Out[32]: True

In [33]: isinstance([1, 2, 3], list)
Out[33]: True

In [34]: isinstance(4 == 5, bool)
Out[34]: True
```

You can call `type()` on any Python object to know its type.

```
In [35]: type(123)
Out[35]: int

In [36]: type(12.34)
Out[36]: float

In [37]: type(None)
Out[37]: NoneType

In [38]: type(False)
Out[38]: bool

In [39]: type([1, 2, 3])
Out[39]: list
```

The Numeric types

`int` creation and methods

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

```
 In: x = 5
 In: type(x)
Out: int
```

Pressing `<tab>` following a dot after the `x` would show you all the methods associated with objects of type `int`

```
x.bit_length   x.conjugate    x.denominator     x.imag         x.numerator    x.real
```

Here, `.bit_length()`, for example, gives you 'the number of bits necessary to represent self in binary.'

Learn more about each of these by running commands like

```
x.conjugate?
```

`float` creation and methods

Any number with a decimal point is stored internally as an instance of type `float`. These can be defined in standard or scientific notation.

```
  In: y = 3.142; z = 3e8
  In: type(y), type(z)
 Out: (float, float)
```

Pressing `<tab>` following a dot after the `x` would show you all the methods associated with objects of type `float`

```
y.as_integer_ratio  y.fromhex          y.imag             y.real
y.conjugate         y.hex              y.is_integer
```

Learn more about each of these by running commands like

```
y.is_integer?
```

Conversion

`int`s can be converted into `float`s using the `float()` function, and vice-versa

```
 In: float(x)
Out: 5.0

 In: int(y)
Out: 3
```

Numbers stored as text can also be converted to `int`s or `float`s

```
In : int('1123')
Out: 1123

In : float('3.142')
Out: 3.142
```

Division

By default, integer division in Python is going to return an `int` value.

```
 In: 55/7
Out: 7
```

To modify this behavior, you have two options - to affect a global or a local change. You may import the default division behavior of Python 3, or you may simply convert the denominator to a float.

```
# Globally
from __future__ import division

# Locally
 In: 55/7.0
Out: 7.857142857142857

 In: 55/float(7)
Out: 7.857142857142857
```

The str type

`str` creation and methods

A lot of programmers favor Python owing to its flexible and powerful built-in string processing capabilities.

Objects of type `str` are created with the single `'` or double `"` quotes.

```
str_1 = 'This is a string'
str_2 = "This too is a string."
```

Properties

- Strings are **immutable**; it is impossible to modify a string in-place (you would have to make a copy)
- Strings are **iterables**; you can run loops over strings.

Methods

To view a full list of the methods available to objects of type `str`, fire up the IPython Shell and create an `str` object. Place a dot after the object name, and hit `<tab>`

```
In [1]: str_1.
str_1.capitalize  str_1.find        str_1.isspace     str_1.partition   str_1.rstrip      str_1.
str_1.center      str_1.format      str_1.istitle     str_1.replace     str_1.split       str_1.
str_1.count       str_1.index       str_1.isupper     str_1.rfind       str_1.splitlines  str_1.
str_1.decode      str_1.isalnum     str_1.join        str_1.rindex      str_1.startswith
str_1.encode      str_1.isalpha     str_1.ljust       str_1.rjust       str_1.strip
str_1.endswith    str_1.isdigit     str_1.lower       str_1.rpartition  str_1.swapcase
str_1.expandtabs  str_1.islower     str_1.lstrip      str_1.rsplit      str_1.title
```

As always, go forth and explore the different methods and their arguments by placing a `?` after the method name and hitting `Shift + Enter`

```
<str_obj_name>.<method>?
```

Here, I shall show you a few of the more useful methods

```
In [4]: str_1.upper()                  # convert to uppercase
Out[4]: 'A QUICK BROWN FOX'

In [5]: str_1.replace('fox', 'dog')    # replace values
Out[5]: 'a quick brown dog'

In [6]: str_1.split(' ')               # split string on a delimiter. returns a list.
Out[6]: ['a', 'quick', 'brown', 'fox']

In [7]: str_1.find('quick')            # returns index of first instance of a character/word
Out[7]: 2

In [8]: str_1.count('o')               # returns count of a character
Out[8]: 2

In [9]: str_1.endswith('ox')           # returns boolean if patterns match
Out[9]: True
```

String Arithmetic

The `+` operator concatenates strings, and the `*` operator creates copies
[Note] I've included some whitespaces inside the strings to enhance readability.

