Objects and classes in Python Documentation

Release 0.1

Jonathan Fine

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Decorators

This section cover *the decorator syntax* and the concept of a decorator (or decorating) callable.

Decorators are a syntactic convenience, that allows a Python source file to say what it is going to do with the result of a function or a class statement before rather than after the statement. Decorators on function statements have been available since Python 2.4, and on class statements since Python 2.6.

In this section we describe the decorator syntax and give examples of its use. In addition, we will discuss functions (and other callables) that are specifically designed for use as decorators. They are also called decorators.

You can, and in medium sized or larger projects probably should, write your own decorators. The decorator code might, unfortunately, be a little complex. But it can greatly simplify the other code.

The decorator syntax

The decorator syntax uses the @ character. For function statements the following are equivalent:

```
# State, before defining f, that a_decorator will be applied to it.
@a_decorator
def f(...):
    ...
```

```
def f(...):
    ...

# After defining f, apply a_decorator to it.
f = a_decorator(f)
```

The benefits of using the decorator syntax are:

1. The name of the function appears only once in the source file.
2. The reader knows, before the possibly quite long definition of the function, that the decorator function will be applied to it.

The decorator syntax for a class statement is same, except of course that it applies to a class statement.

Bound methods

Unless you tell it not to, Python will create what is called a bound method when a function is an attribute of a class and you access it via an instance of a class. This may sound complicated but it does exactly what you want.

```
>>> class A(object):
...     def method(*argv):
...         return argv
>>> a = A()
>>> a.method
<bound method A.method of <A object at 0x...>>
```

When we call the bound method the object *a* is passed as an argument.

```
>>> a.method('an arg')
(<A object at 0x...>, 'an arg')
>>> a.method('an arg')[0] is a
True
```

staticmethod()

A static method is a way of suppressing the creation of a bound method when accessing a function.

```
>>> class A(object):
...     @staticmethod
...     def method(*argv):
...         return argv
>>> a = A()
>>> a.method
<function method at 0x...>
```

When we call a static method we don't get any additional arguments.

```
>>> a.method('an arg')
('an arg',)
```

classmethod()

A class method is like a bound method except that the class of the instance is passed as an argument rather than the instance itself.

```
>>> class A(object):
...     @classmethod
...     def method(*argv):
...         return argv
>>> a = A()
>>> a.method
<bound method type.method of <class 'A'>>
```

When we call a class method the class of the instance is passed as an additional argument.

```
>>> a.method('an arg')
(<class 'A'>, 'an arg')
>>> a.method('an arg')[0] is A
True
```

In addition, class methods can be called on the class itself.

```
>>> A.method('an arg')
(<class 'A'>, 'an arg')
```

The `call()` decorator

Suppose we want to construct a lookup table, say containing the squares of positive integers for 0 to *n*.

For n small we can do it by hand:

```
>>> table = [0, 1, 4, 9, 16]
>>> len(table), table[3]
(5, 9)
```

Because the formula is simple, we could also use a list comprehension:

```
>>> table = [i * i for i in range(5)]
>>> len(table), table[3]
(5, 9)
```

Here's another way, that uses a helper function (which we will call *table*). For a table of squares list comprehension is better, because we can write an expression that squares. But for some tables a complex sequence of statements is required.

```
>>> def table(n):
...     value = []
...     for i in range(n):
...         value.append(i*i)
...     return value
>>> table = table(5)
```

We call the helper function *table* for three related reasons

1. It indicates the purpose of the function.
2. It ensures that the helper function is removed from the namespace once the table has been constructed.
3. It conforms to the decorator syntax.

As before, we test the table and find that it works. >>> len(table), table[3] (5, 9)

```
>>> def call(*argv, **kwargs):
...     def call_fn(fn):
...         return fn(*argv, **kwargs)
...     return call_fn
```

```
>>> @call(5)
... def table(n):
...     value = []
...     for i in range(n):
...         value.append(i*i)
...     return value
```

```
>>> len(table), table[3]
(5, 9)
```

Nesting decorators

The decorator syntax can be nested. The following example is similar to the list comprehension approach, except that it uses a generator function rather than a generator expression.

```
>>> @list
... @call(5)
... def table(n):
...     for i in range(n):
...         yield i * i
```

We read this as saying:

> The value of *table* is the list obtained by iterating over the function evaluated at n equal to 5.

The purpose of this example is illustrate some of the concepts. We are not saying that it is, or is not good programming practice. That will depend, in part, on the context.

As before, we test the table and find that it works.

```
>>> len(table), table[3]
(5, 9)
```

Class decorators before Python 2.6

Prior to Python 2.6 one could not write

```
@a_decorator
class MyClass(...):

      # possibly many lines of code.
```

If you need to support earlier versions of Python, I recommend that you develop in Python 2.6 or later. This allows your mind and keyboarding to use decorators. Once the decorating code is stable refactor it to support earlier versions of Python, as follows.

```
# @a_decorator
class MyClass(...):

      # possibly many lines of code.

MyClass = a_decorator(MyClass)    # if changed, change decorator comment.
```

This approach allows you to think and largely code using the class decorator point of view, at the cost of having to keep the decorator comment up to date when the decorator changes.

CHAPTER 2

Constructing classes

There are two basic ways of constructing classes in Python. The best known way is to use Python's `class` statement. The other way is to use Python's `type()` function. This page covers the statement way. *type(name, bases, dict)* is more powerful and sometimes more convenient. However, the statement approach is the better way to get started and in ordinary programming is the most convenient.

The empty class

We will start with the empty class, which is not as empty as it looks.

```
>>> class A(object):
...     pass
```

Like most Python objects, our empty class has a dictionary. The dictionary holds the attributes of the object.

```
>>> A.__dict__
<dictproxy object at 0x...>
```

Even though our class is empty, its dictionary (or more exactly dictproxy) is not.

```
>>> sorted(A.__dict__.keys())
['__dict__', '__doc__', '__module__', '__weakref__']
```

Attributes __doc__ and __module__ are there for documentation, and to give better error messages in tracebacks. The other attributes are there for system purposes.

In addition, our class two attributes that are not even listed in the dictionary. The __bases__ attribute is the list of base classes provided in the original class statement.

```
>>> A.__bases__
(<type 'object'>,)
```

The method resolution order (mro) attribute __mro__ is computed from the bases of the class. It provides support for multiple inheritance.

```
>>> A.__mro__
(<class 'A'>, <type 'object'>)
```

For now the important thing is that even the empty class has attributes. (For IronPython and Jython the attributes are slightly different.)

CHAPTER 3

dict_from_class()

In this section we define a functions that gets a dictionary from a class. This dictionary contains all the information supplied in the body of a class statement, except for the doc-string.

The __dict__ of the empty class

Here's our empty class again:

```
>>> class A(object):
...     pass
```

As seen in *Constructing classes*, even for the empty class its class dictionary has entries. Handling these always-there entries is a nuisance when deconstructing classes. Here, once again, is the list of entries.

```
>>> sorted(A.__dict__.keys())
['__dict__', '__doc__', '__module__', '__weakref__']
```

The __dict__ and __weakref__ entries are there purely for system purposes. This makes them easier to deal with.

The class docstring __doc__ is None unless the user supplies a value.

```
>>> A.__doc__ is None
True
```

```
>>> class A2(object):
...     'This is the docstring'
```

```
>>> A2.__doc__
'This is the docstring'
```

Ordinarily, __module__ is the name of the module in which the class is defined. However, because of the way Sphinx uses doctest, it gets the name of the module wrong. Please don't worry about this. Despite what it says below, it's the name of the module.

```
>>> A.__module__
'__builtin__'
```

Is the doc-string part of the body?

Soon we will define a function that copies the body, as a dictionary, out of a class. But first we must answer the question: Is the doc-string part of the body of a class?

There is no completely satisfactory answer to this question, as there are good arguments on both sides. We choose NO, because for example using the -OO command line option will *remove doc-strings*, and so they are not an essential part of the body of the class. (However, -OO does *not* remove doc-strings produced explicitly, by assigning to __doc__.)

The keys to be excluded are precisely the ones that the empty class (which has an empty body) has.

```
>>> _excluded_keys = set(A.__dict__.keys())
```

Definition of `dict_from_class()`

This function simply filters the class dictionary, copying only the items whose key is not excluded.

```
>>> def dict_from_class(cls):
...     return dict(
...         (key, value)
...         for (key, value) in cls.__dict__.items()
...         if key not in _excluded_keys
...         )
```

As expected, the empty class has an empty body.

```
>>> dict_from_class(A)
{}
```

Here's a class whose body is not empty.

```
>>> class B(object):
...     'This docstring is not part of the body.'''
...     s = 'a string'
...     def f(self): pass
```

We get what we expect for the body. (See [somewhere] for why we need the __func__.)

```
>>> dict_from_class(B) == dict(s='a string', f=B.f.__func__)
True
```

Here's another way of expressing the same truth.

```
>>> sorted(dict_from_class(B).items())
[('f', <function f at 0x...>), ('s', 'a string')]
```

CHAPTER 4

property_from_class()

This section shows how using a class decorator, based upon *dict_from_class()*, can make it much easier to define complex properties. But first we review properties.

About properties

The `property()` type is a way of 'owning the dot' so that attribute getting, setting and deletion calls specified functions.

One adds a property to a class by adding to its a body a line such as the following, but with suitable functions for some or all of fget, fset and fdel. One can also specify doc to give the property a doc-string.

```
attrib = property(fget=None, fset=None, fdel=None, doc=None)
```

If all one wants is to specify fset (which is a common case) you can use property as a decorator. This works because fget is the first argument.

For example, to make the area of a rectangle a read-only property you could write:

```
@property
def attrib(self):
    return self.width * self.length
```

Suppose now you have a property that you wish to both get and set. Here's the syntax we'd like to use.

```
@property_from_class
class attrib(object):
    '''Doc-string for property.'''

    def fget(self):
        '''Code to get attribute goes here.'''

    def fset(self):
        '''Code to set attribute goes here.'''
```

We will now construct such a decorator.

Definition of `property_from_class()`

This function, designed to be used as a decorator, is applied to a class and returns a property. Notice how we pick up the doc-string as a separate parameter. We don't have to check for unwanted keys in the class dictionary - `property()` will do that for us.

```
>>> def property_from_class(cls):
...
...     return property(doc=cls.__doc__, **dict_from_class(cls))
```

Using `property_from_class()`

Here is an example of its use. We add a property called value, which stores its data in _value (which by Python convention is private). In this example, we validate the data before it is stored (to ensure that it is an integer).

```
>>> class B(object):
...     def __init__(self):
...         self._value = 0
...
...     @property_from_class
...     class value(object):
...         '''The value must be an integer.'''
...         def fget(self):
...             return self._value
...         def fset(self, value):
...             # Ensure that value to be stored is an int.
...             assert isinstance(value, int), repr(value)
...             self._value = value
```

Here we show that `B` has the required properties.

```
>>> b = B()
>>> b.value
0
```

```
>>> b.value = 3
```

```
>>> b.value
3
```

```
>>> B.value.__doc__
'The value must be an integer.'
```

```
>>> b.value = 'a string'
Traceback (most recent call last):
AssertionError: 'a string'
```

Unwanted keys

If the class body contains a key that property does not accept we for no extra work get an exception (which admittedly could be a clearer).

```
>>> @property_from_class
... class value(object):
...     def get(self):
```

```
...         return self._value
Traceback (most recent call last):
TypeError: 'get' is an invalid keyword argument for this function
```

CHAPTER 5

Deconstructing classes

In *Constructing classes* we saw how to construct a class (by using the **class** keyword). In this section we see how to reverse the process.

