1.3.6 Also See

For more on lexical matters and Python styles, see:

- Code Like a Pythonista: Idiomatic Python -- http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html.
- Style Guide for Python Code -- http://www.python.org/dev/peps/pep-0008/

1.3.7 Code Evaluation

Understanding the Python execution model -- How Python evaluates and executes your code. Python evaluates a script or module from the top to the bottom, binding values (objects) to names as it proceeds.

Evaluating expressions -- Expressions are evaluated in keeping with the rules described for operators, above.

Creating names/variables -- Binding -- The following all create names (variables) and bind values (objects) to them: (1) assignment, (2) function definition, (3) class definition, (4) function and method call, (5) importing a module, ...

First class objects -- Almost all objects in Python are first class. Definition: An object is first class if: (1) we can put it in a structured object; (2) we can pass it to a function; and (3) we can return it from a function.

References -- Objects (or references to them) can be shared. What does this mean?

- The object(s) satisfy the identity test operator `is`, that is, `obj1 is obj2` returns `True`.
- The built-in function `id(obj)` returns the same value, that is, `id(obj1) == id(obj2)` is `True`.
- The consequences for mutable objects are different from those for immutable objects.
- Changing (updating) a mutable object referenced through one variable or container also changes that object referenced through other variables or containers, because *it is the same object*.
- `del()` -- The built-in function `del()` removes a reference, not (necessarily) the object itself.

1.4 Built-in Data Types

1.4.1 Strings

1.4.1.1 What strings are

In Python, strings are immutable sequences of characters. They are immutable in that in

order to modify a string, you must produce a new string.

1.4.1.2 When to use strings

Any text information.

1.4.1.3 How to use strings

Create a new string from a constant:

```
s1 = 'abce'
s2 = "xyz"
s3 = """A
multi-line
string.
"""
```

Use any of the string methods, for example:

```
>>> 'The happy cat ran home.'.upper()
'THE HAPPY CAT RAN HOME.'
>>> 'The happy cat ran home.'.find('cat')
10
>>> 'The happy cat ran home.'.find('kitten')
-1
>>> 'The happy cat ran home.'.replace('cat', 'dog')
'The happy dog ran home.'
```

Type "help(str)" or see http://www.python.org/doc/current/lib/string-methods.html for more information on string methods.

You can also use the equivalent functions from the string module. For example:

```
>>> import string
>>> s1 = 'The happy cat ran home.'
>>> string.find(s1, 'happy')
4
```

See string - Common string operations -- http://www.python.org/doc/current/lib/module-string.html for more information on the string module.

There is also a string formatting operator: "%". For example:

```
>>> state = 'California'
>>> 'It never rains in sunny %s.' % state
'It never rains in sunny California.'
>>>
>>> width = 24
>>> height = 32
>>> depth = 8
>>> print 'The box is %d by %d by %d.' % (width, height, depth, )
The box is 24 by 32 by 8.
```

Things to know:

- Format specifiers consist of a percent sign followed by flags, length, and a type character.
- The number of format specifiers in the target string (to the left of the "%" operator) must be the same as the number of values on the right.
- When there are more than one value (on the right), they must be provided in a tuple.

You can learn about the various conversion characters and flags used to control string formatting here: String Formatting Operations -- http://docs.python.org/library/stdtypes.html#string-formatting-operations.

You can also write strings to a file and read them from a file. Here are some examples.

Writing - For example:

```
>>> outfile = open('tmp.txt', 'w')
>>> outfile.write('This is line #1\n')
>>> outfile.write('This is line #2\n')
>>> outfile.write('This is line #3\n')
>>> outfile.close()
```

Notes:

- Note the end-of-line character at the end of each string.
- The `open()` built-in function creates a file object. It takes as arguments (1) the file name and (2) a mode. Commonly used modes are "r" (read), "w" (write), and "a" (append).
- See Built-in Functions: open() -- http://docs.python.org/library/functions.html#open for more information on opening files. See Built-in Types: File Objects -- http://docs.python.org/library/stdtypes.html#file-objects for more information on how to use file objects.

