Lecture 5: Python Classes and Objects

IPEC Winter School 2015
B-IT

Dr. Tiansi Dong & Dr. Joachim Köhler
class FirstClass:
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)

X = FirstClass()
Y = FirstClass()

X.setdata("Monkey King")
Y.setdata(3.14159)

X.display()  # Monkey King
Y.display()  # 3.14159

X.data = "queen"
X.display()  # "queen"
X.newdata = "new"
Basics of Class Inheritance

```python
class SecondClass(FirstClass):
    def display(self):
        print('value: "{}"'.format(self.data))

Z = SecondClass()
Z.setdata("1.414")

Z.display()
value: 1.414
```

A class can inherit from superclasses

Superclasses are listed in parentheses followed the class name

Changes can be made for inherited attributes, these changes do not affect attributes in superclasses
Attribute searching in a class tree
- From bottom to top
- From left to right

Class 1
.x
.y

Ins1
.x
.z

Class 2
.x
.z

Class 3
.w
.z

Ins1.x refers to Class1.x
Ins1.z refers to Class2.z
Why use Classes?

- Classes are factories which produce objects
- Relations between classes
  - Inheritance
  - composition

```python
class C2: ..
class C3: ..
class C1(C2,C3):
    def setname(self, who):
        self.name = who
```

```python
Obj1 = C1()
Obj2 = C1()
Obj1.setname('tom')
Obj2.setname('ana')
print(Obj1.name)
```
class ThirdClass(SecondClass):
    def __init__(self, value):
        self.data = value
    def __add__(self, other):
        return ThirdClass(self.data + other)
    def __str__(self):
        return '[ThirdClass: %s]' % self.data
    def mul(self, other):
        self.data *= other

>>> A = ThirdClass('abc')
>>> A.display()
value: abc
>>> print(A)
[ThirdClass: abc]
>>> B = A + 'xyz'
>>> print(B)
[ThirdClass: abcxyz]
>>> A.mul(3)
>>> print(A)
[ThirdClass: abcabcabc]
>>> class rec: pass
>>> rec.name = 'Bob'
>>> rec.age = 20
>>> print(rec.age)
20
>>> x = rec()
>>> y = rec()
>>> x.name, y.name
('Bob', 'Bob')
>>> x.name = 'Sue'

>>> rec.name, x.name, y.name
('Bob', 'Sue', 'Bob')

>>> rec.__dict__.keys()
dict_keys(['__module__', 'name', 'age', '__dict__', '__weakref__', '__doc__'])

>>> getattr(rec, '__module__')
'__main__'

>>> list(x.__dict__.keys())
['name']

>>> list(y.__dict__.keys())
[]

>>> def upperName(self):
    return self.name.name.upper()

>>> upperName(x)
'SUE'

>>> rec.method = upperName

>>> x.method()
'SUE'

>>> y.method()
'BOB'

>>> rec.__dict__.keys()
dict_keys(['__module__', 'name', 'age', '__dict__', '__weakref__', '__doc__'])

>>> getattr(rec, '__module__')
'__main__'

>>> list(x.__dict__.keys())
['name']

>>> list(y.__dict__.keys())
[]
A Realistic Example for OOP in Python

- We will go through an example of OOP design for the information recording and processing about people
- The OOP design process will be instructed step by step
- All of the main ideas in Python's OOP will be demonstrated
- Python's class system is really just two things
  - A matter of searching for an attribute in a tree of objects
  - A special first argument for functions
Step 1: Making Instances (1/2)

- The information about a person in a company
  - Name
  - Job
  - Payment

- Create a module file person.py and type:

```python
# file person.py

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
```
Step 1: Making Instances (2/2)

- Testing before going on

```python
# file person.py
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay

if __name__ == '__main__':
    Bob = Person('Bob Smith')
    Sue = Person('Sue Jones', job='dev', pay=10000)
    print(Bob.name, Bob.job, Bob.pay)
    print(Sue.name, Sue.job, Sue.pay)

% python person.py
Bob Smith None 0
Sue Jones dev 10000
```
Step 2: Adding Behaviors (1/2)

- We can test interactively some operations, and add some codes into the self-test part

```python
>>> name = 'Bob Smith'
>>> name.split()
['Bob', 'Smith']
>>> name.split()[-1]
'Smith'
>>> pay = 10000
>>> pay *= 1.10
11000.0
```

```python
If __name__ == '__main__':
    ...
    print(Bob.name.split()[-1])
    Sue.pay *=1.10
    print(Sue.pay)
```
Step 2: Adding Behaviors (2/2)

- Coding new methods in the class (encapsulation)

```python
# file persona.py

class Person:
    def __init__(...):
        ...
        def lastName(self):
            return self.name.split()[-1]
        def giveRaise(self, percent):
            self.pay = int(self.pay * (1+percent))

If __name__ == '__main__':
    ...
    print(Bob.lastName(), Sue.lastName())
    Sue.giveRaise(0.10)
    print(Sue.pay)
```
Step 3: Operator Overloading

- Overloading the print function through `__str__()`

```python
# file persona.py

class Person:
    ...
    
    def __str__(self):
        return '[Person: %s, %s, %s]' % (self.name, self.job, self.pay)

If __name__ == '__main__':
    ...
    print(Bob)
    Sue.giveRaise(.10)
    print(Sue)

[Person: Bob Smith, None, 0]
[Person: Sue Jones, dev, 11000]
```
Step 4: Customizing behavior by subclassing

- Managers in a company have a bonus percent, additional to the given percent of raise.

```python
# file persona.py

class Person:
    ...

class Manager(Person):
    def giveRaise(self, percent, bonus = .10):
        self.pay = int(self.pay*(1+percent+bonus))

If __name__ == '__main__':
    ...
    Tom = Manager('Tom Jones', 'mgr', 50000)
    Tom.giveRaise(.10)
    print(Tom)

[Person: Tom Jones, mgr, 60000]
```
Step 4: Customizing behavior by subclassing

- Managers in a company have a bonus percent, additional to the given percent of raise.

```python
# file persona.py

class Person:
    ...

class Manager(Person):
    def giveRaise(self, percent, bonus = .10):
        self.pay = int(self.pay*(1+percent+bonus))

If __name__ == '__main__':
    ...
    Tom = Manager('Tom Jones', 'mgr', 50000)
    Tom.giveRaise(.10)
    print(Tom)

[Person: Tom Jones, mgr, 60000]
```
Step 5: Customizing constructors

- Managers certainly have the job 'Manager'.

```python
# file persona.py

class Person:
    ...

class Manager(Person):
    def __init__(self, name, pay):
        Person.__init__(self, name, 'mgr', pay)
    ...

if __name__ == '__main__':
    ...
    Tom = Manager('Tom Jones', 50000)
    ...

[Person: Tom Jones, mgr, 60000]
```
Step 6: Using Introspection Tools (1/3)

Class introspection with `__class__`, `__class__.__name__`, `__dict__`

```python
>>> from person import Person
>>> bob = Person('Bob Smith')
>>> bob.__class__
<class 'person.Person'>
>>> bob.__class__.__name__
'Person'
>>> list(bob.__dict__.keys())
['pay', 'job', 'name']
>>> for key in bob.__dict__:
...     print(key, '=>', bob.__dict__[key])
...     pay => 0
...     job => None
...     name => Bob Smith
```
A generic display tool

```python
#File classtools.py

class AttrDisplay:
    '''
    Overload print function, so that all attributes of a
class/object will be displayed
    '''

def gatherAttrs(self):
    attrs = []
    for key in sorted(self.__dict__):
        attrs.append('%s=%s' % (key, getattr(self, key)))
    return ', '.join(attrs)

def __str__(self):
    Return ' [%s: %s] ' % (self.__class__.__name__, self.gatherAttrs())
```
Step 6: Using Introspection Tools (3/3)

- Using generic display in the person.py

```python
#File person.py

from classtools import AttrDisplay

class Person(AttrDisplay):
    def __init__(...):
        ...
    def lastName(self):
        ...
    def giveRaise(self, percent):
        ...
        
    def __str__(self):
        ...

if __name__ == '__main__':
    Bob = Person('Bob Smith')
    print(Bob)

[Person: job=None, name=Bob Smith, pay=0]
```
Step 7: Storing Objects in a Database (1/2)

- shelf uses `pickle` and `dbm` modules to store Python objects on a file by key

```python
# File makedb.py

from person import Person, Manager
bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

import shelf
db = shelf.open('persondb')
for obj in (bob, sue, tom):
    db[obj.name] = obj
db.close()
```
Step 7: Storing Objects in a Database (2/2)

- **Exploring a** `shelve` **database**

```python
>>> import glob

>>> glob.glob('person')
['person.py', 'person.pyc', 'persondb.bak', 'persondb.dat', 'persondb.dir']

>>> import shelve

>>> db = shelve.open('persondb')

>>> len(db)
3

>>> list(db.keys())
['Tom Jones', 'Sue Jones', 'Bob Smith']

>>> bob = db['Bob Smith']

>>> print(bob)
[Person: job=None, name=Bob Smith, pay=0]
```
Details of Python OOP: class

- The `class` statement in Python is not a declaration, rather a statement for object creation and assignment.

  ```python
class <name>(superclass,...):
    data = value
    def method(self, ...):
      self.member = value
  ```

- Because `class` is a compound statement, any sort of statement can be nested inside its body
  - A name assigned with a value is a class attribute
  - A name assigned with a function is a class method

- Class attributes are not the same as its object attributes
Details of Python OOP: class attributes

- Class attributes are not the same as its object attributes

```python
>>> class CData:
    Data = 13

>>> x = CData()
>>> y = Cdata()
>>> x.Data, y.Data
(13, 13)
>>> CData.Data = 25
>>> x.Data, y.Data, CData.Data
(25, 25, 25)
>>> x.Data = 0
>>> x.Data, y.Data, CData.Data
(0, 25, 25)
>>> CData.Data = 100
>>> x.Data, y.Data, CData.Data
(0, 100, 100)
```
• How about class methods?

```python
>>> class CData:
    Data = 13
    def printer(self):
        print(“hello"

>>> x = CData()
>>> x.printer()
hello
```

instance.method(args...) ==> class.method(instance, args, ...)

