Dive into Python Class

Knowing python class step-by-step

Created by **Jim Yeh** / **@jimyeh00**

About Me

- A front-to-end web developer
- Use Python since 2006
- Enjoy writing python code

Outline

- Introduce New-style class
- descriptor
- function
- super

Basic knowing

- Knowledge of OO

Classic class and new-style class

- Different syntax
- type of object
- Inheritance

Syntax

```
>>> class OldObj:
...     pass
>>> type(OldObj)
<type 'classobj'>
```

```
>>> class NewObj(object):
...     pass
>>> type(NewObj)
<type 'type'>
```

type of object

```
>>> old_instance = OldObj()
>>> type(old_instance)
<type 'instance'>
```

```
>>> new_instance = NewObj()
>>> type(new_instance)
<class '__main__.NewObj'>
```

Inheritance

- For classic classes, the search is depth-first, left-to-right in the order of occurrence in the base class list
- For new-style classes, search in an mro order

What's New?

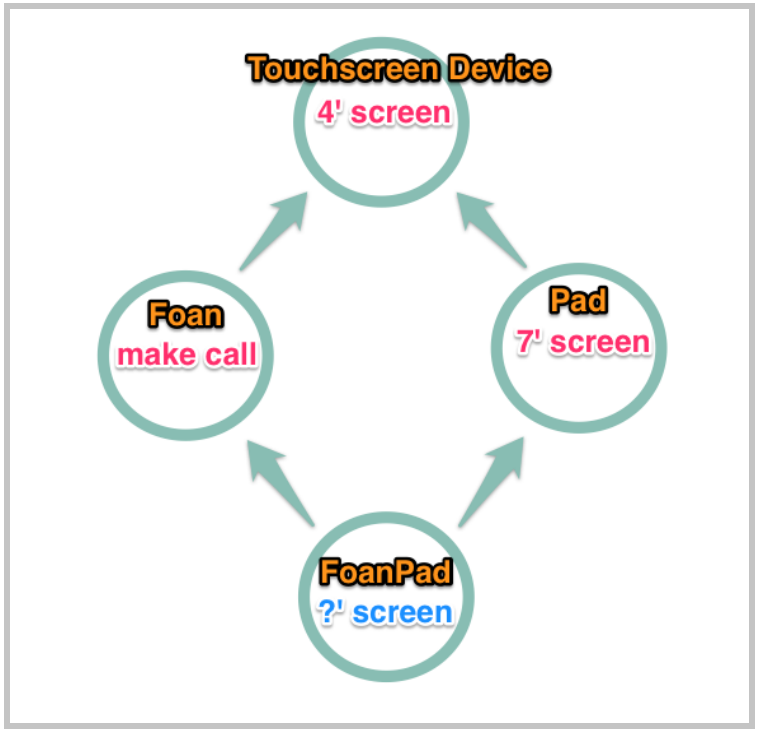
1. MRO
2. property
3. classmethod / staticmethod
4. descriptor (not a **decorator**)
5. super
6. __new__ and __metaclass__

MRO

Method Resolution Order

It is the **order** that a new-style class
uses to search for methods and attributes.

Diamond Problem




```
class TouchScreenDevice:
    screen_size = 4

class Foan(TouchScreenDevice):
    def make_call(self, number):
        print "Call " + number


class Pad(TouchScreenDevice):
    screen_size = 7

class FoanPad(Foan, Pad):
    pass
```

In classic inheritance, the search order is

FoanPad -> Foan -> TouchScreenDevice -> Pad

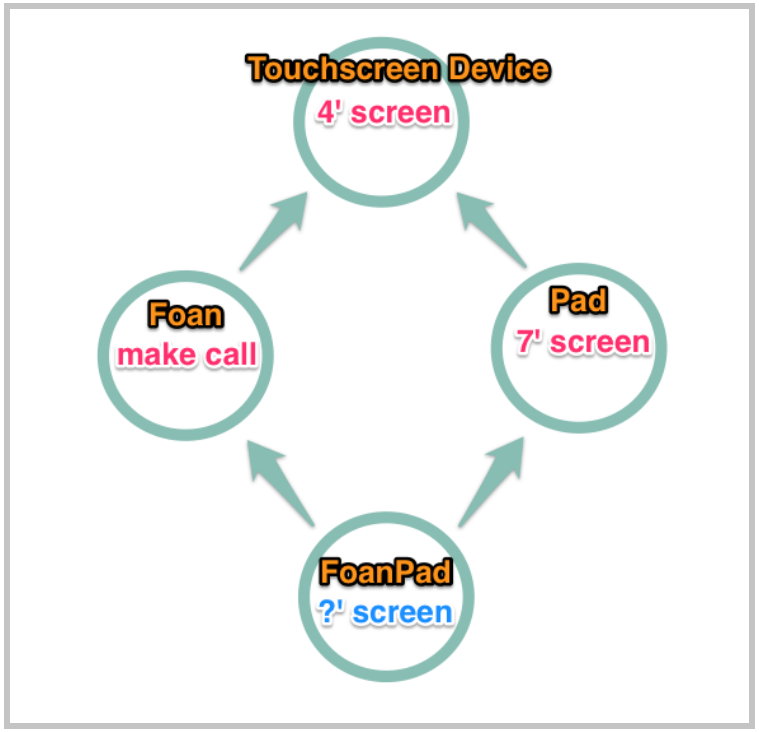
That is to say, FoanPad.screen_size = 4

C3 linearization

The implementation of MRO in python

- The right class is next to the left class.
- The parent class is next to the child class

Example



1. FoanPad -> Foan -> Pad
2. Foan -> TouchScreen
3. Pad -> TouchScreen
4. FoanPad -> Foan -> Pad -> **TouchScreen**

```
>>> FoanPad.mro()
[<class '__main__.FoanPad'>, <class '__main__.Foan'>, <class '__main__.
Pad'>, <class '__main__.TouchScreenDevice'>, <type '
object'>]
```

property

A implementation of get / set function in OO

Example

```
class Student(object):

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def get_name(self):
        return self.first_name + " " + self.last_name

    def set_name(self, first_name):
        self.first_name = first_name

    name = property(get_name, set_name)
```

```
>>> me = Student("Jim", "Yeh")
>>> me.name
'Jim Yeh'
>>> me.name = Joe
>>> me.name
'Joe Yeh'
```

classmethod

A implementation of the overloading-like feature in C++

Example

```
class Host(object):
    def __init__(self, name, os):
        self.name = name
        self.os = os

    def _from_linux(cls, name):
        return cls(name, "linux")

    from_linux = classmethod(_from_linux)
```

```
>>> h = Host.from_linux("My Server")
>>> h.os
```

staticmethod

An isolated function

Example

```
class Host(object):
    def __init__(self, name, os):
        self._name = name
        self._os = os

    def _version():
        return "1.0.0"

    version = staticmethod(version)
```

```
>>> h = Host("My Host", "Linux")
>>> h.version()
```

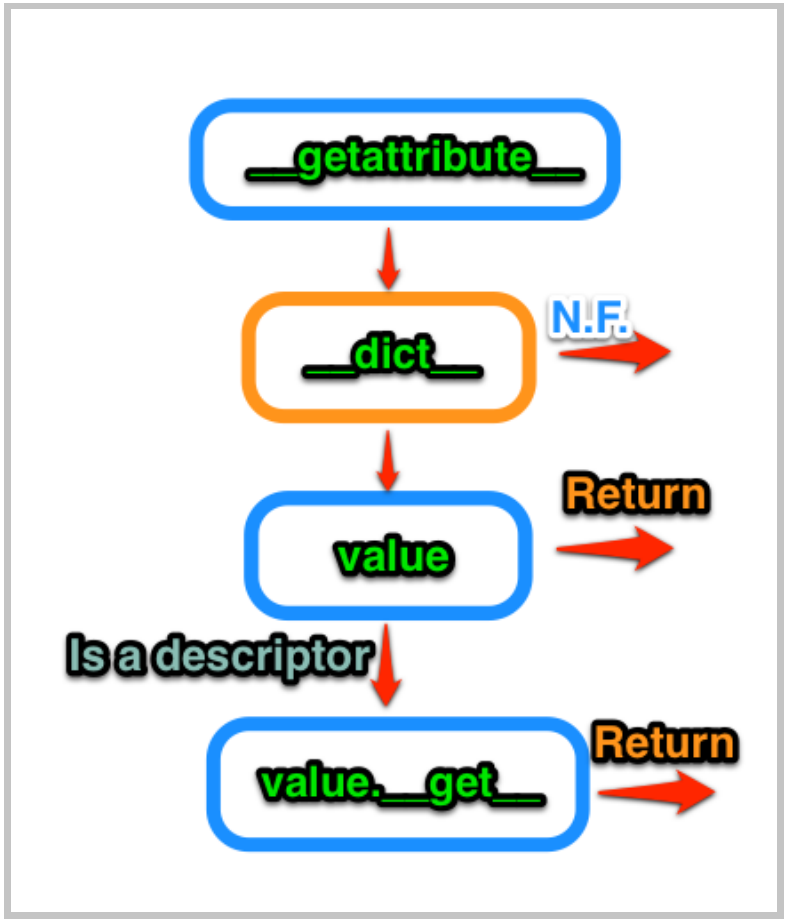
Before get into descriptor

The lookup chain of attribute/method

1. __getattribute__
2. __dict__
3. descriptor
4. __getattr__
5. AttibuteError

Classic lookup chain

New Mechanism in New-style class



__getattribute__ only work in new-style class

A descriptor class

It is the mechanism behind properties, methods, static methods, class methods, function, and super.

Descriptor Protocol

They are three specific methods.

```
Descriptor.__get__(self, obj, type=None) --> value

Descriptor.__set__(self, obj, value) --> None

Descriptor.__delete__(self, obj) --> None
```

Definition

If any of the methods in the **descriptor protocol** are defined for a class, its instance is said to be a descriptor object.

Example

```
class MyDescriptor(object):

    def __init__(self):
        self.val = "Init"

    def __get__(self, obj, type=None):
        return self.val

    def __set__(self, obj, val):
        if type(val) != str:
            raise TypeError("The value must be a string.")
        self.val = "The value I assigned to the variable is: %s" % val

    def __delete__(self, obj):
        self.val = None
```

Special cases

- data descriptor
  An object which defines both __get__ and __set__ function.
- non-data descriptor
  An object which only define __get__ function.

How to use Descriptor class

Basic Usage

```
class MyCls(object):
    my_desc = MyDescriptor()
```

```
>>> inst = MyCls()
>>> print inst.my_desc
'Init'
```

How it works?

What happens when an instance method is called?

We know

```
>>> MyCls.__dict__
dict_proxy({'my_desc': <__main__.MyDescriptor object at 0x1078b9c50>})
```

When you invoke

```
>>> inst.my_desc
```

According to **the lookup chain**, its "__get__" function is invoked.

```
>>> MyCls.__dict__["my_desc"].__get__(inst, MyCls)
```

Caveats

- The mechanism of descriptor object won't work if you assign it on an instance.
- A non-data descriptor will be replaced by attribute assignment.

Built-in descriptors

1. property
2. staticmethod / classmethod
3. functions
4. super

functions

There is an implict function class

```
>>> func = lambda x: x
>>> type(func)
<type 'function'>
```

Besides, every function is a **non-data descriptor** class

```
>>> func.__get__
<method-wrapper '__get__' of function object at 0x1078a17d0>

>>> func.__set__
Traceback (most recent call last):
AttributeError: 'function' object has no attribute '__set__'
```

Function(method) in a class

```
class FuncTestCls(object):
    def test(self):
        print "test"

>>> print type(FuncTestCls.__dict__['test'])
<type 'function'>
```

As you can see, it's a function.

Invoke by instance

As we have seen before,

```
>>> inst = FuncTestCls()
>>> inst.test
>>> FuncTestCls.__dict__['test'].__get__(inst, FuncTestCls)
<bound method FuncTestCls.test of <__main__.FuncTestCls object at 0x107
90b9d0>>
```

__call__

The place where a function context is put into.

```
def func(x, y):
    return x + y
```

```
>>> func.__call__(1, 2)
>>> 3
```

partial function

```
import functools
def func(a, b, c):
    print a, b, c
partial_func = functools.partial(func, "I am Jim.",)
```

```
>>> partial_func("Hey!", "Ha!")
>>> I am Jim. Hey! Ha!
```

__get__ function in function class

It returns a partial function whose first argument, known as self, is replaced with the instance object.

PSEUDO CODE

```
import functools
def __get__(self, instance, cls):
    return functools.partial(self.__call__, instance)
```

Let's review the **example**.

Additional usage

By the fact that a function is a descriptor object, every function can be invoked by an instance.

```
def inst_func(self):
    print self

class MyCls(object): pass

>>> print inst_func.__get__(MyCls(), MyCls)
>>> <bound method MyCls.inst_func of <__main__.MyCls object >>
```

Bound / Unbound

A function is said to be a bound method if its first variable is replaced by instance/class through __get__ function.
Otherwise, it is an unbound method.

Example - Bound method

```
>>> class C(object):
...     def test(self):
...         print "ttest"

>>> c = C()
>>> c.test
<bound method C.test of <__main__.C object at 0x10cf5a6d0>>
```

What is super

super is a function which returns a **proxy object** that delegates method calls to a parent or sibling class(according to MRO).

Basic usage of super

Consider the following example:

```
class A(object):
    attr = 1
    def method(self):
        print "I am A"

class B(A):
    attr = 1
    def method(self):
        super(B, self).method()
        print "I am B"

>>> b = B()
>>> b.method()
I am A
I am B
```

Fact

super is a kind of class

```
>>> sup_B = super(B)
>>> type(sup_B)
<type 'super'>
```

super is not a parent class

```
>>> A == super(B)
False
```

You have to delegate a target to super before you use it

```
>>> sup_B = super(B)
>>> sup_B.method
Traceback (most recent call last):
AttributeError: 'super' object has no attribute 'method'
```

super doesn't know who you want to delegate.

Try this:

```
>>> super(B, b).method
<bound method B.method of <__main__.B object at 0x105d84990>>
```

Again, what is super?

- Actually, it is a **descriptor object**.
- What super(B, b) does is super(B).__get__(b)

```
>>> proxy_b = sup_B.__get__(b)
>>> proxy_b.method
<bound method B.method of <__main__.B object>>
```

Conclude of super

```
super(B) != A
super(B, b) != super(B).__get__(b)
super(B, b).method == super(B).__get__(b).method
```

Q & A