To use the **class** keyword you have to specify:

1. A name for your class.
2. A tuple of bases.
3. A class body.

In this section we see how to get this information back again. Let's do the easy stuff first. Here's our empty class again:

```
>>> class A(object):
...     pass
```

Here's how to get the name of the class:

```
>>> A.__name__
'A'
```

And here's how to get the bases:

```
>>> A.__bases__
(<type 'object'>,)
```

[To be continued.]

CHAPTER 6

type(name, bases, dict)

According to its docstring, there are two ways to call the `type()` builtin.

```
>>> print type.__doc__
type(object) -> the object's type
type(name, bases, dict) -> a new type
```

In this section we explore how to use `type()` to construct new classes.

Constructing the empty class

As usual, we start with the empty class. The __name__ attribute of the class need not be the same as the name of the variable in which we store the class. When at top-level (in the module context) the class command binds the class to the module object, using the name of the class as the key.

When we use type, there is no link between the __name__ and the binding.

```
>>> cls = type('A', (object,), {})
```

The new class has the name we expect.

```
>>> cls.__name__
'A'
```

Its docstring is empty.

```
>>> cls.__doc__ is None
True
```

It does not have a __module__ attribute, which is surprising.

```
>>> cls.__module__
Traceback (most recent call last):
AttributeError: __module__
```

This class does not have a __module__ attribute because to the things that Sphinx does when running the doctest. Ordinarily, the class will have a __module__ attribute.

```
>>> sorted(cls.__dict__.keys())
['__dict__', '__doc__', '__weakref__']
```

The lack of a __module__ attribute explains the string representation of the class.

```
>>> cls
<class 'A'>
```

Constructing any class

We obtained the empty class, whose __dict__ has only the system keys, by passing the empty dictionary to `type()`. We obtain more interesting classes by passing a non-empty dictionary. We can at the same time pass more interesting bases, in order to achieve inheritance.

Specifying __doc__, __name__ and __module__

Let's try to use the dict argument to specify these special attributes.

```
>>> body = dict(__doc__='docstring', __name__='not_A', __module__='modname')
>>> cls2 = type('A', (object,), body)
```

We have set the __docstring__ and __module__ attributes, but the __name__ is still **A**.

```
>>> cls2.__doc__, cls2.__name__, cls2.__module__
('docstring', 'A', 'modname')
```

CHAPTER 7

Subclassing `int`

We subclass in order to create a new class whose behaviour is inherited from the base classes, except that the new class can also override and add behaviour. Object creation is behaviour. For most classes it is enough to provide a different __init__ method, but for immutable classes one often have to provide a different __new__ method.

In this section we explain why __new__ is needed, and give examples of its use. But first we review mutation.

Mutable and immutable types

Some objects in Python, such as dictionaries and lists, can be changed. We can change these objects after they have been made. This is called mutation. The types `dict` and `list` are called *mutable types*.

```
>>> x = []
>>> x.append(1)
>>> x
[1]
```

Some other objects, such as strings and tuples, cannot be changed. Once made they cannot be changed. They are called *immutable types*.

```
>>> y = 'abc'
>>> y[0] = 'A'
Traceback (most recent call last):
TypeError: 'str' object does not support item assignment
```

Enumerated integers and named tuples

We will use enumerated integers as an example in this section. In Python, booleans are an example of an enumerated integer type.

However, our task in this section is not to use booleans but to understand them. This will allow us to create our own subclasses of int and of immutable types.

The `bool` type

Here we review the `bool` type in Python.

Comparisons return a boolean, which is either True or False.

```
>>> 1 < 2, 1 == 2
(True, False)
```

True and False are instance of the `bool` type. >>> type(True), type(False) (<type 'bool'>, <type 'bool'>)

The `bool` type inherits from `int`.

```
>>> bool.__bases__
(<type 'int'>,)
```

Because True and False are (in the sense of inherit from) integers, we can do arithmetic on them.

```
>>> True + True
2
>>> False * 10
0
```

We can even use boolean expressions as numbers (although doing so might result in obscure code).

```
>>> a = 3; b = 4
>>> (a < b) * 10 + (a == b) * 20
10
```

Emulating `bool` - the easy part

In this subsection, as preparation for enumerated integers, we will start to code a subclass of `int` that behave like `bool`. We will start with string representation, which is fairly easy.

```
>>> class MyBool(int):
...     def __repr__(self):
...         return 'MyBool.' + ['False', 'True'][self]
```

This give us the correct string representations. >>> f = MyBool(0) >>> f MyBool.False

```
>>> t = MyBool(1)
>>> t
MyBool.True
```

But compare

```
>>> bool(2) == 1
True
```

with

```
>>> MyBool(2) == 1
False
```

In fact we have

```
>>> MyBool(2) == 2
True
>>> MyBool(2)
```

```
Traceback (most recent call last):
IndexError: list index out of range
```

Emulating `bool` - what goes wrong

In many classes we use __init__ to mutate the newly constructed object, typically by storing or otherwise using the arguments to __init__. But we can't do this with a subclass of `int` (or any other immuatable) because they are immutable.