Reading an entire file -- example:

```
>>> infile = file('tmp.txt', 'r')
>>> content = infile.read()
>>> print content
This is line #1
This is line #2
This is line #3

>>> infile.close()
```

Notes:

- Also consider using something like `content.splitlines()`, if you want to divide content in lines (split on newline characters).

Reading a file one line at a time -- example:

```
>>> infile = file('tmp.txt', 'r')
>>> for line in infile:
```

```
...     print 'Line:', line
...
Line: This is line #1

Line: This is line #2

Line: This is line #3

>>> infile.close()
```

Notes:

- Learn more about the `for:` statement in section for: statement.
- "infile.readlines()" returns a list of lines in the file. For large files use the file object itself or "infile.xreadlines()", both of which are iterators for the lines in the file.
- In older versions of Python, a file object is *not* itself an iterator. In those older versions of Python, you may need to use `infile.readlines()` or a `while` loop containing `infile.readline()` For example:

```
>>> infile = file('tmp.txt', 'r')
>>> for line in infile.readlines():
...     print 'Line:', line
...
```

A few additional comments about strings:

- A string is a special kind of sequence. So, you can index into the characters of a string and you can iterate over the characters in a string. For example:

```
>>> s1 = 'abcd'
>>> s1[1]
'b'
>>> s1[2]
'c'
>>> for ch in s1:
...   print ch
...
a
b
c
d
```

- If you need to do fast or complex string searches, there is a regular expression module in the standard library. re re - Regular expression operations -- http://docs.python.org/library/re.html.
- An interesting feature of string formatting is the ability to use dictionaries to supply the values that are inserted. Here is an example:

```
names = {'tree': 'sycamore', 'flower': 'poppy', 'herb':
'arugula'}

print 'The tree is %(tree)s' % names
print 'The flower is %(flower)s' % names
print 'The herb is %(herb)s' % names
```

1.4.2 Sequences -- Lists and Tuples

1.4.2.1 What sequences are

There are several types of sequences in Python. We've already discussed strings, which are sequences of characters. In this section we will describe lists and tuples. See Built-in Types: Sequence Types - str, unicode, list, tuple, buffer, xrange -- http://docs.python.org/library/stdtypes.html#sequence-types-str-unicode-list-tuple-buffer-xrange for more information on Python's built-in sequence types.

Lists are dynamic arrays. They are arrays in the sense that you can index items in a list (for example "mylist[3]") and you can select sub-ranges (for example "mylist[2:4]"). They are dynamic in the sense that you can add and remove items after the list is created.

Tuples are light-weight lists, but differ from lists in that they are immutable. That is, once a tuple has been created, you cannot modify it. You can, of course, modify any (modifiable) objects that the tuple contains, in other words that it refers to.

Capabilities of lists:

- Append an item.
- Insert an item (at the beginning or into the middle of the list).
- Add a list of items to an existing list.

Capabilities of lists and tuples:

- Index items, that is get an item out of a list or tuple based on the position in the list (relative to zero, the beginning of the sequence).
- Select a subsequence of contiguous items (also known as a slice).
- Iterate over the items in the list or tuple.

1.4.2.2 When to use sequences

- Whenever you want to process a colletion of items.
- Whenever you want to iterate over a collection of items.
- Whenever you want to index into a collection of items.

Collections -- Not all collections in Python are ordered sequences. Here is a comparison of some different types of collections in Python and their characteristics:

- String -- ordered, characters, immutable
- Tuple -- ordered, heterogeneous, immutable
- List -- ordered, heterogeneous, mutable
- Dictionary -- unordered, key/values pairs, mutable
- Set -- unordered, heterogeneous, mutable, unique values

1.4.2.3 How to use sequences

To create a list use square brackets. Examples:

```
>>> items = [111, 222, 333]
>>> items
[111, 222, 333]
```

Create a new list or copy an existing one with the list constructor:

```
>>> trees1 = list(['oak', 'pine', 'sycamore'])
>>> trees1
['oak', 'pine', 'sycamore']
>>> trees2 = list(trees1)
>>> trees2
['oak', 'pine', 'sycamore']
>>> trees1 is trees2
False
```

