Table of Contents

Python Attributes and Methods.....1
Shalabh Chaturvedi.....1
Before You Begin.....2
Chapter 1. New Attribute Access.....2
The Dynamic __dict__.....2
From Function to Method.....4
Creating Descriptors.....5
Two Kinds of Descriptors.....6
Attribute Search Summary.....8
Descriptors In The Box.....8
Chapter 2. Method Resolution Order.....10
The Problem (Method Resolution Disorder).....10
The "Who's Next" List.....11
A Super Solution.....12
Computing the MRO.....14
Chapter 3. Usage Notes.....18
Special Methods.....18
Subclassing Built-in Types.....19
Related Documentation.....21
Colophon.....22

Python Attributes and Methods

Shalabh Chaturvedi

Copyright © 2005-2009 Shalabh Chaturvedi

All Rights Reserved.

About This Book

Explains the mechanics of object attribute access for new-style Python objects:

- how functions become methods
- how descriptors and properties work
- determining method resolution order

New-style implies Python version 2.2 and upto and including 3.x. There have been some behavioral changes during these version but all the concepts covered here are valid.

This book is part of a series:

1. Python Types and Objects
2. Python Attributes and Methods [you are here]

This revision:
Discuss | Latest version | Cover page

Author: shalabh@cafepy.com

Table of Contents

Before You Begin
1. New Attribute Access
    The Dynamic `__dict__`
    From Function to Method
    Creating Descriptors
    Two Kinds of Descriptors
    Attribute Search Summary
    Descriptors In The Box
2. Method Resolution Order
    The Problem (Method Resolution Disorder)
    The "Who's Next" List
    A Super Solution
    Computing the MRO
3. Usage Notes
    Special Methods
    Subclassing Built-in Types
Related Documentation

List of Figures

2.1. The Diamond Diagram

2.2. A Simple Hierarchy
2.3. Beads on Strings - Unsolvable
2.4. Beads on Strings - Solved
2.5. Multiple Solutions

List of Examples

1.1. Simple attribute access
1.2. A function is more
1.3. A simple descriptor
1.4. Using a descriptor
1.5. Non-data descriptors
1.6. Hiding a method
1.7. Built-in descriptors
1.8. More on properties
2.1. Usual base call technique
2.2. Base call technique fails
2.3. Making a "Who's Next" list
2.4. Call next method technique
2.5. One super technique
2.6. Using super with a class method
2.7. Another super technique
3.1. Special methods work on type only
3.2. Forwarding special method to instance
3.3. Subclassing <type 'list'>
3.4. Using `__slots__` for optimization
3.5. Customizing creation of subclasses

Before You Begin

Some points you should note:

- This book covers the new-style objects (introduced a long time ago in Python 2.2). Examples are valid for Python 2.5 and all the way to Python 3.x.
- This book is not for absolute beginners. It is for people who already know Python (some Python at least), and want to know more.
- You should be familiar with the different kinds of objects in Python and not be confused when you come across the term *type* where you expected *class*. You can read the first part of this series for background information - Python Types and Objects.

Happy pythoneering!

Chapter 1. New Attribute Access

The Dynamic `__dict__`

What is an attribute? Quite simply, an attribute is a way to get from one object to another. Apply the power of the almighty dot - `objectname.attributename` - and voila! you now have the handle to another object. You also have the power to create attributes, by assignment: `objectname.attributename = notherobject`.

Which object does an attribute access return, though? And where does the object set as an attribute end up? These questions are answered in this chapter.

Example 1.1. Simple attribute access

```
>>> class C(object):
...     classattr = "attr on class"  ❶
...
>>> cobj = C()
>>> cobj.instattr = "attr on instance"  ❷
>>>
>>> cobj.instattr  ❸
'attr on instance'
>>> cobj.classattr ❹
'attr on class'
>>> C.__dict__['classattr'] ❺
'attr on class'
>>> cobj.__dict__['instattr'] ❻
'attr on instance'
>>>
>>> cobj.__dict__ ❼
{'instattr': 'attr on instance'}
>>> C.__dict__ ❽
{'classattr': 'attr on class', '__module__': '__main__', '__doc__': None}
```

❶ Attributes can be set on a class.

❷ Or even on an instance of a class.

❸ ❹ Both, class and instance attributes are accessible from an instance.

❺ ❻ Attributes really sit inside a dictionary-like `__dict__` in the object.

❼ ❽ `__dict__` contains only the *user-provided* attributes.

Ok, I admit 'user-provided attribute' is a term I made up, but I think it is useful to understand what is going on. Note that `__dict__` is itself an attribute. We didn't set this attribute ourselves, but Python provides it. Our old friends `__class__` and `__bases__` (none which appear to be in `__dict__` either) also seem to be similar. Let's call them *Python-provided* attributes. Whether an attribute is Python-provided or not depends on the object in question (`__bases__`, for example, is Python-provided only for classes).

We, however, are more interested in *user-defined* attributes. These are attributes provided by the user, and they usually (but not always) end up in the `__dict__` of the object on which they're set.

When accessed (for e.g. `print objectname.attributename`), the following objects are searched in sequence for the attribute:

1. The object itself (`objectname.__dict__` or any *Python-provided* attribute of `objectname`).
2. The object's type (`objectname.__class__.__dict__`). Observe that only `__dict__` is searched, which means only *user-provided* attributes of the class. In other words `objectname.__bases__` may not return anything even though `objectname.__class__.__bases__` does exist.
3. The bases of the object's class, their bases, and so on. (`__dict__` of each of `objectname.__class__.__bases__`). More than one base does not confuse Python, and should not concern us at the moment. The point to note is that all bases are searched

until an attribute is found.

