Global Startup

MEXICO 2015, iLAB XALAPA

http://tiny.cc/GSLMEX15


MIT Global Startup
Object Oriented Programming in Python

Objects

- A unique instance of a data structure that's defined by its class.
- An object comprises both state (data) and behavior (methods)
- Objects are made up of attributes and methods
- Object instances are specific realizations of a class.

Class

- A user-defined prototype for an object that defines any object of the class.
- The attributes are data members and methods.

- Class includes two members: form and object.
- The example in the following can reflect what is the difference between object and form for class.

```
class A:
    i = 123
    def __init__(self):
        self.i = 12345

print A.i
print A().i
```

```
>>>
123
12345
```

Invoke form: just invoke data or method in the class, so i=123

Invoke object: instantialize object Firstly, and then invoke data or Methods.
Here it experienced __init__(), i=12345

Encapsulation

Only what is necessary is exposed (public interface) to the outside. Implementation details are hidden to provide abstraction. Abstraction should not leak implementation details. Abstraction allows us to break up a large problem into understandable parts

Inheritance

“a dog (subclass) is a mammal (parent / superclass)"
Subclass is derived from / inherits / extends a parent class.
Override parts with specialized behavior and extend it with additional functionality.
Liskov substitution principle: What works for the parent class should also work for any subclass.
Python supports a limited form of multiple inheritance as well.

Polymorphism

Different subclasses can be treated like the parent class, but execute their specialized behavior. Example: When we let a mammal make a sound that is an instance of the dog class, then we get a barking sound.

Inheritance


```
class Person:
    def speak(self):
        print 'I can speak'

class Man(Person):
    def wear(self):
        print 'I wear shirt'

class Woman(Person):
    def wear(self):
        print 'I wear Skirt'

man = Man()
man.wear()
man.speak()
```

```
>>>
I wear shirt
I can speak
```

Inheritance in Python is simple,
Just like JAVA, subclass can invoke
Attributes and methods in superclass.

From the example, **Class Man** inherits **Class Person**, and invoke **speak()** method In **Class Person**

Inherit Syntax:

```
class subclass(superclass):
        ...
        ...
```

In Python, it supports multiple inheritance,
In the next slide, it will be introduced.

Multiple Inheritance

- Python supports a limited form of multiple inheritance.
- A class definition with multiple base classes looks as follows:

```
class DerivedClass(Base1, Base2, Base3 …)
        <statement-1>
        <statement-2>
        …
```

- The only rule necessary to explain the semantics is the resolution rule used for class attribute references. This is depth-first, left-to-right. Thus, if an attribute is not found in DerivedClass, it is searched in Base1, then recursively in the classes of Base1, and only if it is not found there, it is searched in Base2, and so on.

Self

- "Self" in Python is like the pointer "this" in C++. In Python, functions in class access data via "self".

```
class Person:
    def __init__(self,name):
        self.name = name
    def PrintName(self):
        print self.name

P = Person('Yang Li')
print P.name
P.PrintName()
```

- "Self" in Python works as a variable of function but it won't invoke data.

Encapsulation

- In Python, there is no keywords like 'public', 'protected' and 'private' to define the accessibility. In other words, In Python, it acquiesce that all attributes are public.

- But there is a method in Python to define Private:

  Add "__" in front of the variable and function name can hide them when accessing them from out of class.

An example of Private


```
class Person:
    def __init__(self):

        self.A = 'Yang Li'
        self.__B = 'Yingying Gu'

    def PrintName(self):
        print self.A
        print self.__B

P = Person()
```

Public variable

Private variable

Invoke private variable in class

```
>>> P.A
'Yang Li'
>>> P.__B
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    P.__B
AttributeError: Person instance has no attribute '__B'
>>> P.PrintName()
Yang Li
Yingying Gu
```

Access public variable out of class, succeed

Access private variable our of class, fail

Access public function but this function access Private variable __B successfully since they are in the same class.

Python OOPS Basics - 1

- All classes are derived from `object` (*new-style classes*).

```
class Dog(object):
    pass
```

- Python objects have data and function attributes (*methods*).

```
class Dog(object):
    def bark(self):
        print "Wuff!"

snowy = Dog()
snowy.bark()  # first argument (self) is bound to this Dog instance
snowy.a = 1  # added attribute a to snowy
```

- Always define your data attributes first in `__init__`.

```
class Dataset(object):
    def __init__(self):
        self.data = None
    def store_data(self, raw_data):
        ... # process the data
        self.data = processed_data
```

Python OOPS Basics - 2

- Class attributes are shared across all instances.

```
class Platypus(Mammal):
    latin_name = "Ornithorhynchus anatinus"
```

- Use `super` to call a method from a superclass.

```
class Dataset(object):
    def __init__(self, data):
        self.data = data

class MRIDataset(Dataset):
    # __init__ does not have to follow the Liskov principle
    def __init__(self, data, parameters):
        # here has the same effect as calling
        # Dataset.__init__(self)
        super(MRIDataset, self).__init__(data)
        self.parameters = parameters

mri_data = MRIDataset([1,2,3], {'amplitude': 11})
```

Note: In Python 3 `super(B, self)` can be written `super()`.

Python OOPS Basics - 3

- *Special / magic* methods start and end with two underscores ("dunder") and customize standard Python behavior (e.g., operator overloading).

```
class My2Vector(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return My2Vector(self.x+other.x, self.y+other.y)

v1 = My2Vector(1, 2)
v2 = My2Vector(3, 2)
v3 = v1 + v2
```

- *Properties* allow you to add behavior to data attributes:

```
class My2Vector(object):
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        print "returning x, which is {}".format(self._x)
        return self._x

    def set_x(self, x):
        print "setting x to {}".format(x)
        self._x = x

    x = property(get_x, set_x)

v1 = My2Vector(1, 2)
x = v1.x  # uses the getter, which prints the value
v1.x = 4  # uses the setter, printing the value
```

Helps with refactoring (can replace direct attribute access with a property).

How to come up with a good structure for classes and modules?

- KIS & iterate. When you see the same pattern for the third time then it might be a good time to create an abstraction (refactor).
- Sometimes it helps to sketch with pen and paper.
- Classes and their inheritance often have no correspondence to the real-world, be pragmatic instead of perfectionist.
- *Design principles* tell you in an abstract way what a good design should look like (most come down to *loose coupling*).
- *Testability* (with unittests) is a good design criterium.
- *Design Patterns* are concrete solutions for reoccurring problems.

References:

http://www.cs.colorado.edu/~kena/classes/5448/f12/presentation-materials/li.pdf

https://python.g-node.org/python-summerschool-2013/_media/wiki/oop/oo_design_2013.pdf