Python for Test Automation

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	August 2012		SF

Contents

Chapter 1

UnitTest

1.1 Lesson objectives

In this lesson you will learn to write tests with the builtin `unittest` module. Using `unittest` allows you to write more robust tests that require setup or cleanup.

1.2 UnitTest

The `unittest` module may feel a little verbose and "un-pythonic". This is probably a reflection of it's origin as a straight port of the JUnit testing framework written for the Java programming language. Tests written with the help of unittest must be written inside of classes - Java, unlike Python, does not allow purely procedural code and unittest reflects this.

While we haven't yet learned any of the syntax or concepts related to Object Oriented Programming in Python, I'll provide you with the outline of a test and you can start writing your own.

Why might we want to use the more cumbersome `unittest` module? You've probably written enough doctests by now to recognize some of their limitations: output parsing can be a little fragile due to embedding strings inside of docstrings, there isn't any way to structure your test suite (each line of code is a test), and you're limited by readability constraints. It probably wouldn't be very friendly to put 20 lines of doctest on a 5 line function - but sometimes that is what you need to test all the edge cases. Some functions also depend on their environment - and a doctest that has 10 lines of setup and 1 line of actual test isn't necessarily contributing much to the intended documentation purpose of docstrings.

The unittest module provides a runner to execute our tests, means of grouping tests into logical units, and the concept of setup and teardown to provide a consistent environment for our tests. Let's see how all these concepts work by looking at a simple example:

unitttest1.py

```
import unittest

class ObviousTest(unittest.TestCase):

    def setUp(self):
        self.num = 1

    def test_math(self):
        self.assertEqual(self.num, 1)

    def test_divides_by_zero(self):
        self.assertRaises(ZeroDivisionError, lambda : 1/0)

if __name__ == '__main__':
    unittest.main()
```

Let's explain the OOP boilerplate you will need to write tests. A TestCase is a class that inherits from unittest.TestCast. In Python we create inheritance by passing a list of parent classes in parentheses after the name of our class. If you aren't familiary with OOP concepts don't panic! For the moment we'll just think of a class as a container for functions.

The functions defined inside a class all require a mandatory first parameter conventionally named `self`. We'll explain this later - for the moment you can just follow the pattern.

We can run our tests by running the script - when run as a script the provided `unittest.main()` function takes care of finding all the TestCases, calling their `setUp` method first if provided (functions defined inside a class are called methods), calling each of the methods whose name start with *test*, finally calling a `tearDown` method if provided.

```
$ python unitest1.py
..
Ran 2 tests in 0.000s

OK
```

Our test class has three methods. The `setUp` method allows us to prepare the environment in which our tests will be run - if we need to create files, prepare data, etc then `setUp` is the place to do it. Next we have our "test" methods. When writing doctests each line of code was a test but in unittest you will define your tests as functions.

Each test can contain whatever Python code it likes - but usually calls convenience methods supplied by the unittest.TestCase base class. You can access these methods by dereferencing `self` and methods exist to assert that two values are equal or inequal, or even "almost" equal, that a value is False or True, or even to check that a callable raises an exception.

The tests in our `ObviousTest` class aren't doing anything very profound - `test_divides_by_zero` confirms that an anonymous function that attempts to divide by zero does in fact raise an exception. In this case we are testing a failure condition and the test passes if the failure occurs as expected. In `test_math` we are testing that two numeric values are equal and as expected these tests pass.

You might pay attention to the output. The `unittest` test runner outputs a dot (.) for each sucessful test, an "F" for any tests that do not pass, and an "E" for tests that failed to run due to errors in the test or in code it tested. With a little experience you'll quickly become addicted to that sucessful test output ....

1.2.1 Lab

Earlier you completed a short lab that tested using functional programming constructs like map, filter, and sorted. You've already written doctests for this code - now remove any doctests that don't improve the documentation of your functions and create a new file `unittest1.py`. Copy and paste the unittest example above and write new methods to test the functions in functional1.py. How many tests should you write? How can you tell when you've exhaustively tested your code?

1.3 *UnitTest* Follow-up

Doctests are quick to write and may improve your documentation. They also make sense when you need code samples for your documentation and want to make sure that the documentation doesn't lie. Use unittests, however, whenever you need to thoroughly test something. As we will see there is an ecosystem of test runners, reports, plugins, and tools that work well with tests written using `unittest`.

Chapter 2

Code Organization

2.1 Lesson objectives

In this lesson you'll learn some of the tools you need as your scripts get to be bigger than just the contents of a single file. Python's module system gives us a valuable organizational pattern for organizing our programs. After this session you'll understand

- namespaces
- how to import modules
- how to write your own modules and packages

2.2 TOPIC: Namespaces

Inside a single python module (file) there are multiple namespaces. The global namespace represents the full contents of the file, while inside each function there is a local namespace. The contents of these namespaces can be accessed as dictionaries using the functions `globals()` and `locals()` respectively.

When objects (variables, functions, etc) are defined in the file, they manipulate the global namespace. Anything defined inside a function manipulates its local namespace. Notice in the example below that the namespace is manipulated as the file is read. In other words, the first print statement occurs before the interpreter is aware of the presence of the function definition.

Note
The following example makes use of `import`, which will be explained in the next section. It also uses `pprint.pformat` which converts a dictionary into a string in a manner that is more easily read when printed.