If all this hunting around fails to find a suitably named attribute, Python raises an `AttributeError`. The type of the type (`objectname.__class__.__class__`) is never searched for attribute access on an object (`objectname` in the example).

The built-in `dir()` function returns a list of *all* attributes of an object. Also look at the inspect module in the standard library for more functions to inspect objects.

The above section explains the general mechanism for *all* objects. Even for classes (for example accessing `classname.attrname`), with a slight modification: the bases of the class are searched before the class of the class (which is `classname.__class__` and for most types, by the way, is `<type 'type'>`).

Some objects, such as built-in types and their instances (lists, tuples, etc.) do not have a `__dict__`. Consequently user-defined attributes cannot be set on them.

We're not done yet! This was the short version of the story. There is more to what can happen when setting and getting attributes. This is explored in the following sections.

From Function to Method

Continuing our Python experiments:

Example 1.2. A function is more

```
>>> class C(object):
...     classattr = "attr on class"
...     def f(self):
...             return "function f"
...
>>> C.__dict__ ❶
{'classattr': 'attr on class', '__module__': '__main__',
 '__doc__': None, 'f': <function f at 0x008F6B70>}
>>> cobj = C()
>>> cobj.classattr is C.__dict__['classattr'] ❷
True
>>> cobj.f is C.__dict__['f'] ❸
False
>>> cobj.f ❹
<bound method C.f of <__main__.C instance at 0x008F9850>>
>>> C.__dict__['f'].__get__(cobj, C) ❺
<bound method C.f of <__main__.C instance at 0x008F9850>>
```

❶ Two innocent looking class attributes, a string 'classattr' and a function 'f'.

❷ Accessing the string really gets it from the class's `__dict__`, as expected.

❸ Not so for the function! Why?

❹ Hmm, it does look like a different object. (A bound method is a callable object that calls a function (`C.f` in the example) passing an instance (`cobj` in the example) as the first argument in addition to passing through all arguments it was called with. This is what makes method calls on instance work.)

5. Here's the spoiler - this is what Python did to create the bound method. While looking for an attribute for an instance, if Python finds an object with a `__get__()` method inside the class's `__dict__`, instead of returning the object, it calls the `__get__()` method and returns the result. Note that the `__get__()` method is called with the instance and the class as the first and second arguments respectively.

It is only the presence of the `__get__()` method that transforms an ordinary function into a *bound method*. There is nothing really special about a function object. Anyone can put objects with a `__get__()` method inside the class `__dict__` and get away with it. Such objects are called *descriptors* and have many uses.

Creating Descriptors

Any object with a `__get__()` method, and optionally `__set__()` and `__delete__()` methods, accepting specific parameters is said to follow the *descriptor protocol*. Such an object qualifies as a descriptor and can be placed inside a class's `__dict__` to do something special when an attribute is retrieved, set or deleted. An empty descriptor is shown below.

Example 1.3. A simple descriptor

```
class Desc(object):
    "A descriptor example that just demonstrates the protocol"

    def __get__(self, obj, cls=None): ❶
        pass

    def __set__(self, obj, val): ❷
        pass

    def __delete__(self, obj): ❸
        pass
```

1. Called when attribute is read (eg. `print objectname.attrname`). Here `obj` is the object on which the attribute is accessed (may be `None` if the attribute is accessed directly on the class, eg. `print classname.attrname`). Also `cls` is the class of `obj` (or the class, if the access was on the class itself. In this case, `obj` is None).
2. Called when attribute is set on an instance (eg. `objectname.attrname = 12`). Here `obj` is the object on which the attribute is being set and `val` is the object provided as the value.
3. Called when attribute is deleted from an instance (eg. `del objectname.attrname`). Here `obj` is the object on which the attribute is being deleted.

What we defined above is a class that can be instantiated to create a descriptor. Let's see how we can create a descriptor, attach it to a class and put it to work.

Example 1.4. Using a descriptor

```
class C(object):
    "A class with a single descriptor"
    d = Desc() ❶

cobj = C()
```

```
x = cobj.d ❷
cobj.d = "setting a value" ❸
cobj.__dict__['d'] = "try to force a value" ❹
x = cobj.d ❺
del cobj.d ❻

x = C.d ❼
C.d = "setting a value on class" ❽
```

❶ Now the attribute called `d` is a descriptor. (This uses `Desc` from previous example.)

❷ Calls `d.__get__(cobj, C)`. The value returned is bound to `x`. Here `d` means the instance of `Desc` defined in ❶. It can be found in `C.__dict__['d']`.

❸ Calls `d.__set__(cobj, "setting a value")`.

❹ Sticking a value directly in the instance's `__dict__` works, but...

❺ is futile. This still calls `d.__get__(cobj, C)`.

❻ Calls `d.__delete__(cobj)`.

❼ Calls `d.__get__(None, C)`.

❽ Doesn't call anything. This replaces the descriptor with a new string object. After this, accessing `cobj.d` or `C.d` will just return the string `"setting a value on class"`. The descriptor has been kicked out of `C`'s `__dict__`.

