# DIVE INTO PYTHON CLASS

## KNOWING PYTHON CLASS STEP-BY-STEP

Created by Jim Yeh / @jimyeh00

# ABOUT ME

- A front-to-end software developer
- Use Python since 2006
- Enjoy writing python code
- Little help on PyConTW website

# OUTLINE

- Introduce New-style class
- Descriptor
- super

# 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 old-style 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

# WHY WE NEED NEW-STYLE CLASS

- Unifying types and classes
- Providing meta-model (__metaclass__)
- Being able to subclass most built-in types

# WHAT'S NEW?

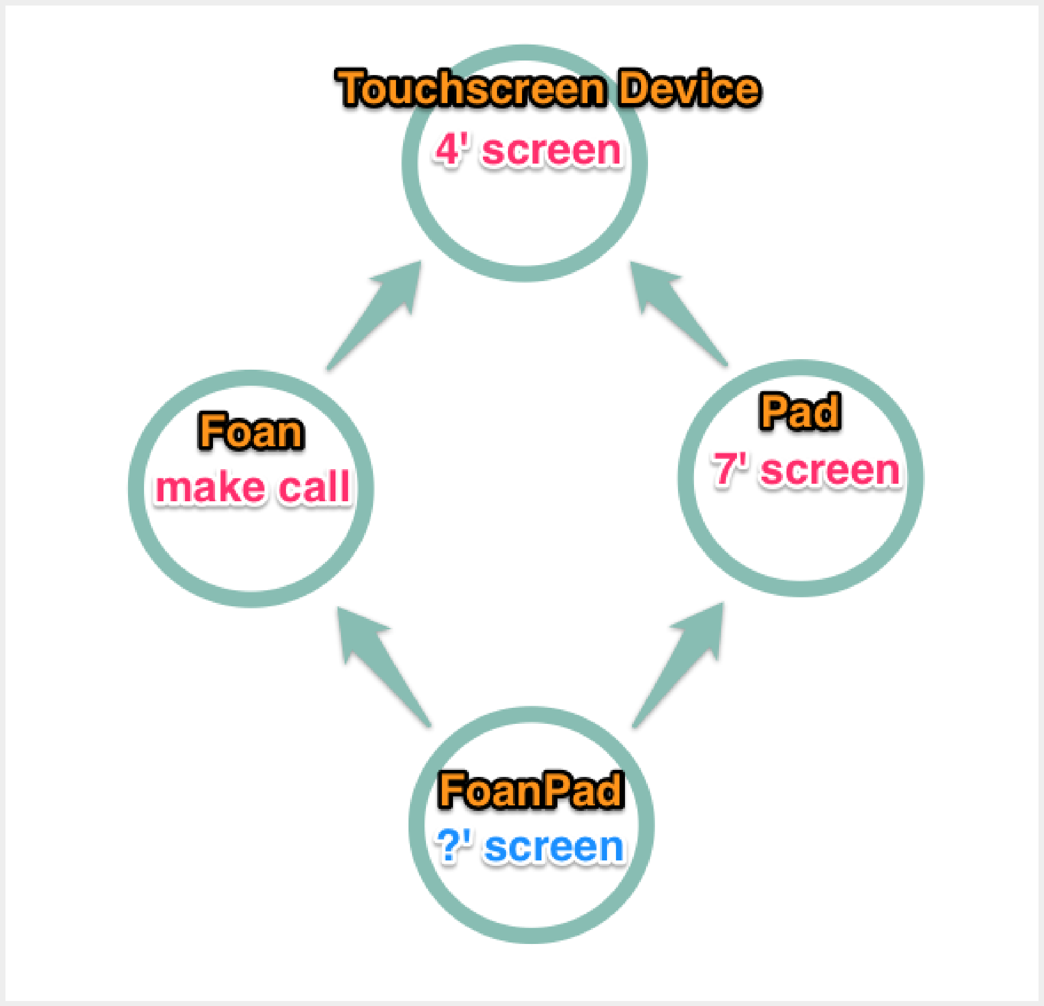
1. MRO
2. property
3. classmethod / staticmethod
4. descriptor (not a **decorator**)
5. super
6. __new__ and __metaclass__

# MRO

Method Resolution Order

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

# DIAMOND PROBLEM



# 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. ~~FoanPad, Foan, TouchScreen, Pad~~
3. FoanPad, Foan, Pad, **TouchScreen**

# PROPERTY

A implementation of getter / setter function in OO

# EXAMPLE

```python
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'
```

# CLASSMETHOD

A implementation of the overloading feature in C++

# EXAMPLE

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

    @classmethod
    def from_linux(cls, name):
        return cls(name, "linux")
```

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

# STATICMETHOD

An isolated function in a class

# EXAMPLE

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

    @staticmethod
    def version():
        return "1.0.0"
```

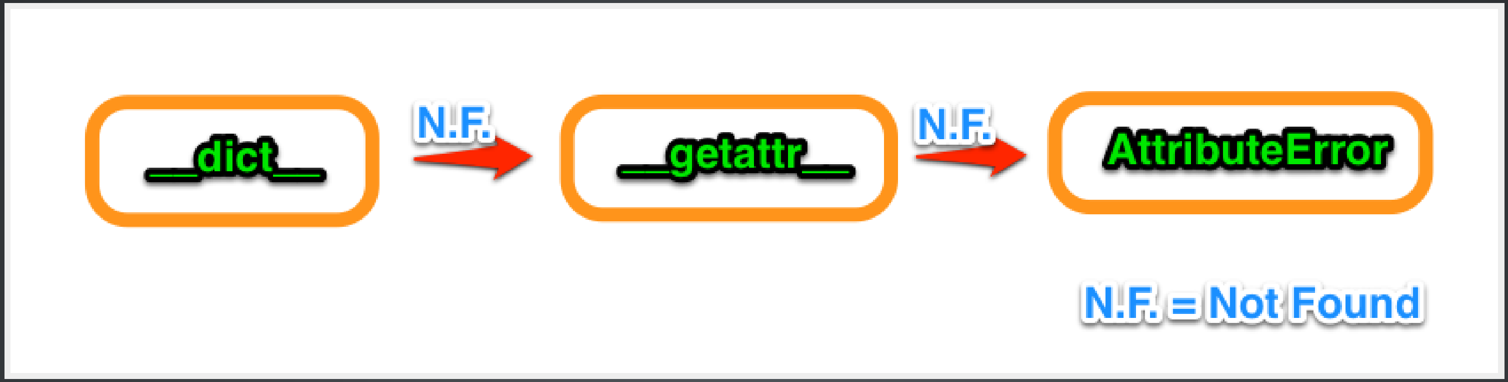
```python
>>> h = Host("My Host", "Linux")
>>> h.version()
```