To create a tuple, use commas, and possibly parentheses as well:

```
>>> a = (11, 22, 33, )
>>> b = 'aa', 'bb'
>>> c = 123,
>>> a
(11, 22, 33)
>>> b
('aa', 'bb')
>>> c
(123,)
>>> type(c)
<type 'tuple'>
```

Notes:

- To create a tuple containing a single item, we still need the comma. Example:

```
>>> print ('abc',)
('abc',)
>>> type(('abc',))
<type 'tuple'>
```

To add an item to the end of a list, use `append()`:

```
>>> items.append(444)
>>> items
[111, 222, 333, 444]
```

To insert an item into a list, use `insert()`. This example inserts an item at the beginning of a list:

```
>>> items.insert(0, -1)
>>> items
[-1, 111, 222, 333, 444]
```

To add two lists together, creating a new list, use the `+` operator. To add the items in one list to an existing list, use the `extend()` method. Examples:

```
>>> a = [11, 22, 33,]
>>> b = [44, 55]
>>> c = a + b
```

```
>>> c
[11, 22, 33, 44, 55]
>>> a
[11, 22, 33]
>>> b
[44, 55]
>>> a.extend(b)
>>> a
[11, 22, 33, 44, 55]
```

You can also push items onto the right end of a list and pop items off the right end of a list with `append()` and `pop()`. This enables us to use a list as a stack-like data structure. Example:

```
>>> items = [111, 222, 333, 444,]
>>> items
[111, 222, 333, 444]
>>> items.append(555)
>>> items
[111, 222, 333, 444, 555]
>>> items.pop()
555
>>> items
[111, 222, 333, 444]
```

And, you can iterate over the items in a list or tuple (or other collection, for that matter) with the `for:` statement:

```
>>> for item in items:
...   print 'item:', item
...
item: -1
item: 111
item: 222
item: 333
item: 444
```

For more on the `for:` statement, see section for: statement.

1.4.3 Dictionaries

1.4.3.1 What dictionaries are

A dictionary is:

- An associative array.
- A mapping from keys to values.
- A container (collection) that holds key-value pairs.

A dictionary has the following capabilities:

- Ability to iterate over keys or values or key-value pairs.
- Ability to add key-value pairs dynamically.

- Ability to look-up a value by key.

For help on dictionaries, type:

```
>>> help dict
```

at Python's interactive prompt, or:

$ pydoc dict

at the command line.

It also may be helpful to use the built-in `dir()` function, then to ask for `help` on a specific method. Example:

```
>>> a = {}
>>> dir(a)
['__class__', '__cmp__', '__contains__', '__delattr__',
'__delitem__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault',
'update', 'values']
>>>
>>> help(a.keys)
Help on built-in function keys:

keys(...)
    D.keys() -> list of D's keys
```

More information about dictionary objects is available here: Mapping types - dict -- http://docs.python.org/library/stdtypes.html#mapping-types-dict.

1.4.3.2 When to use dictionaries

- When you need look-up by key.
- When you need a "structured" lite-weight object or an object with named fields. (But, don't forget classes, which you will learn about later in this document.)
- When you need to map a name or label to any kind of object, even an executable one such as a function.

1.4.3.3 How to use dictionaries

Create a dictionary with curly bracets. Items in a dictionary are separate by commas. Use a colon between each key and its associated value:

```
>>> lookup = {}
>>> lookup
{}
>>> states = {'az': 'Arizona', 'ca': 'California'}
>>> states['ca']
```

```
'California'
```

or:

```
>>> def fruitfunc():
...    print "I'm a fruit."
>>> def vegetablefunc():
...    print "I'm a vegetable."
>>>
>>> lookup = {'fruit': fruitfunc, 'vegetable': vegetablefunc}
>>> lookup
{'vegetable': <function vegetablefunc at 0x4028980c>,
'fruit': <function fruitfunc at 0x4028e614>}
>>> lookup['fruit']()
I'm a fruit.
>>> lookup['vegetable']()
I'm a vegetable.
```

or:

```
>>> lookup = dict((('aa', 11), ('bb', 22), ('cc', 33)))
>>> lookup
{'aa': 11, 'cc': 33, 'bb': 22}
```

Note that the keys in a dictionary must be immutable. Therefore, you can use any of the following as keys: numbers, strings, tuples.