Note that when accessed from the class itself, only the `__get__()` method comes in the picture, setting or deleting the attribute will actually replace or remove the descriptor.

Descriptors work only when attached to classes. Sticking a descriptor in an object that is not a class gives us nothing.

Two Kinds of Descriptors

In the previous section we used a descriptor with both `__get__()` and `__set__()` methods. Such descriptors, by the way, are called *data descriptors*. Descriptors with only the `__get__()` method are somewhat weaker than their cousins, and called *non-data descriptors*.

Repeating our experiment, but this time with non-data descriptors, we get:

Example 1.5. Non-data descriptors

```
class GetonlyDesc(object):
    "Another useless descriptor"

    def __get__(self, obj, typ=None):
        pass

class C(object):
    "A class with a single descriptor"
    d = GetonlyDesc()

cobj = C()

x = cobj.d ❶
cobj.d = "setting a value" ❷
x = cobj.d ❸
```

```
del cobj.d ❹

x = C.d ❺
C.d = "setting a value on class" ❻
```

❶ Calls `d.__get__(cobj, C)` (just like before).

❷ Puts `"setting a value"` in the instance itself (in `cobj.__dict__` to be precise).

❸ Surprise! This now returns `"setting a value"`, that is picked up from `cobj.__dict__`. Recall that for a data descriptor, the instance's `__dict__` is bypassed.

❹ Deletes the attribute `d` from the instance (from `cobj.__dict__` to be precise).

❺ ❻ These function identical to a data descriptor.

Interestingly, not having a `__set__()` affects not just attribute setting, but also retrieval. What is Python thinking? If on setting, the descriptor gets fired and puts the data somewhere, then it follows that the descriptor only knows how to get it back. Why even bother with the instance's `__dict__`?

Data descriptors are useful for providing *full control* over an attribute. This is what one usually wants for attributes used to store some piece of data. For example an attribute that gets *transformed* and saved somewhere on setting, would usually be *reverse-transformed* and returned when read. When you have a data descriptor, it controls all access (both read and write) to the attribute on an instance. Of course, you could still directly go to the *class* and replace the descriptor, but you can't do that from an instance of the class.

Non-data descriptors, in contrast, only provide a value when an instance itself does not have a value. So setting the attribute on an instance *hides* the descriptor. This is particularly useful in the case of functions (which are non-data descriptors) as it allows one to hide a function defined in the class by attaching one to an instance.

Example 1.6. Hiding a method

```
class C(object):
    def f(self):
        return "f defined in class"

cobj = C()

cobj.f() ❶

def another_f():
    return "another f"

cobj.f = another_f

cobj.f() ❷
```

❶ Calls the bound method returned by `f.__get__(cobj, C)`. Essentially ends up calling `C.__dict__['f'](cobj)`.

❷ Calls `another_f()`. The function `f()` defined in `C` has been hidden.

Attribute Search Summary

This is the long version of the attribute access story, included just for the sake of completeness.

When retrieving an attribute from an object (`print objectname.attrname`) Python follows these steps:

1. If `attrname` is a special (i.e. Python-provided) attribute for `objectname`, return it.
2. Check `objectname.__class__.__dict__` for `attrname`. If it exists and is a data-descriptor, return the descriptor result. Search all bases of `objectname.__class__` for the same case.
3. Check `objectname.__dict__` for `attrname`, and return if found. If `objectname` is a class, search its bases too. If it is a class and a descriptor exists in it or its bases, return the descriptor result.
4. Check `objectname.__class__.__dict__` for `attrname`. If it exists and is a non-data descriptor, return the descriptor result. If it exists, and is not a descriptor, just return it. If it exists and is a data descriptor, we shouldn't be here because we would have returned at point 2. Search all bases of `objectname.__class__` for same case.
5. Raise `AttributeError`

Note that Python first checks for a *data* descriptor in the class (and its bases), then for the attribute in the object `__dict__`, and then for a *non-data* descriptor in the class (and its bases). These are points 2, 3 and 4 above.

The *descriptor result* above implies the result of calling the `__get__()` method of the descriptor with appropriate arguments. Also, checking a `__dict__` for `attrname` means checking if `__dict__["attrname"]` exists.

Now, the steps Python follows when *setting* a user-defined attribute (`objectname.attrname = something`):

1. Check `objectname.__class__.__dict__` for `attrname`. If it exists *and is a data-descriptor*, use the descriptor to set the value. Search all bases of `objectname.__class__` for the same case.
2. Insert `something` into `objectname.__dict__` for key `"attrname"`.
3. Think "Wow, this was much simpler!"

What happens when setting a Python-provided attribute depends on the attribute. Python may not even allow some attributes to be set. Deletion of attributes is very similar to setting as above.

Descriptors In The Box

Before you rush to the mall and get yourself some expensive descriptors, note that Python ships with some very useful ones that can be found by simply looking in the box.

Example 1.7. Built-in descriptors

```
class HidesA(object):

    def get_a(self):
        return self.b - 1
```

```
    def set_a(self, val):
        self.b = val + 1

    def del_a(self):
        del self.b

    a = property(get_a, set_a, del_a, "docstring") ❶


    def cls_method(cls):
        return "You called class %s" % cls

    clsMethod = classmethod(cls_method) ❷


    def stc_method():
        return "Unbindable!"

    stcMethod = staticmethod(stc_method) ❸
```

❶ A *property* provides an easy way to call functions whenever an attribute is retrieved, set or deleted on the instance. When the attribute is retrieved from the class, the getter method is not called but the property object itself is returned. A docstring can also be provided which is accessible as `HidesA.a.__doc__`.