You might try

```
>>> class InitBool(int):
...     def __init__(self, value):
...         self = bool(value)
```

but it won't work. Look at this - nothing has changed.

```
>>> x = InitBool(2)
>>> x == 2
True
```

This line of code

```
self = bool(value)
```

is deceptive. It does change the value bound to the *self* in __init__, but it does not change the object that was passed to __init__.

You might also try

```
>>> class InitBool2(int):
...     def __init__(self, value):
...         return bool(value)
```

but when called it raises an exception

```
>>> x = InitBool2(2)
Traceback (most recent call last):
TypeError: __init__() should return None, not 'bool'
```

Emulating `bool` - using __new__

The solution to the problem is to use __new__. Here we will show that it works, and later we will explain elsewhere exactly what happens. [where?].

```
>>> class NewBool(int):
...     def __new__(cls, value):
...         return int.__new__(cls, bool(value))
```

This works - no exception and 2 is converted into 1.

```
>>> y = NewBool(2)
>>> y == 1
True
```

We'll go carefully through this definition of __new__.

1. We define __new__, which like __init__ has a special role in object creation. But it's role is to do with creation of a new object, and not the initialisation of an already created object.

2. The function __new__ has *two* parameters. The first parameter is a class. The way we've called it, it will be the NewBool class.

3. The function __new__ returns a value.

4. The value returned is

```
int.__new__(cls, bool(value))
```

Understanding *int.__new__*

Here's the docstring for _int.__new__.

```
>>> print int.__new__.__doc__
T.__new__(S, ...) -> a new object with type S, a subtype of T
```

Let's try it, with S and T equal.

```
>>> z = int.__new__(int, 5)  # (*)
>>> z == 5
True
>>> type(z)
<type 'int'>
```

Thus, we see that line (*) is very much like or perhaps the same as *int(5)*. Let's try another example.

```
>>> int('10')
10
>>> int.__new__(int, '21')
21
```

The docstring above says that S must be a subtype of T. So let's create one.

```
>>> class SubInt(int): pass
```

And now let's use it as an argument to int.__new__.

```
>>> subint = int.__new__(SubInt, 11)
```

Now let's test the object we've just created. We expect it to be an instance of SubInt, and to be equal to 11.

```
>>> subint == 11
True
>>> type(subint)
<class 'SubInt'>
```

There we have it. Success. All that's required to complete the emulation of `bool` is to put all the pieces together.

Note: The key to subclassing immutable types is to use __new__ for both object creation and initialisation.

Exercise Create a class EmulBool that behaves like the `bool` builtin.

Exercise (Hard). Parameterize EmulBool. In other words, create an EnumInt such that

```
X = EnumInt(['False', 'True'])
```

creates a class X that behave like EmulBool.

CHAPTER 8

Subclassing `tuple`

Recall that with EmulBool in *Subclassing int* we had to define a __new__ method because we need to adjust the values passed to EmulBool before the instance was created.

The desired properties of Point

Since Python 2.6, `namedtuple` has been part of the `collections` module. We can use it to provide an example of what is required.

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ('x', 'y'))
```

Here are some facts about the Point class.

1. Point is a subclass of tuple.

   ```
   >>> Point.__bases__
   (<type 'tuple'>,)
   ```

2. Two arguments are used to initialise a point.

   ```
   >>> p = Point(1, 2)
   ```

3. A point has items at 0 and at 1.

   ```
   >>> p[0], p[1]
   (1, 2)
   ```

4. We can access these items using the names *x* and *y*.

   ```
   >>> p.x, p.y
   (1, 2)
   ```

Exercise Write an implementation of Point, that satisfies the above. (Please use the hints - they are there to help you.)

Hint To pass 1, 2 and 3 only three lines of code are required.

Hint To pass 4 use *property*, which replaces getting an attribute by a function call.

Hint The elegant way to pass 4 is to use `operator.itemgetter()`. Use this, and you'll need only another 3 lines of code in order to pass 4.

Answer

1. Point is a subclass of tuple.

```
>>> class Point(tuple):
...     def __new__(self, x, y):
...         return tuple.__new__(Point, (x, y))
```

```
>>> Point.__bases__
(<type 'tuple'>,)
```

2. Two arguments are used to initialise a point.

```
>>> p = Point(1, 2)
```

3. A point has items at 0 and at 1.

```
>>> p[0], p[1]
(1, 2)
```

4. We can access these items using the names *x* and *y*.

```
>>> import operator
```

```
>>> Point.x = property(operator.itemgetter(0))
>>> Point.y = property(operator.itemgetter(1))
```

```
>>> p.x, p.y
(1, 2)
```

CHAPTER 9

What happens when you call a class?

In this section we describe, in some detail, what happens when you call a class.

Creation and initialisation

Recall that every object has a type (sometimes known as a class).

```
>>> type(None), type(12), type(3.14), type([])
(<type 'NoneType'>, <type 'int'>, <type 'float'>, <type 'list'>)
```

The result of calling a class C is, ordinarily, an initialised object whose type is C. In Python this process is done by two functions

- __new__ returns an object that has the right type
- __init__ initialises the object created by __new__

To explain we will do the two steps one at a time. This will also clarify some details. But before we begin, we need a simple class.

