**17Python\_08(OOP基础+进阶)+re模块**

**一 OOP-OOP简介-基本概念**

OOP:面向对象编程

• 实现了数据属性和行为属性的融合

**1.1 基本概念**

• 类(Class): 用来描述 具有相同的属性和方法的对象的**集合**。它定义了该集合中每个对象所共有的属性和方法。**对象是类的实例**。

• 实例/对象: 类的一个具体体现。

• 方法: 类中定义的函数。

• 对象: 通过类定义的数据结构实例。对象包括两个数据成员(类变量和实例变量)和方法。

**1.2 创建类**

• 使用 class 语句来创建一个新类,class 之后为类的名称并以冒号结尾

• 类名建议使用驼峰形式

class BearToy:

pass

**1.3 创建实例**

• 类是蓝图,实例是根据蓝图创建出来的具体对象

tidy = BearToy()

**二 OOP绑定方法**

**2.1 构造器方法**

• 当实例化类的对象是,构造器方法默认自动调用

• 实例本身作为第一个参数,传递给self

class BearToy:

def \_\_init\_\_(self, size, color):

self.size = size

slef.color = color

if \_\_name\_\_ == '\_\_main\_\_':

tidy = BearToy('small', 'orange')

**2.2 其他绑定方法**

• 类中定义的方法需要绑定在具体的实例,由实例调用

• 实例方法需要明确调用

class BearToy:

def \_\_init\_\_(self, size, color):

self.size = size

self.color = color

def speak(self):

print('hahaha')

if \_\_name\_\_ == '\_\_main\_\_':

tidy = BearToy('small', 'orange')

tidy.speak()

**roles.py**

class Warrior:

def \_\_init\_\_(self, name, weapon):

# self不是关键字,可以是任何名字,表示实例本身

# 绑定在对象身上的属性,在类中任意位置可用

self.name = name # 可理解为 实例.姓名=吕布

self.weapon = weapon ##可理解为 实例.武器=方天画戟

def speak(self, words):

# 方法自己的参数或变量,就是函数的局部变量

print('我是%s,%s' % (self.name, words))

def attack(self, target):

print('正在攻击: %s' % target)

if \_\_name\_\_ == '\_\_main\_\_':

# 创建实例时,自动调用\_\_init\_\_方法,实例自动作为第一个参数

lb = Warrior('吕布', '方天画戟') # 创建实例lb

print(lb.name)

print(lb.weapon)

lb.speak('马中赤兔,人中吕布')

print('\*' \* 30)

zf = Warrior('张飞', '丈八蛇矛')

print(zf.name)

print(zf.weapon)

zf.speak('我乃燕人张飞张翼德')

print('\*' \* 30)

lb.attack('董卓')

zf.attack('曹操')

**三 OOP进阶-组合和派生**

**3.1 什么是组合**

**• 两个类有明显的不同**

**• 一个类的属性是另一个类的实例,用组合**

• 类被定义后,目标就是要把它当成一个模块来使用,并把这些对象嵌入到你的代码中去

• 组合就是让不同的类混合并加入到其它类中来增加功能和代码重用性

• 可以在一个大点的类中创建其它类的实例,实现用一些其它属性和方法来增强原来的类对象

**3.2 组合应用**

• 两个类明显不同

• 一个类是另一个类的组件

class Manufacture:

def \_\_init\_\_(self, phone, email):

self.phone = phone

self.email = email

class BearToy:

def \_\_init\_\_(self, size, color, phone, email):

self.size = size

self.color = color

self.vendor = Manufacture(phone, email)

**roles2.py**

class Warrior:

def \_\_init\_\_(self, name, weapon):

self.name = name

self.weapon = weapon

class Weapon:

def \_\_init\_\_(self, name, strength, type):

self.name = name

self.strength = strength

self.type = type

if \_\_name\_\_ == '\_\_main\_\_':

ji = Weapon('方天画戟', '88', '物理伤害')

# print(ji.name,ji.strength,ji.type)

lb = Warrior('吕布', ji)

print(lb.weapon.name, lb.weapon.type)