❷ A *classmethod* is similar to a regular method, except that is passes the class (and not the instance) as the first argument to the function. The remaining arguments are passed through as usual. It can also be called directly on the class and it behaves the same way. The first argument is named `cls` instead of the traditional `self` to avoid confusion regarding what it refers to.

❸ A *staticmethod* is just like a function outside the class. It is never *bound*, which means no matter how you access it (on the class or on an instance), it gets called with exactly the same arguments you pass. No object is inserted as the first argument.

As we saw earlier, Python functions are descriptors too. They weren't descriptors in earlier versions of Python (as there were no descriptors at all), but now they fit nicely into a more generic mechanism.

A property is always a data-descriptor, but not all arguments are required when defining it.

Example 1.8. More on properties

```
class VariousProperties(object):

    def get_p(self):
        pass

    def set_p(self, val):
        pass

    def del_p(self):
        pass

    allOk = property(get_p, set_p, del_p)  ❶

    unDeletable = property(get_p, set_p) ❷
```

```
    readOnly = property(get_p) ❸
```

❶ Can be set, retrieved, or deleted.

❷ Attempting to delete this attribute from an instance will raise `AttributeError`.

❸ Attempting to set or delete this attribute from an instance will raise `AttributeError`.

The getter and setter functions need not be defined in the class itself, any function can be used. In any case, the functions will be called with the instance as the first argument. Note that *where* the functions are passed to the property constructor above, they are not bound functions anyway.

Another useful observation would be to note that subclassing the class and redefining the getter (or setter) functions is not going to change the property. The property object is *holding on* to the actual functions provided. When kicked, it is going to say "Hey, I'm holding this function I was given, I'll just call this and return the result.", and not "Hmm, let me look up the *current* class for a method called 'get_a' and then use that". If that is what one wants, then defining a new descriptor would be useful. How would it work? Let's say it is initialized with a string (i.e. the name of the method to call). On activation, it does a `getattr()` for the method name on the class, and use the method found. Simple!

Classmethods and staticmethods are non-data descriptors, and so can be *hidden* if an attribute with the same name is set directly on the instance. If you are rolling your own descriptor (and not using properties), it can be made read-only by giving it a `__set__()` method but raising `AttributeError` in the method. This is how a property behaves when it does not have a setter function.

Chapter 2. Method Resolution Order

The Problem (Method Resolution Disorder)

Why do we need Method Resolution Order? Let's say:

1. We're happily doing OO programming and building a class hierarchy.
2. Our usual technique to implement the `do_your_stuff()` method is to first call `do_your_stuff()` on the base class, and then do our stuff.

Example 2.1. Usual base call technique

```
class A(object):
    def do_your_stuff(self):
        # do stuff with self for A
        return

class B(A):
    def do_your_stuff(self):
        A.do_your_stuff(self)
        # do stuff with self for B
        return

class C(A):
    def do_your_stuff(self):
```

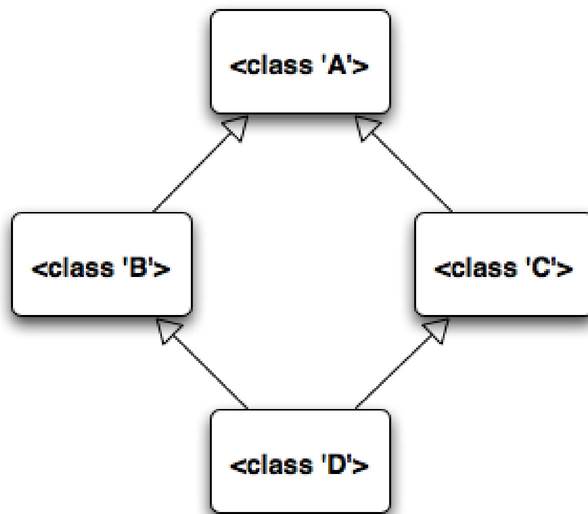
```
        A.do_your_stuff(self)
        # do stuff with self for C
        return
```

3. We subclass a new class from two classes and end up having the same superclass being accessible through two paths.

Example 2.2. Base call technique fails

```
class D(B,C):
    def do_your_stuff(self):
        B.do_your_stuff(self)
        C.do_your_stuff(self)
        # do stuff with self for D
        return
```

Figure 2.1. The Diamond Diagram



4. Now we're stuck if we want to implement `do_your_stuff()`. Using our usual technique, if we want to call both `B` and `C`, we end up calling `A.do_your_stuff()` twice. And we all know it might be dangerous to have `A` do its stuff twice, when it is only supposed to be done once. The other option would leave either `B`'s stuff or `C`'s stuff not done, which is not what we want either.

There are messy solutions to this problem, and clean ones. Python, obviously, implements a clean one which is explained in the next section.

The "Who's Next" List

Let's say:

1. For each class, we arrange *all* superclasses into an ordered list without repetitions, and insert the class itself at the start of the list. We put this list in an class attribute called `next_class_list` for our use later.

