Protocols

or: The Magicians Secrets Revealed

What is a "protocol"?

Protocols are similar to interfaces, in that they define a collection of methods an object must support to implement that protocol, but different in that they have language-level recognition and syntactic support. A great example is the iteration protocol...

Iteration

```
for item in some_collection:
    print item
```

How does the above code work for all collection types?

The secret is the iteration protocol. As long as `some_collection` implements a couple of methods it can be used as an iterator.

http://docs.python.org/library/stdtypes.html#iterator-types

Iteration - Example

```
def reverse_iter(seq):
    position = len(seq) - 1
    while True:
        if position < 0:
            raise StopIteration
        yield seq[position]
        position -= 1

class BackwardsSequence(list):
    def __iter__(self):
        return reverse_iter(self)
```

Other Protocols

Here are a few of the many protocols Python supports for customizing your objects interaction with the language:

Comparison (`__eq__`, `__gt__`, `__lt__`)
Containers (`__contains__`, `__setitem__`, `__getitem__`)
Iterators (`__iter__`, `next`)
Context Managers (`__enter__`, `__exit__`)
Stringification (`__str__`, `__unicode__`, `__repr__`)
Descriptors (`__get__`, `__set__`)
Instance Creation (`__new__`, `__metaclass__` attribute)

Container Protocol

By implementing the appropriate methods, it's simple to make your class act like one of the built in container types.

The following example creates a set of "proxy" classes that wrap a Cassandra library to support indexed lookup:

```
keyspace['column_family']['rowkey']['column_name']
```

Container - Example (1/3)

```
from pycassa import *

class KeyspaceProxy(object):
    def __init__(self, keyspace, connection_pool):
        self._keyspace = keyspace
        self._pool = connection_pool

    def __getitem__(self, cf):
        return ColumnFamilyProxy(self._pool, cf)
```

Container - Example (2/3)

```
class ColumnFamilyProxy(object):
    def __init__(self, pool, cf_name):
        self._cf = ColumnFamily(pool, cf_name)

    def __getitem__(self, rowkey):
        return RowProxy(self._cf, rowkey)
```

Container - Example (3/3)

```
class RowProxy(object):
    def __init__(self, column_family, rowkey):
        self._cf = column_family
        self._rowkey = rowkey

    def __getitem__(self, column_name):
        row = self._cf.get(self._rowkey, [column_name])
        return row[column_name]

    def __setitem__(self, column_name, value):
        self._cf.insert(self._rowkey,
                        {column_name: value})
```

Attribute Access

Another commonly used special method is `__getattr__`. It is called when `your_obj.attr_name` is accessed and does *not* exist, (i.e. right before an AttributeError is raised).

```
class Mock(object):
    def __getattr__(self, attr_name):
        setattr(self, attr_name) = Mock()
        return self.attr_name
```

This is a simplification of an actual mocking library I use called mock.

Descriptors

Descriptors are the mechanism by which method and attribute binding happens in Python. When you perform an attribute access on an object (e.g. `object.attribute`) Python gives `attribute` an opportunity to customize it's behaviour at access time, **if** it implements the descriptor protocol.

The `object.attribute` call gets transformed into something like `attribute.__get__(object)` and the return value of `__get__` is what appears to be `object.attribute` to the calling code.

Descriptor - Example

```
# Attach a new function to an existing object

class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

me = Person("Stephen", 27)
```

Descriptor - Example

```
def sleep(self):
    if self.age <= 25:
        print "later"
    else:
        print "zzz"

# Bind the sleep method to myself
me.sleep = sleep.__get__(me)
me.sleep() # Prints "zzz"
```

The End

Python protocols allow you to take full advantage of the complete syntax of the language in your own objects. Writing library code against simple built-in types that can be used with any type that supports the operations needed is a great example of *duck-typing* in Python: if it walks like a duck and talks like a duck, code that only needs a duck shouldn't care that it's a subclass of `DeadlyRobot` with a `DuckMixin`.