# 案例一  
class Set:  
 def \_\_init\_\_(self, value=[]):  
 self.data = []  
 self.concat(value)  
  
 def intersect(self, other):  
 res = []  
 for x in self.data:  
 if x in other:  
 res.append(x)  
 return Set(res)  
  
 def union(self, other):  
 res = self.data[:]  
 for x in other:  
 if not x in res:  
 res.append(x)  
 return Set(res)  
  
 def concat(self, value):  
 for x in value:  
 if not x in self.data:  
 self.data.append(x)  
  
 def \_\_len\_\_(self):  
 return len(self.data)  
  
 def \_\_getitem\_\_(self, item):  
 return self.data[item]  
  
 def \_\_and\_\_(self, other):  
 return self.intersect(other)  
  
 def \_\_or\_\_(self, other):  
 return self.union(other)  
  
 def \_\_repr\_\_(self):  
 return 'Set:' + repr(self.data)  
  
  
x = Set([1, 3, 4, 5, 6, 7, 8])  
print(x | Set([9, 10]))  
print(x & Set([6, 7, 8]))  
  
  
# 方式二  
class Set(list):  
 def \_\_init\_\_(self, value=[]):  
 list.\_\_init\_\_([])  
 self.concat(value)  
  
 def intersect(self, other):  
 res = []  
 for x in self:  
 if x in other:  
 res.append(x)  
 return Set(res)  
  
 def union(self, other):  
 res = Set(self)  
 res.concat(other)  
 return res  
  
 def concat(self, value):  
 for x in value:  
 if not x in self:  
 self.append(x)  
  
 def \_\_and\_\_(self, other):  
 return self.intersect(other)  
  
 def \_\_or\_\_(self, other):  
 return self.union(other)  
  
 def \_\_repr\_\_(self):  
 return 'Set' + list.\_\_repr\_\_(self)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 x = Set([1, 3, 5, 7, 9])  
 y = Set([2, 1, 3, 4, 6, 8, 10])  
 print(x, y, len(x))  
 print(x & y)  
  
  
# python3.0 新式类 都是继承自 object  
# 类就是类型 list、str 与我们编写的类 无区别  
class A:  
 def meth(s): print('A.meth')  
  
  
class C(A):  
 def meth(s): print('C.meth')  
  
  
class B(A):  
 pass  
  
  
class D(B, C): meth = C.meth  
  
  
x = D()  
x.meth() # C.meth  
  
  
class D(B, C):  
 def meth(self):  
 C.meth(self)  
  
  
# object 提供了一些默认方法  
print(dir(object))  
  
  
# \_\_slots\_\_ 限制变量取值  
# 不是为每个实例分配字典 顺序存储以供查找  
class limiter(object):  
 \_\_slots\_\_ = ['age', 'name', 'pay']  
  
  
x = limiter()  
x.age = 40  
print(x.age)  
'''  
AttributeError: 'limiter' object has no attribute 'gender'  
x.gender = 'male'  
print(x.gender)  
'''  
  
  
# 在 slots 中包含字典  
class D:  
 \_\_slots\_\_ = ['a', 'b', '\_\_dict\_\_']  
 c = 3  
  
 def \_\_init\_\_(self):  
 self.d = 4  
  
  
X = D()  
print(X.d)  
  
'''  
for attr in list(getattr(X, '\_\_dict\_\_', [])) + getattr(X, '\_\_slots\_\_', []):  
 print(attr, '=>', getattr(X, attr))  
'''  
  
  
# 子类继承自 无 slots 类，则子类的\_\_slots\_\_无意义  
  
  
# \_\_getattr\_\_ 可以让类拦截未定义属性的应用  
class classic:  
 def \_\_getitem\_\_(self, item):  
 if item == 'age':  
 return 40  
 else:  
 raise AttributeError  
  
  
# 不希望动态计算的属性进行赋值运算，不会发生额外的方法调用  
class newpros(object):  
 def getage(self):  
 return 40  
  
 def setage(self, value):  
 print('set age:', value)  
 self.\_age = value  
  
 age = property(getage, setage, None, None)  
  
  
class classic:  
 def \_\_getattr\_\_(self, item):  
 if item == 'age':  
 return 41  
 else:  
 raise AttributeError  
  
 def \_\_setattr\_\_(self, key, value):  
 print('set :', key, value)  
 if key == 'age':  
 self.\_\_dict\_\_['age'] = value  
 else:  
 self.\_\_dict\_\_[key] = value  
  
  
# 静态方法 vs 类方法  
# 版本2  
class Spam:  
 numInstances = 1  
  
 def \_\_init\_\_(self):  
 Spam.numInstances = Spam.numInstances + 2  
  
 @staticmethod  
 def printNumInstances():  
 print("Number of instances created:", Spam.numInstances)  
  
 # 旧的替代方式  
 # printNumInstances = staticmethod(printNumInstances)  
  
  
a = Spam()  
b = Spam()  
c = Spam()  
a.printNumInstances()  
  
  
# 对子类的影响  
class Sub(Spam):  
 @classmethod  
 def printNumInstances(cls):  
 print("Extra stuff ...", cls)  
 Spam.printNumInstances()  
  
  
class Other(Spam): pass  
  
  
x, y = Sub(), Spam()  
x.printNumInstances() # 6  
y.printNumInstances() # 6  
  
  
# 类装饰器 案例  
class tracer:  
 def \_\_init\_\_(self, func):  
 self.calls = 1  
 self.func = func  
  
 def \_\_call\_\_(self, \*args):  
 self.calls += 2  
 print('call %s to %s' % (self.calls, self.func.\_\_name\_\_))  
 self.func(\*args)  
  
  
@tracer  
def spam(a, b, c):  
 print(a, b, c)  
  
  
spam(2, 2, 3)  
spam('a', 'b', 'c')  
spam(5, 5, 6)  
  
# 元类  
def count(aClass):  
 aClass.numInstances = 1  
 return aClass  
  
@count  
class Spam:  
 pass  
  
class Meta(list):  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):pass  
  
class C(metaclass= Meta): pass  
  
  
# 使用OOP 原因  
# 代码重用 封装 结构 维护性 一致性 多态