Example 2.3. Making a "Who's Next" list

```
B.next_class_list = [B,A]

C.next_class_list = [C,A]

D.next_class_list = [D,B,C,A]
```

2. We use a different technique to implement `do_your_stuff()` for our classes.

Example 2.4. Call next method technique

```
class B(A):
    def do_your_stuff(self):
        next_class = self.find_out_whos_next()
        next_class.do_your_stuff(self)
        # do stuff with self for B

    def find_out_whos_next(self):
        l = self.next_class_list           # l depends on the actual instance
        mypos = l.index(B)  ❶              # Find this class in the list
        return l[mypos+1]                  # Return the next one
```

The interesting part is how we `find_out_whos_next()`, which depends on which instance we are working with. Note that:

- Depending on whether we passed an instance of `D` or of `B`, `next_class` above will resolve to either `C` or `A`.
- We have to implement `find_out_whos_next()` for each class, since it has to have the class name hardcoded in it (see ❶ above). We cannot use `self.__class__` here. If we have called `do_your_stuff()` on an instance of `D`, and the call is traversing up the hierarchy, then `self.__class__` will be `D` here.

Using this technique, each method is called only once. It appears clean, but seems to require too much work. Fortunately for us, we neither have to implement `find_out_whos_next()` for each class, nor set the `next_class_list`, as Python does both of these things.

A Super Solution

Python provides a class attribute `__mro__` for each class, and a type called `super`. The `__mro__` attribute is a tuple containing the class itself and all of its superclasses without duplicates in a predictable order. A `super` object is used in place of the `find_out_whos_next()` method.

Example 2.5. One super technique

```
class B(A): ❶
    def do_your_stuff(self):
        super(B, self).do_your_stuff() ❷
        # do stuff with self for B
```

❷ The `super()` call creates a *super* object. It finds the next class after `B` in `self.__class__.__mro__`. Attributes accessed on the *super* object are searched on the next class and returned. Descriptors are resolved. What this means is accessing a method (as above) returns a *bound* method (note the `do_your_stuff()` call does not pass self). When using `super()` the first parameter should always be the same as the class in which it is being used (❶).

If we're using a class method, we don't have an instance `self` to pass into the `super` call. Fortunately for us, `super` works even with a class as the second argument. Observe that above, `super` uses `self` only to get at `self.__class__.__mro__`. The class can be passed directly to `super` as shown below.

Example 2.6. Using super with a class method

```
class A(object):
    @classmethod ❶
    def say_hello(cls):
        print 'A says hello'

class B(A):
    @classmethod
    def say_hello(cls):
        super(B, cls).say_hello() ❷
        print 'B says hello'

class C(A):
    @classmethod
    def say_hello(cls):
        super(C, cls).say_hello()
        print 'C says hello'

class D(B, C):
    @classmethod
    def say_hello(cls):
        super(D, cls).say_hello()
        print 'D says hello'

B.say_hello() ❸
D.say_hello() ❹
```

❶ This example is for classmethods (not instance methods).

❷ Note we pass `cls` (the class and not the instance) to super().

❸ This prints out:

A says hello
B says hello

❹ This prints out (observe each method is called only once):

A says hello
C says hello
B says hello
D says hello

There is yet another way to use `super`:

Example 2.7. Another super technique

```
class B(A):

    def do_your_stuff(self):
        self.__super.do_your_stuff()
        # do stuff with self for B

B._B__super = super(B) ❶
```

When created with only a type, the `super` instance behaves like a descriptor. This means (if `d` is an instance of `D`) that `super(B).__get__(d)` returns the same thing as `super(B,d)`. In ❶ above, we munge an attribute name, similar to what Python does for names starting with double underscore *inside* the class. So this is accessible as `self.__super` within the body of the class. If we didn't use a class specific attribute name, accessing the attribute through the instance `self` might return an object defined in a subclass.

While using `super` we typically use only one `super` call in one method even if the class has multiple bases. Also, it is a good programming practice to use `super` instead of calling methods directly on a base class.

A possible pitfall appears if `do_your_stuff()` accepts different arguments for `C` and `A`. This is because, if we use `super` in `B` to call `do_your_stuff()` on the *next* class, we don't know if it is going to be called on `A` or `C`. If this scenario is unavoidable, a case specific solution might be required.

Computing the MRO

One question as of yet unanswered is how does Python determine the `__mro__` for a type? A basic idea behind the algorithm is provided in this section. This is not essential for just using `super`, or reading following sections, so you can jump to the next section if you want.

Python determines the *precedence* of types (or the order in which they should be placed in any `__mro__`) from two kinds of constraints specified by the user:

1. If `A` is a superclass of `B`, then `B` has precedence over `A`. Or, `B` should always appear *before* `A` in all `__mro__`s (that contain both). In short let's denote this as `B > A`.
2. If `C` appears before `D` in the list of bases in a class statement (eg. `class Z(C,D):`), then `C > D`.

In addition, to avoid being ambiguous, Python adheres to the following principle:

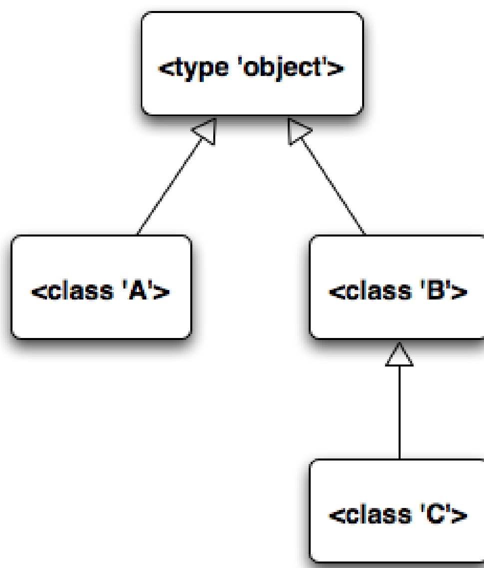
3. If `E > F` in one scenario (or one `__mro__`), then it should be that `E > F` in all scenarios (or all `__mro__`s).