```
>>> class A(object):
...     def __init__(self, arg):
...         self.arg = arg
```

We will explain what happens when Python executes the following.

```
a = A('an arg')
```

First, Python creates an object that has the right type. (The temporary *tmp* is introduced just to explain what happens. Python stores its value at a nameless location.)

```
>>> tmp = A.__new__(A, 'an arg')
>>> type(tmp)
<class 'A'>
```

But it has not been initialised.

```
>>> tmp.arg
Traceback (most recent call last):
AttributeError: 'A' object has no attribute 'arg'
```

Second, Python runs our initialisation code.

```
>>> tmp.__init__('an arg')
>>> tmp.arg
'an arg'
```

Finally, Python stores the value at *a*.

```
>>> a = tmp
```

The default __new__

We did not define a __new__ method for our class A, but all the same Python was able to call A.__new__. How is this possible?

For an instance of a class C, getting an attribute proceeds via the method resolution order of C. Something similar, but with important differences, happens when getting an attribute from C itself (rather than just an instance).

Here's proof that A.__new__ and object.__new__ are the same object. We show this in two different, but equivalant, ways.

```
>>> A.__new__ is object.__new__
True
>>> id(A.__new__) == id(object.__new__)
True
```

This explains how it is that Python can call A.__new__ even though we did not supply such a function ourselves.

For another example, we subclass *int*.

```
>>> class subint(int): pass
>>> subint.__new__ is int.__new__
True
```

Summary

Suppose C is a class. When you call, say

```
C(*argv, **kwargs)
```

the following happens.

1. C.__new__ is found.
2. The result of the following call is stored, say in tmp.

   ```
   C.__new__(C, *argv, **kwargs)
   ```

3. tmp.__init__ is found.
4. The result of the following is return as the value of the class call.

   ```
   self.__init__(*argv, **kwargs)
   ```

5. (Not discussed.) If tmp is not an instance of C (which includes subclasses of C) then steps 3 and 4 are omitted.

CHAPTER 10

Metaclass

Every object has a type

In Python, every object has a type, and the type of an object is an instance of type.

```
>>> type(0)
<type 'int'>
>>> isinstance(type(0), type)
True
>>> class A(object): pass
>>> type(A)
<type 'type'>
>>> a = A()
>>> type(a)
<class 'A'>
```

Even type has a type which is an instance of type (although it's a little silly).

```
>>> type(type)
<type 'type'>
>>> isinstance(type, type)
True
```

The metaclass of an object

The metaclass of an object is defined to be the type of its type.

```
>>> def metaclass(obj):
...     return type(type(obj))
```

```
>>> metaclass(0)
<type 'type'>
```

```
>>> metaclass(metaclass)
<type 'type'>
```

It's quite hard to create an object whose metaclass is not type.

A trivial non-type metaclass

In Python anything that is a type can be subclassed. So we can subclass type itself.

```
>>> class subtype(type): pass
```

We can now use subtype in pretty much the same way as type itself. In particular we can use it to construct an empty class.

```
>>> cls = subtype('name', (object,), {})
```

Let's look at the type and metaclass of cls.

```
>>> type(cls), metaclass(cls)
(<class 'subtype'>, <type 'type'>)
```

Notice that type(cls) is not type. This is our way in. Here's an instance of cls, followed by its type and metaclass.

```
>>> obj = cls()
>>> type(obj), metaclass(obj)
(<class 'name'>, <class 'subtype'>)
```

We have just constructed an object with a non-trivial metaclass. The metaclass of obj is subtype.

A non-trivial example

When Python executes

```
obj[key]
```

behind the scenes it executes

```
obj.__getitem__[key]
```

Here's an example:

```
>>> class A(object):
...     def __getitem__(self, key):
...         return getattr(self, key)
```

```
>>> obj = A()
>>> obj['name']
Traceback (most recent call last):
AttributeError: 'A' object has no attribute 'name'
```

```
>>> obj.name = 'some value'
>>> obj['name']
'some value'
```

What's the point?

There are two main reasons for introducing and using a metaclass, or in other words a subclass of type.

1. We wish to create classes whose behaviour requires special methods or other properties on the type of the class. This sounds and is odd, but can useful. In *JavaScript objects* we use it to create an elegant and simple implementation in Python of JavaScript object semantics.
2. We wish to make the class statement construct a class differently, somewhat as `bool()` construct a number differently from an `int()`. This is described in *The __metaclass__ attribute*, which is the next section.

CHAPTER 11

The __metaclass__ attribute

The __metaclass__ attribute was introduced to give the programmer some control over the semantics of the class statement. In particular it eases the transition from old-style classes (which are not covered in this tutorial) and new-style classes (simply called classes in this tutorial).

Automatic subclassing of object

If at the told of a module you write:

```
__metaclass__ = type
```

then class statements of the form:

```
class MyClass:
    pass
```

will automatically be new-style. In other words, you don't have to explicitly place object in the list of bases. (This behaviour is a consequence of the semantics of __metaclass__.)

Review of type(name, bases, body) and class statement

Recall that the type command, called like so

```
cls = type(name, bases, body)
```

constructs the class cls, as does the class statement

```
class cls(...):

    # body statements go here
```

The __metaclass__ attribute provides a link between these two ways of constructing classes.

The basic principle of the __metaclass__

Ordinarily, a class statement results in a call to type, with name, bases and body as arguments. However, this can be changed by

1. Assigning __metaclass__ as an class body attribute.
2. Assigning __metaclass__ as a module attribute.
3. Placing a suitable class in the bases of the class statement.

Method (1) is used above, in *Automatic subclassing of object*. To explain (2) we will introduce a very silly example.

A very silly example

It's not necessary for the __metaclass__ attribute to be type or a subclass of type. It could be any callable.

Here it is a function that returns a string.

```
>>> class very_silly(object):
...     def __metaclass__(*argv):
...         return 'This is very silly.'
```

The variable `silly` bound by the class statement is a string. In fact, it is the return value of the __metaclass__ attribute.

```
>>> very_silly
'This is very silly.'
```

A less silly example

Here's a less silly example. We define the __metaclass__ to return the argument vector passed to it. This consists of name, bases and body.

```
>>> class silly(object):
...     def __metaclass__(*argv):
...         return argv
```

The variable silly is now bound to the value of argv. So it is a tuple of length 3, and it can be unpacked into name, bases and body.

```
>>> type(silly), len(silly)
(<type 'tuple'>, 3)
>>> name, bases, body = silly
```

The name, and bases are much as we expect them.

```
>>> name == 'silly', bases ==(object,)
(True, True)
```

The body has, as could be expected, a __metaclass__ key, which has the expected value.

```
>>> sorted(body.keys())
['__metaclass__', '__module__']
>>> silly[2]['__metaclass__']
<function __metaclass__ at 0x...>
```

A __metaclass__ gotcha

A class statement, if it does not raise an exception, assigns a value to a variable. Ordinarily, this value is a direct instance of type, namely

```
type(name, bases, body)
```

However, using __metaclass__ above allows the value assigned by a class statement to be any object whatsover. In the very silly example the value assigned by the class statement was a string. This is a violates the principle of least surprise, and that is the main reason why the example is very silly (and not that it does nothing useful).

With decorators, which are available on class statements since Python 2.6, the same effect as the silly example can be obtained without resort to complex magic.

A decorator example

Here we produce something similar to the silly example. First we define a decorator

```
>>> from jfine.classtools import dict_from_class
>>> def type_argv_from_class(cls):
...     d = cls.__dict__
...     name = cls.__name__
...     body = dict_from_class(cls)
...     bases = cls.__bases__
...     return name, bases, body
```

Now we use the decorator. There is no magic. The class statement produces a class, and the decorator function `type_args_from_class()` produces an argument vector from the class.

```
>>> @type_argv_from_class
... class argv(object):
...      key = 'a value'
```

When we unpack `argv` we get what we expect.

```
>>> name, bases, body = argv
>>> name
'argv'
>>> bases
(<type 'object'>,)
>>> body
{'key': 'a value'}
```

CHAPTER 12

Decorators versus __metaclass__

Whenever a __metaclass__ is used, one could also use a decorator to get effectively the same result. This section discusses this topic.

For an example we use the concept of a Bunch, as discussed in Alex Martelli's excellent book *Python in a Nutshell*. As he says, a Bunch is similar to the struct type in C.

Bunch using decorators

Here we give a construction based on the decorator point of view. First we define a function, which can be used as a decorator, that returns a bunch class.

```
>>> def bunch_from_dict(a_dict, name='a_bunch'):
...
...     __slots__ = sorted(a_dict.keys())
...     defaults = dict(a_dict)
...     bases = (BaseBunch,)
...
...     def __init__(self, **kwargs):
...         for d in defaults, kwargs:
...             for key, value in d.items():
...                 setattr(self, key, value)
...
...     body = dict(__slots__=__slots__, __init__=__init__)
...     return type(name, bases, body)
```

We now need to implement the BaseBunch class, from which the return bunch classes will inherit __repr__ and, if we wish, other attributes.

```
>>> class BaseBunch(object):
...     def __repr__(self):
...         body = ', '.join([
...             '%s=%r' % (key, getattr(self, key))
...             for key in self.__slots__
...         ])
...         return '%s(%s)' % (self.__class__.__name__, body)
```

Here's an example of the creation of a Point class.

```
>>> Point = bunch_from_dict(dict(x=0, y=0), 'Point')
```

And here are examples of its use.

```
>>> Point(x=1, y=3)
Point(x=1, y=3)
>>> Point()
Point(x=0, y=0)
```

We can also use bunch_from_dict as a decorator.

```
>>> from jfine.classtools import dict_from_class
>>> @bunch_from_dict
... @dict_from_class
... class RGB(object):
...       'This is a docstring.'
...       red = green = blue = 0
```

We could, of course, introduce a new decorator `bunch_from_class()` to make life a little easier for the user.

Here's an example of the use of the RGB class. It shows that the name of the class is not being properly picked up. This is an interface problem rather than a problem with the decorator approach. The name is available to be used, but the interface is not making it available. Similar remarks apply to the docstring.

```
>>> RGB(blue=45, green=150)
a_bunch(blue=45, green=150, red=0)
```

Bunch using __metaclass__

The code here is based on the __metaclass__ implementation of Bunch, given in *Python in a Nutshell*. The API is:

```
class Point(MetaBunch):

    x = 0.0
    y = 0.0
```

The base class `MetaBunch()` is defined by:

```
class MetaBunch(object):

    __metaclass__ = metaMetaBunch
```

The real work is done in

```
class metaMetaBunch(type):

    def __new__(cls, name, bases, body):

        # Creation of new_body similar to bunch_from_dict.
        # ... but first need to 'clean up' the body.
        new_body = ... # Computed from body

        # Creation of new instance similar to bunch_from_dict.
        # ... but here can't use type(name, bases, new_body)
        return type.__new__(cls, name, bases, new_body)
```

where I've omitted the crucial code that computes the new_body from the old. (My focus here is on the logic of __metaclass_ and not the construction of the new body.)

How __metaclass__ works

In Python the class statement creates the class body from the code you have written, placing it in a dictionary. It also picks up the name and the bases in the first line of the class statement. These three arguments, (name, bases, body) are then passed to a function.

The __metaclass__ attribute is part of determining that function. If __metaclass__ is a key in the body dictionary then the value of that key is used. This value could be anything, although if not callable an exception will be raised.

In the example above, the MetaBunch class body has a key __metaclass__, and so its value metaMetaBunch is used. It is metaMetaBunch that is used to create the value that is stored at MetaBunch.

What is that value? When we instantiate metaMetaBunch we use its __new__ method to create the instance, which is an instance of type. In particular, the code that creates the new_body is run on the body of MetaBunch.

Now what happens when we subclass MetaBunch. One might think that

- because Point inherits from MetaBunch
- and because MetaBunch has a __metaclass__ in its body
- and that __metaclass__ has value metaMetaBunch

it follows that metaMetaBunch is use to construct the Point class.

But this is gotcha. Even though the conclusion is correct the reasoning is not. What happens is that

- Python looks for __metaclass__ in the body of Point
- but it's not there so it looks at the bases of Point
- and in the bases it finds MetaBunch
- whose type is metaMetaBunch

and so it uses that instead of type when constructing Point.

Discussion

Here are the main differences between the two approaches.

The decorator approach

- Syntax differs from ordinary class statement.
- Awkward if class decorators are not available.
- As is, the name is not picked up.
- Easier to construct Bunch classes dynamically.
- The Point class is an instance of type.

The __metaclass__ approach

- Syntax the same as ordinary class statement.
- 'Magic' takes place behind the scenes.
- Requires more knowledge to implement.
- Awkward to construct Bunch classes dynamically.
- The Point class is an instance of MetaBunch.

My view is that using decorators is simpler than using __metaclass__, particularly if the decorator syntax is available.

CHAPTER 13

JavaScript objects

Like Python classes

In JavaScript all objects are part of an inheritance tree. The **create** function adds a node to the inheritance tree.

```
// A JavaScript object.
js> root = {}

// Prototype inheritance.
js> create = function (obj) {
        var f = function () {return this;};
        f.prototype = obj;
        return new f;
    }

js> a = create(root)
js> b = create(a)

js> a.name = 5
js> a.name
5
js> b.name
5
```

In Python classes inherit in the same way.

```
>>> root = type              # Most classes are instance of type.
>>> class a(root): pass
>>> class b(a): pass         # Class inheritance.

>>> a.name = 5               # Just like JavaScript.
>>> a.name
5
>>> b.name
5
```

class explanation

In Python we can subclass anything whose type is **type** (or a subclass of type). A subclass (and its instances) inherits properties from the super-class.

```
>>> type(root) == type(a) == type(b) == type
True
```

Custom item methods

In JavaScript attribute and item access are the same.

```
js> a = create(root)

js> a.name = 5
js> a['name']
5

js> a['key'] = 6
js> a.key
6

js> a[1] = 6
js> a['1']
6
```

In Python we can defined our own item methods. (The programmer owns the dot.)

```
>>> class A(object):
...
...     def __getitem__(self, key):
...         return getattr(self, str(key))
...     def __setitem__(self, key, value):
...         return setattr(self, str(key), value)

>>> a = A()
>>> a.name = 5

>>> a['name']
5

>>> a['key'] = 6
>>> a.key
6

>>> a[1] = 6
>>> a['1']
6
```

Because **type(a)** is **A**, which has the special item methods, we get the special item behaviour.

```
>>> type(a) is A
True
```

On metaclass

Using previous definition, we cannot subclass **a** to create **b**.

```
>>> class b(a): pass
Traceback (most recent call last):
    class b(a): pass
TypeError: Error when calling the metaclass bases
    object.__new__() takes no parameters
```

This is because **a** is not a type. The solution involves Python metaclasses (an advanced topic).

```
>>> isinstance(a, type)
False
```

metaclass construction

We will subclass type, not object, and add to it the special item methods.

```
>>> class ObjectType(type):
...
...     def __getitem__(self, key):
...         return getattr(self, str(key))
...
...     def __setitem__(self, key, value):
...         return setattr(self, str(key), value)
```

Here is a fancy way of calling **ObjectType**.

```
>>> class root(object):
...     __metaclass__ = ObjectType
```

Here is a more direct (and equivalent) construction (create an instance of **ObjectType**, whose instances are objects).

```
>>> root = ObjectType('root', (object,), {})
>>> isinstance(root(), object)
True
```

metaclass demonstration

```
>>> class a(root): pass
>>> class b(a): pass

>>> a.name = 5
>>> a.name
5
>>> b.name
5
>>> a['name']
5
>>> b['name']
5

>>> a[1] = 6
>>> a['1']
6
```

metaclass explanation

Because **type(root)** is a subclass of type we can subclass root.

```
>>> issubclass(type(root), type)
True
```

Because the **type(root)** is **ObjectType**, which has special item methods, we get the special item behaviour.

```
>>> type(root) == type(a) == type(b) == ObjectType
True
```

Never instantiated

We can't call JavaScript objects (unless they are a function). But **create** creates ordinary JavaScript objects.

```
js> a = create(root)
js> a(1, 2, 3)
TypeError: a is not a function
```

We will monkey-patch the previous Python class, to provide custom behaviour when called.