• In class methods, the first argument is usually `self`, however, technically, only its position is significant, you can use other names.
• Simple example of class inheritance

```python
>>> class Super:
    def method(self):
        print('in Super.method')

>>> class Sub(Super):
    def method(self):
        print('starting Sub.method')
        Super.method(self)
        print('ending Sub.method')

>>> x = Super()
>>> x.method()
in Super.method

>>> y = Sub()
>>> y.method()
starting Sub.method
in Super.method
ending Sub.method
```
Details of Python OOP: class inheritance

- Methods in sub-class can use methods in its super-class in ways as follow
  - Inheritance
    - Just use the method defined in the super-class
  - Replacement
    - Define a new method in the sub-class
  - Extension
    - Besides of running the method defined in the super-class, run some new operations in the sub-class
  - Realization
    - Implement an abstract method in the super-class
Abstract Superclass

- Example of an abstract superclass

```python
>>> class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!

>>> x = Super()
>>> x.delegate()
AssertionError: action must be defined!
```

- An abstract method can not be called before being implemented in the subclass
Namespaces: the whole story

- Names can be unqualified names and qualified names
  - Namespaces of unqualified names follow LEGB rule
  - Local, Enclosing function local, Global, Built-in
  - Namespaces of qualified names follow object namespace

```python
# names.py
X = 11

def f():
    print(X)

def g():
    X=22
    print(X)

class C:
    X=33
    def m(self):
        X=44
        self.X=55

def __name__ == '__main__':
    if __name__ == '__main__':
        print(X)
        f()
        g()
        print(X)
        Obj = C()
        print(Obj.X)
        Obj.m()
        print(Obj.X)
        print(C.X)
        print(C.m.X)
        print(g.X)
    elif __name__ == 'names.py':
        print(X)
        import names
        print(X)
        print(names.X)
        names.f()
        names.g()
        print(names.C.X)
        ins = names.C()
        print(ins.X)
        ins.m()
        print(ins.X)
```
Namespaces: the whole story

- Imagine: variables are toilet paper; functions are toilets; a namespace is WC; classes are namespaces and create toilets;

```python
# names.py
X = 11

def f():
    print(X)

def g():
    X=22
    print(X)

class C:
    X=33
    def m(self):
        X=44
        self.X=55

if __name__ == '__main__':
    print(X)     #11
    f()          #11
    g()          #22
    print(X)     #11

Obj = C()
print(Obj.X) #33
Obj.m()

print(Obj.X) #55
print(C.X)   #33
print(C.m.X)
print(g.X)

# file.py
import names

X = 66
print(X)
print(names.X)

names.f()
names.g()

names.C()

print(names.C.X)
ins = names.C()
print(ins.X)
ins.m()
print(ins.X)
```
Namespaces: the whole story

Namespace follows the “toilet paper” rule: you choose the toilet paper nearest to you
Namespace dictionary and links

- Use `__class__`, `__bases__`, `__dict__` attributes, we can visit the structure of namespace
  - `__class__` links to the class
  - `__bases__` returns a tuple containing links to superclasses
  - `__dict__` returns all attribute-values of an object

```python
>>> class super:
    def hello(self):
        self.x = 'hi'

>>> class sub(super):
    def hola(self):
        self.y = 'ho'

>>> X = sub()
>>> X.__dict__
{}  # X has no attributes
>>> super.__bases__
(<class '__main__.super'>,)
>>> sub.__bases__
(<class '__main__.super'>,)

>>> X.hello()
>>> X.__dict__
{'x': 'hi'}  # X gets an attribute 'x' after calling hello

>>> X.hola()
>>> X.__dict__
{'x': 'hi', 'y': 'ho'}  # X gets an attribute 'y' after calling hola
```
def classtree(cls, indent):
    print('.' * indent + cls.__name__)
    for supercls in cls.__bases__:
        classtree(supercls, indent+3)

def instancetree(inst):
    print('Tree of %s % inst)
classtree(inst.__class__, 3)

def selftest():
class A: pass
class B(A): pass
class C(A): pass
class D(B,C): pass
class E: pass
class F(D,E): pass
instancetree(B())
instancetree(F())

if __name__ == '__main__':
    selftest()