We can satisfy the constraints if we build the `__mro__` for each new class `C` we introduce, such that:

1. *All* superclasses of `C` appear in the `C.__mro__` (plus `C` itself, at the start), and
2. The *precedence* of types in `C.__mro__` does not conflict with the precedence of types in `B.__mro__` for each `B` in `C.__bases__`.

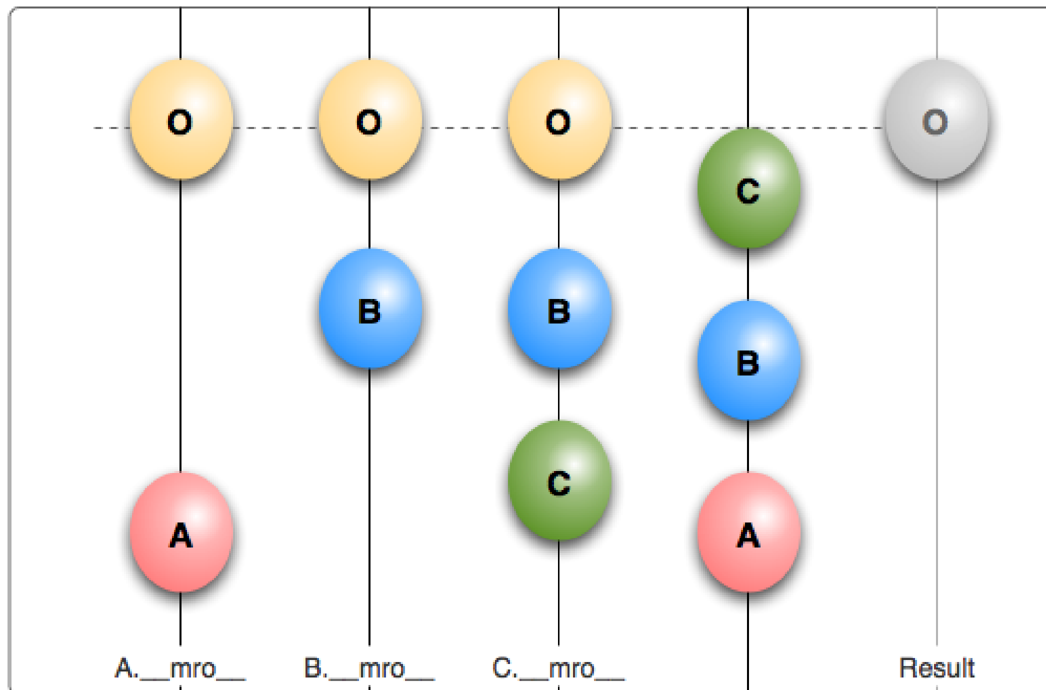
Here the same problem is translated into a game. Consider a class hierarchy as follows:

Figure 2.2. A Simple Hierarchy



Since only single inheritance is in play, it is easy to find the `__mro__` of these classes. Let's say we define a new class as `class N(A,B,C)`. To compute the `__mro__`, consider a game using abacus style beads over a series of strings.

