中原大學
Chung Yuan Christian University

OO programming and design (python ver.)

- Define a class in python
- In python, instance variables are called **attribute**
- Class methods must have an extra first parameter self in the method definition. (self is similar to the keyword this in Java)

```
class Animal():
    def __init__(self, name):
        self.name = name

    def eat(self):
        print('eat!')
```

- Create an object

```
ani = Animal('john')
ani.eat()
```

When we invoke ani.eat() method, this is automatically converted by Python into Animal.eat(ani), this is all the special self is about

- The __init__() method is similar to the constructor in Java.
- It is run as soon as an object of a class is instantiated. The method is useful to do some initializations
- Attributes created in __init__() are called **instance attributes**

- On the other hand, attributes that are declared directly inside the class are called **class attributes**. They have the same value for all class instances (i.e., the static things in Java)

```
class Animal():

    str = 'Animals on the earth'

    def __init__(self, name, age):
        self.name = name

print(Animal.str)
```

- Encapsulation
- Python does not have access modifiers (public, private, protected)
- In fact, attributes are always regarded as public
- To reach encapsulation, python uses some conventions, For example:

- Name mangling
- A name prefixed with two underscores (e.g. __name) should be treated as a non-public part of the API (whether it is a method or a data member)

```
class Animal():
    def __init__(self, name, age):
        self.name = name
        self._age = 10  # regarded as private
```

- Python uses the property decorator to behave as getter and setter

```
class Animal():

    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        self.__name = name

ani = Animal('lucky')
print(ani.name)
```

If you do not write the setter, when you try to write something like ani.name = 'Mary', you will get an AttributeError Exception

- Inheritance

In fact, this is equivalent to
`class Animal(object):`

```
class Animal:

    def __init__(self):
        pass

    def walk(self):
        print('walking')

class Dog(Animal):

    def bark(self):
        print('woof!')

ani = Dog()
ani.walk()
```

Put the superclass type as the first paramter

- Abstract
- Python does not have the abstract keyword. Instead, it uses ***decorators***

- There are several ways to mark abstract method in python. For example, you have to import abc (Abstract Base Classes) library

- If the subclasses do not override this method, it will raise a TypeError: Can't instantiate abstract class Animal with abstract methods setname

```
from abc import ABC, abstractmethod

class Animal(ABC):

    def __init__(self, name):
        self.__name = name

    @abstractmethod
    def setname(self):
        return NotImplemented

class Dog(Animal):
    pass

ani = Dog('lucky')
```

- Polymorphism
- Polymorphism is less observed in python because it supports dynamic typing (i.e., you can change the type of a variable at run-time)
- However, polymorphism is almost everywhere in Python

- Duck typing
- "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

- For example, python supports a data structure – list, which can be seen as an array that can store any objects

- Something that matches its behavior to anything then it will consider a thing of that category to which it matches.

- static method

- Use the decorator @staticmethod to mark a method as static

```
class Animal:
    def __init__(self):
        pass

    @staticmethod
    def aStaticMethod(msg):
        print(msg)

Animal.aStaticMethod('hello')
```

Note that the static method does not pass self into it

- In Java, static methods and static variables are regarded as class-level
- That is, in Java static methods can access static variables which are in the same class
- However, this isn't work in python. Static methods cannot access static variables

```
class Animal:
    place = 'earth'

    def __init__(self):
        pass

    @staticmethod
    def aStaticMethod():
        print(place)  # NameError
```

- Instead, python use the decorator @classmethod which indicates that a method will take a class as the input

```
class Animal:
    place = 'earth'

    def __init__(self):
        pass

    @classmethod
    def aClassMethod(cls):
        print(cls.place)
```

Use the term cls as the parameter

Design patterns using python

- Simple Factory Pattern
- Goal: design a RPG game. Firstly, we need to create some characters such as barbarians, wizards, and priests.

- Create a factory class whose duty is to create objects for caller.

```
class Character():
    def __init__(self):
        pass
    def echo(self):
        pass

class Barbarian(Character):
    def __init__(self):
        pass
    def echo(self):
        print('I am a barbarian!')

class Priest(Character):
    def __init__(self):
        pass
    def echo(self):
        print('I am a priest!')

class CharacterFactory():
    def create(self, char):
        if char == 'Barbarian':
            return Barbarian()
        elif char == 'Priest':
            return Priest()

c = CharacterFactory().create('Priest')
c.echo()
```

- Factory method
- Define an interface which has a abstract method create()
- Create a concrete class BarbarianFactory which implements the create() behavior

```
import abc

class CharacterFactory(metaclass=abc.ABCMeta):

    def __init__(self):
        pass
    @abc.abstractmethod
    def create(self, name):
        return NotImplemented

class Barbarian():
    def __init__(self, name):
        self.name = name
    def echo(self):
        print('I am ' + self.name)

class BarbarianFactory(CharacterFactory):
    def __init__(self):
        pass

    def create(self, name):
        return Barbarian(name)

b = BarbarianFactory().create('john')
b.echo()
```

- Abstract Factory method
- Chance to use: when you need many factory methods

```
import abc

class EquipmentFactory(metaclass = abc.ABCMeta):
    def __init__(self):
        pass
    @abc.abstractmethod
    def createWeapon(self):
        return NotImplemented

    @abc.abstractmethod
    def createArmor(self):
        return NotImplemented

class CharacterFactory():

    def createWeapon(self):
        return Sword()
    def createArmor(self):
        return Chainmail()
```

- Builder Pattern
- User need to create an object with lots of possible configuration options
- Remember: the caller knows nothing about how to create a Trip instance.

```
class Trip():
    def __init__(self):
        self._date = None
        self._hotel = None

    def __str__(self):
        return "{0},{1}".format(self._date,self._hotel)

class TripBuilder():
    def __init__(self):
        self._trip = Trip()

    def setDate(self, date):
        self._trip._date = date

    def setHotel(self, hotel):
        self._trip._hotel = hotel

class ManageTrip():

    def __init__(self):
        self._builder = TripBuilder()

    def manage(self, date, hotel):
        self._builder.setDate(date)
        self._builder.setHotel(hotel)

    @property
    def trip(self):
        return self._builder._trip
```

- Singleton
- One class will only have one instance

- Note that python does not have real `private' things

```
class SingletonDemo():
    _instance = None

    @staticmethod
    def getInstance():
        if SingletonDemo._instance is None:
            SingletonDemo()
        return SingletonDemo._instance

    def __init__(self):
        if self._instance is None:
            SingletonDemo._instance = self
        else:
            raise Exception('only one instance is available.')
```

- Command pattern
- Encapsulate a request on a specific objec

```
class Light:
    def on(self):
        print('turn on the light!')

class Vacuum:
    def start(self):
        print('vacuum starts cleaning.')
```

```
lc = LightCommand(Light())
vc = VacuumCommand(Vacuum())
rc = [lc, vc]
for iot in rc:
    iot.execute()
```

In this loop, we think iot is a device that can be executed.

```
class Command(ABC):
    @abstractmethod
    def execute(self):
        return NotImplemented

class LightCommand(Command):

    def __init__(self, light):
        self.__light = light

    def execute(self):
        self.__light.on()

class VacuumCommand(Command):

    def __init__(self, vacuum):
        self.__vacuum = vacuum

    def execute(self):
        self.__vacuum.start()
```

- Summary
- Any programming language is good for patterns
- Python is an 100% object-oriented language so many design patterns can be elegantly applied in python programming