Test for the existence of a key in a dictionary with the `in` operator:

```
>>> if 'fruit' in lookup:
...   print 'contains key "fruit"'
...
contains key "fruit"
```

or, alternatively, use the (slightly out-dated) `has_key()` method:

```
>>> if lookup.has_key('fruit'):
...   print 'contains key "fruit"'
...
contains key "fruit"
```

Access the value associated with a key in a dictionary with the indexing operator (square brackets):

```
>>> print lookup['fruit']
<function fruitfunc at 0x4028e614>
```

Notice that the above will throw an exception if the key is not in the dictionary:

```
>>> print lookup['salad']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'salad'
```

And so, the `get()` method is an easy way to get a value from a dictionary while avoiding an exception. For example:

```
>>> print lookup.get('fruit')
<function fruitfunc at 0x4028e614>
>>> print lookup.get('salad')
None
>>> print lookup.get('salad', fruitfunc)
<function fruitfunc at 0x4028e614>
```

A dictionary is an iterator object that produces its keys. So, we can iterate over the keys in a dictionary as follows:

```
>>> for key in lookup:
...     print 'key: %s' % key
...     lookup[key]()
...
key: vegetable
I'm a vegetable.
key: fruit
I'm a fruit.
```

And, remember that you can sub-class dictionaries. Here are two versions of the same example. The keyword arguments in the second version require Python 2.3 or later:

```
#
# This example works with Python 2.2.
class MyDict_for_python_22(dict):
    def __init__(self, **kw):
        for key in kw.keys():
            self[key] = kw[key]
    def show(self):
        print 'Showing example for Python 2.2 ...'
        for key in self.keys():
            print 'key: %s  value: %s' % (key, self[key])

def test_for_python_22():
    d = MyDict_for_python_22(one=11, two=22, three=33)
    d.show()

test_for_python_22()
```

A version for newer versions of Python:

```
#
# This example works with Python 2.3 or newer versions of Python.
#   Keyword support, when sub-classing dictionaries, seems to have
#   been enhanced in Python 2.3.
class MyDict(dict):
    def show(self):
        print 'Showing example for Python 2.3 or newer.'
        for key in self.keys():
            print 'key: %s  value: %s' % (key, self[key])

def test():
    d = MyDict(one=11, two=22, three=33)
    d.show()
```

```
test()
```

Running this example produces:

```
Showing example for Python 2.2 ...
key: one  value: 11
key: three  value: 33
key: two  value: 22
Showing example for Python 2.3 or newer.
key: three  value: 33
key: two  value: 22
key: one  value: 11
```

A few comments about this example:

- Learn more about classes and how to implement them in section Classes and instances.
- The class `MyDict` does not define a constructor (`__init__`). This enables us to re-use the contructor from super-class `dict` and any of its forms. Type "help dict" at the Python interactive prompt to learn about the various ways to call the dict constructor.
- The show method is the specialization added to our sub-class.
- In our sub-class, we can refer to any methods in the super-class (dict). For example: `self.keys()`.
- In our sub-class, we can refer the dictionary itself. For example: `self[key]`.

1.4.4 Files

1.4.4.1 What files are

- A file is a Python object that gives us access to a file on the disk system.
- A file object can be created ("opened") for reading ("r" mode), for writing ("w" mode), or for appending ("a" mode) to a file.
- Opening a file for writing erases an existing with that path/name. Opening a file for append does not.

1.4.4.2 When to use files

Use a file object any time you wish to read from or write to the disk file system.