Figure 2.3. Beads on Strings - Unsolvable

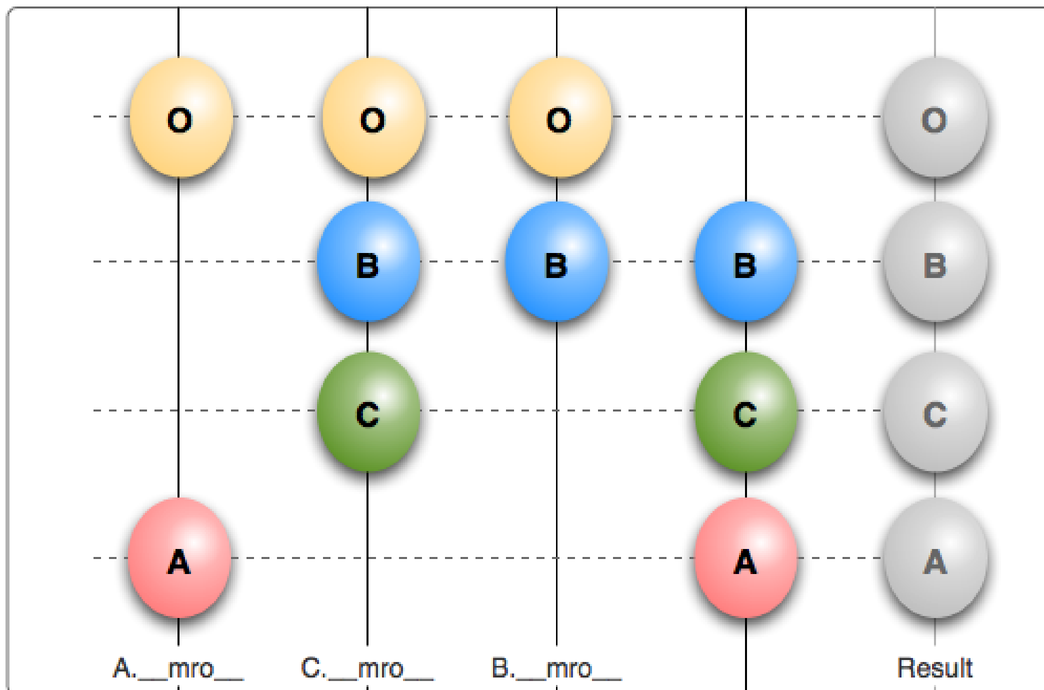


Beads can move freely over the strings, but the strings cannot be cut or twisted. The strings from left to right contain beads in the order of `__mro__` of each of the bases. The rightmost string contains one bead for each base, in the order the bases are specified in the class statement.

The objective is to line up beads in rows, so that each row contains beads with only one label (as done with the `O` bead in the diagram). Each string represents an ordering constraint, and if we can reach the goal, we would have an order that satisfies all constraints. We could then just read the labels off rows from the bottom up to get the `__mro__` for `N`.

Unfortunately, we cannot solve this problem. The last two strings have `C` and `B` in different orders. However, if we change our class definition to `class N(A,C,B)`, then we have some hope.

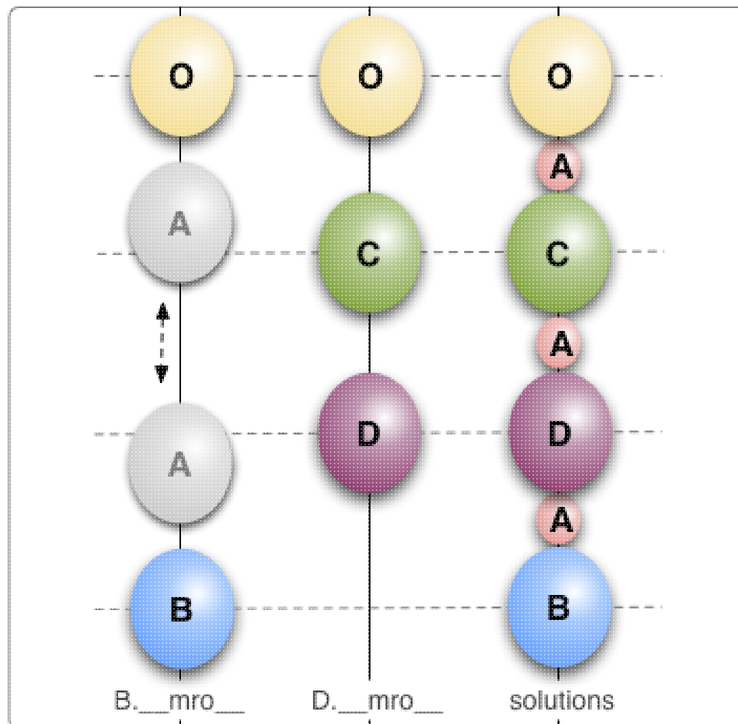
Figure 2.4. Beads on Strings - Solved



We just found out that `N.__mro__` is `(N,A,C,B,object)` (note we inserted `N` at the head). The reader can try out this experiment in real Python (for the unsolvable case above, Python raises an exception). Observe that we even swapped the position of two strings, keeping the strings in the same order as the bases are specified in the class statement. The usefulness of this is seen later.

Sometimes, there might be more than one solution, as shown in the figure below. Consider four classes `class A(object)`, `class B(A)`, `class C(object)` and `class D(C)`. If a new class is defined as `class E(B, D)`, there are multiple possible solutions that satisfy all constraints.

Figure 2.5. Multiple Solutions



Possible positions for `A` are shown as the little beads. The order can be kept unambiguous (more correctly, *monotonic*) if the following policies are followed:

1. Arrange strings from left to right in order of appearance of bases in the class statement.
2. Attempt to arrange beads in rows moving from bottom up, and left to right. What this means is that the MRO of `class E(B, D)` will be set to: `(E,B,A,D,C,object)`. This is because `A`, being left of `C`, will be selected first as a candidate for the second row from bottom.

This, essentially, is the idea behind the algorithm used by Python to generate the `__mro__` for any new type. The formal algorithm is formally explained elsewhere [mro-algorithm].

Chapter 3. Usage Notes

This chapter includes usage notes that do not fit in other chapters.

Special Methods

In Python, we can use methods with special name like `__len__()`, `__str__()` and `__add__()` to make objects convenient to use (for example, with the built-in functions `len()`, `str()` or with the '`+`' operator, etc.)

Example 3.1. Special methods work on type only

```
class C(object):
    def __len__(self):   ❶
        return 0

cobj = C()

def mylen():
```

```
    return 1

cobj.__len__ = mylen ❷

print len(cobj) ❸
```

❶ Usually we put the *special* methods in a class.
❷ We can try to put them in the instance itself, but it doesn't work.
❸ This goes straight to the class (calls `C.__len__()`), not to the instance.

The same is true for all such methods, putting them on the instance we want to use them with does not work. If it did go to the instance then even something like `str(C)` (`str` of the class `C`) would go to `C.__str__()`, which is a method defined for an *instance* of `C`, and not `C` itself.

A simple technique to allow defining such methods for each instance separately is shown below.

