Python不同版本的类

Python2.2之前类是没有共同的祖先的，之后，引入object类，它是所有类的共同祖先类object。
Python2中为了兼容，分为古典类（旧式类）和新式类。
Python3中全部都是新式类。
新式类都是继承自object的，新式类可以使用super。

```
# 以下代码在Python2.x中运行
# 古典类（旧式类）
class A: pass


# 新式类
class B(object): pass


print(dir(A))
print(dir(B))
print(A.__bases__)
print(B.__bases__)

# 古典类
a = A()
print(a.__class__)
print(type(a))  # <type 'instance'>

# 新式类
b = B()
print(b.__class__)
print(type(b))
```

多继承

OCP原则：多用“继承”、少修改
继承的用途：增强基类、实现多态

多态
在面向对象中，父类、子类通过继承联系在一起，如果可以通过一套方法，就可以实现不同表现，就是多态。

一个类继承自多个类就是多继承，它将具有多个类的特征。

多继承弊端

多继承很好的模拟了世界，因为事物很少是单一继承，但是舍弃简单，必然引入复杂性，带来了冲突。

如同一个孩子继承了来自父母双方的特征。那么到底眼睛像爸爸还是妈妈呢？孩子究竟该像谁多一点呢？

多继承的实现会导致编译器设计的复杂度增加，所以现在很多语言也舍弃了类的多继承。

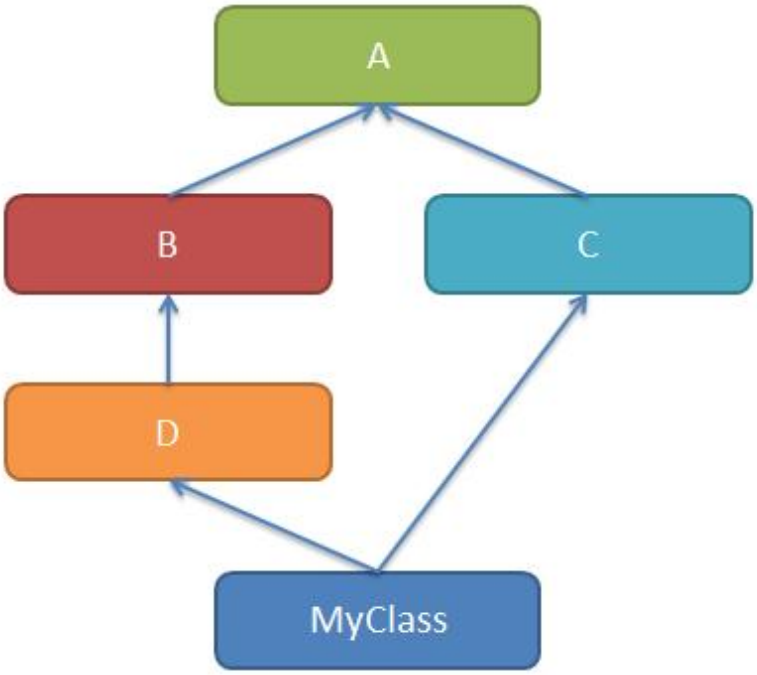
C++支持多继承；Java舍弃了多继承。
Java中，一个类可以实现多个接口，一个接口也可以继承多个接口。Java的接口很纯粹，只是方法的声明，继承者必须实现这些方法，就具有了这些能力，就能干什么。

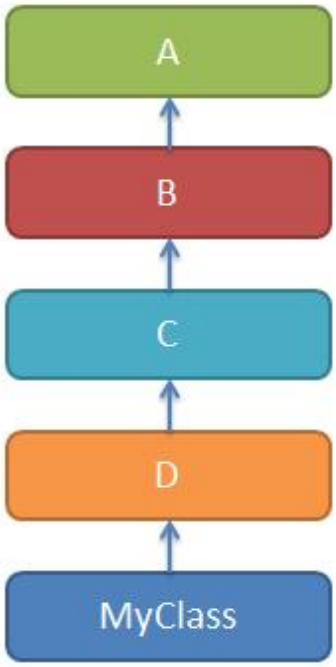
多继承可能会带来二义性，例如，猫和狗都继承自动物类，现在如果一个类多继承了猫和狗类，猫和狗都有shout方法，子类究竟继承谁的shout呢？