1.4.4.3 How to use files

Here is an example that (1) writes to a file, then (2) appends to that file, and finally, (3) reads from the file:

```
def write_file(outfilename):
    outfile = open(outfilename, 'w')
    outfile.write('Line # 1\n')
    outfile.write('Line # 2\n')
```

```
    outfile.write('Line # 3\n')
    outfile.close()

def append_file(outfilename):
    outfile = open(outfilename, 'a')
    outfile.write('Line # 4\n')
    outfile.write('Line # 5\n')
    outfile.close()

def read_file(infilename):
    infile = open(infilename, 'r')
    for line in infile:
        print line.rstrip()
    infile.close()

def test():
    filename = 'temp_file.txt'
    write_file(filename)
    read_file(filename)
    append_file(filename)
    print '-' * 50
    read_file(filename)

test()
```

1.4.4.4 Reading Text Files

To read a text file, first create a file object. Here is an example:

```
inFile = open('messages.log', 'r')
```

Then use the file object as an iterator or use one or more of the file object's methods to process the contents of the file. Here are a few strategies:

- Use `for line in inFile:` to process one line at a time. You can do this because (at least since Python 2.3) file objects obey the iterator protocol, that is they support methods `__iter__()` and `next()`. For more on the iterator protocol see Python Standard Library: Iterator Types -- http://docs.python.org/library/stdtypes.html#iterator-types.
  Example:

```
>>> inFile = file('tmp.txt', 'r')
>>> for line in inFile:
...     print 'Line:', line,
...
Line: aaaaa
Line: bbbbb
Line: ccccc
Line: ddddd
Line: eeeee
>>> inFile.close()
```

- For earlier versions of Python, one strategy is to use "inFile.readlines()", which creates a list of lines.

- If your want to get the contents of an entire text file as a collection of lines, use `readlines()`. Alternatively, you could use `read()` followed by `splitlines()`. Example:

```
>>> inFile = open('data2.txt', 'r')
>>> lines = inFile.readlines()
>>> inFile.close()
>>> lines
['aaa bbb ccc\n', 'ddd eee fff\n', 'ggg hhh iii\n']
>>>
>>> inFile = open('data2.txt', 'r')
>>> content = inFile.read()
>>> inFile.close()
>>> lines = content.splitlines()
>>> lines
['aaa bbb ccc', 'ddd eee fff', 'ggg hhh iii']
```

- Use `inFile.read()` to get the entire contents of the file (a string). Example:

```
>>> inFile = open('tmp.txt', 'r')
>>> content = inFile.read()
>>> inFile.close()
>>> print content
aaa bbb ccc
ddd eee fff
ggg hhh iii
>>> words = content.split()
>>> print words
['aaa', 'bbb', 'ccc', 'ddd', 'eee', 'fff', 'ggg',
'hhh', 'iii']
>>> for word in words:
...     print word
...
aaa
bbb
ccc
ddd
eee
fff
ggg
hhh
iii
```

1.5 Simple Statements

Simple statements in Python do *not* contain a nested block.

1.5.1 print statement

Alert: In Python version 3.0, the `print` statement has become the `print()` built-in function. You will need to add parentheses.

The print statement sends output to stdout.

Here are a few examples:

```
print obj
print obj1, obj2, obj3
print "My name is %s" % name
```

Notes:

- To print multiple items, separate them with commas. The print statement inserts a blank between objects.
- The print statement automatically appends a newline to output. To print without a newline, add a comma after the last object, or use "sys.stdout", for example:

```
print 'Output with no newline',
```

  which will append a blank, or:

```
import sys
sys.stdout.write("Some output")
```

- To re-define the destination of output from the print statement, replace sys.stdout with an instance of a class that supports the write method. For example:

```
import sys

class Writer:
    def __init__(self, filename):
        self.filename = filename
    def write(self, msg):
        f = file(self.filename, 'a')
        f.write(msg)
        f.close()

sys.stdout = Writer('tmp.log')
print 'Log message #1'
print 'Log message #2'
print 'Log message #3'
```

More information on the print statement is at The print statement -- http://docs.python.org/reference/simple_stmts.html#the-print-statement.

1.5.2 Assignment statement

The assignment operator is =.

Here are some of the things you can assign a value to:

- A name (variable)
- An item (position) in a list. Example:

```
>>> a = [11, 22, 33]
>>> a
[11, 22, 33]
>>> a[1] = 99
>>> a
[11, 99, 33]
```

- A key in a dictionary. Example:

```
>>> names = {}
>>> names['albert'] = 25
>>> names
```