---

organization-1-namespaces.py

```
'''Some documentation for this file.'''
import pprint

print 'globals before def: %s\n' % pprint.pformat(globals(), indent=4)

def simple():
  print 'locals before a: %s\n' % locals()
  a = 'simple'
  print 'locals after a: %s\n' % locals()
  return a

print 'globals after def: %s\n' % pprint.pformat(globals(), indent=4)

simple()
```

```
$ python organization-1-namespaces.py
globals before def: {   '__builtins__': <module '__builtin__' (built-in)>,
    '__doc__': 'Some documentation for this file.',
    '__file__': 'samples/organization-1-namespaces.py',
    '__name__': '__main__',
    '__package__': None,
    'pprint': <module 'pprint' from '/usr/lib/python2.7/pprint.pyc'>}

globals after def: {   '__builtins__': <module '__builtin__' (built-in)>,
    '__doc__': 'Some documentation for this file.',
    '__file__': 'samples/organization-1-namespaces.py',
    '__name__': '__main__',
    '__package__': None,
    'pprint': <module 'pprint' from '/usr/lib/python2.7/pprint.pyc'>,
    'simple': <function simple at 0x405088b4>}

locals before a: {}

locals after a: {'a': 'simple'}
```

2.3 TOPIC: Importing modules

Have another look at an example similar to the one above. Notice that the modules that are imported are present in the global namespace.

organization-2-imports.py

```
import collections
import pprint

d = collections.deque()
d.append('a')
d.appendleft('b')

pprint.pprint(globals())
```

```
$ python organization-2-imports.py
{'__builtins__': <module '__builtin__' (built-in)>,
 '__doc__': None,
 '__file__': 'samples/organization-2-imports.py',
 '__name__': '__main__',
 '__package__': None,
 'collections': <module 'collections' from '/usr/lib/python2.7/collections.pyc'>,
 'd': deque(['b', 'a']),
 'pprint': <module 'pprint' from '/usr/lib/python2.7/pprint.pyc'>}
```

Objects from this module are accessed using dotted notation.

Alternatively, you can import the specific element from the module, using the `from ... import` syntax.

organization-3-import-submodule.py

```
from collections import deque
from pprint import pprint

d = deque()
d.append('a')
d.appendleft('b')

pprint(globals())
```

```
$ python organization-3-import-submodule.py
{'__builtins__': <module '__builtin__' (built-in)>,
 '__doc__': None,
 '__file__': 'samples/organization-3-import-submodule.py',
 '__name__': '__main__',
 '__package__': None,
 'd': deque(['b', 'a']),
 'deque': <type 'collections.deque'>,
 'pprint': <function pprint at 0x4051b3e4>}
```

It is also possible to import an entire namespace. This should be done with caution, as it can lead to unexpected elements in your namespace and conflicts in naming.

organization-4-import-all.py

```
from collections import *
from pprint import pprint

d = deque()
d.append('a')
d.appendleft('b')

pprint(globals())
```

```
$ python organization-4-import-all.py
{'Callable': <class '_abcoll.Callable'>,
 'Container': <class '_abcoll.Container'>,
 'Counter': <class 'collections.Counter'>,
 'Hashable': <class '_abcoll.Hashable'>,
 'ItemsView': <class '_abcoll.ItemsView'>,
 'Iterable': <class '_abcoll.Iterable'>,
 'Iterator': <class '_abcoll.Iterator'>,
 'KeysView': <class '_abcoll.KeysView'>,
 'Mapping': <class '_abcoll.Mapping'>,
 'MappingView': <class '_abcoll.MappingView'>,
 'MutableMapping': <class '_abcoll.MutableMapping'>,
 'MutableSequence': <class '_abcoll.MutableSequence'>,
 'MutableSet': <class '_abcoll.MutableSet'>,
 'OrderedDict': <class 'collections.OrderedDict'>,
 'Sequence': <class '_abcoll.Sequence'>,
 'Set': <class '_abcoll.Set'>,
 'Sized': <class '_abcoll.Sized'>,
 'ValuesView': <class '_abcoll.ValuesView'>,
 '__builtins__': <module '__builtin__' (built-in)>,
 '__doc__': None,
 '__file__': 'samples/organization-4-import-all.py',
 '__name__': '__main__',
 '__package__': None,
 'd': deque(['b', 'a']),
 'defaultdict': <type 'collections.defaultdict'>,
 'deque': <type 'collections.deque'>,
 'namedtuple': <function namedtuple at 0x40510614>,
 'pprint': <function pprint at 0x4051b3e4>}
```

2.4 TOPIC: Creating Modules

Once your code starts to get bigger than a script, you will want to start organizing it into modules. Unlike some other languages (Java for example) each file in Python is a module. Directories can also be used as a further layer of organization with some care.

Using the file-only model, functions can be created in another file in the same directory (or somewhere in the $PYTHONPATH) and imported using the filename and the function name.

tools.py

```
def shorten(toolong):
    return toolong[:2]
```

complexity-4-file-module.py

```
from tools import shorten

print shorten('abcdef')
```

```
$ python complexity-4-file-module.py
ab
```

As code starts to require even more organization, perhaps with multiple types of utility functions, this file could be moved to a subdirectory. In order to use a directory as a module, it is required that the special file `__init__.py` be present in the directory, which can be empty. Hierarchical modules like this - modules that contain modules - are called packages.

```
$ ls tools2
tools2/__init__.py
tools2/strings.py
```

tools2/strings.py

```
def shorten(toolong):
    return toolong[:2]
```

complexity-5-directory-module.py

```
from tools2 import strings

print strings.shorten('abcdef')
```

```
$ python complexity-5-directory-module.py
ab
```