# BEFORE GET INTO DESCRIPTOR

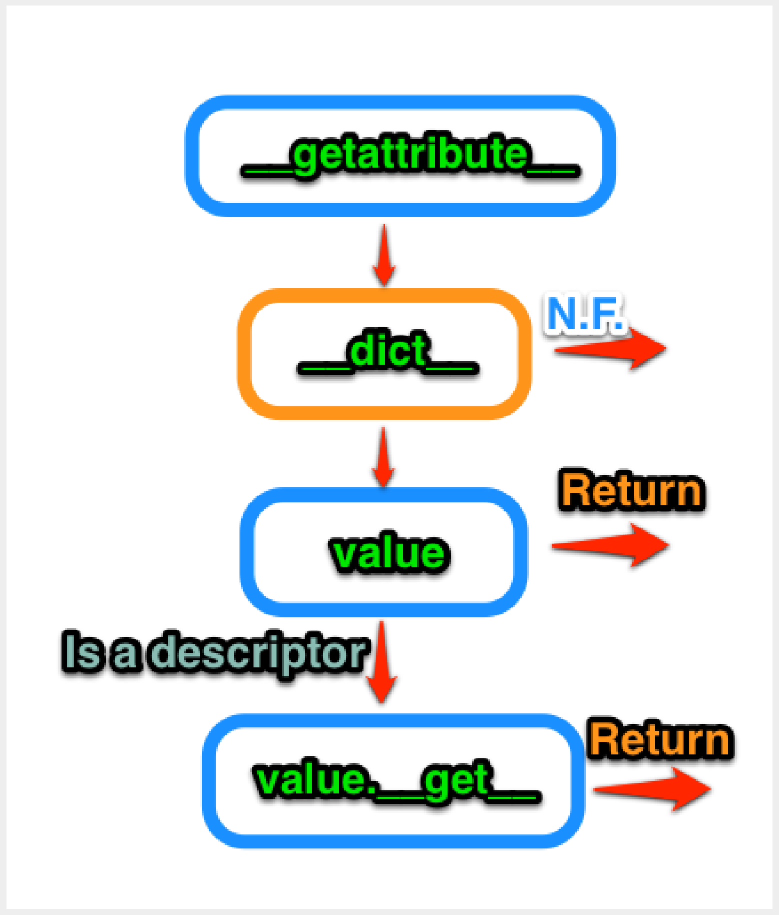
The lookup chain of attribute/method

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, 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 an object, it is said to be a descriptor.

# EXAMPLE

```python
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
  A object which defines both __get__ and __set__ function.
- non-data descriptor
  A object which only define __get__ function.

# HOW TO USE DESCRIPTOR CLASS

# USAGE

## 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

# PROPERTY - DESCRIPTOR VERSION

```python
class Student(object):
    def __init__(self, name):
        self._name = name
    class NameDescriptor(object):
        def __get__(self, obj, type=None):
            return obj._name
        def __set__(self, obj, val):
            obj._name = val
    name = NameDescriptor()
```

```
>>> s = Student("Jim")
>>> s.name
'Jim'
>>> s.__dict__
{'_name': 'Jim'}
>>> s.name = "Willy"
>>> s.name
'Willy'
>>> s.__dict__
{'_name': 'Willy'}
```

# CLASSMETHOD - DESCRIPTOR VERSION

```python
class SimulateClassMethod(object):
    def __init__(self, f):
        self._func = f
    def __get__(self, instance, owner):
        return self._func.__get__(owner, owner)

class Host(object):
    def __init__(self, name, os):
        self._name = name
        self._os = os
    def _from_windows(cls, name):
        return cls(name, "windows")
    from_windows = SimulateClassMethod(_from_windows)
    def _from_linux(cls, name):
        return cls(name, "linux")
    from_linux = SimulateClassMethod(_from_linux)
```

# STATICMETHOD - DESCRIPTOR VERSION

```
class SimulateStaticMethod(object):
    def __init__(self, f):
        self._func = f
    def __get__(self, instance, owner):
        return self._func

class Host(object):
    def __init__(self, name, os):
        self._name = name
        self._os = os
    def _version():
        return "1.0.0"
    version = SimulateStaticMethod(_version)
```

# ENCAPSULATION FACTORY

We know property can be encapsulate by a property decorator. But, what if there are lots of property?

# 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.

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

PSEUDO CODE

# 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

The result of replacing the first variable of a function by instance / class variable

# EXAMPLE - BOUND / UNBOUND

```
>>> class C(object):
...     def test(self):
...         print "ttest"
...     ttest = classmethod(test)
...     testt = test.__get__(C, C)
```

```
>>> C.test
<unbound method C.test>
>>> C().test
<bound method C.test of <__main__.C object at 0x10cf5a6d0>>
>>> C.testt
<bound method C.test of <class '__main__.C'>>
>>> C.ttest
<bound method type.test of <class '__main__.C'>>
```

# WHAT IS SUPER

super is a function which returns a **proxy object** that delegates method calls to a parent or sibling class of type.

# 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).test()
        print "I am B"

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

# AGAIN, WHAT IS SUPER?

Continue the previous example...

```
>>> 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.

Actually, it is a **descriptor object**.

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

# COMPARISON

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

# Q&A