```
In [11]: str_1 + ' jumps over the' + ' lazy dog'
Out[11]: 'a quick brown fox jumps over the lazy dog'

In [14]: 'mango ' * 5
Out[14]: 'mango mango mango mango mango '
```

A note on object Mutability (and Immutability)

In CS terminology, we say that an object is

- **Mutable** when it can be 'modified in place', i.e. we can alter/change/mutate the contents of a variable without changing its location in memory.
- **Immutable** objects are those that cannot be modified once they have been declared. Functional Programming languages like Scala prefer immutable code. When you try to modify an immutable object, a copy is created.

> [Note]
> The `id()` function can be used to retrieve the memory address of an object (guaranteed to be unique for objects existing simultaneously in the interpreter.)
> The `cast()` function from the `ctypes` module can retrieve the contents of a memory address.

First we look at Python **strings** - an *immutable* type.

In the example below, trying to change the first character of the string `'hello'` from `'h'` to `'H'` gives you an error. This is because strings cannot be modified in place. The int `4859938208` represents the memory address of the location where `x` *points to* and where `'hello'` is *physically stored*.

```
In[1]:
x = 'hello'
print id(x)
x[0] = 'H'

4859938208

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

However, if we try something like this

```
In[2]:
x = 'hello'
memloc_1 = id(x)
print "Before: x points to the string '{}', stored at {}".format(x, memloc_1)

x += 'world'
memloc_2 = id(x)
print "After : x points to the string '{}', stored at {}".format(x, memloc_2)

import ctypes
print "Stored at {}, is '{}'".format(memloc_1, ctypes.cast(memloc_1, ctypes.py_object).value)
print "Stored at {}, is '{}'".format(memloc_2, ctypes.cast(memloc_2, ctypes.py_object).value)

Out[2]:
Before: x points to the string 'hello', stored at 4859938208
After : x points to the string 'helloworld', stored at 4859940176
Stored at 4859938208, is 'hello'
Stored at 4859940176, is 'helloworld'
```

Here, when we concatenate another string, `'world'`, to our string `x`, a copy is created and the new string is added to it. Note how the memory address changes too (from `4859938208` to `4859940176`). Thus, the new string is stored at an entirely different location.

In summary, `mutable` objects are those that cannot be modified in-place. You can, however, use them as a building block in another object, in which case the new object will be stored at a new memory location.

Now let's use a **list**, which is a *mutable* container.

```
In[2]:
 # Declare x as a list, and make x and y point to it
x = [1, 2, 3]
y = x

print "Before:\nx points to {}, and y points to {}".format(id(x), id(y))
print "x contains {} and y contains {}".format(x, y)

 # Modify y alone
y += [3, 2, 1]

print "After :\nx points to {}, and y points to {}".format(id(x), id(y))
print "x contains {} and y contains {}".format(x, y)

Out[2]:
Before:
x points to 4362771848, and y points to 4362771848
x contains [1, 2, 3] and y contains [1, 2, 3]

After :
x points to 4362771848, and y points to 4362771848
x contains [1, 2, 3, 3, 2, 1] and y contains [1, 2, 3, 3, 2, 1]
```

We see that any change to `y` also modifies `x`, and all mutations happen **in-place**. As a general rule, remember (and this might be true for most OOP languages)

> Primitive types are probably immutable. Container-like types are probably mutable.

Python Semantics

Here I present to you two fundamental concepts of Python. Understanding the first well will empower you to do great Pythonic things and taking heed of the second will help you avoid common errors.

Everything in Python is an `Object`

Much like other OOP languages, Python's object model is remarkably consistent. Every number, string, data structure, function, class, module etc. exists inside the interpreter in its own "box" which is referred to as a Python **object**.