```
>>> def raise_not_a_function(obj, *argv, **kwargs):
...     raise TypeError, obj.__name__ + ' is not a function'

>>> ObjectType.__call__ = raise_not_a_function

>>> a(1, 2, 3)
Traceback (most recent call last):
     a(1, 2, 3)
TypeError: a is not a function
```

Conclusion

JavaScript objects are like Python classes (because they inherit like Python classes).

For JavaScript attribute and item access are the same. This is achieved in Python by providing custom item methods.

In Python the custom item methods must be placed on the type of the object (or a superclass of its type).

Ordinary JavaScript objects are not functions and cannot be called. A Python class can be called (to create an instance of the object). But we can override this behaviour by supplying a custom method for call.

To summarize: ..

> **JavaScript objects are like Python classes with custom item methods (on the metaclass) that are never instantiated.**

It's worth saying again:

> **JavaScript objects are like Python classes with custom item methods (on the metaclass) that are never instantiated.**

CHAPTER 14

Exercise: A line from a file

We want to read lines from one of more files. We want each line to

- be a string
- have a filename attribute
- have a linenumber attribute

Recall that we can already iterate over the lines of a file.

The interface I suggest is

```
filename = 'myfile.txt'
f = open(filename)
labelled_lines = LabelledLines(f, filename)
```

The behavior we'd like is for this code

```
for line in labelled_lines:
    print (line.filename, line.linenumber, line)
```

to produce output like

```
('myfile.txt', 0, 'First line\n')
('myfile.txt', 1, 'Second line\n')
('myfile.txt', 2, 'Third line\n')
```

CHAPTER 15

Exercise: Property from class decorator

A property is a way of providing a class with virtual or protected attributes. They can be a good way of hiding and protecting implementation details. The area property of a rectangle is a good example of a read-only property.

Some properties are both read and write. There may also be a case for write-only properties. One can also delete a property. To help the programmer, a property can have a docstring.

The signature for property is

```
property(fget=None, fset=None, fdel=None, doc=None)
```

The exercise is to make a decorator that simplifies the creation of complex property attributes. The interface I suggest is

```
class MyClass(object):

    @property_from_class
    class my_property(object):
        '''This is to be the doc string for the property.'''

        def fget(self):
            pass            # code goes here

        def fset(self):
            pass            # code goes here

        def fdel(self):
            pass            # code goes here
```

Any or all of fget, fset, fdel can be omitted, as can the docstring. It should be an error to 'use the wrong keyword'.

CHAPTER 16

Exercise: Named integers

The bool type produces True and False, which are something like named integers. The exercise is to produce, from something like a mapping, a subclass of int that provides named integers.

Just as True and False look nicer that 0 or 1 when a quantity is a boolean, so named integers look nicer with symbolic constants.

The interface should be something like

```
my_items = [(0, 'zero'), (1, 'one'), (2, 'two')]
OneTwoThree = whatsit(my_items)
```

The behavior should be something like

```
z = OneTwoThree('zero')
str(z) == 'zero'

t = OneTwoThree(2)
str(t) == 'two'
```

Thus, any string or integer (within range) can be used to produce a named integer. Out of range values should produce an exception.

CHAPTER 17

Exercise: Subset of a set

The task here is to produce a memory efficient way of representing a subset of a given base set. We will use bytes, and for simplicity we assume that the base set has at most eight elements.

In Python 2.6 bytes is an alternative name for string, and in Python 3 it is a separate type in its own right (with a somewhat different interface). Please do the exercise in Python 2.6 (or earlier, with bytes equal to str).

The interface I suggest is something like

```
all_vowels = 'aeiou'
SubsetOfVowels = SubsetClassFactory(all_vowels)

my_vowels = SubsetOfVowels('ie')
set(my_vowels) == set(['i', 'e'])
ord(my_vowels[0]) == 2 + 4
```

Don't deal with set operations, such as intersection or complement.

By the way, an application would be dealing with large numbers of subsets of a largish set (set 255 elements), using numpy to store the data and do the hard work. So set operation details would have to fit in with numpy.

CHAPTER 18

Exercise: Class to and from class data

Sometimes __metaclass__ is used to amend the operation of a class statement. However always the same change can be done with a decorator function, and often this is clearer and easier.

The exercise here is to produce some class from class decorators. The first task is to produce two decorators whose composition is trivial.

In other words this

```
@class_from_class_data
@class_data_from_class
class MyClass(object):
      pass
```

should be equivalent to this

```
class MyClass(object):
      pass
```

Once we have done this, it's a lot easier to modify classes during construction, because so to speak the input-output has already been dealt with. Simply write a function that changes or creates a class data object.

The decorator function class_data_from_class should produce class_data, which we can regard as a tuple.

The decorator function class_from_class_data should produce a class from the class data.

Note: Don't assume that the type of MyClass is type. It could be a subclass of type.

CHAPTER 19

Exercise: Your own class to class decorator

This is an open exercise. The task is to find a situation where you need to change a class during its construction via a class statement, and then to write a class to class decorator that does this.

Please use the class_to_class_data and class_data_to_class decorators, either the ones supplied by the tutorial or your own (if you think they are better).

Here's a template, to get you started.

```
def my_class_decorator(cls):

    class_data = class_data_from_class(cls)
    new_class_data = whatever_you_want_it_to_be
    new_cls = class_from_class_data(new_class_data)
    return new_cls
```

Here are some ideas

- Check that the class supplies certain methods
- Perform other checks on the class
- Change methods so all calls are logged
- Supply extra utility methods to the class
- Refactor existing code that depends on __metaclass__
- search