解决方案
实现多继承的语言，要解决二义性，深度优先或者广度优先。

Python多继承实现

```
class ClassName(基类列表):
    类体
```





左图是多继承，右图是单一继承

多继承带来路径选择问题，究竟继承哪个父类的特征呢

Python使用MRO（method resolution order）解决基类搜索顺序问题。

- 历史原因，MRO有三个搜索算法：
  - 经典算法，按照定义从左到右，深度优先策略。2.2之前
    左图的MRO是MyClass,D,B,A,C,A
  - 新式类算法，经典算法的升级，重复的只保留最后一个。2.2
    左图的MRO是MyClass,D,B,C,A,object
  - C3算法，在类被创建出来的时候，就计算出一个MRO有序列表。2.3之后，Python3唯一支持的算法
    左图中的MRO是MyClass,D,B,C,A,object的列表
    C3算法解决多继承的二义性

多继承的缺点

当类很多，继承复杂的情况下，继承路径太多，很难说清什么样的继承路径。
Python语法是允许多继承，但Python代码是解释执行，只有执行到的时候，才发现错误。

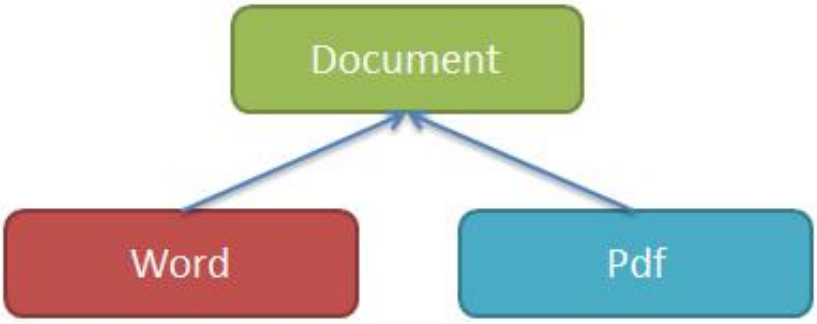
团队协作开发，如果引入多继承，那代码将不可控。

不管编程语言是否支持多继承，都应当避免多继承。

Python的面向对象，我们看到的太灵活了，太开放了，所以要团队守规矩。

Mixin

类有下面的继承关系



文档Document类是其他所有文档类的抽象基类；
Word、Pdf类是Document的子类。

需求：为Document子类提供打印能力
思路：
1、在Document中提供print方法

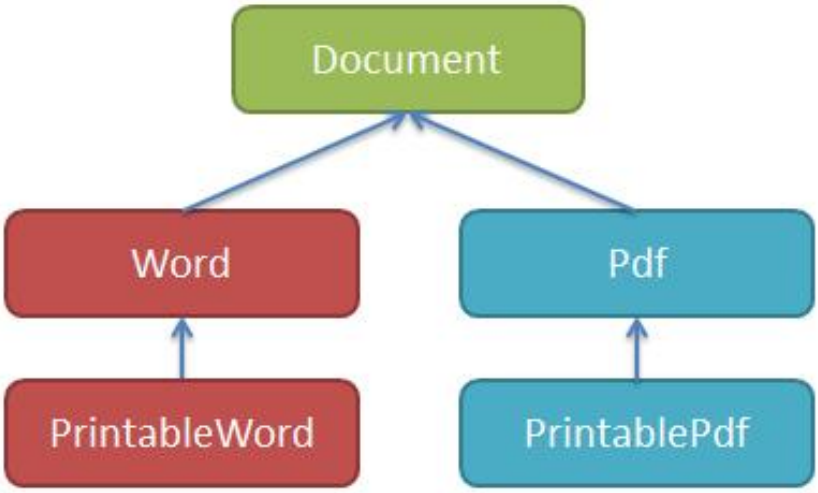
```
class Document:
    def __init__(self, content):
        self.content = content

    def print(self):
        raise NotImplementedError()

class Word(Document): pass
class Pdf(Document): pass
```

基类提供的方法不应该具体实现，因为它未必适合子类的打印，子类中需要覆盖重写。
print算是一种能力 -- 打印功能，不是所有的Document的子类都需要的，所有，从这个角度出发，有点问题。

2、需要打印的子类上增加
如果在现有子类上直接增加，违反了OCP的原则，所以应该继承后增加。因此有下图




```
class Printable:
    def print(self):
        print(self.content)

class Document: # 第三方库，不允许修改
    def __init__(self, content):
        self.content = content
class Word(Document): pass # 第三方库，不允许修改
class Pdf(Document): pass # 第三方库，不允许修改

class PrintableWord(Printable, Word): pass
print(PrintableWord.__dict__)
print(PrintableWord.mro())

pw = PrintableWord('test string')
pw.print()
```

看似不错，如果需要还要提供其他能力，如何继承？
应用于网络，文档应该具备序列化的能力，类上就应该实现序列化。
可序列化还可能分为使用pickle、json、messagepack等。

这个时候发现，类可能太多了，继承的方式不是很好了。
功能太多，A类需要某几样功能，B类需要另几样功能，很繁琐。

3、装饰器
用装饰器增强一个类，把功能给类附加上去，那个类需要，就装饰它

```
def printable(cls):
    def _print(self):
        print(self.content, '装饰器')
```

```
    cls.print = _print
    return cls

class Document: # 第三方库，不允许修改
    def __init__(self, content):
        self.content = content
class Word(Document): pass # 第三方库，不允许修改
class Pdf(Document): pass # 第三方库，不允许修改

@printable # 先继承，后装饰
class PrintableWord(Word): pass
print(PrintableWord.__dict__)
print(PrintableWord.mro())

pw = PrintableWord('test string')
pw.print()

@printable
class PrintablePdf(Word): pass
```

优点：
简单方便，在需要的地方动态增加，直接使用装饰器

4、Mixin

先看代码

```
class Document: # 第三方库，不允许修改
    def __init__(self, content):
        self.content = content


class Word(Document): pass # 第三方库，不允许修改
class Pdf(Document): pass # 第三方库，不允许修改

class PrintableMixin:
    def print(self):
        print(self.content, 'Mixin')

class PrintableWord(PrintableMixin, Word): pass
print(PrintableWord.__dict__)
```

```
print(PrintableWord.mro())

def printable(cls):
    def _print(self):
        print(self.content, '装饰器')
    cls.print = _print
    return cls

@printable
class PrintablePdf(Word): pass
print(PrintablePdf.__dict__)
print(PrintablePdf.mro())
```

Mixin就是其它类混合进来，同时带来了类的属性和方法。
这里看来Mixin类和装饰器效果一样，也没有什么特别的。但是Mixin是类，就可以继承。

```
class Document: # 第三方库，不允许修改
    def __init__(self, content):
        self.content = content


class Word(Document): pass # 第三方库，不允许修改
class Pdf(Document): pass # 第三方库，不允许修改

class PrintableMixin:
    def print(self):
        print(self.content, 'Mixin')

class PrintableWord(PrintableMixin, Word): pass
print(PrintableWord.__dict__)
print(PrintableWord.mro())

pw = PrintableWord('test string')
pw.print()

class SuperPrintableMixin(PrintableMixin):
    def print(self):
        print('~' * 20)  # 打印增强
        super().print()
        print('~' * 20)  # 打印增强
```

```
# PrintableMixin类的继承
class SuperPrintablePdf(SuperPrintableMixin, Pdf): pass

print(SuperPrintablePdf.__dict__)
print(SuperPrintablePdf.mro())

spp = SuperPrintablePdf('super print pdf')
spp.print()
```

Mixin类

Mixin本质上就是多继承实现的。
Mixin体现的是一种组合的设计模式。

在面向对象的设计中，一个复杂的类，往往需要很多功能，而这些功能有来自不同的类提供，这就需要很多的类组合在一起。
从设计模式的角度来说，多组合，少继承。

Mixin类的使用原则

- Mixin类中不应该显式的出现__init__初始化方法
- Mixin类通常不能独立工作，因为它是准备混入别的类中的部分功能实现
- Mixin类的祖先类也应是Mixin类

使用时，Mixin类通常在继承列表的第一个位置，例如class PrintableWord(PrintableMixin, Word): pass

Mixin类和装饰器
这两种方式都可以使用，看个人喜好。
如果还需要继承就得使用Mixin类的方式。

练习

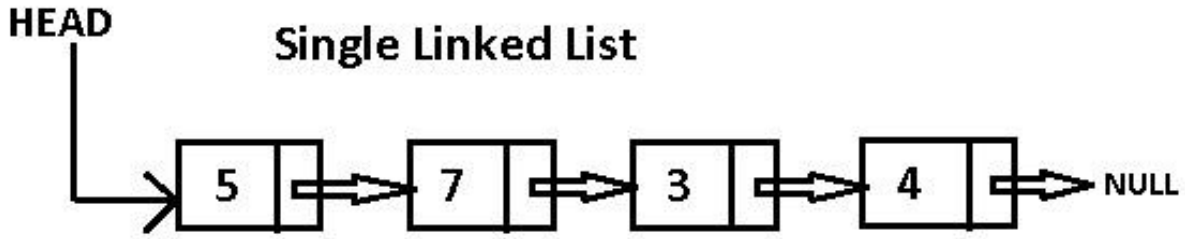
1、Shape基类，要求所有子类都必须提供面积的计算，子类有三角形、矩形、圆。
2、上题圆类的数据可序列化

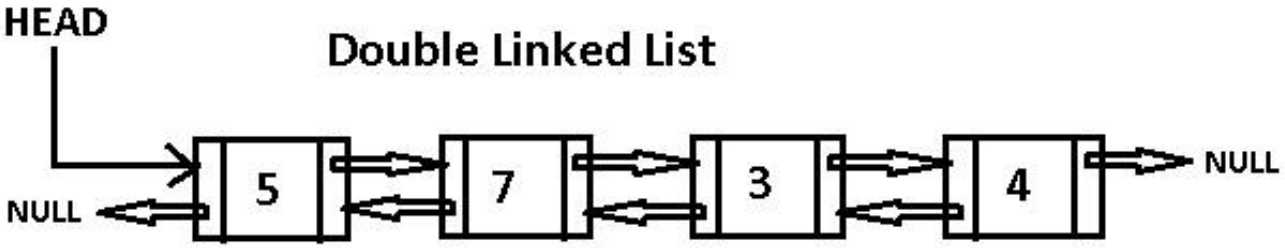
作业

用面向对象实现LinkedList链表

单向链表实现append、iternodes方法

双向链表实现append、pop、insert、remove、iternodes方法





参考

1、Shape基类，要求所有子类都必须提供面积的计算，子类有三角形、矩形、圆。

```
import math


class Shape:
    @property
    def area(self):
        raise NotImplementedError('基类未实现')


class Triangle(Shape):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c


    @property
    def area(self):
        p = (self.a+self.b+self.c)/2
        return math.sqrt(p*(p-self.a)*(p-self.b)*(p-self.c))
```

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height


class Circle(Shape):
    def __init__(self, radius):
        self.d = radius * 2

    @property
    def area(self):
        return math.pi * self.d * self.d * 0.25


shapes = [Triangle(3,4,5), Rectangle(3,4), Circle(4)]
for s in shapes:
    print('The area of {} = {}'.format(s.__class__.__name__,s.area))
```

2、圆类的数据可序列化

```
import json
import msgpack

class SerializableMixin:
    def dumps(self, t='json'):
        if t == 'json':
            return json.dumps(self.__dict__)
        elif t == 'msgpack':
            return msgpack.packb(self.__dict__)
        else:
            raise NotImplementedError('没有实现的序列化')

class SerializableCircleMixin(SerializableMixin, Circle):
    pass
```

```
scm = SerializableCircleMixin(4)
print(scm.area)
s = scm.dumps('msgpack')
print(s)
```