Each object in Python has the following things associated with it.

- a data **type** (for example, 'string' or 'function')
- internal **data**
- some metadata (for example, `shape,) called **attributes**
- some functions (for example, `count, join, append`) called **methods** that have access to the internal data
    - In other words, a `method` is a function that “belongs to” an object

These can be accessed using the dot `<obj-name>.<attribute-or-method>` syntax.
Methods are followed by a `()` as they mostly take additional parameters.

> Though we won't discuss `classes` here right now, it is important to remember that every declared object belongs to a class, and **inherits** traits declared inside that class.

Python uses `Pass by Reference` by default

This concept might be tricky to grasp even for people with experience in other languages, and outright bewildering for programming novices.

So, pay very careful attention here.

> Python is *dynamically typed*.
> This means that you don't have to declare the type of a variable before you assign a value to it.

Through assignment statements we create something called a '*binding*' where a Python name (a variable) is associated with an object (a value.) This binding is (in CS jargon) a **pointer**, which (in plain English) means that it stores the *address of some value in memory* and not the value itself.

Since its only 'holding references' or 'pointing to' objects in memory, it is possible to do things like

```
var_1 = 3.142
var_1 = "Huskies"
var_1 = ['a', 'e', 'i', 'o', 'u']
```

Statically-typed languages like Java or C, you'd have to write things like `int num_1 = 12;` Once declared, `num_1` cannot hold an object or value of any other type than `int`.

And, now comes the fun part.

When we're dealing **mutable** objects, if we have two variable referring to the same object, modifying one would also change the value for other.

Observe:

```
In[1]:

 # Declare a list and assign it to another
var_1 = ['a', 'e', 'i', 'o', 'u']
var_2 = var_1

print "Before Modification | Var_1: {}, and Var_2: {}".format(var_1, var_2)

 # Change the first list
var_1.append('z')

print "After Modification | Var_1: {}, and Var_2: {}".format(var_1, var_2)

Out[1]:

Before Modification:
Var_1: ['a', 'e', 'i', 'o', 'u'], and
Var_2: ['a', 'e', 'i', 'o', 'u']
After Modification:
Var_1: ['a', 'e', 'i', 'o', 'u', 'z'], and
Var_2: ['a', 'e', 'i', 'o', 'u', 'z']
```

You will see that even though you only appended an object (a character) to `var_1`, this changed has also echoed in `var_2` which was pointing at the same location in memory. Well, that was kind of obvious if you thought about it in terms of memory locations and addresses.

Remember,

> Pointers hold references to objects in memory, not the objects themselves.

Okay let's break it down.

- Say the list `['a', 'e', 'i', 'o', 'u']` was stored at the memory location `7ec58d1f968c`.
- Both `var_1` and `var_2` were pointing to this address.
- When we modified `var_1`, we changed what it was pointing to, i.e. the list `['a', 'e', 'i', 'o', 'u']`
- The same location now stored a *mutated* version of the original list, pointed at by `var_1` and `var_2`
- So effectively `var_2` was just an innocent witness to all of this.

Now, if we were to bind `var_1` to a totally different location, what do you think would happen to `var_2`?

```
In[2]:
var_1 = ['x', 'y', 'z']
print "After Reassignment:\nVar_1: {}, and \nVar_2: {}".format(var_1, var_2)

Out[2]:
After Reassignment:
Var_1: ['x', 'y', 'z'], and
Var_2: ['a', 'e', 'i', 'o', 'u', 'z']
```

Why is this important, you ask?

Well, because when you're working with big data and you have fifty variables to keep track of and a mix of mutable and immutable objects in memory, it might get quite annoying if you couldn't figure out why a particular data structure had changed seemingly all on its own. I don't want you to pull your hair out trying to investigate inadvertent mutations (also called *side effects*).

Each time you find yourself writing things like `this_var = that_var`, stop and think about the consequences.

---

5. Data Structures and Methods - Tuples, Lists, Dictionaries, Sets

Python Data Structures are simple, but quite powerful. Understanding them well and mastering their use is critical for a programmer to write efficient code for doing data science.