**3.3 创建子类**

**• 两个类非常像是,只是有一些不同**

**• 通过一个类派生出另一个类,也可以说一个类继承于一个类**

• 当类之间有显著的不同,并且较小的类是较大的类所需要的组件时组合表现得很好;但当设计“相同的类但有一些不同的功能”时,派生就是一个更加合理的选择了

• OOP 的更强大方面之一是能够使用一个已经定义好的类,扩展它或者对其进行修改,而不会影响系统中使用现存类的其它代码片段

• OOD(面向对象设计)允许类特征在子孙类或子类中进行继承

• 创建子类只需要在圆括号中写明从哪个父类继承即可

class BearToy:

def \_\_init\_\_(self, size, color):

self.size = size

self.color = color

... ...

class NewBearToy(BearToy):

pass

**3.4 继承**

• 继承描述了基类的属性如何“遗传”给派生类

• 子类可以继承它的基类的任何属性,不管是数据属性还是方法

class BearToy:

def \_\_init\_\_(self, size, color):

self.size = size

self.color = color

... ...

class NewBearToy(BearToy):

pass

if \_\_name\_\_ == '\_\_main\_\_':

tidy = NewBearToy('small', 'orange')

tidy.speak()

**roles3**

class Role:

def \_\_init\_\_(self, name, weapon):

# self不是关键字,可以是任何名字,表示实例本身

# 绑定在对象身上的属性,在类中任意位置可用

self.name = name # 可理解为 实例.姓名=吕布

self.weapon = weapon ##可理解为 实例.武器=方天画戟

def speak(self, words):

# 方法自己的参数或变量,就是函数的局部变量

print('我是%s,%s' % (self.name, words))

def attack(self, target):

print('正在攻击: %s' % target)

class Warrior(Role):

# 子类可以继承父类(基类)的所有方法

pass

class Mage(Role):

pass

if \_\_name\_\_ == '\_\_main\_\_':

lb = Warrior('吕布', '方天画戟')

lj = Mage('李靖', '宝塔')

lb.speak('人中吕布,马中赤兔')

lj.speak('宝塔镇河妖')

**3.5 通过继承覆盖方法**

• 如果子类中有和父类同名的方法,父类方法将被覆盖

• 如果需要访问父类的方法,则要调用一个未绑定的父类方法,明确给出子类的实例

class BearToy:

def \_\_init\_\_(self, size, color, phone, email):

self.size = size

self.color = color

self.vendor = Manufacture(phone, email)

... ...

class NewBearToy(BearToy):

def \_\_init\_\_(self, size, color, phone, email, date):

super(NewBearToy, self).\_\_init\_\_(size, color, phone,email)

self.date = date

myclass.py

class A:

def \_\_init\_\_(self, a, b, c, d):

self.a = a

self.b = b

self.c = c

self.d = d

class B(A):

def \_\_init\_\_(self, a, b, c, d, e):

#A.\_\_init\_\_(self, a, b, c, d) #此2行效果一致

super(B, self).\_\_init\_\_(a, b, c, d) #此2行效果一致

self.e = e

if \_\_name\_\_ == '\_\_main\_\_':

b1 = B(10, 2, 30, 40, 50)

print(b1.a, b1.e)

**3.5 多重继承**

• python允许多重继承,子类可以有多个父类,子类自动继承父类的所有方法

• 对象查找方法时,按自下向上,从左到右的原则查找

>>> class A:

... def foo(self):

... print('foo method’)

>>> class B:

... def bar(self):

... print('bar method’)

>>> class C(A, B):

... pass

>>> c = C()

>>> c.foo()

foo method

>>> c.bar()

bar method

**myclass2.py**

class A:

def func1(self):

print('A func')

def fun4(self):

print('AAAA func4')

class B:

def func2(self):

print('B func')

def fun4(self):

print('BBBB func4')

class C(A,B): #继承父类方法时,从左到右,先继承左边方法?????

def func3(self):

print('C func')

def func4(self):

print('CCCC func4')

if \_\_name\_\_ == '\_\_main\_\_':

c1=C()

c1.func1()

c1.func2()

c1.func3()

c1.func4()

#查找方法时,从下到上查找方法.在此例中,c1.func4()首先找到的是C类里的方法

**四 OOP进阶-特殊方法**

• 为了实现

**4.1 \_\_init\_\_方法**

• 实例化类实例时默认会调用的方法

class BearToy:

\_\_init\_\_(self, size, color):

self.size = size

slef.color = color

if \_\_name\_\_ == '\_\_main\_\_':

tidy = BearToy('small', 'orange')

**4.2 \_\_str\_\_方法**

• 打印/显示实例时调用方法

• 返回字符串

class BearToy:

def \_\_init\_\_(self, size, color):

self.size = size

slef.color = color

def \_\_str\_\_(self):

return '<Bear: %s %s>' % (self.size, self.color)

if \_\_name\_\_ == '\_\_main\_\_':

tidy = BearToy('small', 'orange')

print(tidy)

**4.3 \_\_call\_\_方法**

• 用于创建可调用的实例

class BearToy:

def \_\_init\_\_(self, size, color):

self.size = size

slef.color = color

def \_\_call\_\_(self):

print('I am a %s bear' % self.size)

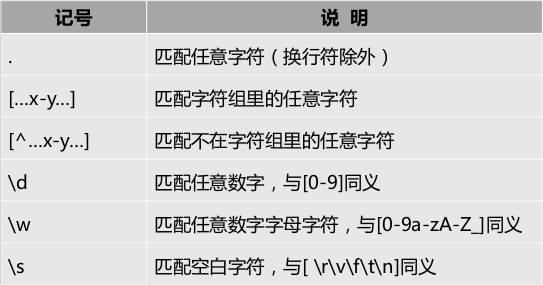
if \_\_name\_\_ == '\_\_main\_\_':

tidy = BearToy('small', 'orange')

print(tidy)

**五 re模块-正则表达式**

**5.1 匹配单个字符**



**\D \W \S 是 \d \w \s的取反,匹配^ $ -等放开头或结尾有特殊效果的字**

**符,匹配时需要更改这些字符的位置,见5.3**

**^在中括号中开头表示取反**

**5.2 匹配一组字符(注意有写记号需要转义符\)**



**5.3 其他元字符**



匹配单词边际使用 \<单词\>

**案例:为MAC地址加冒号(MAC)地址为16进制**

192.168.1.1 000C29123456

192.168.1.2 5254A3802B32

:%s/\(..\)\(..\)\(..\)\(..\)\(..\)\(..\)$/\1:\2:\3:\4:\5:\6/

192.168.1.1 00:0C:29:12:34:56

192.168.1.2 52:54:A3:80:2B:32

**六 re模块-核心函数和方法**

**6.1 match函数**

• 尝试用正则表达式模式从**字符串的开头**匹配,如果匹配成功,则返回一个匹配对象;否则返回None

**6.2 search函数**

• 在**字符串中**查找正则表达式模式的第一次出现,如果匹配成功,则返回一个匹配对象;否则返回None

**6.3 group函数**

• 使用match或search匹配成功后,返回的匹配对象可以通过group方法获得匹配内容

>>> import re

>>> re.match('f..','food')

#在food的开头匹配f.. ,匹配到返回匹配对象,否则返回None

<re.Match object; span=(0, 3), match='foo'>

>>> print(re.match('f..','seafood'))

None

>>> re.search('f..','seafood') #在字符串中匹配f..

<re.Match object; span=(3, 6), match='foo'>

>>> m=re.search('f..','seafood')

>>> m.group() #匹配到之后,用group方法返回匹配结果

'foo'

**6.4 findall函数**

• 在字符串中查找正则表达式模式的所有(非重复)出现;返回一个匹配对象的**列表**

>>> re.findall('f..','seafood is food')

['foo', 'foo']

**6.5 finditer函数**

• 和findall()函数有相同的功能,但返回的不是列表而是**迭代器**;对于每个匹配,该迭代器返回一个匹配对象

#finditer返回的是由匹配对象构成的迭代器

>>> re.finditer('f..','seafood is food')

<callable\_iterator object at 0x7f28fb10adc0>

>>> list(re.finditer('f..', 'seafood is food'))

[<re.Match object; span=(3, 6), match='foo'>, <re.Match object; span=(11, 14), match='foo'>]

>>> for m in re.finditer('f..', 'seafood is food'):

... m.group()

... print(m.group())

...

foo

foo

**6.6 compile函数**

• 对正则表达式模式进行编译,返回一个正则表达式对象

• 不是必须要用这种方式,但是在大量匹配的情况下,可以提升效率

**6.7 split方法**

• 根据正则表达式中的分隔符把字符分割为一个列表,并返回成功匹配的列表

• 字符串也有类似的方法,但是正则表达式更加灵活

**以.和-作为分隔符,切割字符串**

>>> re.split('-|\.','hello-word.tar.gz')

['hello', 'word', 'tar', 'gz']

**6.8 sub方法**

• 把被替换字符串中所有匹配正则表达式的旧字符串替换成新字符串

>>> re.sub('X','tedu','Hello X.Hi X')

'Hello tedu.Hi tedu'

re.sub(‘旧字符串’,’新字符串’,’被替换字符串’)

**6.9 为了提升效率,建议将正则表达式的模式先编译**

>>> patt=re.compile('f..')

>>> m=patt.search('seafood')

>>> m.group()

'foo'

>>> patt.findall('seafood is food')

['foo', 'foo']

**6.10 字典排序:**

**字典本身没有顺序,不能排序**

**需要将其转换成其他序列类型**

>>> result={'172.40.58.150': 10, '172.40.58.124': 6, '172.40.58.101': 10, '127.0.0.1': 121, '192.168.4.254': 103, '192.168.2.254': 110, '201.1.1.254': 173, '201.1.2.254': 119, '172.40.0.54': 391, '172.40.50.116': 244}

...

>>> **alist=list(result.items())**

>>> alist

[('172.40.58.150', 10), ('172.40.58.124', 6), ('172.40.58.101', 10), ('127.0.0.1', 121), ('192.168.4.254', 103), ('192.168.2.254', 110), ('201.1.1.254', 173), ('201.1.2.254', 119), ('172.40.0.54', 391), ('172.40.50.116', 244)]

**复杂列表排序:**

**列表的sort方法,只支持一个名为key的参数**

**key应该是一个函数**

**该换上处理列表的每一项,处理结果作为排序依据**

>>> **def get\_second(seq):**

...  **return seq[-1]**

>>> **alist.sort(key=get\_second)**

>>> alist

[('172.40.58.124', 6), ('172.40.58.150', 10), ('172.40.58.101', 10), ('192.168.4.254', 103), ('192.168.2.254', 110), ('201.1.2.254', 119), ('127.0.0.1', 121), ('201.1.1.254', 173), ('172.40.50.116', 244), ('172.40.0.54', 391)]

**上面写法,可以用以下一行解决,并实现降序排列**

>>> **alist.sort(key=lambda seq:seq[-1],reverse=True)**

>>> alist

[('172.40.0.54', 391), ('172.40.50.116', 244), ('201.1.1.254', 173), ('127.0.0.1', 121), ('201.1.2.254', 119), ('192.168.2.254', 110), ('192.168.4.254', 103), ('172.40.58.150', 10), ('172.40.58.101', 10), ('172.40.58.124', 6)]

**案例3:分析apache访问日志(字典排序见6.10)**

• 编写一个apche日志分析脚本

1. 统计每个客户端访问apache服务器的次数

2. 将统计信息通过字典的方式显示出来

3. 分别统计客户端是Firefox和MSIE的访问次数

4. 分别使用函数式编程和面向对象编程的方式实现

**countpatt.py(函数式)**

import re

def count\_patt(fname, patt):

result = {} # 定义空字典,用于保存结果

cpatt = re.compile(patt) # 编译模式,提升效率

with open(fname) as fobj:

for line in fobj:

m = cpatt.search(line)

if m: # 如果匹配到了

key = m.group() # 获取key

result[key] = result.get(key, 0) + 1

return result

if \_\_name\_\_ == '\_\_main\_\_':

fname = 'access\_log'

ip = '^(\d+\.){3}\d+'

br = 'Firefox|MSIE|Chrome'

result1 = count\_patt(fname, ip)

result2 = count\_patt(fname, br)

print(result1)

print(result2)

**counterpatt.py2(OOP式)**

import re

class CountPatt:

def \_\_init\_\_(self,fname):

self.fname=fname

def count\_patt(self, patt):

result = {} # 定义空字典,用于保存结果

cpatt = re.compile(patt) # 编译模式,提升效率

with open(self.fname) as fobj:

for line in fobj:

m = cpatt.search(line)

if m: # 如果匹配到了

key = m.group() # 获取key

result[key] = result.get(key, 0) + 1

return result

if \_\_name\_\_ == '\_\_main\_\_':

fname = 'access\_log'

ip = '^(\d+\.){3}\d+'

br = 'Firefox|MSIE|Chrome'

cp1 = CountPatt(fname)

result1 = cp1.count\_patt(ip)

result2 = cp1.count\_patt(br)

print(result1)

print(result2)

print('\*'\*30)

cp2=CountPatt('/etc/passwd')

result3=cp2.count\_patt('nologin$|bash$')

print(result3)