2.4.1 Preventing execution on import

You might pay attention to the special variable `__name__`. In the code samples above `__name__` appears as a variable in `globals()` and has the value "*main*". Python provides this variable and automatically sets it to one of two values depending on the execution context.

testmodule.py

```
print __name__
if __name__ == '__main__':
   print "Run program"
```

If we run this file directly the value of `__name__` is the string "*main*".

```
$ python testmodule.py
__main__
Run program
```

However if we import this file as a module the value of the variable `__name__` is set to the string value of the containing module. This means our `if` statement doesn't evaluate to `True` and any code hidden behind the if statement doesn't run. It is Python convention to end every script with an `if __name__` block to ensure that importing your script has no side effects.

```
$ python
Python 2.6.5
>>> import testmodule
testmodule
```

2.5 *Code Organization* Follow-up

Unfortunately most of the labs in this course are not substantial enough to give you real world experience with structuring your code in packages. It's worth noting, however, that you have been creating modules each time you create a new .py file. Be sure to name your python files using valid Python identifiers so they can be imported.

You also learned about the special `__name__` variable and how to use it to execute Python code only when run directly. Be sure to use this idiom to hide the main body of your scripts hidden from execution when your script is imported as a module.

Chapter 3

Advanced Iteration

3.1 Lesson objectives

This lesson will explore list comprehensions and generators. List comprehensions are a great feature in Python, allowing us to filter and transform lists in succint but readable fashion. Generator expressions and generator functions allow us more efficient iteration or the ability to chaining multiple iterations into one operations. And finally we'll explore the `itertools` module to see some tools that help us with advanced iteration.

3.2 TOPIC: List Comprehensions

Python provides list comprehension syntax to simplify the task of generating new lists from existing lists (or from any other iterable data type). A list comprehension is simply a 3 part expression inside square braces that produces a new list from an old one. The three parts are

1. an item in the new list
2. a for loop that defines the old list
3. an optional if statement that filters the original list

In the earlier example of writing to a file, the `\n` character was stored with each color in the list so that it could be passed to the `writelines()` method of the file object. Storing the data in this way would make other processing challenging, so a more common model would be to create a list with line feeds based on a list without them.

Without list comprehensions, this operation would look something like the following:

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = []
>>> for color in colors:
...     color_lines.append('{0}\n'.format(color))
...
>>> color_lines
['red\n', 'yellow\n', 'blue\n']
```

We could use the `map()` function to perform this task. Map takes a function that maps values from an original list to a new list.

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = map(lambda c: '{0}\n'.format(c), colors)
>>> color_lines
['red\n', 'yellow\n', 'blue\n']
```

List comprehensions perform this task equally well, but provide additional functionality in an easier package. To accomplish the same task with a list comprehension, use the following syntax:

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = [color + "\n" for color in colors]
>>> color_lines
['red\n', 'yellow\n', 'blue\n']
```

A conditional filter can also be included in the creation of the new list, like so:

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = [color + "\n' for color in colors if 'l' in color]
>>> color_lines
['yellow\n', 'blue\n']
```

Read the list comprehension aloud - it should read naturally. Can you tell what the expression is doing? It should be more easily understandable than the equivalent nested `map` and `filter` calls while remaining much more succint than the `for` loop and nested `if` equivalent.

More than one list can be iterated over, as well, which will create a pass for each combination in the lists.

```
>>> colors = ['red', 'yellow', 'blue']
>>> clothes = ['hat', 'shirt', 'pants']

>>> colored_clothes = ['{0} {1}'.format(color, garment) for color in colors for garment in ↩
    clothes]

>>> colored_clothes
['red hat', 'red shirt', 'red pants', 'yellow hat', 'yellow shirt', 'yellow pants', 'blue ↩
    hat', 'blue shirt', 'blue pants']
```

Nesting list comprehensions typically diminishes their readability and should be approached with caution.

3.2.1 LAB

Copy your earlier completion of functional1.py to a new file called listcomp1.py. Adjust the code you wrote in unittest1.py to make sure you're testing the new file. Modify any examples that use `map` or `filter` to use a list comprehension instead. You may be able to get rid of the lambda functions you used with map and filter. When you're done, run unittest1.py to make sure that changing the **implementation** didn't break anything.

3.3 TOPIC: Generator Expressions

Storing a new list as the output of a list comprehension is not always optimal behavior. Particularly in a case where that list is intermediary or where the total size of the contents is quite large.

For such cases, a slightly modified syntax (replacing square brackets with parentheses) leads to the creation of a generator instead of a new list. The generator will produce the individual items in the list as each one is requested, which is generally while iterating over that new list.

```
>>> colors = ['red', 'yellow', 'blue']
>>> color_lines = ("\n" + color for color in colors)
>>> color_lines
<generator object <genexpr> at 0x10041ac80>
>>> color_lines.next()
'red\n'
>>> color_lines.next()
'yellow\n'
>>> color_lines.next()
'blue\n'
```

```
>>> color_lines.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

It's important to realise that a generator does not produce a list, it is a promise to produce a list in the future! Until you iterate on a generator nothing has happened besides the creation of your generator object.

Even when the generator does run - instead of creating a single list from the loop and conditions it produces a single value and a position. For each call to `.next()` it produces a single value and updates the current position.

There are downsides - you can't index a generator and you can only iterate over it once. But the performance implications (particularly memory usage) can be dramatic. Code that uses a list comprehension that produces a new list from an original (large) list takes twice the memory of the same code that simply changes the square brackets in the list comprehension to parentheses to make a generator expression instead.

3.4 TOPIC: Generator Functions

The type of object created by the generator expression in the previous section is unsurprisingly called a generator. This is a term for a type of iterator that generates values on demand.

While the generator expression notation is very compact, there may be cases where there is more logic to be performed than can be effectively expressed with this notation. For such cases, a generator function can be used.

A generator function uses the `yield` statement in place of `return` and usually does so inside a loop. When the interpreter sees this statement, it will actually return a generator object from the function. Each time the `next()` function is called on the generator object, the function will be executed up to the next `yield`. When the function completes, the interpreter will raise a StopIteration error to the caller.

```
>>> def one_color_per_line():
...     colors = ['red', 'yellow', 'blue']
...     for color in colors:
...         yield '{0}\n'.format(color)
...
>>> gen = one_color_per_line()
>>> gen
<generator object one_color_per_line at 0x10041acd0>
>>> gen.next()
'red\n'
>>> gen.next()
'yellow\n'
>>> gen.next()
'blue\n'
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Note that a second call to the same generator function will return a new generator object (shown at a different address) as each generator should be capable of maintaining its own state.

```
>>> gen = one_color_per_line()
>>> gen
<generator object one_color_per_line at 0x10041ad20>
```

Of course, the more typical use case would be to allow the calls to `next()` to be handled by a `for ... in` loop.

```
>>> for line in one_color_per_line():
...     print line,
...
```

```
red
yellow
blue
```

You might be wondering why you would ever use generators. Most answers tend to revolve around performance. The potential memory efficiency is most obvious but you can also often improve the execution speed by using generators to interleave CPU and IO bound work.

Aside from performance, however, the coolest thing about generators is the way they let you reshape iteration with trivial amounts of code. Consider writing a function that takes an arbitrary number of iterable data sources and returns a single object that can be iterated over to produce one record at a time. You could:

1. Iterate over all the data sources and accumulate each record in an output list
2. Build an iterable object that stores the data sources with a .next() method that keeps track of which data source it's currently on, returning one record at a time.

The yield keyword makes this sort of reshaping simply a matter of yielding inside a doubly nested list.

3.4.1 LAB

Let's tackle writing this function "test driven development"-style. TDD emphasizes writing a minimal test that fails, then supplying the minimal implementation required to make the test pass, and iterating the process until requirements are met. We'll try writing our tests before our application code.

1. First create a new file *test_yield.py*. Create the `unittest.TestCase` boilerplate and make sure to call the `unittest.main()` method in an `if __name__ == "__main__"` block. Create a new file *chained_iterators.py* that will contain our application code.
2. Now lets add tests for the function we haven't yet created! Import our *chained_iterators* module and write a test that calls a function `chained_iterators.chain` that doesn't yet exist. Verify the test fails and then create the function (the minimal implementation is probably a `def` statement and a `pass` function body) and verify that the test passes.
3. Let's see how many tests we can write on the way to completing the lab. Next might be a test to make sure you can pass an arbitrary number of arguments, then perhaps write the code to actually join the arguments together. You might test that the return value is in fact an iterator and test the edge cases in terms of number of arguments (no arguments passed, only 1, 2 and more). What happens when a scalar value like an integer is passed to the function?

3.5 TOPIC: Iteration Helpers: `itertools`

Iteration is a big part of the flow of Python and aside from the builtin syntax, there are some handy tools in the itertools package to make things easier. They also tend to make things run faster.

3.5.1 `chain()`

The `chain()` method accepts an arbitrary number of iterable objects as arguments and returns an iterator that will iterate over each iterable in turn. Once the first is exhausted, it will move onto the next.

Without the `chain()` function, iterating over two lists would require creating a copy with the contents of both or adding the contents of one to the other.

```
>>> l1 = ['a', 'b', 'c']
>>> l2 = ['d', 'e', 'f']
>>> l1.extend(l2)
>>> l1
['a', 'b', 'c', 'd', 'e', 'f']
```

It's much more efficient to use the `chain()` function which only allocates additional storage for some housekeeping data in the iterator itself.

```
>>> import itertools
>>> l1 = ['a', 'b', 'c']
>>> l2 = ['d', 'e', 'f']

>>> chained = itertools.chain(l1, l2)
>>> chained
<itertools.chain object at 0x100431250>

>>> [l for l in chained]
['a', 'b', 'c', 'd', 'e', 'f']
```

3.5.2 `izip()`

`izip()` is almost identical to the `zip()` builtin, in that it pairs up the contents of two lists into an iterable of 2-tuples. However, where `zip()` allocates a new list, `izip()` only returns an iterator.

```
>>> name = ['Jimmy', 'Robert', 'John Paul', 'John']
>>> instruments = ['Guitar', 'Vocals', 'Bass', 'Drums']

>>> zepp = zip(name, instruments)
>>> zepp
[('Jimmy', 'Guitar'), ('Robert', 'Vocals'), ('John Paul', 'Bass'), ('John', 'Drums')]

>>> zepp = itertools.izip(name, instruments)
>>> zepp
<itertools.izip object at 0x100430998>

>>> [musician for musician in zepp]
[('Jimmy', 'Guitar'), ('Robert', 'Vocals'), ('John Paul', 'Bass'), ('John', 'Drums')]
```

3.6 *Advanced Iterations* Follow-up

In this lesson you learned how to use list comprehensions to filter or modify lists. Practice list comprehensions until you are very comfortable writing and reading them. Generator functions and generator expressions might be used less - but they give us important efficiency gains and you'll use generators later on when you enable iteration on our own types.

Chapter 4

Object-Oriented Programming

4.1 Lesson objectives

This lesson will review Python's support for object oriented programming. Some topics in OOP are beyond the scope of this class but you will learn

- how to create your own types with `class`
- how to support Python operators and protocols (like emulation or indexing) on your custom types
- how to create static functions as part of your Classes
- how to use inheritance and `super`
- how to separate the interface from implementation in your classes

4.2 TOPIC: Classes

A class is defined in Python using the `class` statement. The syntax of this statement is `class <ClassName>(superclass)`. In the absence of anything else, the superclass should always be `object`, the root of all classes in Python.

Note
`object` is technically the root of "new-style" classes in Python, but new-style classes today are as good as being the only style of classes.

Here is a basic class definition for a class with one method. There are a few things to note about this method:

1. The single argument of the method is `self`, which is a reference to the object instance upon which the method is called, is explicitly listed as the first argument of the method. In the example, that instance is `a`. This object is commonly referred to as the "bound instance."
2. However, when the method is called, the `self` argument is inserted implicitly by the interpreter — it does not have to be passed by the caller.
3. The attribute `__class__` of `a` is a reference to the class object `A`
4. The attribute `__name__` of the class object is a string representing the name, as given in the class definition.

Also notice that "calling" the class object (`A`) produces a newly instantiated object of that type (assigned to `a` in this example). You can think of the class object as a factory that creates objects and gives them the behavior described by the class definition.

```
>>> class A(object):
...     def whoami(self):
...         return self.__class__.__name__
...

>>> a = A()

>>> a
<__main__.A object at 0x100425d90>

>>> a.whoami()
'A'
```

The most commonly used special method of classes is the `__init__()` method, which is an initializer for the object. The arguments to this method are passed in the call to the class object.

Notice also that the arguments are stored as object attributes, but those attributes are not defined anywhere before the initializer.

Attempting to instantiate an object of this class without those arguments will fail.

```
>>> class Song(object):
...     def __init__(self, title, artist):
...         self.title = title
...         self.artist = artist
...     def get_title(self):
...         return self.title
...     def get_artist(self):
...         return self.artist
...

>>> unknown = Song()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3 arguments (1 given)
```

Notice again that one argument was actually provided (`self`) and only title and artist are considered missing.

So, calling `Song` properly gives an instance of Song with an artist and title. Calling the `get_title()` method returns the title, but so does just referencing the `title` attribute. It is also possible to directly write the instance attribute. Using boilerplate getter / setter methods is generally considered unnecessary. There are ways to create encapsulation that will be covered later.

```
>>> leave = Song('Leave', 'Glen Hansard')

>>> leave
<__main__.Song object at 0x100431050>

>>> leave.get_title()
'Leave'

>>> leave.title
'Leave'

>>> leave.title = 'Please Leave'

>>> leave.title
'Please Leave'
```

One mechanism that can be utilized to create some data privacy is a preceding double-underscore on attribute names. However, it is possible to find and manipulate these variables if desired, because this approach simply mangles the attribute name with the class name. The goal in this mangling is to prevent clashing between "private" attributes of classes and "private" attributes of their superclasses.

```
>>> class Song(object):
...     def __init__(self, title, artist):
...         self.__title = title
...         self.__artist = artist
...     def get_title(self):
...         return self.__title
...     def get_artist(self):
...         return self.__artist
...
>>> leave = Song('Leave', 'Glen Hansard')

>>> leave.get_title()
'Leave'

>>> leave.__title
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Song' object has no attribute '__title'

>>> leave._Song__title
'Leave'
```

4.2.1 Emulation

Python provides many special methods on classes that can be used to emulate other types, such as functions, iterators, containers and more.

Functions In order to emulate a function object, a class must define the method `__call__()`. If the call operator `()` is used on an instance of the class, this method will be called behind the scenes. Here is an example that performs the same task as the adding closure in the functional programming section.

```
>>> class Adder(object):
...     def __init__(self, extra):
...         self.extra = extra
...     def __call__(self, base):
...         return self.extra + base
...
>>> add2 = Adder(2)
>>> add2(3)
5
>>> add5 = Adder(5)
>>> add5(3)
8
>>> add2(1)
3
```

Iterators When an object is used in a `for ... in` statement, the object's `__iter__()` method is called and the returned value should be an iterator. At that point, the interpreter iterates over the result, assigning each object returned from the iterator to the loop variable in the `for ... in` statement.

This example class implements the `__iter__()` method and returns a generator expression based on whatever arguments were passed to the initializer.

```
>>> class Lister(object):
...     def __init__(self, *args):
...         self.items = tuple(args)
...     def __iter__(self):
...         return (i for i in self.items)
...
```

```
>>> l = Lister('a', 'b', 'c')

>>> for letter in l:
...     print letter,
...
a b c
```

Here is the same example using a generator function instead of a generator expression.

```
>>> class Lister(object):
...     def __init__(self, *args):
...         self.items = tuple(args)
...     def __iter__(self):
...         for i in self.items:
...              yield i
...

>>> l = Lister('a', 'b', 'c')

>>> for letter in l:
...     print letter,
...
a b c
```

4.2.2 `classmethod` and `staticmethod`

A class method in Python is defined by creating a method on a class in the standard way, but applying the `classmethod` decorator to the method.

Notice in the following example that instead of `self`, the class method's first argument is named `cls`. This convention is used to clearly denote the fact that in a class method, the first argument received is not a bound instance of the class, but the class object itself.

As a result, class methods are useful when there may not be an existing object of the class type, but the type of the class is important. This example shows a "factory" method, that creates Song objects based on a list of tuples.

Also notice the use of the `__str__()` special method in this example. This method returns a string representation of the object when the object is passed to `print` or the `str()` builtin.

```
>>> class Song(object):
...     def __init__(self, title, artist):
...         self.title = title
...         self.artist = artist

...     def __str__(self):
...         return ('"%(title)s" by %(artist)s' %
...                 self.__dict__)

...     @classmethod
...     def create_songs(cls, songlist):
...         for artist, title in songlist:
...             yield cls(title, artist)
...
>>> songs = (('Glen Hansard', 'Leave'),
...         ('Stevie Ray Vaughan', 'Lenny'))

>>> for song in Song.create_songs(songs):
...     print song
...
"Leave" by Glen Hansard
"Lenny" by Stevie Ray Vaughan
```

Static methods are very similar to class methods and defined using a similar decorator. The important difference is that static methods receive neither an instance object nor a class object as the first argument. They only receive the passed arguments.

As a result, the only real value in defining static methods is code organization. But in many cases a module-level function would do the same job with fewer dots in each call.

```
>>> class Song(object):
...     def __init__(self, title, artist):
...         self.title = title
...         self.artist = artist

...     def __str__(self):
...         return ('"%(title)s" by %(artist)s' %
...                 self.__dict__)

...     @staticmethod
...     def create_songs(songlist):
...         for artist, title in songlist:
...             yield Song(title, artist)
...
>>> songs = (('Glen Hansard', 'Leave'),
...         ('Stevie Ray Vaughan', 'Lenny'))

>>> for song in Song.create_songs(songs):
...     print song
...
"Leave" by Glen Hansard
"Lenny" by Stevie Ray Vaughan
```

4.2.3 Lab

oop-1-parking.py

```
'''
>>> # Create a parking lot with 2 parking spaces
>>> lot = ParkingLot(2)
'''


'''
>>> # Create a car and park it in the lot
>>> car = Car('Audi','R8', '2010')
>>> lot.park(car)
>>> car = Car('VW', 'Vanagon', '1981')
>>> lot.park(car)
>>> car = Car('Buick','Regal', '1988')
>>> lot.park(car)
'Lot Full'
>>> lot.spaces = 3
>>> lot.park(car)
>>> car.make
'Buick'
>>> car.model
'Regal'
>>> car.year
'1988'
>>> for c in lot:
...     print c
2010 Audi R8
1981 VW Vanagon
1988 Buick Regal
```

```
>>> for c in lot.cars_by_age():
...     print c
1981 VW Vanagon
1988 Buick Regal
2010 Audi R8
>>> for c in lot:
...     print c
2010 Audi R8
1981 VW Vanagon
1988 Buick Regal
'''



if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Once you have written classes that pass all the doctests, consider that the doctests do have some value both as documentation and as tests. While we might want to run additional unittests as well we don't want to abandon our doctests. The `doctest` module does provide a function that can return a UnitTest test suite. Depending on your version of Python you may be able to easily find and run these tests - unittest in Python >2.7 includes test discovery.

We'll be using other test discovery tools - but lets learn how to do this the manual way. Create a separate file to hold unit tests and use the `doctest.DocTestSuite` function to create a suite. Make sure it runs - you'll have to look at the docs to see how to run a unittest suite.

4.3 TOPIC: Inheritance

As noted in the first class definition example above, a class defines a superclass using the parentheses list in the class definition. The model for overloading methods is very similar to most other languages: define a method in the child class with the same name as that in the parent class and it will be used instead.

oop-1-inheritance.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True

class PercussionInstrument(Instrument):
    def has_strings(self):
        return False

guitar = Instrument('guitar')
drums = PercussionInstrument('drums')

print 'Guitar has strings: {0}'.format(guitar.has_strings())
print 'Guitar name: {0}'.format(guitar.name)
print 'Drums have strings: {0}'.format(drums.has_strings())
print 'Drums name: {0}'.format(drums.name)
```

```
$ python oop-1-inheritance.py
Guitar has strings: True
Guitar name: guitar
Drums have strings: False
Drums name: drums
```

Calling Superclass Methods Python has a `super()` builtin function instead of a keyword and it makes for slightly clunky syntax. The result, however, is as desired, which is the ability to execute a method on a parent or superclass in the body of the overloading method on the child or subclass.

In this example, an overloaded `__init__()` is used to hard-code the known values for every guitar, saving typing on every instance.

oop-2-super.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True

class StringInstrument(Instrument):
    def __init__(self, name, count):
        super(StringInstrument, self).__init__(name)
        self.count = count

class Guitar(StringInstrument):
    def __init__(self):
        super(Guitar, self).__init__('guitar', 6)

guitar = Guitar()

print 'Guitar name: {0}'.format(guitar.name)
print 'Guitar count: {0}'.format(guitar.count)
```

```
python oop-2-super.py
Guitar name: guitar
Guitar count: 6
```

There is an alternate form for calling methods of the superclass by calling them against the unbound class method and explicitly passing the object as the first parameter. Here is the same example using the direct calling method.

oop-3-super-alt.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True

class StringInstrument(Instrument):
    def __init__(self, name, count):
        Instrument.__init__(self, name)
        self.count = count

class Guitar(StringInstrument):
    def __init__(self):
        StringInstrument.__init__(self, 'guitar', 6)

guitar = Guitar()

print 'Guitar name: {0}'.format(guitar.name)
print 'Guitar count: {0}'.format(guitar.count)
```

```
python oop-3-super-alt.py
Guitar name: guitar
Guitar count: 6
```

Multiple Inheritance Python supports multiple inheritance using the same definition format as single inheritance. Just provide an ordered list of superclasses to the class definition. The order of superclasses provided can affect method resolution in the case of conflicts, so don't treat it lightly.

The next example shows the use of multiple inheritance to add some functionality to a class that might be useful in many different kinds of classes.

oop-4-multiple.py

```
class Instrument(object):
    def __init__(self, name):
        self.name = name
    def has_strings(self):
        return True


class Analyzable(object):
    def analyze(self):
        print 'I am a {0}'.format(self.__class__.__name__)


class Flute(Instrument, Analyzable):
    def has_strings(self):
        return False


flute = Flute('flute')
flute.analyze()
print flute.__class__.__mro__
```

```
$ python oop-4-multiple.py
I am a Flute
(<class '__main__.Flute'>, <class '__main__.Instrument'>, <class '__main__.Analyzable'>, < ↩
    type 'object'>)
```

Resolution order of methods and class attributes is determined by the order of inheritance. Python will look in the left-most classes first, ascending the tree of parents and checking the base `object` class last of all. If you are uncertain about the "method resolution order" of your class you can check the `__mro__` attribute on the class to see the order in which Python looks for attribute resolution.

Abstract Base Classes Python recently added support for abstract base classes. Because it is a more recent addition, its implementation is based on existing capabilities in the language rather than a new set of keywords. To create an abstract base class, override the metaclass in your class definition (metaclasses in general are beyond the scope of this course, but they define how a class is created). Then, apply the `abstractmethod` decorator to each abstract method. Note that both `ABCMeta` and `abstractmethod` need to be imported.

Here is a simple example. Notice that the base class cannot be instantiated, because it is incomplete.

oop-5-abc.py

```
from abc import ABCMeta, abstractmethod
import sys
import traceback

class Instrument(object):
    __metaclass__ = ABCMeta
    def __init__(self, name):
        self.name = name
    @abstractmethod
    def has_strings(self):
        pass
```

```
class StringInstrument(Instrument):
    def has_strings(self):
        return True


guitar = StringInstrument('guitar')
print 'Guitar has strings: {0}'.format(guitar.has_strings())
try:
    guitar = Instrument('guitar')
except:
    traceback.print_exc(file=sys.stdout)
```

```
$ python oop-5-abc.py
Guitar has strings: True
Traceback (most recent call last):
  File "samples/oop-5-abc.py", line 22, in <module>
    guitar = Instrument('guitar')
TypeError: Can't instantiate abstract class Instrument with abstract methods has_strings
```

One feature of abstract methods in Python that differs from some other languages is the ability to create a method body for an abstract method. This feature allows common, if incomplete, functionality to be shared between multiple subclasses. The abstract method body is executed using the `super()` method in the subclass.

oop-6-abcbody.py

```
from abc import ABCMeta, abstractmethod

class Instrument(object):
    __metaclass__ = ABCMeta

    def __init__(self, name):
        self.name = name

    @abstractmethod
    def has_strings(self):
        print 'checking for strings in %s' % \
            self.name


class StringInstrument(Instrument):

    def has_strings(self):
        super(StringInstrument,
                self).has_strings()
        return True


guitar = StringInstrument('guitar')
print 'Guitar has strings: {0}'.format(guitar.has_strings())
```

```
$ python oop-6-abcbody.py
checking for strings in guitar
Guitar has strings: True
```

4.3.1 Lab

oop-2-pets.py

```
'''
>>> cat = Cat('Spike')
```

```
>>> cat.speak()
'Spike says "Meow"'
>>> dog = Dog('Bowzer')
>>> cat.can_swim()
False
>>> dog.can_swim()
True
>>> dog.speak()
'Bowzer says "Woof"'
>>> fish = Fish('Goldie')
>>> fish.speak()
"Goldie can't speak"
>>> fish.can_swim()
True
>>> generic = Pet('Bob')
Traceback (most recent call last):
    ...
TypeError: Can't instantiate abstract class Pet with abstract methods can_swim
'''

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

4.4 TOPIC: Encapsulation

As mentioned previously, while Python does not support declarations of attribute visibility (public, private, etc), it does provide mechanisms for encapsulation of object attributes. There are three approaches with different levels of capability that can be used for this purpose. In this course we won't cover the usage of descriptors, but be aware that descriptors provide a very powerful API for encapsulating attribute access. Python itself uses descriptors to implement *properties* which provide a simpler and, in most cases, powerful enough mechanism to encapsulate data access. We'll also discuss using the *getattr* family of magic methods to provide generic interception of the . operator.

4.4.1 Intercepting Attribute Access

When an attribute of an object is accessed using dot-notation, there are three special methods of the object that may get called along the way.

For lookups, two separate methods are called: `__getattribute__(self, name)` is called first, passing the name of the attribute that is being requested. Overriding this method allows for the interception of requests for any attribute. By contrast, `__getattr__()` is only called when `__getattribute__()` fails to return a value. So this method is useful if only handling undefined cases.

For setting attributes only one method, `__setattr__(self, name, value)` is called. Note that inside this method body, calling `self.name = value` will lead to infinite recursion. Use the superclass `object.__setattr__(self, name, value)` instead.

The following example shows a course with two attributes `capacity` and `enrolled`. A third attribute `open` is calculated based on the other two. However, setting it is also allowed and forces the enrolled attribute to be modified.

oop-7-intercept.py

```
import traceback
import sys

class Course(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.enrolled = 0
```