Here we will learn about Tuples, Lists, Dictionaries and Sets - each of which are characterized by

- how data is stored in them
- methods that work on them
- use-cases they're most suitable for

Tuples `tuple`

A tuple is a sequence of Python objects that is

- one-dimensional,
- fixed-length,
- immutable.

Creation: We can create tuples in two ways

- A comma-separated sequence of values assigned to a variable (optional: placed inside parentheses)
- Calling `tuple()` on any sequence/iterator (eg. a `list`)

Syntax:

```
tup = 1, 2, 6
nested_tup = (1, 3, 5), (2, 8), [‘a’, ‘b’])

tuple([1, 2, 7])
tuple(‘forests’)
```

Subsetting: Elements can be accessed with square brackets `[]`, with indexes beginning with 0.

```
nested_tup[0]
```

Immutability: Once a tuple is created, it's not possible to modify which object is stored at an index.

```
nested_tup[1] = (1, 3)

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

> [Note]
> The objects stored in the tuple (eg. a list) might be mutable, so they can be altered (but not moved.)

```
nested_tup[2].append('c')
```

This line of code would not produce an error because the element stored at index 2 of `nested_tup` is a list, which we know to be a *mutable* object.

Concatenation: The '+' operator joins tuples to form longer tuples.

```
(4, None, 'foo') + (6, 0) + ('bar',)
```

Tuple Unpacking: In an assignment statement, corresponding elements of a tuple will be assigned to respective objects on the right-hand-side (given that the number of objects is the same as length of the tuple.) This makes is very easy to swap variables, a task that requires a third variable in other languages.

```
a, b, c = (1, 2, 3)      # equivalent to writing a=1; b=2; c=3
a, b = b, a              # a and b swap values
```

Tuple Methods: Press `<tab>` following a dot after the tuple object. Though there aren't too many Tuple methods, `count` and `index` are useful ones.

```
In:
tup_1 = ('a', 'p', 'p', 'l', 'e')
print tup_1
print tup_1.count('p')
print tup_1.index('l')

Out:
('a', 'p', 'p', 'l', 'e')
2
3
```

Lists `list`

A Python list is simply an **ordered** collection of values or objects. It is similar to what in other languages might be called an array, but with some added functionality. Lists are

- one-dimensional
- containers for collections of objects of any type

- variable-length
- mutable, ie, their contents can be modified in-place

Creation: Lists can be defined using

- square brackets `[]` or
- using the `list()` function
- Python functions that produce lists (such as `range`), and Comprehensions

```
int_list = [1, 2, 3]
mixed_list = ["string", 0.1, True]
list_of_lists = [int_list, mix_list, ['A', 'B']]
x = range(10)
```

Subsetting: Single elements can be accessed using their index or position

```
x[4]     # fetches the 5th element
x[-1]    # fetches the last element
```

Subsets of lists (smaller lists) can be accessed using **integer slicing**

```
x[:4]       # first four elements
x[4:]       # all elements from fourth to the end
x[1:5]      # second to fifth
x[-3:]      # last three
x[::-1]     # reverse the list
```

List Methods: List objects in Python have convenient methods to append, remove, extract, insert and sort elements.

Adding elements

`.append()`, adds a single element at the end of the list.

```
x = ['x', 'y', 'z']
x.append('a')
```

`.extend()`, adds multiple elements at the end of an existing list.

```
x.extend([10, 11, 12])
```

[Note] Lists can be also be combined/concatenated using the `+` operator.

```
[1, 2, 3] + ['a', 'b', 'c']
```

`.insert()`, to insert elements at a specific index (this is an expensive operation)

```
x.insert(3, 'a')
```

Removing elements

`.pop()`, removes and returns an element at a particular index (default: from the end)

```
x.pop()
```

`.remove()`, takes an element as input and removes its first occurrence

```
x.remove(5)
```

Sorting

`.sort()` is used to sort a list in the ascending order (default action) in-place

```
In[1]:
x = range(10, 1, -1)
print "Before Sort:", x
x.sort()
print "After  Sort:", x

