Python Data Model

Python's Data Model: Overview

The Data Model specifies how objects, attributes, methods, etc. function and interact in the processing of data.

The Python Language Reference (https://docs.python.org/2/reference/index.html) provides a clear introduction to Python's lexical analyzer, data model, execution model, and various statement types.

A familiarity with the data model can be helpful when trying to solve problems and in becoming conversant on StackOverflow and other discussion sites.

Special ("Magic") Attributes

All objects contain "private" attributes that may be methods that are indirectly called, or internal "meta" information for the object.

The **__dict__** attribute shows any attributes stored in the object.

```
>>> list.__dict__.keys()
['__getslice__', '__getattribute__', 'pop', 'remove', '__rmul__', '__lt__
 '__init__', 'count', 'index', '__delslice__', '__new__', '__contains__',
 '__doc__', '__len__', '__mul__', 'sort', '__ne__', '__getitem__', 'inser
 '__setitem__', '__add__', '__gt__', '__eq__', 'reverse', 'extend', '__de
 '__reversed__', '__imul__', '__setslice__', '__iter__', '__iadd__', '__l
 '__hash__', '__ge__']
```

The **dir()** function will show the object's available attributes, including those available through inheritance.

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '_
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__geti
 '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__'
 '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce
 '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
 '__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append', '
 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

In this case, dir(list) includes attributes not found in list.__dict__. What class(es) does list inherit from? We can use **__bases__** to see:

```
>>> list.__bases__
(object,)
```

This is a tuple of classes from which **list** inherits - in this case, just the super object **object**.

```
>>> object.__dict__.keys()
['__setattr__', '__reduce_ex__', '__new__', '__reduce__', '__str__', '__f
 '__getattribute__', '__class__', '__delattr__', '__subclasshook__', '__r
 '__hash__', '__sizeof__', '__doc__', '__init__']
```

Of course this means that any object that inherits from **object** will have the above attributes. Most if not all built-in objects inherit from **object**.

"Introspection" Special Attributes

The name, module, file, arguments, documentation, and other "meta" information for an object can be found in special attributes.

For example, this line placed within a function prints the function name, which can be useful for debugging:

```
if debug:  print 'entering {}()'.format(sys._getframe().f_code.co_name )
```

If **debug** is set to **True**, the **frame** object's function name can be printed:

```
entering myfunc()
```

Below is a partial listing; available attributes are discussed in more detail on the data model documentation page (https://docs.python.org/2/reference/datamodel.html).

user-defined functions

__doc__	doc string
__name__	this function's name
__module__	module in which this func is defined
__defaults__	default arguments
__code__	the "compiled function body" of bytecode of this function. Code objects can be inspected with the inspect (https://docs.python.org/2.7/library/inspect.html) module and "disassembled" with the dis (https://docs.python.org/2.7/library/dis.html) module.
__globals__	global variables available from this function
__dict__	attributes set in this function object by the user

user-defined methods

im_class	class for this method
__self__	instance object
__module__	name of the module

modules

__dict__	globals in this module
__name__	name of this module

__doc__	docstring
__file__	file this module is defined in

classes

__name__	class name
__module__	module defined in
__bases__	classes this class inherits from
__doc__	docstring

class instances (objects)

im_class	class
im_self	this instance

Object Inspection And Modification Built-in Functions

Object Inspection

isinstance()	Checks to see if this object is an instance of a class (or parent class)
issubclass()	Checks to see if this class is a subclass of another
callable()	Checks to see if this object is callable
hasattr()	Checks to see if this object has an attribute of this name

Object Attribute Modification

setattr()	sets an attribute in an object (using a string name)
getattr()	retrieves an attribute from an object (using a string name)
delattr()	deletes an attribute from an object (using a string name)

Special Attributes: "operator overloading"

Some special attributes are methods, usually called implictly as the result of function calls, use of operators, subscripting or slicing, etc.

We can replace any operator and many functions with the corresponding magic methods to achieve the same result:

```
var = 'hello'
var2 = 'world'

print var + var2         # helloworld
print var.__add__(var2)  # helloworld

print len(var)           # 5
print var.__len__()      # 5

if 'll' in var:
    print 'yes'

if var.__contains__('ll'):
    print 'yes'
```

Here is an example of a new class, **Number**, that reproduces the behavior of a number in that you can add, subtract, multiply, divide them with other numbers.

```
class Number(object):
  def __init__(self, start):
    self.data = start
  def __sub__(self, other):
    return Number(self.data - other)
  def __add__(self, other):
    return Number(self.data + other)
  def __mul__(self, other):
    return Number(self.data * other)
  def __div__(self, other):
    return Number(self.data / float(other))
  def __repr__(self):
    print "Number value: ",
    return str(self.data)

X = Number(5)
X = X - 2
print X               # Number value: 3
```

Of course this means that existing built-in objects make use of these methods -- you can find them listed from the object's **dir()** listing.