Example 3.2. Forwarding special method to instance

```
class C(object):
    def __len__(self):
        return self._mylen() ❶

    def _mylen(self): ❷
        return 0

cobj = C()

def mylen():
    return 1

cobj._mylen = mylen ❸

print len(cobj) ❹
```

❶ We call another method on the instance,
❷ for which we provide a default implementation in the class.
❸ But it can be overwritten (rather *hidden*) by setting on the instance.
❹ This now calls `mylen()`.

Subclassing Built-in Types

Subclassing built-in types is straightforward. Actually we have been doing it all along (whenever we subclass `<type 'object'>`). Some built-in types (`types.FunctionType`, for example) are not subclassable (not yet, at least). However, here we talk about subclassing `<type 'list'>`, `<type 'tuple'>` and other basic data types.

Example 3.3. Subclassing <type 'list'>

```
>>> class MyList(list): ❶
...     "A list that converts appended items to ints"
...     def append(self, item): ❷
...         list.append(self, int(item)) ❸
...
>>>
>>> l = MyList()
>>> l.append(1.3) ❹
>>> l.append(444)
>>> l
[1, 444] ❺
>>> len(l) ❻
2
>>> l[1] = 1.2 ❼
>>> l
[1, 1.2]
>>> l.color = 'red' ❽
```

❶ A regular class statement.

❷ Define the method to be overridden. In this case we will convert all items passed through `append()` to integers.

❸ Upcall to the base if required. `list.append()` works like an unbound method, and is passed the instance as the first argument.

❹ Append a float and...

❺ watch it automatically become an integer.

❻ Otherwise, it behaves like any other list.

❼ This doesn't go through append. We would have to define `__setitem__()` in our class to massage such data. The upcall would be to `list.__setitem__(self,item)`. Note that the *special* methods such as `__setitem__` exist on built-in types.

❼ We can set attributes on our instance. This is because it has a `__dict__`.

Basic lists do not have `__dict__` (and so no user-defined attributes), but ours does. This is usually not a problem and may even be what we want. If we use a *very* large number of `MyList`s, however, we could optimize our program by telling Python not to create the `__dict__` for instances of `MyList`.

Example 3.4. Using `__slots__` for optimization

```
class MyList(list):
    "A list subclass disallowing any user-defined attributes"

    __slots__ = [] ❶

ml = MyList()
ml.color = 'red' # raises exception! ❷

class MyListWithFewAttrs(list):
    "A list subclass allowing specific user-defined attributes"

    __slots__ = ['color'] ❸

mla = MyListWithFewAttrs()
mla.color = 'red' ❹
mla.weight = 50 # raises exception! ❺
```

❶ The `__slots__` class attribute tells Python to not create a `__dict__` for instances of this type.
❷ Setting any attribute on this raises an exception.
❸ `__slots__` can contain a list of strings. The instances still don't get a real dictionary for `__dict__`, but they get a *proxy*. Python reserves space in the instance for the specified attributes.
❹ Now, if an attribute has space reserved, it can be used.
❺ Otherwise, it cannot. This will raise an exception.

The purpose and recommended use of `__slots__` is for optimization. After a type is defined, its slots cannot be changed. Also, every subclass must define `__slots__`, otherwise its instances will end up having `__dict__`.

We can create a list even by instantiating it like any other type: `list([1,2,3])`. This means `list.__init__()` accepts the same argument (i.e. any iterable) and initializes a list. We can customize initialization in a subclass by redefining `__init__()` and *upcalling* `__init__()` on the base.

Tuples are immutable and different from lists. Once an instance is created, it cannot be changed. Note that the instance of a type already exists when `__init__()` is called (in fact the instance is passed as the first argument). The `__new__()` static method of a type is called to *create* an instance of the type. It is passed the type itself as the first argument, and passed through other initial arguments (similar to `__init__()`). We use this to customize immutable types like a tuple.

Example 3.5. Customizing creation of subclasses

```
class MyList(list):

    def __init__(self, itr): ❶
        list.__init__(self, [int(x) for x in itr])

class MyTuple(tuple):

    def __new__(typ, itr): ❷
        seq = [int(x) for x in itr]
        return tuple.__new__(typ, seq) ❸
```

❶ For a list, we massage the arguments and hand them over to `list.__init__()`.
❷ For a tuple, we have to override `__new__()`.
❸ A `__new__()` should *always* return. It is supposed to return an instance of the type.

The `__new__()` method is not special to immutable types, it is used for all types. It is also converted to a static method automatically by Python (by virtue of its name).

Related Documentation

[descrintro] Unifying types and classes in Python 2.2. Guido van Rossum.

[pep-252] Making Types Look More Like Classes. Guido van Rossum.

[pep-253] Subclassing Built-in Types. Guido van Rossum.

[descriptors-howto] How-To Guide for Descriptors. Raymond Hettinger.

[mro-algorithm] The Python 2.3 Method Resolution Order. Michele Simionato.

Colophon

This book was written in DocBook XML. The HTML version was produced using DocBook XSL stylesheets and `xsltproc`. The PDF version was produced using `htmldoc`. The diagrams were drawn using OmniGraffe [1]. The process was automated using Paver [2].

[1] http://www.omnigroup.com/
[2] http://www.blueskyonmars.com/projects/paver/

Feedback

Comment on this book here: discussion page. I appreciate feedback!