Out[1]:
Before Sort: [10, 9, 8, 7, 6, 5, 4, 3, 2]
After  Sort: [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The `reverse=True` parameter sorts the list in reverse order.

```
In[2]:
y = list('abcde')
print "Before Sort:", y
y.sort(reverse=True)
print "After  Sort:", y

Out[2]:
Before Sort: ['a', 'b', 'c', 'd', 'e']
After  Sort: ['e', 'd', 'c', 'b', 'a']
```

List Functions In addition to *methods*, Python also has a few **general-purpose functions** that work with lists.

```
x = list('just a string')
```

`len()`, returns the number of elements in the list

```
 In[1]: len(x)
Out[1]: 13
```

`in()`, checks whether an element belongs to a list and returns a `boolean`

```
 In[2]: 't' in x
Out[2]: True
```

`sorted()`, returns a new list from the elements of given list, sorted in ascending order (default action)

```
 In[3]: sorted(x)
Out[3]: [' ', ' ', 'a', 'g', 'i', 'j', 'n', 'r', 's', 's', 't', 't', 'u']
```

`reversed()`, yields an iterator to go over the list in reverse order

```
 In[4]: reversed(x)
Out[4]: <listreverseiterator at 0x1041af310>
```

List Unpacking Works a lot like `tuple` unpacking, where you can assign list elements to objects using an assignment statement.

```
a, b, c, d = [1, 2, 3, 4]
```

Dictionary `dict`

`dict` is likely the most important built-in Python data structure. It is a **flexibly-sized unordered mutable collection** of `key-value pairs`, where key and value are Python objects. We frequently use dictionaries as a simple way to represent **structured data**.

```
doc_info = {
    "author" : "dushyantk",
    "title" : "Data Science in Python",
    "chapters" : 10,
    "tags" : ["#data", "#science", "#datascience", "#python", "#analysis"]
}
```

Creation: `dicts` are created using curly braces `{}` and using colons `:` to separate keys and values.

```
empty_dict = {}
a_dict = {‘key_1’:12, ‘key_2’:36}
b_dict = {'a': 'a string', 'b' : [1, 2, 3, 4]}
```

[Note] We can check if a `dict` contains a key using the in keyword

```
 In[1]: ‘d’ in b_dict
Out[1]: False
```

Subsetting: We can access or insert the value/object associated with a key by using the curly brackets

```
 In[2]: b_dict['a']
Out[2]: ‘a string’

 In[3]: b_dict[‘c’] = 3.142; b_dict
Out[3]: {'a': 'a string', 'b': [1, 2, 3, 4], 'c': 3.142}
```

`dict` **Methods**

`.keys(), .values(), .items()` will return the keys, values, key-value pairs of the `dict`

```
In[1]:
print "Keys  :", a_dict.keys()
print "Values:", a_dict.values()
print "Pairs :", a_dict.items()

Out[1]:
Keys  : ['key_1', 'key_2']
Values: [12, 36]
Pairs : [('key_1', 12), ('key_2', 36)]
```

Removing Elements

`del my_dict[key]` will remove the key-value pair associated with passed key. It doesn't return anything.

```
del b_dict[‘a’]
```

`.pop()` - works in much the same fashion and returns a value.

```
 In[2]: b_dict.pop(‘c’)
Out[2]: 3.142
```

Merging Dicts

`.update()` will merge two given dictionaries. Doesn't return a value.

```
b_dict.update(a_dict)
```

Handling Missing Keys

`.get()` is used to fetch a value from a `dict` given a key, and behaves gracefully for missing keys. If the key is found, it returns the associated value. It returns a default value if the key isn’t contained in the dict. This ensures that an exception is not raised.

```
 In[3]: b_dict['d']

-------------------------
KeyError: 'd'

 In[4]: b_dict.get(‘d’, ‘Nothing found’)
Out[4]: 'Nothing Found'
```

Sets `set()`

A set is an **unordered collection** of **unique** elements. They can be thought of being like dicts, but keys only - no values.

Creation: A set can be created in two ways:

Using the `set()` function,

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

Using a `set` literal with curly braces:

```
 In: {2, 2, 2, 1, 3, 3}
Out: set([1, 2, 3])
```

Sets support mathematical set operations like union, intersection, difference, and symmetric difference.

Examples

```
In:
setx = {1, 2, 3, 4}
sety = {3, 4, 5, 6}

print setx.union(sety)
print setx.intersection(sety)
print setx.difference(sety)

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

Searching for elements within a collection

We know that the `in` keyword can be used to find if an item belongs to a collection.

> Note that checking whether a `list` contains a value is a **lot slower** than `dicts` and `sets`

This is because Python makes a *linear scan* across the values of the list, whereas the others (set and dicts based on hash tables) can make the check in constant time.

- `list` - linear scan, so longer for long lists
- `set` and `dict` - hash table scan, so constant time

Example

```
In:
list_1 = range(10000)
set_1 = set(list_1)

 In: %timeit 1123 in list_1
Out: 100000 loops, best of 3: 23.8 µs per loop

 In: %timeit 1123 in set_1
Out: 100000 loops, best of 3: 194 ns per loop
```

Here we use the `%timeit` function to measure how long it takes for each statement to run. As we can see, searching for an item in a collection is many orders of magnitude faster for sets than for lists.

This can lead to significant speedup in code execution time, especially in big data applications.

---

6. Conditional Operators

Booleans are produced in Python when we

- Compare objects using binary operators like `==`, `!=`, `>`, `<`, `<=`, `>=`, `in`, `is` and so on
- Evaluate conditional expressions like `a > 5 & b < 0`
- Use special funtions and methods like `isinstance`, `islower`, `is_integer` etc
- Calling `bool()` on objects of various types

The two boolean values are `True` and `False`. Note that they are case-sensitive.

The `bool()` function and Truthiness

All Python objects have an inherent notion of their True- or Falseness. These can be found by calling `bool()` on the object.

The `bool()` function

- when called on any non-empty string will always return `True`
- when called on integers (other than 0 and 0.0) will return `True`
- when called on `None` will return `False`
- when called on any Python objects will return a `False` if the object is empty
    - For example, all of the following expressions will evaluate to `False`

```
    bool("")    # empty string
    bool([])    # empty list
    bool({})    # empty dict
```

This property can be very useful if you want to conditionally break out of a loop when a data structure becomes empty.

[Note] To check if a value is a null-value, we use the `is` operator

```
x = None

x == None       # works, but not Pythonic
x is None       # returns True
```

Truthiness of a `list`

The `all()` function takes as input a `list` and returns `True` if every element evaluates to `True` (is *truthy*). The `any()` function also takes a `list` as input and returns `True` any one or more of its elements evaluate to `True`.

Examples

```
In [24]: all(['a string', 12345, list('abc')])
Out[24]: True

In [25]: all(['a string', 12345, list()])
Out[25]: False

In [26]: any([False, None, [], {}, 0.0])
Out[26]: False

In [27]: any([False, None, [], {}, 1.0])
Out[27]: True
```

`if elif else`

As the name implies, `if` statements execute particular sections of code depending on some tested condition(s). For conditions that have more than two possible values, the `elif` clause can be used.

```
x = 5
if x < 0:
    print('x is negative')
elif x % 2:
    print('x is positive and odd')
else:
    print('x is even and non-negative')
```

Ternary `if else`

This is effectively a condensation of a standard if-else block into a single line.

Syntax

```
<action-if-true> if <condition> else <action-if-false>
```

Example

```
if name == 'John':
    print 'How are you?'
else:
    print 'Who are you?'
```

can be rewritten as

```
'How are you?' if name == 'John' else 'Who are you?'
```

This syntatical sugar comes in handy during the creation of new variables in a data set, that are derived from other existing variables, through the application of some conditional logic.

7. Loops and Flow Control

`for` loops, `continue` and `break`

These are meant for iteration tasks over a collection (a Python data structure like a Tuple or List.)

Syntax

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

Example: Here we print out the squares of the first five natural numbers

```
In[1]:
for x in [1, 2, 3, 4, 5]:
    print x ** 2

Out[1]:
1 4 9 16 25
```

The **continue** keyword advances the for loop to the next iteration - skipping the remainder of the block.

Example: The following loop sums up values, ignoring instances of `None`

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

The **break** keyword is used to altogether exit the `for` loop.

Example: This code sums elements of the list until a 5 is reached

