《Python 教程》

- 讲师: 魏明择
- 时间: 2019

目录

[TOC]

面向对象编程 Object-Oriented Programming

- 什么是对象
    - 对象是指现实中的物体或实体
- 什么是面向对象
    - 把一切看成对象（实例），用各种对象之间的关系来描述事务。
- 对象都有什么特征
    - 对象有很多属性(名词)
        - 姓名, 年龄, 性别,
    - 对象有很多行为(动作,动词)
        - 学习,吃饭,睡觉,踢球, 工作
- 什么是类:
    - 拥有相同属性和行为的对象分为一组,即为一个类
    - 类是用来描述对象的工具,用类可以创建此类的对象(实例)
- 面向对象 示意

```
车(类)  ------>> BYD  E6(京A.88888) 实例, 对象
       \
        \.--->> BMW  X5(京B.00000) 实例(对象)

狗(类)  ------>> 小京巴(户籍号:000001)
       \
        \.--->> 导盲犬（户籍号:000002）

int(类)  ---->> 100 (对象)
        \
         \.-->> 200 (对象)
```

class语句

- 类的创建语句语法:

```
class 类名（继承列表):
    "类文档字符串"
    实例方法(类内的函数method) 定义
    类变量(class variable) 定义
    类方法(@classmethod) 定义
    静态方法(@staticmethod) 定义
```

> 注: 继承列表可以省略，省略继承列表表示类继承自object.

- 类的作用:
  - 创建一个类
  - 类用于描述对象的行为和属性
  - 类用于创建此类的一个或多个对象（实例）
- 支持面向对象的语言:
  - Python / C++/ C# / Java / Swift
- 面向过程的语言
  - C / Pascal / Delphi /Visual Basic
- 类的创建示例

```
# file : class.py
class Dog:  # 定义一个Dog类
    pass
```

- 类的创建的说明:
  - 类名必须为标识符(与变量的命名相同,建议首字母大写)
  - 类名实质上就是变量，它绑定一个类
- 类和对象

类	对象（也叫实例）
class	object (instance)

构造函数

- 构造函数调用表达式

```
类名([创建传参列表])
```

- 构造函数作用

    - 创建这个类的实例对象，并返回此实例对象的引用关系

- 用类创建对象示例：

```
# file : constructor.py
class Dog:
    pass

# 创建第一个实例：
dog1 = Dog()
print(id(dog1))  # 打印这个对象的ID
# 创建第二个实例对象
dog2 = Dog()  # dog2 绑定一个Dog类型的对象
print(id(dog2))

lst1 = list()
print(id(lst1))
lst2 = list()
print(id(lst2))
```

- 实例说明

    - 实例有自己的作用域和名字空间,可以为该实例添加实例变量（也叫属性)
    - 实例可以调用类方法和实例方法
    - 实例可以访问类变量和实例变量

实例方法(instance method)

```
```
class 类名(继承列表):
    def 实例方法名(self, 参数1, 参数2, ...):
        "文档字符串"
        语句块
```
```

- 实例方法的作用

    - 用于描述一个对象的行为,让此类型的全部对象都拥有相同的行为

- 实例方法说明

    - 实例方法的实质是函数，是定义在类内的函数
    - 实例方法至少有一个形参，第一个形参绑定调用这个方法的实例,一般命名为"self"
    - 实例方法名是类属性

- 实例方法的调用语法

```

```

```
实例.实例方法名(调用传参)
# 或
类名.实例方法名(实例, 调用传参)
```

- 带有实例方法的简单的Dog类

```
# file: instance_method.py
class Dog:
    """这是一个种小动物的定义
    这种动物是狗(犬)类，用于创建各种各样的小狗
    """
    def eat(self, food):
        '''此方法用来描述小狗吃东西的行为'''
        print("小狗正在吃", food)
    def sleep(self, hour):
        print("小狗睡了", hour, "小时!")
    def play(self, obj):
        print("小狗正在玩", obj)
```

dog1 = Dog() dog1.eat("骨头") dog1.sleep(1) dog1.play('球')

```
dog2 = Dog()
dog2.eat("窝头")
dog2.sleep(2)
dog2.play('飞盘')

>>> help(Dog) # 可以看到Dog类的文档信息
```

实例属性 attribute（也叫实例变量)

- 每个实例可以有自己的变量，称为实例变量(也叫属性)
- 属性的使用语法

```
实例.属性名
```

- 属性的赋值规则
  - 首次为属性赋值则创建此属性.
  - 再次为属性赋值则改变属性的绑定关系.
- 作用

    - 记录每个对象自身的数据

- 属性使用示例:

```
# file : attribute.py
class Dog:
    def eat(self, food):
         print(self.color, '的', self.kinds, '正在吃', food)
    pass
# 创建一个实例:
dog1 = Dog()
dog1.kinds = "京巴"  # 添加属性
dog1.color = "白色"
dog1.color = "黄色"  # 改变属性的绑定关系
print(dog1.color, '的', dog1.kinds)

dog2 = Dog()
dog2.kinds = "藏獒"
dog2.color = "棕色"
print(dog2.color, '的', dog2.kinds)
```

- 实例方法和实例属性(实例变量)结合在一起用:

```
class Dog:
    def eat(self, food):
        print(self.color, '的',
              self.kinds, '正在吃', food)


# 创建第一个对象
dog1 = Dog()
dog1.kinds = '京巴'  # 添加属性kinds
dog1.color = '白色'  # 添加属性color
# print(dog1.color, '的', dog1.kinds)  # 访问属性
dog1.eat("骨头")

dog2 = Dog()
dog2.kinds = '牧羊犬'
dog2.color = '灰色'
# print(dog2.color, '的', dog2.kinds)  # 访问属性
dog2.eat('包子')
```

- 练习:

    - 定义一个'人'类:

```
class Human:
    def set_info(self, name, age, address='不详'):
```

```
            '''此方法用来给人对象添加'姓名'、'年龄'和 '家庭住址' 属性
            # 此处自己实现
        def show_info(self):
            '''此处显示此人的信息'''
            # 此处自己实现

    # 如:
    s1 = Human()
    s1.set_info('小张', 20, '北京市朝阳区')
    s2 = Human()
    s2.set_info('小李', 18)``
    s1.show_info()  # 小张 今年 20 岁, 家庭住址: 北京市朝阳区
    s2.show_info()  # 小李 今年 18 岁, 家庭住址: 不详
```

- day17 pm

del 语句删除实例属性

```
```
del 对象.实例属性名
```
```

- del 语句 最终总结:
    1. 删除变量
    2. 删除列表中的元素
    3. 删除字典中的键值对
    4. 删除对象的属性(也叫实例变量)

初始化方法

- 初始化方法的作用:
    - 对新创建的对象添加属性
- 初始化方法的语法格式:

```
class 类名(继承列表):
    def __init__(self[, 形参列表]):
        语句块
# [] 代表其中的内容可省略
```

- 初始化方法的说明:
    - 初始化方法名必须为`__init__` 不可改变
    - 初始化方法会在构造函数创建实例后自动调用,且将实例自身通过第一个参数self传入`__init__`方法
    - 构造函数的实参将通过`__init__`方法的 参数列表 传入到 `__init__`方法中

    - 初始化方法内如果需要return语句返回，则必须返回None

- 初始化方法示例：

```
# file : init_method.py
class Car:
    def __init__(self, c, b, m):
        self.color = c  # 颜色
        self.brand = b  # 品牌
        self.model = m  # 型号
    def run(self, speed):
        print(self.color, "的", self.brand, self.model, "正在以",
speed, "公里/小时的速度行驶")
    def change_color(self, c):
        self.color = c

a4 = Car("红色", "奥迪", "A4")
a4.run(199)
a4.change_color("黑色")
a4.run(230)
```

析构方法

- 语法格式

```
class XXX:
    def __del__(self):
        ...
```

- 说明:

    - 析构函数在对象被销毁时被自动调用
    - python语言建议不要在对象销毁时做任何事情,因为销毁的时间难以确定

- 析构方法示例

```
# file : del_method.py
class Car:
    def __init__(self, name):
        self.name = name
        print("汽车", name, "对象已创建!")
    def __del__(self):
        print("汽车", self.name, "对象已销毁")

c1 = Car("BYD E6")
del c1
```

预置实例属性

- `__dict__` 属性
  - 用于绑定一个存储此实例自身属性的字典
  - `__dict__` 属性示例：

```
class Dog:
    pass

dog1 = Dog()
print(dog1.__dict__)
dog1.kinds = "京巴"
print(dog1.__dict__)
dog1.color = "白色"
print(dog1.__dict__)
```

- `__class__` 属性
  - `__class__` 属性绑定创建此实例的类
  - `__class__` 属性作用
    - 可以借助于此属性来访问创建此实例的类
  - 示例：

```
class Dog:
    pass

dog1 = Dog()
print(dog1.__class__)
dog2 = dog1.__class__()
print(dog2.__class__)
```

用于类的函数

函数	说明
isinstance(obj, class_or_tuple)	返回这个对象obj 是否是 某个类的对象,或者某些类中的一个类的对象,如果是返回True,否则返回False
type(obj)	返回对象的类型

- day18

类属性

- 类属性是类的属性，此属性属于类，不属于此类的实例

- 作用:
    - 通常用来存储该类创建的对象的共有属性
- 类属性说明
    - 类属性,可以通过该类直接访问
    - 类属性,可以通过类的实例直接访问
    - 类属性可以通过此类的对象的__class__属性间接访问
- 类属性示例

```
# file : class_attribute.py
class Human:
    total_count = 0  # 创建类属性
    def __init__(self, name):
        self.name = name
        self.__class__.total_count += 1

print(Human.total_count)
h1 = Human("小张")
print(h1.total_count)
```

类的文档字符串

- 类内第一个没有赋值给任何变量的字符串为类的文档字符串
- 类的文档字符串可以通过help函数查看
- 类的文档字符串可以用 类的 __doc__ 属性访问
- 类的文档字符串示例:

```
class Dog:
    """Dog类的文档字符串
    此文档字符串用于描述类的信息"""

>>> help(Dog)  # 查看上述文档字符串
```

类的 __slots__ 列表

- 作用:
    - 限定一个类创建的实例只能有固定的实例属性(实例变量)
    - 不允许对象添加列表以外的实例属性(实例变量)
    - 防止用户因错写属性的名称而发生程序错误。
- 说明:

    - __slots__ 属性是一个列表，列表的值是字符串
    - 含有__slots__属性的类所创建的实例对象没有__dict__属性, 即此实例不用字典来存储对象的实例属性（实例变量）

- 示例

```
class Human:
    __slots__ = ["name", "age"]
    def __init__(self, name, age):
        self.name, self.age = name, age
    def show_info(self):
        print(self.name, self.age)

h1 = Human("Tarena", 15)
h1.show_info()
h1.Age = 16
h1.show_info()
print(h1.age)
h1.Age = 100  # 出错，s1对象没有Age属性
```

类方法 @classmethod

- 类方法是用于描述类的行为的方法，类方法属于类，不属于该类创建的对象
- 说明
    - 类方法需要使用@classmethod装饰器定义
    - 类方法至少有一个形参，第一个形参用于绑定类，约定写为'cls'
    - 类和该类的实例都可以调用类方法
    - 类方法不能访问此类创建的对象的实例属性
- 类方法示例

```
class A:
    v = 0
    @classmethod
    def set_v(cls, value):
        cls.v = value
    @classmethod
    def get_v(cls):
        return cls.v
print(A.get_v())
A.set_v(100)
print(A.get_v)
a = A()
print(A.get_v())
```

静态方法 @staticmethod

- 静态方法是定义在类的内部函数，此函数的作用域是类的内部
- 说明
    - 静态方法需要使用@staticmethod装饰器定义
    - 静态方法与普通函数定义相同，不需要传入self实例参数和cls类参数
    - 静态方法只能凭借该类或类创建的实例调用
    - 静态方法不能访问类属性和实例属性
- 静态方法示例

```
class A:
    @staticmethod
    def myadd(a, b):
        return a+b

print(A.myadd(100, 200))
a = A()
print(a.myadd(300, 400))
```

- day18 pm

继承(inheritance) 和 派生(derived)

- 什么是继承 / 派生
    - 继承是从已有的类中派生出新的类，新类具有原类的数据属性和行为，并能扩展新的能力。
    - 派生类就是从一个已有类中衍生出新类，在新的类上可以添加新的属性和行为
- 为什么继承/派生
    - 继承的目的是延续旧的类的功能
    - 派生的目地是在旧类的基础上添加新的功能
- 继承/派生的作用
    - 用继承派生机制，可以将一些共有功能加在基类中。实现代码的共享。
    - 在不改变基类的代码的基础上改变原有类的功能
- 继承/派生名词：
    - 基类(base class)/超类(super class)/父类(father class)
    - 派生类(derived class)/子类(child class)

单继承

- 单继承的语法:

```
class 类名(基类名):
    语句块
```

- 单继承说明:
  - 单继承是指派生类由一个基类衍生出来的
- 单继承的示例1:

```
class Human:  # 人类的共性
    def say(self, what):  # 说话
        print("说: ", what)
    def walk(self, distance):  # 走路
        print("走了", distance, "公里")

class Student(Human):
    def study(self, subject):  # 学习
        print("学习:", subject)

class Teacher(Human):
    def teach(self, language):
        print("教:", language)

h1 = Human()
h1.say("天气真好!")
h1.walk(5)

s1 = Student()
s1.walk(4)
s1.say("感觉有点累")
s1.study("python")

t1 = Teacher()
t1.teach("面向对象")
t1.walk(6)
t1.say("一会吃点什么好呢")
```py
```

- 继承说明:
  - Python3任何类都直接或间接的继承自object类
  - object类是一切类的超类
- 类的 `__base__` 属性(在继承之后讲)
  - `__base__`属性用来记录此类的基类
  - 详见 : `>>> help(__builtins__)`

覆盖 override
```

- 覆盖是指在有继承关系的类中，子类中实现了与基类同名的方法,在子类的实例调用该方法时，实际调用的是子类中的覆盖版本,这种现象叫覆盖
- 作用:
  - 实现和父类同名，但功能不同的方法
- 覆盖示例

```
# file : override.py
class A:
    def work(self):
        print("A.work 被调用!")

class B(A):
    '''B类继承自A类'''
    def work(self):
        print("B.work 被调用!!!")
    pass

b = B()
b.work()  # 请问调用谁？ B

a = A()
a.work()  # 请问调用谁？ A
```

子类显式调用基类的覆盖方法

- 问题?
  - 当在覆盖发生时，子类对象能否调用父类中的方法?
- 子类对象显式调用基类方法的方式:
  - 基类名.方法名(实例，实际调用参数, ...)

super函数

函数	说明
super(cls, obj)	返回绑定超类的实例(要求obj必须为cls类型的实例)
super()	返回绑定超类的实例,等同于:super(**class**, 实例方法的第一个参数)，必须用在方法内调用

- super函数 作用
  - 借助super() 返回的实例间接调用其父类的覆盖方法
- super函数 示例:

```
class A(object):
    def work(self):
        print("A类的work被调用")

class B(A):
    def work(self):
        print("B类的work被调用")
    def super_work(self):
        self.work(). # 调用自身的work 方法
        # 以下两种方式来调用父类的方法
        super(B, self).work()
        super().work()

b = B()
# b.work() # 调用子类方法，但没办法调用超类方法
super(B, b).work()  # 调用超类方法
b.super_work()
```

- 显式调用基类的初始化方法:

```
# 当子类中实现了__init__ 方法，基类的初始化方法并不会被调用.
def __init__(self, ....):
    ....
```

- 显示调用基类的初始化方法示例

```
# file : super_init.py
class Human:
    def __init__(self, n, a):
        self.name, self.age = n, a
    def infos(self):
        print("姓名:", self.name)
        print("年龄:", self.age)

class Student(Human):
    def __init__(self, n, a, s=0):
        super().__init__(n,a)
        self.score = s
    def infos(self):
        super().infos()
        print("成绩:", self.score)

s = Student("wei", 35, 100)
```

- day19

用于类的函数

函数	说明
issubclass(cls, class_or_tuple)	判断一个类是否继承自其它的类,如果此类cls是class 或 tuple中的一个派生子类则返回True,否则返回False

- issubclass 示例:

```
# file : issubclass.py
class A: pass
class B(A): pass
class C(B): pass

issubclass(C, (A, B))  # True
issubclass(C, (int, str))  # False
issubclass(D, (int, A))    # True
```

封装 enclosure

- 封装是指隐藏类的实现细节，让使用者不用关心这些细节;
- 封装的目的是让使用者通过尽可能少的方法(或属性)操作对象
- Python的封装是假的（模拟的）封装
- 私有属性和方法
    - python类中以双下划线(__)开头，不以双下划线结尾的标识符为私有成员,私有成员只能使用方法来进行访问和修改
        - 以__开头的属性为类的私有属性，在子类和类外部无法直接使用
        - 以__开头的方法为私有方法，在子类和类外部无法直接调用
- 私有属性和方法 示例:

```
# file : enclosure.py
class A:
    def __init__(self):
        self.__p1 = 100  # 私有属性
    def __m1(self):  # 私有方法
        print("__m1(self) 方法被调用")
    def showA(self):
        self.__m1()
        print("self.__p1 = ", self.__p1)
class B(A):
    def __init__(self):
        super().__init__()
    def showB(self):
        self.__m1()  # 出错，不允许调用
        print("self.__p1 = ", self.__p1)  # 出错，不允许调用

a = A()
```

```
a.showA()
a.__m1()        # 出错，不允许调用
v = self.__p1   # 出错，不允许调用
b = B()
b.showB()
```

多态 polymorphic

- 什么是多态:
  - 字面意思"多种状态"
  - 多态是指在有继承/派生关系的类中，调用基类对象的方法,实际能调用子类的覆盖方法的现象叫多态
- 状态:
  - 静态(编译时状态)
  - 动态(运行时状态)
- 多态说明:
  - 多态调用的方法与对象相关，不与类型相关
  - Python的全部对象都只有"运行时状态(动态)", 没有"C++语言"里的"编译时状态(静态)"
- 多态示例:

```
class Shape:
    def draw(self):
        print("Shape的draw()被调用")

class Point(Shape):
    def draw(self):
        print("正在画一个点!")

class Circle(Point):
    def draw(self):
        print("正在画一个圆!")

def my_draw(s):
    s.draw()  # 此处显示出多态

shapes1 = Circle()
shapes2 = Point()
my_draw(shapes1)  # 调用Circle 类中的draw
my_draw(shapes2)  # Point 类中的draw
```

- 面向对象编程语言的特征:
  - 继承
  - 封装

- 多态

多继承 multiple inheritance

- 多继承是指一个子类继承自两个或两个以上的基类
- 多继承的语法:

```
class 类名(基类名1, 基类名2[, ...]):
    ...
```

- 多继承说明
  - 一个子类同时继承自多个父类，父类中的方法可以同时被继承下来
  - 如果两个父类中有同名的方法，而在子类中又没有覆盖此方法时，调用结果难以确定
- 多继承的示例:

```
# file : multiple_inherit.py
class Car:
    def run(self, speed):
        print("汽车以", speed, "km/h的速度行驶")

class Plane:
    def fly(self, height):
        print("飞行以海拔", height, "米的高度飞行")

class PlaneCar(Car, Plane):
    """PlaneCar类 , 同时继承自汽车和飞机"""

p1 = PlaneCar()
p1.fly(10000)
p1.run(300)
```

- 多继承的问题(缺陷):
  - 标识符(名字空间)冲突的问题
  - 要谨慎使用多继承

多继承的 MRO (Method Resolution Order) 问题

- 类的__mro__属性
  - 用来记录类的方法查找顺序
- 示例:

```
class A:
    def go(self):
        print("A")
class B(A):
    def go(self):
        print("B")
class C(A):
    def go(self):
        print("C")
class D(B,C):
    def go(self):
        print("D")
print(D.__mro__) # (<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

- 钻石继承:

```
   A
  / \
 /   \
B     C
 \   /
  \ /
   D
```

- day19 pm

函数重写(overwrite)

- 在自定义类内添加相应的方法,让自定义类创建的实例像内建对象一样进行内建函数操作

对象转字符串函数重写

```
- repr 返回一个附合Python语法规则且能代表此对象的表达式字符串, 通常:
    - eval(repr(obj)) == obj
- str(obj) 通过给定的对象返回一个字符串(这个字符串通常是给人阅读的)

> 注: repr(obj) 函数和str(obj)函数都是返回一个对象的字符串
```

- 对象转字符串函数重写方法
  - repr() 函数的重载方法:
    - `def __repr__(self)`
  - str() 函数的重载方法:
    - `def __str__(self)`
      - 如果没有 `__str__(self)` 方法，则返回repr(obj)函数结果代替

- str(obj)函数调用方法说明

  1. str(obj)函数先查找obj.**str**() 方法,调用此方法并返回结果
  2. 如果`obj.__str()__`方法不存在，则调用`obj.__repr__`方法并返回结果
  3. 如果`obj.__repr__`方法不存在，则调用object类的`__repr__`实例方法显示'<**main**.MyNumber object at 0x102a8ea20>'格式的字符串

- str/repr函数重写示例

```
# file : mynumber.py
class MyNumber:
    "此类用于定义一个自定义的类，用于演示str/repr函数重写"
    def __init__(self, value):
        "构造函数,初始化MyNumber对象"
        self.data = value
    def __repr__(self):
        "转换为eval能够识别的字符串"
        print('__repr__(self)方法被调用 ')
        return "MyNumber(%d)" % self.data

    def __str__(self):
        "转换为普通字符串"
        return "%d" % self.data

n1 = MyNumber(100)
n2 = MyNumber(200)
print("repr(n1) ===>", repr(n1)
print("str(n2) ===>", str(n2))
```

内建函数重写

- `__abs__` abs(obj) 函数调用
- `__len__` len(obj) 函数调用
- `__reversed__` reversed(obj) 函数调用
- `__round__` round(obj) 函数调用
- 内建函数 重写示例

```
# file : len_overwrite.py
class MyList:
    def __init__(self, iterable=()):
        self.data = [x for x in iterable]
    def __repr_(self):
        return "MyList(%s)" % self.data
    def __len__(self):
        print("__len__(self) 被调用!")
        return len(self.data)
```

```
    def __abs__(self):
        print("__len__(self) 被调用!")
        return MyList((abs(x) for x in self.data))

myl = MyList([1, -2, 3, -4])
print(len(myl))
print(abs(myl))
```

数值转换函数重写

- __complex__ complex(obj) 函数调用
- __int__ int(obj) 函数调用
- __float__ float(obj) 函数调用
- __bool__ bool(obj) 函数调用(稍后会讲)
- 数值转换函数重写示例

```
class MyNumber:
    "此类用于定义一个自定义的类，用于演示函数重写"
    def __init__(self, value):
        "构造函数,初始化MyNumber对象"
        self.data = value
    def __float__(self):
        "float函数重写"
        print("__float__ is called")
        return float(self.data)

n1 = MyNumber('100')
print(float(n1))
```

布尔测试函数重写

- 布尔测试函数重写 格式

```
def __bool__(self):
    ...
```

- 作用:
  - 用于bool(obj)函数取值
  - 用于if语句真值表达式中
  - 用于while语句真值表达式中
- 布尔测试函数重写说明:

  - 当自定义类内有__bool__(self)方法时,以此方法的返回值作为bool(obj)的返回值
  - 当不存在__bool__(self)方法时,bool(x)返回__len__(self)方法的返回值是否为零来测试布尔值
  - 当不存在__len__(self)方法时，则直接返回True

- 布尔测试函数重写示例: (将此示例上调至 abs. len 方法重写)

```
# file : bool.py
class MyList:
    def __init__(self, iterable=()):
        self.data = [x for x in iterable]
    def __repr__(self):
        return "MyList(%s)" % self.data
    def __bool__(self):
        print("__bool__(self) 被调用!")
        return any(self.data)
    def __len__(self):
        print("__len__(self) 被调用!")
        return len(self.data)
    # def __abs__(self):
    #     print("__len__(self) 被调用!")
    #     return MyList((abs(x) for x in self.data))


myl = MyList([1, -2, 3, -4])
print(len(myl))
# print(abs(myl))
if myl:
    print("myl为真值")
else:
    print("myl为假值")

myl = MyList(count = 10, value = 1)
if myl:
    print("myl为真值")
else:
    print("myl为假值")
```

- 对象的属性管理函数

函数	说明
getattr(obj, name[, default])	从一个对象得到对象的属性；getattr(x, 'y') 等同于x.y; 当属性不存在时,如果给 出default参数,则返回default,如果没有给出default 则产生一个AttributeError错误
hasattr(obj, name)	用给定的name返回对象obj是否有此属性,此种做法可以避免在getattr(obj, name)时引发错误
setattr(obj, name, value)	给对象obj的名为name的属性设置相应的值value, setattr(x, 'y', v) 等同于 x.y = v

函数	说明
delattr(obj, name)	删除对象obj中的name属性, delattr(x, 'y') 等同于 del x.y

- 对象的属性管理函数示例:

```
class Dog:
    pass
d = Dog()
d.color = '白色'
v = getattr(d, 'color')   # 等同于 v = d.color
v = getattr(d, 'kinds')  # 出错,没有d.kinds属性
v = getattr(d, 'kinds', '没有这个属性')  # v= '没有这个属性'
hasattr(d, 'color')  # True
hasattr(d, 'kinds')  # False
setattr(d, 'kinds', '京巴')  # 等同于d.kinds ='京巴'
hasattr(d, 'kinds')  # True
delattr(d, 'kinds')  # 等同于 del d.kinds
hasattr(d, 'kinds')  # False
```

迭代器(高级)

- 什么是迭代器
  - 可以通过next函数取值的对象，就是迭代器
- 迭代器协议
  - 迭代器协议是指对象能够使用next函数获取下一项数据，在没有下一项数据时触发一个StopIteration 异常来终止迭代的约定
- 迭代器协议的实现方法
  - 在类内需要用 `__next__(self)` 方法来实现迭代器协议
- 迭代对象的语法形式

```
class MyIterator:
    def __next__(self):
        迭代器协议
        return 数据
```

- 什么是可迭代对象
  - 是指能用iter(obj)函数返回迭代器的对象(实例)
  - 可迭代对象内部要定义`__iter__(self)`方法来返回迭代器对象(实例)
- 可迭代对象的语法形式

```
class MyIterable:
    def __iter__(self):
        语句块
        return 迭代器
```

- 迭代器协议示例

```
class MyList:
    def __init__(self, iterable=()):
        self.data = [x for x in iterable]
    def __repr__(self):
        return 'MyList(%r)' % self.data
    def __iter__(self):
        return MyListIterator(self.data)

class MyListIterator:
    def __init__(self, lst):
        self.lst_data = lst  # 绑定要迭代的列表
        self.cur_index = 0  # 迭代的起始位置
    def __next__(self):
        if self.cur_index >= len(self.lst_data):
            raise StopIteration  # 发送迭代结束通知
        r = self.lst_data[self.cur_index]
        self.cur_index += 1
        return r  # 返回此次提供的数据

L = MyList("ABCD")
print(L)  # MyList(['A', 'B', 'C', 'D'])
for x in L:
    print(x)  # A B C D E
```

- 练习:
  - 实现一个与系统内建的range类相同功能的类:

```
class MyRange:
    def __init__(self, ...):
        ...
    def __iter__(self):
        ...
# 测试调用如下:
L = list(MyRange(5))
print(L)  # [0, 1, 2, 3, 4]
print(sum(MyRange(1, 101)))  # 5050
L2 = [x**2 for x in MyRange(1, 10, 3)]
print(L2)  # [1, 16, 49]
for x in MyR  ange(10, 0, -3):
    print(x)# 10, 7, 4, 1
```

- day20

异常(高级)

- 异常回顾（可以用于异常的语句):
  - try-except # 捕获异常,得到异常通知,并将程序从异常流程转为正常流程
  - try-finally # 做任何流程(正常流程/异常流程)都必须要执行的语句
  - raise # 发送异常通知并进入异常流程
  - assert # 根据条件发送AssertionError类型的异常通知

with 语句

- with 语句语法

```
with 表达式1 [as 变量1], 表达式1 [as 变量1], ...:
    语句块
```

- with 语句的作用
  - 使用于对资源进行访问的场合。确保使用过程中不管是否发生异常，都会执行必要的“清理”操作，并释放资源。
    - 例如文件使用后自动关闭，线程中锁的自动获取和释放等(线程后面会学)
- with 语句 说明
  - 执行表达式,用as子句中的变量绑定生成的对象
  - with语句并不改变异常的状态
- with 语句示例

```
# 用with 语句自动关闭文件
with open("../day19.txt", 'r') as f:
    for l in f:
        print(l)
        # 以下为可能触发异常的语句
        int(input("请输入数字: "))
        # raise ZeroDivisionError

print("程序结束")
```

- 环境管理器
  - 类内有 __enter__ 和 __exit__ 实例方法的类被称为环境管理器
  - 能够用with进行管理的对象必须是是环境管理器
  - __enter__ 将在进入with语句时被调用并返回 由as 变量 管理的对象
  - __exit__ 将在离开with语句时被调用,且可以用参数来判断在离开with语句时是否有异常发生并做出相应的处理

- 最简单的环境管理器的代码编写示例