Special Attributes: Reimplementing __repr__ and __str__

__**str**__ is invoked when we print an object or convert it with **str()**; __**repr**__ is used when __**str**__ is not available, or when we view an object at the Python interpreter prompt.

```
class Number(object):
  def __init__(self, start):
    self.data = start
  def __str__(self):
    return str(self.data)
  def __repr__(self):
    return 'Number(%s)' % self.data

X = Number(5)
print X          # 5  (uses __str__ -- without repr or str, would be <__m
```

__str__ is intended to display a human-readable version of the object; **__repr__** is supposed to show a more "machine-faithful" representation.

Special attributes available in class design

Here is a short listing of attributes available in many of our standard objects.

You view see many of these methods as part of the attribute dictionary through **dir()**.

There is also a more exhaustive exhaustive list (http://www.rafekettler.com/magicmethods.html#reflection) with explanations provided by Rafe Kettler.

object construction and destruction:

__init__	object constructor
__del__	del x (invoked when reference count goes to 0)
__new__	special 'metaclass' constructor

object rendering:

__repr__	"under the hood" representation of object (in Python interpreter)
__str__	string representation (i.e., when printed or with str()

object comparisons:

__lt__	<
__le__	<=
__eq__	==
__ne__	!=
__gt__	>

__ge__	>=
__nonzero__	(bool(), i.e. when used in a boolean test)

calling object as a function:

__call__	when object is "called" (i.e., with **()**)

container types:

__len__	handles len() function
__getitem__	subscript access (i.e. mylist[0] or mydict['mykey'])
__missing__	handles missing keys
__setitem__	handles dict[key] = value
__delitem__	handles del dict[key]
__iter__	handles looping
__reversed__	handles reverse() function
__contains__	handles 'in' operator
__getslice__	handles slice access
__setslice__	handles slice assignment
__delslice__	handles slice deletion

attribute access:

__getattr__	object.attr read: attribute may not exist
__getattribute__	object.attr read: attribute that already exists
__setattr__	object.attr write
__delattr__	object.attr deletion (i.e., del this.that)

'descriptor' class methods (discussed next session)

__get__	when an attribute w/descriptor is read
__set__	when an attribute w/descriptor is written
__delete__	when an attribute w/descriptor is deleted with **del**

numeric types:

__add__	addition with +	
__sub__	subtraction with -	
__mul__	multiplication with *	
__div__	division with \/	
__floordiv__	"floor division", i.e. with //	
__mod__	modulus	

Variable Naming Conventions

Underscores are used to designate variables as "private" or "special".

lower-case separated by underscores	**my_nice_var**	"public", intended to be exposed to users of the module and/or class
underscore before the name	**_my_private_var**	"non-public", *not* intended for importers to access (additionally, "from modulename import *" doesn't import these names)
double-underscore before the name	**__dont_inherit**	"private"; its name is "mangled", available only as _classname__dont_inherit
double-underscores before and after the name	**__magic_me__**	"magic" attribute or method, specific to Python's internal workings

```
class GetSet(object):

    instance_count = 0

    __mangled_name = 'no privacy!'

    def __init__(self,value):
        self._attrval = value
        instance_count += 1

    def getvar(self):
        print('getting the "var" attribute')
        return self._attrval

    def setvar(self, value):
        print('setting the "var" attribute')
        self._attrval = value

cc = GetSet(5)
cc.var = 10
print cc.var
print cc.instance_count

print cc._attrval                 # "private", but available:  10
print cc.__mangled_name           # "private", apparently not available..
print cc._GetSet__mangled_name    # ...and yet, accessible through "mangl

cc.__newmagic__ = 10              # MAGICS ARE RESERVED BY PYTHON -- DON'
```

Internal Types

Some implicit objects can provide information on code execution.

