Object-Oriented Programming in Python
Part 2: Advance on OOP

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

Object-Oriented Programming

Instance vs Class Attributes

class C:
    def __init__(self):
        self.class_attribute = "a value"
    def __str__(self):
        return self.class_attribute

[15:18]$ python3
>>> from C import C
>>> c = C()
>>> print(c)
a value
>>> c.class_attribute = "another value"
>>> c.class_attribute
'another value'
>>> c1 = C()
>>> c1.instance_attribute = "another value"
>>> c1.instance_attribute
'another value'
>>> c.instance_attribute
Traceback (most recent call last):
  File "stdin", line 1, in <module>
AttributeError: 'C' object has no attribute 'instance_attribute'
>>> C.another_class_attribute = 42
>>> c1.another_class_attribute, c.another_class_attribute
(42, 42)

C does not describe c1

Object-Oriented Programming

Alternative Way to Access Attributes: __dict__

>>> c.__dict__
{'class_attribute': 'a value'}
>>> c1.__dict__
{'class_attribute': 'a value', 'instance_attribute': 'another value'}
>>> c.__dict__['class_attribute'] = 'the answer'
>>> print(c)
the answer

__dict__ is an attribute
- it is a dictionary that contains the user-provided attributes
- it permits introspection

Let's dynamically change how things are printed.

>>> def introspect(self):
...     result="
...         for k,v in self.__dict__.items():
...             result += k + ': ' + v + '\n'
...     return result
...     return introspect
...     return introspect
>>> introspect
<function introspect at 0x7f05e5b52400>
>>> print(C.__dict__['f'])
<function f at 0x7f05e5b52400>
>>> d = D()
>>> d.class_attribute is D.__dict__['class_attribute']
True
>>> d.f is D.__dict__['f']
False

Functions are not accessed through the dictionary of the class.
- they must be bound to an instance

A bound method is a callable object that calls a function passing an instance as the first argument.
Object-Oriented Programming

Descriptors

```python
class Desc(object):
    "A descriptor example that just demonstrates the protocol"
    def __get__(self, obj, type):
        return self
    def __set__(self, obj, val):
        self.val = val
    def __delete__(self, obj):
        print('deleting')

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

References

Special Resolution Method

Attributes

__slots__

methods

super

__mro__ & diamond problem

__dict__

attributes

instance vs class

Descriptors

Object-Oriented Programming

Method Resolution Disorder: the Diamond Problem

```python
class A(object):
    def do_your_stuff(self):
        do_stuff_for_A()
        return

class B(A):
    def do_your_stuff(self):
        do_stuff_for_B()
        return

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

Two copies of A

- if do_your_stuff() is called once B or C is incomplete,
- if called twice it could have undesired side-effects

Computing the method resolution order (MRO)

There are a class attribute __mro__ for each type and a super

- __mro__ keeps the list of the superclasses without duplicates in a predictable order
- super is used in place of the find_out_whos_next()

```python
class A:
    def do_stuff(self):
        print('A')
        return

class B(A):
    def do_stuff(self):
        print('B')

class C(A):
    def do_stuff(self):
        print('C')

class D(B,C):
    def do_stuff(self):
        print('D')
```

Class B(A), B do the do_stuff for A

- if A is a superclass of B then B>A
- if C precedes D in the list of bases in a class statement then C>D
- if E is in one scenario then E>F in all scenarios

```python
>>> do_stuff() # do stuff for B
B
>>> do_stuff() # do stuff for D
D
>>> do_stuff() # do stuff for C
C
>>> do_stuff() # do stuff for B
B
```
Object-Oriented Programming
Special Methods

Special methods, as `__len__()`, `__str__()`, `__lt__()`, and `__add__()` govern the behavior of some standard operations.

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

def mylen():
    return 1

C().__len__ = mylen
print(len(C()))  # 0
```

Special methods are "class methods" – they cannot be changed through the instance – this goes straight to the type by calling `C.__len__()`

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

    def _mylen(self):
        return 0

mylen() = 1
C().__len__ = mylen
print(len(C()))  # 1
```

To be more flexible – the special method must be forwarded to a method that can be overridden in the instance

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

    def _mylen(self):
        return 1
```

Unfortunately the subtype of list allow the adding of attributes – this is due to the presence of `__dict__`

The presence of `__slots__` in a class definition inhibits the introduction of `__dict__` – this disallows any user-define attributes

```
class MyList(list):
    """A list that converts added items to ints""
    def append(self, item):
        list.append(self, int(item))
    def __setitem__(self, key, item):
        list.__setitem__(self, key, int(item))

m2 = MyList2()
try:
    m2.color = 'red'
except AttributeError:
    pass

m3 = MyList3()
try:
    m3.color = 'red'
    m3.weight = 50
except AttributeError:
    pass
```

References