```
class A:
    '''此类的对象可用于with语句进行管理'''
    def __enter__(self):
        print("已进入with语句")
        return self

    def __exit__(self, exc_type, exc_value, exc_taceback):
        print('已离开with语句')
        if exc_type is None:
            print("with语句内没有发生异常! ")
        else:
            print("with语句内有", exc_type, "类型的异常发生")

try:
    with A() as a:
        print("这是with语句内部的输出")
        int(input("请输入整数: "))
except ValueError:
    print("有ValueError类型的异常发生")
```

运算符重载

- 运算符重载是指让自定义的类生成的对象(实例)能够使用运算符进行操作
- 运算符重载的作用
  - 让自定义类的实例像内建对象一样进行运算符操作
  - 让程序简洁易读
  - 对自定义对象将运算符赋予新的运算规则
- 运算符重载说明:
  - 运算符重载方法的参数已经有固定的含义,不建议改变原有的意义

算术运算符重载

方法名	运算符和表达式	说明
`__add__(self, rhs)`	self + rhs	加法
`__sub__(self, rhs)`	self - rhs	减法
`__mul__(self, rhs)`	self * rhs	乘法
`__truediv__(self, rhs)`	self / rhs	除法
`__floordiv__(self, rhs)`	self // rhs	地板除
`__mod__(self, rhs)`	self % rhs	取模(求余)

方法名	运算符和表达式	说明
`__pow__(self, rhs)`	self ** rhs	幂

```
rhs (right hand side) 右手边
```

- 二元运算符重载方法格式:

```
def __xxx__(self, other):
    ....
```

- 算术运算符重载示例

```
class MyNumber:
    "此类用于定义一个自定义的类，用于演示运算符重载"
    def __init__(self, value):
        "构造函数,初始化MyNumber对象"
        self.data = value
    def __repr__(self):
        "转换为表达式字符串"
        return "MyNumber(%d)" % self.data
    def __add__(self, rhs):
        "加号运算符重载"
        print("__add__ is called")
        return MyNumber(self.data + rhs.data)
    def __sub__(self, rhs):
        "减号运算符重载"
        print("__sub__ is called")
        return MyNumber(self.data - rhs.data)

n1 = MyNumber(100)
n2 = MyNumber(200)
print(n1 + n2)
print(n1 - n2)
```

反向算术运算符重载:

- 当运算符的左侧为内建类型时右侧为自定义类型进行算术运算符运算时，会出现TypeError错误
- 因无法修改内建类型的代码来实现运算符重载，此时需要使用反向算术运算符的重载来完成重载

方法名	运算符和表达式	说明
`__radd__(self, lhs)`	lhs + self	加法
`__rsub__(self, lhs)`	lhs - self	减法
`__rmul__(self, lhs)`	lhs * self	乘法

方法名	运算符和表达式	说明
`__rtruediv__(self, lhs)`	lhs / self	除法
`__rfloordiv__(self, lhs)`	lhs // self	地板除
`__rmod__(self, lhs)`	lhs % self	取模(求余)
`__rpow__(self, lhs)`	lhs ** self	幂

```
lhs (left hand side) 左手边
```

复合赋值算术运算符重载:

- 以复合赋值算术运算符 x += y 为例，此运算符会优先调用 x.**iadd**(y)方法，如果没有__iadd__方法时会将复合赋值运算拆解为 x = x + y,然后调用 x = x.**add**(y)方法；如果再不存在__add__方法则会触发TypeError异常
- 其它复合赋值算术运算符也具有相同的规则

方法名	运算符和复合赋值语句	说明
`__iadd__(self, rhs)`	self += rhs	加法
`__isub__(self, rhs)`	self -= rhs	减法
`__imul__(self, rhs)`	self *= rhs	乘法
`__itruediv__(self, rhs)`	self /= rhs	除法
`__ifloordiv__(self, rhs)`	self //= rhs	地板除
`__imod__(self, rhs)`	self %= rhs	取模(求
`__ipow__(self, rhs)`	self **= rhs	幂

比较运算符的重载

方法名	运算符和复合赋值语句	说明
`__lt__(self, rhs)`	self < rhs	小于
`__le__(self, rhs)`	self <= rhs	小于等于
`__gt__(self, rhs)`	self > rhs	大于
`__ge__(self, rhs)`	self >= rhs	大于等于
`__eq__(self, rhs)`	self == rhs	等于
`__ne__(self, rhs)`	self != rhs	不等于

- 比较运算符通常返回布尔值 True 或 False

位运算符重载

方法名	运算符和表达式	说明
`__invert__(self)`	~ self	取反(一元运算
`__and__(self, rhs)`	self & rhs	位与
`__or__(self, rhs)`	self | rhs	位或
`__xor__(self, rhs)`	self ^ rhs	位异或
`__lshift__(self, rhs)`	self << rhs	左移
`__rshift__(self, rhs)`	self >> rhs	右移

反向位运算符重载

方法名	运算符和表达式	说明
`__rand__(self, lhs)`	lhs & self	位与
`__ror__(self, lhs)`	lhs | self	位或
`__rxor__(self, lhs)`	lhs ^ self	位异或
`__rlshift__(self, lhs)`	lhs << self	左移
`__rrshift__(self, lhs)`	lhs >> self	右移

复合赋值位运算符重载

方法名	运算符和复合赋值语句	说明
`__iand__(self, rhs)`	self &= rhs	位与
`__ior__(self, rhs)`	self |= rhs	位或
`__ixor__(self, rhs)`	self ^= rhs	位异或
`__ilshift__(self, rhs)`	self <<= rhs	左移
`__irshift__(self, rhs)`	self >>= rhs	右移

一元运算符重载

方法名	运算符和表达式	说明
`__neg__(self)`	-self	负号
`__pos__(self)`	+self	正号
`__invert__(self)`	~self	取反

- 一元运算符重载 语法:

```
class 类名:
    def __xxx__(self):
        ....
```

- 一元运算符重载示例

```
# 示例1
class MyList:
    def __init__(self, iterable):
         self.data = [x for x in iterable]
    def __repr__(self):
        return 'MyList(%r)' % self.data
    def __neg__(self):
        '对自定义的列表取 -(负号)操作时，正变负，负变正'
        return MyList([-x for x in self.data])
myl1 = MyList([1, -2, 3, -4, 5])
myl2 = -myl1
print(myl2)
myl3 = +myl1
print(myl3)
```

in/not in 成员运算符重载

- __contains__ 函数调用
- __contains__ 函数格式

```
def __contains__(self, e:'元素'):
    pass
```

方法名	运算符和表达式	说明
__contains__(self, e)	e in self	成员运算

索引和切片运算符重载:

方法名	运算符和表达式	说明
__getitem__(self, i)	x = self[i]	索引/切片取值
__setitem__(self, i, val)	self[i] = val	索引/切片的赋值
__delitem__(self, i)	del self[i]	del语句删除索引/切片

- 作用： -让自定义类型的对象能够支持索引和切片操作

slice函数

- slice函数作用:
  - 用于创建一个slice切片对象,此对象存储一个切片的起始值，终止值，步长信息
- slice函数格式

函数	说明
slice(start=None, stop=None, step=None)	创建一个slice切片对象

- slice 对象的属性
  - s.start 切片的起始值，默认为None
  - s.stop 切片的终止值，默认为None
  - s.step 切片的步长，默认为None
- 切片取值运算符重载

```
class MyList:
    def __init__(self, iterable):
         self.data = [x for x in iterable]
    def __repr__(self):
        return 'MyList(%r)' % self.data
    def __getitem__(self, i):
        '[] 取值运算符重载'
        if type(i) is slice:
            print("正在进行切片取值操作...")
            print("i.start =", i.start)
            print("i.stop =", i.stop)
            print("i.step =", i.step)
        return MyList(self.data[i])


myl1 = MyList([1, -2, 3, -4, 5])
print(myl1[::2])
```

特性属性 @property

- 实现其它语言所拥有的 getter 和 setter功能
- 作用:
  - 用来模拟一个属性
  - 通过@property装饰器可以对模拟属性赋值和取值加以控制
- 特性属性示例

```
# file : property.py
class Student:
    def __init__(self, s):
        self.__score = s
    def getScore(self):
        return self.__score
    def setScore(self, a):
        if a < 0 or a > 100:
            print("年龄超出了正常范围，请核对!")
        self.__score = a
    score = property(getScore, setScore)

s = Student(10)
print(s.score)
```

PEP8编码规范

代码编排

1. 使用4空格缩进，不使用Tab,更不允许用Tab和空格混合缩进
2. 每行最大长度最大79字节，超过部分使用反斜杠折行
3. 类和全局函数定义间隔两个空行,类内方法定义间隔一个空行.其它地方可以不加空行。

文档编排

1. 其中import部分，又按标准、三方和自己编写顺序依次排放，之间空一行。
2. 不要在一句import中导入多个模块，比如不推荐import os, sys。
3. 尽可能用import XX 而不采用from XX import YY引用库,因为可能出现名字冲突。

空格的使用

1. 各种右括号前不用加空格
2. 逗号、冒号、分号前不要加空格。
3. 函数的左括号前不要加空格。如func(1)。
4. 序列的左括号前不要加空格。如list[2]。
5. 操作符左右各加一个空格，不要为了对齐增加空格。
6. 函数默认参数使用的赋值符左右省略空格。
7. 不要将多条语句写在同一行，尽管使用‘;’允许。
8. if/for/while语句中，即使执行语句只有一句，也必须另起一行

- 原则：避免不必要的空格

参考：https://www.douban.com/note/134971609/

以下内容可以参考http://www.cnblogs.com/MarchThree/p/4014775.html