```
    def enroll(self):
        self.enrolled += 1

    def __getattr__(self, name):
        if name == 'open':
            return self.capacity - self.enrolled
        else:
            raise AttributeError('%s not found', name)

    def __setattr__(self, name, value):
        if name == 'open':
            self.enrolled = self.capacity - value
        else:
            object.__setattr__(self, name, value)

    def __str__(self):
        return 'Enrolled: \t{0}\nCapacity:\t{1}\nOpen:\t{2}'.format(
                self.enrolled, self.capacity, self.open)

course = Course(12)
course.enroll()
course.enroll()

print course

course.open = 8

print course
```

```
$ python oop-7-properties.py
Enrolled:   2
Capacity: 12
Open: 10
Enrolled:   4
Capacity: 12
Open: 8
```

4.4.2 Properties

For the simple case of defining a calculated property as shown in the above example, the more appropriate (and simpler) model is to use the `property` decorator to define a method as a propery of the class.

Here is the same example again using the `property` decorator. Note that this approach does not handle setting this property. Also notice that while `open()` is initially defined as a method, it cannot be accessed as a method.

oop-8-properties.py

```
import traceback
import sys

class Course(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.enrolled = 0

    def enroll(self):
        self.enrolled += 1

    @property
```

```
    def open(self):
        return self.capacity - self.enrolled

course = Course(12)
course.enroll()
course.enroll()

print 'Enrolled: \t{0}\nCapacity:\t{1}\nOpen:\t{2}'.format(course.enrolled,
        course.capacity, course.open)

print
try:
    course.open()
except:
    traceback.print_exc(file=sys.stdout)

print
try:
    course.open = 9
except:
    traceback.print_exc(file=sys.stdout)
```

```
$ python oop-8-properties.py
Enrolled:   2
Capacity: 12
Open: 10

Traceback (most recent call last):
  File "samples/oop-8-properties.py", line 25, in <module>
    course.open()
TypeError: 'int' object is not callable

Traceback (most recent call last):
  File "samples/oop-8-properties.py", line 31, in <module>
    course.open = 9
AttributeError: can't set attribute
```

Using the property mechanism, setters can also be defined. A second decorator is dynamically created as `<attribute name>.setter` which must be applied to a method with the **exact same name** but an additional argument (the value to be set).

In this example, we use this additional functionality to encapsulate the speed of a car and enforce a cap based on the type of car being manipulated.

oop-9-propertysetters.py

```
class Car(object):
    def __init__(self, name, maxspeed):
        self.name = name
        self.maxspeed = maxspeed
        self.__speed = 0

    @property
    def speed(self):
        return self.__speed

    @speed.setter
    def speed(self, value):
        s = int(value)
        s = max(0, s)
        self.__speed = min(self.maxspeed, s)
```

```
car = Car('Lada', 32)
car.speed = 100
print 'My {name} is going {speed} mph!'.format(name=car.name, speed=car.speed)

car.speed = 24
print 'My {name} is going {speed} mph!'.format(name=car.name, speed=car.speed)
```

```
$ python oop-9-propertysetters.py
My Lada is going 32 mph!
My Lada is going 24 mph!
```

Properties are a great feature of Python's OOP implementation. Unlike languages that require you to write boilerplate getter/setter methods initially, Python allows you to change your mind after the fact. You can allow plain attribute access and then change your implementation to intercept attribute access without changing the public API of your objects!

4.4.3 Lab

We'll be working on a larger project for the remainder of the class - using the functional and object oriented techniques we've learned so far and expanding our knowledge of testing techniques and tools.

The project description is to build an API and command-line interface to a unified reporting system. We want to extract data from a variety of data sources but output a unified data-stream. We'll use a simple plugin architecture to provide data sources and configurable output formatters/filters.

Let's go ahead and set up the file structure for our lab. We'll create a new directory "reports" and add some files to the new directory:

Directory Listing

```
$ ls -l reports
total 8
-rw-r--r-- 1 simeon simeon    0 2012-08-19 18:46 __init__.py
-rw-r--r-- 1 simeon simeon    0 2012-08-19 18:46 report.py
drwxr-xr-x 2 simeon simeon 4096 2012-08-19 18:52 tests
```

Remember your implementation of `chain` from the *Advanced Iteration* section? Go ahead and create the directory structure above, but copy your *chained_iterators.py* file to *report.py* here and drop your *test_yield.py* into the tests directory.

Our `chain` function will build a pipeline of data but we'd like to intercept specially formatted strings (essentially URI's) and use them to construct iterators to be chained. We'll start with an easy iterator to construct:

1. Create a decorator `accept_sources` in `report.py` that wraps `chain`. We should use `accept_sources to intercept arguments to `chain` and replace any argument strings of the form "file://foo.txt" with an iterator over the lines in the file foo.txt. You shouldn't have to write a class for this!
2. Write tests to verify the correct behavior of the decorator. Write tests to verify that the chain function now can chain the contents of a file with another iterable. Tests should be unittest classes living in `tests/test_decorator.py`.
3. Modify the decorator to support strings of the form "zip://foo.zip/foo.txt" where *foo.zip* is a zip file and *foo.txt* is a text file in the zip file. In this case you should write a class that supports iteration and holds the logic for reading from a zip archive, extracting a file, and returning an iterator over its lines. Be sure your decorator continues to be well tested and your new class also has tests.

4.5 *Object Oriented* Follow-up

This chapter covered the major concepts of Object Oriented programming in Python. You've gained experience writing classes and using Python's emulation features to use your objects in an intuitive and Pythonic manner. You've also explored how inheritance works and used more advanced concepts like the `@property` decorator to separate the details of your implementation from the interface to your objects. This should be enough coverage to enable you to use object oriented thinking and design to solve real world problems in your own programs.