1 基本程式結構

1.1 類別class

self 指的是物件實例(Instance)本身

In [1]:
```
class Dog:
    def __init__(self, name, age): #建構子
        self.name = name  #instance variabels實體變數 是public
        self.age = age
    def showMe(self):
        return "我的名字:"+self.name
    def __str__(self): #被print()列印時會執行此程式，等同於java的toString()
        return "姓名:%s, age:%d" %(self.name, self.age)
```

In [2]:
```
billy = Dog("Billy",5)
billy.showMe()
```

Out[2]:
```
'我的名字:Billy'
```

In [3]:
```
billy.age
```

Out[3]:
```
5
```

In [154]:
```
print (billy)
```

```
姓名:Billy, age:5
```

In [155]:
```
willy = Dog("Willy",2)
print(willy)
```

```
姓名:Willy, age:2
```

```
Java的類別是這樣寫，比較複雜嚴謹
```

```
public class Dog
{
    int age;
    String name;

    public Dog( int  age)
    {
        this.age = age;
    }
    public Dog( String name, int  age)
    {
        this.name = name;
        this.age = age;
    }

    public void ShowMe()
    {
       System.out.println("I am "+ name+"歲數:"+ age);
    }
    @Overide
    public String toString()
    {
    }

    pubic static void main(String[] args)
    {
        Dog billy = new Dog("Billy", 5);
        billy.showMe();
    }

}
```

1.2 驗收：請完成Circle類別

```
class Circle:
    ...
```

```
c = Circle(10)

c.area()

面積:314.159...

c.info()

這是Circle物件，半徑:10

print(c)
半徑:10, 面積:314.159
```

In [ ]:

In [10]:

```
#Another good example
class Account:
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        if amount <= 0:
             raise ValueError('amount must be positive')
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            raise RuntimeError('balance not enough')
        self.balance -= amount

    def __str__(self):
        return 'Account information:{0}, {1}'.format(
            self.name, self.balance)
```

In [11]:
```
acc1 = Account("Huang",1000)
print acc1
```

```
Account information:Huang, 1000
```

In [ ]:
```
```

In [ ]:
```
```

1.3 實體變數 類別變數 靜態變數

- instance variable
- class variable
- static variable

@staticmethod function is nothing more than a function defined inside a class. It is callable without instantiating the class first. It’s definition is immutable via inheritance.

@classmethod function also callable without instantiating the class, but its definition follows Sub class, not Parent class, via inheritance. That’s because the first argument for @classmethod function must always be cls (class).

When to use what? We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases. We generally use static methods to create utility functions.

https://www.geeksforgeeks.org/class-method-vs-static-method-python/ (https://www.geeksforgeeks.org/class-method-vs-static-method-python/)

```
Factory Method模式在一個抽象類別中留下某個建立元件的抽象方法沒有實作，其它與元件操作相關聯的方法都先依賴於元件所定義的介面，而不是依賴於元件的實現， 當您的成品中有一個或多個元件無法確定時，您先確定與這些元件的操作介面，然後用元件的抽象操作介面先完成其它的工作，元件的實作（實現）則推遲至實現元 件介面的子類完成，一旦元件加入，即可完成您的成品。

簡單地說，如果您希望如何建立父類別中用到的物件這件事，是由子類別來決定，可以使用 Factory Method。

就是抽象方法!
```

```
@classmethod
第一個參數永遠綁定為類別物件本身，無論是以實例方法來呼叫，或是以靜態方法來呼叫
```

@staticmethod
如果你在定義類別時希望某個函式，完全不要作為實例的綁定方法，也就是不要將第一個參數綁定為所建立的實例，則可以使用@staticmethod加以修飾。

結論:

utility函數用@staticmethod 比較方便

若有繼承情況，希望子類別也能操作，則使用@classmethod

In [ ]:

In [156]:
```python
#class static method
#exampel from Gossip良葛格
class Some:
    def __init__(self, x):
        self.x = x
        print (self)

    @classmethod
    def service(cls, y):
        print('do service...', cls, y)
```

In [158]:
```python
s = Some(10)
```
```
<__main__.Some object at 0x0000022CA4512390>
```

In [159]:
```python
s.service(20)
Some.service(30)
```
```
do service... <class '__main__.Some'> 20
do service... <class '__main__.Some'> 30
```

In [ ]:

```
In [ ]:
```

```
In [160]: #example from a python book
          class Book:
              price=100  #class variable
              @classmethod
              def display(cls): #cls: class的縮寫
                  print (cls.price)
              def set(self,x):
                  self.price=x    #self.price  instance variable

              def show(self):
                  print (self.price)
```

```
In [161]: b=Book()
```

```
In [162]: Book.display()
```

```
100
```

```
In [163]: b.display()
```

```
100
```

```
In [164]: b.set(200)
```

```
In [165]: b.show()
```

```
200
```

```
In [166]: Book.display()
```

```
100
```

In [ ]:

In [167]:
```
#staticmethod and classmetnod
class Product:
    count = 0 #class variable (public)
    def __init__(self, name):
        self.name=name
        Product.count += 1
    @staticmethod
    def getStaticCount():
        return Product.count
    @classmethod
    def getClassCount(cls):
        print('Class info:%s'  % cls )
        print ('Class method - The product count is: %s'  %cls.count)
```

In [168]:
```
p1=Product('Camera')
p2=Product('Cell')
```

In [169]:
```
Product.getClassCount()
```
```
Class info:<class '__main__.Product'>
Class method - The product count is: 2
```

In [170]:
```
Product.getStaticCount()
```
Out[170]:
```
2
```

In [171]:
```
p1.getClassCount() #都得到相同的結果
```
```
Class info:<class '__main__.Product'>
Class method - The product count is: 2
```

In [ ]:

In [ ]:

In [77]:
```
#example
class Dog:
    count=0  #class variable
    def __init__(self, na, n):
        self.name = na
        self.age = n
        Dog.count += 1
    def __str__(self):
        return "姓名:%s, age:%d" %(self.name, self.age)

    @classmethod
    def getCount(cls):
        return cls.count #Dog.count也可

    @staticmethod
    def getCountS():
        return Dog.count #Dog.count也可
```

In [78]:
```
billy = Dog("Billy",5)
print billy
```

```
姓名:Billy, age:5
```

In [79]:
```
billy.getCount()
```

Out[79]:
```
1
```

In [80]:
```
willy = Dog("Willy",3)
willy.getCount()
```

Out[80]:
```
2
```

In [81]:
```
Dog.count
```

Out[81]:
```
2
```

In [82]:
```
Dog.getCount()
```
Out[82]:
```
2
```

In [1]:
```
class MyMath:
    PI = 3.14  #class variable 共享一個存放位置
    LUCKY_NUMBER = 7
    @classmethod
    def area( cls ,r ):
        return cls.PI * r*r
```

In [3]:
```
print (MyMath.pi)
print (MyMath.area(10))
```
```
3.14
314.0
```

In [ ]:
```

```

In [34]:
```python

# Python program to demonstrate
# use of class method and static method.
from datetime import date

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

    # a class method to create a Person object by birth year.
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # a static method to check if a Person is adult or not.
    @staticmethod
    def isAdult(age):
        return age > 18

person1 = Person('John', 12)
person2 = Person.fromBirthYear('Bill', 1964)

print( person1.age )
print( person2.age )

# print the result
print( Person.isAdult(22) )
```

```
12
54
True
```

In [ ]:
```python
```

1.4 私有變數如何定義(private variable)？

__x = 0 # 私有變數 y = 0 # 公開變數

In [ ]:

```

```

1.5 繼承inheritance與取代override

In [89]:

```python
class A:
    def foo(self):
        print('hello')

class B(A):
    def foo2(self):
        A.foo(self)
        #super(B,self).foo()
```