Traceback objects

Traceback objects become available during an exception. Here's an example of inspection of the exception type using **sys.exc_info()**

```
import sys, traceback
try:
    some_code_i_wrote()
except BaseException, e:
    error_type, error_string, error_tb =  sys.exc_info()
    if not error_type == SystemExit:
        print 'error type:    {}'.format(error_type)
        print 'error string:  {}'.format(error_string)
        print 'traceback:     {}'.format(''.join(traceback.format_excepti
```

Code objects
In CPython (the most common distribution), a code object is a piece of compiled bytecode. It is possible to query this object / examine its attributes in order to learn about bytecode execution. A detailed exploration of code objects can be found here (https://late.am/post/2012/03/26/exploring-python-code-objects.html).

Frame objects
A frame object represents an execution frame (a new frame is entered each time a function is called). They can be found in traceback objects (which trace frames during execution).

f_back	previous stack frame
f_code	code object executed in this frame
f_locals	local variable dictionary
f_globals	global variable dictionary
f_builtins	built-in variable dictionary

Again here is a way to report the name of a function from within it. Note that we're pulling a frame, grabbing the code object of that frame, and reading the attribute **co_name** to read it.

```
if debug:  print 'entering {}()'.format(sys._getframe().f_code.co_name )
```

If **debug** is set to **True**, the **frame** object's function name can be printed:

```
entering myfunc()
```

Subclassing Builtin Objects

Inheriting from a builtin object makes all of that object's functionality available in the inheriting class, giving us the opportunity to override or specialize selected behaviors of the builtin.

One of the advantages of working with new-style classes is the ability to inherit from our old friends the built-in classes (**list**, **dict**, etc.) and make use of their functionality in the inheriting class. We then have the opportunity to *reimplement* any of its methods to turn its functionality to our own (nefarious) purposes.

Here's an example from Guido van Rossum himself, exploring the subclassing of the **dict** built-in type. The purpose of this dict-like class is to automatically return a default value from a dict if the key was not found:

```
class DefaultDict(dict):

    def __init__(self, default=None):
        dict.__init__(self)
        self.default = default

    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            return self.default
    def get(self, key, userdefault):
        if not userdefault:
            userdefault = self.default
        return dict.get(self, key, userdefault)

xx = DefaultDict()

xx['c'] = 5

print xx['c']          # 5
print xx['a']          # None
```

Keep in mind that because **DefaultDict** inherits from **dict**, all other methods (including key/value setting) are available to this object type through inheritance lookup.

Here's another example that lets us create a list object that indexes starting at 1 instead of 0:

```
class MyList(list):         # inherit from list
  def __getitem__(self, index):
    if index == 0:  raise IndexError
    if index > 0: index = index - 1
    return list.__getitem__(self, index)  # this method is called when we
                                                       # a value with subscript
  def __setitem__(self, index, value):
    if index == 0:  raise IndexError
    if index > 0: index = index - 1
    list.__setitem__(self, index, value)

x = MyList(['a', 'b', 'c'])  # __init__() inherited from builtin list

print x                      # __repr__() inherited from builtin list

x.append('spam');            # append() inherited from builtin list

print x[1]                   # 'a' (MyList.__getitem__
                             #      customizes list superclass method)
                             # index should be 0 but it is 1!

print x[4]                   # 'spam' (index should be 3 but it is 4!)
```

The point here is that our **MyList** class acts like a **list** and behaves like a **list** (and *isa* **list** because it inherits from **list**) but when it comes to accessing values, it uses indices that start at 1 instead of 0 because of our method overrides.

WARNING! Avoiding method recursion

Look closely at the two prior class examples, and look for the work being done to update the instance itself:

```
    # from MyList.__getitem__()
    return list.__getitem__(self, index)   # why not self[index]?

    # from MyList.__getitem__()
    list.__setitem__(self, index, value)   # why not self[index] = value?

    # from DefaultDict.__getitem__()
    dict.get(self, key, userdefault)       # why not self.get(key, userde
```

What are we doing in these lines? We are calling the parent method, and passing our own instance to it along with the other arguments needed for the method call. Why shouldn't we just operate on the **dict** or **list** directly? Because this would cause Python to call our own class' method, which would then reach the same operation that calls the method -- we would have created an endless recursive loop, in which a method calls itself endlessly (the program will stop when the **maximum recursive depth** is reached, so it won't overtax your memory.

So the solution is to call the parent class method in each case, which works exactly the way we want (i.e., it stores it or retrieves it normally after our own process has modified the values as desired).