```
until_5 = 0
for value in [1, 4, 2, 0, 7, 5, 1, 4]:
    if value == 5:
        break
    until_5 += value
```

`while` loops

`while` loops are a different type of conditional loop; rather than iterating a specified number of times, according to a given sequence, `while` executes its block of code repeatedly, until its condition is no longer true.

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

Obviously, the body of the while statement should contain code that eventually renders the condition false, otherwise

the loop will never end! An exception to this is if the body of the statement contains a break or return statement; in either case, the loop will be interrupted.

`zip` and `enumerate`

`zip` is a general function that takes a list of sequences as input and returns a list of tuples, where each tuple contains the i-th element from each of the input sequences. Note that the returned list is as long as the shortest sequence.

```
In:     zip(range(5), list('abcde'))

Out:    [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

`enumerate` takes an iterable as its input and returns an iterable which contains pairs of `index, value` Example:

```
In:
for x, y in enumerate('abcde'):
    print x, y

Out:
0 a
1 b
2 c
3 d
4 e
```

`list` and `dict` Comprehensions

Comprehensions are a Pythonic way of condensing loops and conditional routines into a single line of code. They are generally used to

- Create new lists and dictionaries
- Filter existing lists and dictionaries

Syntax

```
 # List Comprehension
[f(x) for x in <iterable> if <condition>]

# Dict Comprehension
{key:value for key, value in <iterable(s)> if <condition>}
```

Examples With comprehension, the following loop (to print the squares of natural numbers)

```
In:
squares = []
for n in range(1, 10):
    squares.append(n**2)

print squares

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

can be re-written succintly as

```
In:
squares = [x**2 for x in range(1, 10)]
print squares

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

Filtering a List using Comprehensions: If we wanted the squares of only even natural numbers,

```
In:
squares_even = [x**2 for x in range(1, 10) if x % 2 == 0]
print squares_even

Out:
[4, 16, 36, 64]
```

Another example: Convert all non-vowels to uppercase

```
In:
alphabets = list('abcdefghijklmnouvwxyz')
[a.upper() for a in alphabets if a not in set('aeiou')]

Out:
['B', 'C', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'M', 'N', 'V', 'W', 'X', 'Y', 'Z']
```

Creating a dict using a comprehension

We can use both `zip` and `enumerate` to stitch together Python sequences, one of which would provide the `key`s for the `dict` and one that would provide the `value`s.

```
# Using Zip
In:  {k:v for k, v in zip(range(5), 'abcde')}
Out: {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}

# Using Enumerate
In:  {k:v for k, v in enumerate('abcde')}
Out: {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
```

Example: Create a dictionary with the counts of every alphabet in a word (keys -> alphabets, values -> counts)

```
In:
s = 'mississippi'
{x:s.count(x) for x in set(s)}

Out:
{'i': 4, 'm': 1, 'p': 2, 's': 4}
```

Example: Create a dictionary of the form (value: value squared), for multiples of 5 under 51

```
In:
{x: x**2 for x in range(1, 51) if x % 5 == 0}

Out:
{35: 1225, 5: 25, 40: 1600, 10: 100, 45: 2025, 15: 225, 50: 2500, 20: 400, 25: 625, 30: 900}
```

Note how the keys in a dictionary are **unordered.**

User Defined Functions and Handling Errors

Simply put, you can think of functions as groups of code that have a name and can be called using parentheses. But in practice, functions are perhaps the most powerful and convenient way of accomplishing two critical tasks:

- code **organization** (think about your project as being composed of modules, bundle up code in functions for each)
- code **reuse** (functionality that you will use in multiple places)

As a direct extension of the famous DRY principle (*Don't Repeat Yourself*) if you ever find yourself copy-pasting code written earlier in a script to do a similar task, stop and write a function. As a guiding principle, take a leaf from the UNIX philosophy, and

> Write short functions that do one thing, but do it really well.

Every function in Python is characterized by the following steps:

1. Take some argument(s)
2. Flow it through the body of the function
3. Return object(s)

Functions in Python can take **positional** and **keyword** arguments. Note that when passing both kinds of arguments to a function, the positional arguments should be listed first. It is also possible to declare **default arguments**, which will take the values specified with the function's definition, unless overriden by explicitly passing parameters.

Syntax

Function Definition:

```
def func-name(<parameters>):
    """
    Function doctring
    """
    <function body>
    return <object>
```

Function Call:

```
func-name(<arguments>)
```

Example:

```
In:
def add_one(num):
    return num + 1

add_one(99)

Out:
100
```

Functions in Python are objects too, just like everything else. They can therefore be passed as arguments to other functions.

`*args` and `**kwargs`

If you're declaring a function and are not sure exactly what arguments it would take or will be passed by the user, you can use `*args` for positional and `**kwargs` for keyword arguments as placeholders. The `*` before `args` tells the interpreter to collect positional arguments in a sequence, while the `**` before the `kwargs` tells the interpreter to collect named arguments in a dict.

Here's an example of a function that takes any number of arguments and prints them out.

```
In:
def flex_func(*args, **kwargs):
    """
    The user may pass any number of positional/named arguments to this function
    """
    print args
    print kwargs

flex_func(3.142, 200, alpha=-0.33, beta=0.07)

Out:
(3.142, 200)
{'alpha': -0.33, 'beta': 0.07}
```

Lambda (or Anonymous) Functions

In addition to defining named functions with the `def` statement, Python allows you to define temporary, anonymous functions called `lambdas` that are used to bundle code intended for one time use only.

Lambdas are used extensively in `pandas` for applying functions to rows and/or columns of a table, and also with Python functions like `map`, `filter` and `reduce` that work on sequences and allow for by-element processing.

Here's an example to add 10 to every number in a list of natural numbers

```
 In: map(lambda x: x + 10, range(10))
Out: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

> [Note] `map` takes a function and applies to to every element of a given sequence.

We could an equivalent of this statement as:

```
 In: def addTen(x):
        return x + 10
     map(addTen, range(10))
Out: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

We see here that `lambda`s save us a bit of typing as we accomplish the same task.

Exception Handling

Of the three kinds of Errors commonly encountered in programming (Syntax, Runtime, Semantic), Python has built-in capabilities to handle runtime errors or exceptions gracefully. This is the very foundation of building robust programs. You anticipate that certain parts of your code might fail, and make necessary provisions to avoid your program from crashing.

Runtime Errors come in many flavors - they can happen over Type Mismatches, Division by Zero etc.

Example: Python's `float` function is capable of casting a string to a floating point number, but fails with `ValueError` on improper inputs:

```
In [343]: float('1.2345')
Out[343]: 1.2345
In [344]: float('something')

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

`try` and `except`

Suppose we wanted a version of **float()** 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:

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

The code in the `except` part of the block will only be executed if the code in the `try` block raises an exception.

```
In [346]: attempt_float('1.2345')
Out[346]: 1.2345

In [347]: attempt_float('something')
In [348]: 'something'

In [350]: attempt_float((1, 2))

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

This time, the program failed because we had not provisioned for a third possible kind of input value. When the function encountered a `tuple`, it found no directive to deal with it, so it raised a `TypeError`.

We can **catch multiple exception types** by writing a tuple of exception types:

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

`finally` with a `try-except` block

If you want some code to be executed regardless of whether the code in the try block succeeds or not, use `finally`:

```
f = open(path, 'w')
try:
    write_to_file(f)
finally:
    f.close()
```

Here, the file handle `f` will always get closed.

`else` with a `try-except` block

Similarly, you can have code that executes only if the try: block succeeds using else: f = open(path, 'w')

```
try:
    write_to_file(f)
except:
    print 'Failed'
else:
    print 'Succeeded'
finally:
    f.close()
```

Revision Questions

- Write a function that takes as input a list of numbers (both positive and negative), and returns a new list with the signs reversed (ie, positive turned to negative and vice-versa)
- Use `map` and a lambda function to find all perfect cubes less than 10e5