In [90]:

```python
b = B()
b.foo2()
```

```
hello
```

In [ ]:

```

```

In [91]:

```python
class A(object):
    def foo(self):
        print('hello')

class B(A):
    def foo2(self):
        #A.foo(self)
        super(B,self).foo()
```

In [92]:

```python
b = B()
b.foo2()
```

```
hello
```

In [ ]:

In [ ]:

In [ ]:

In [174]:
```
class A(object):
    def foo(self):
        print('hello')

class B(A):
    def foo(self): #取代
        print('hello2')
        #A.foo(self)
        #super(B,self).foo()
```

In [175]:
```
b = B()
b.foo()
```

```
hello2
```

In [176]:
```
b.isAdult(20)
```

Out[176]:
```
True
```

In [ ]:

In [66]:
```
class WithClass ():
    def __init__(self):
        self.value = "Bob"
    def my_func(self):
        print(self.value)

class WithoutClass():
    value = "Bob"
    def my_func(self):
        print(self.value) #self.value竟然可以操作到 class variable??
```

In [29]:
```
c1 = WithClass()
```

In [31]:
```
c1.my_func()
```

```
Bob
```

In [ ]:
```
```

In [32]:
```
c2 = WithoutClass()
```

In [39]:
```
c2.my_func()
```

```
Bob2
```

In [34]:
```
c2.value
```

Out[34]:
```
'Bob'
```

In [35]:
```
c2.value="Bob2"
```

In [37]:
```
c3 = WithoutClass()
```

In [38]:
```
c3.my_func()
```
```
Bob
```

In [40]:
```
WithoutClass.value
```
Out[40]:
```
'Bob'
```

In [42]:
```
c3.y=6
```

In [ ]:

In [67]:
```
class A(object):

  label="Amazing"

  def __init__(self,d):
      self.data=d
      #self.label="GO"#????

  def say(self):
      #self.label="Amazing2" #注意self.label是class variable
      print("%s %s!"%(self.label,self.data)) #注意self.label是class variable

class B(A):
  label="Bold"  # overrides A.label

A(5).say()      # Amazing 5!
B(3).say()      # Bold 3!
```
```
Amazing2 5!
Amazing2 3!
```

In [68]:
```
a=A(5)
```

```
In [69]: a.say()

Amazing2 5!

In [70]: b=B(3)

In [71]: b.say()

Amazing2 3!

In [72]: a.say()

Amazing2 5!

In [73]: a.label="HI"

In [ ]:

In [ ]:

In [74]: a.say()

Amazing2 5!

In [75]: b.say()

Amazing2 3!

In [76]: c= A(7)

In [77]: c.say()

Amazing2 7!
```

In [ ]:

```

```

In [ ]:

```

```

In [57]:

```python
class MyClass:
    static_elem = 123

    def __init__(self):
        self.object_elem = 456

c1 = MyClass()
c2 = MyClass()
```

In [58]:

```python
# Initial values of both elements
c1.static_elem, c1.object_elem
```

Out[58]:

```
(123, 456)
```

In [59]:

```python
c2.static_elem, c2.object_elem
```

Out[59]:

```
(123, 456)
```

In [60]:

```python
# Let's try changing the static element
MyClass.static_elem = 999
```

In [61]:

```python
c1.static_elem, c1.object_elem
```

Out[61]:

```
(999, 456)
```

In [62]:

```python
c2.static_elem, c2.object_elem
```

Out[62]:

```
(999, 456)
```

In [63]:
```
# Now, let's try changing the object element
c1.object_elem = 888

c1.static_elem, c1.object_elem
```

Out[63]:
```
(999, 888)
```

In [65]:
```
c2.static_elem, c2.object_elem
```

Out[65]:
```
(999, 456)
```

In [ ]: