1.构造方法和普通方法重载

#B A B和A方法名一样

#python是动态语言，重载不会考虑参数类型或者个数，只要方法名一样，就会重载

class Bird:

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

self.hungry = True

print("Bird构造方法")

def eat(self):

if self.hungry:

print("吃虫子")

self.hungry = False

else:

print("不饿")

b=Bird()

b.eat()

b.eat()

b.eat()

class SongBird(Bird):

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

super().\_\_init\_\_()

self.sound="唱歌"

print("SongBird 构造方法")

def sing(self):

super().eat()

print(self.sound)

print("="\*10)

sb=SongBird()

sb.sing()

sb.eat()

print("--------------")

class Myclass0:

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

print("MyClass0")

class MyClass1(Myclass0):

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

print("MyClass1")

class MyClass2(MyClass1):

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

print("MyClass2")

class MyClass3(MyClass2):

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

print("MyClass3")

super(MyClass1, self).\_\_init\_\_()

p=MyClass3()

C:\ProgramData\Anaconda3\python.exe F:/development/py/li\_ning/ch10/demo3\_super.py

Bird构造方法

吃虫子

不饿

不饿

==========

Bird构造方法

SongBird 构造方法

吃虫子

唱歌

不饿

--------------

MyClass3

MyClass0

Process finished with exit code 0

2.自定义序列

'''

特殊方法

\_\_methodname\_\_

\_\_init\_\_

\_\_len\_\_(self):返回序列中元素的个数，使用len函数时会调用这个特殊方法

\_\_getitem\_\_(self,key):返回key对应的值，在对序列使用[...]操作时调用该方法

\_\_setitem\_\_(self,key,val):设置key对应的值，在修改序列元素时调用该方法

\_\_delitem\_\_(self,key)：在对序列使用del时调用

#d=FactorialDict()

#d['4!']=n

print((d['4!']))#1\*2\*3\*4=24

'''

class FactorialDict:

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

self.dict={}

def factorial(self,n):

if n==0 or n==1:

return 1

else:

return n\*self.factorial(n-1)

def \_\_getitem\_\_(self, item):

print("getitem方法被调用，key={}".format(item))

if item in self.dict:

return self.factorial(self.dict[item])

else:

return 0

def \_\_setitem\_\_(self, key, value):

print("setitem被调用，key={}".format(key))

self.dict[key] = int(value)

def \_\_delitem\_\_(self, key):

print("delitem被调用,key={}".format(key))

del self.dict[key]

def \_\_len\_\_(self):

print("len被调用")

return len(self.dict)

d=FactorialDict()

d['4!']=4

d['7!']=7

print('4!','=',d['4!'])

print('len={}'.format(len(d)))

del d['7!']

print('len={}'.format(len(d)))

3.从内建列表、字符串、字典继承序列

#list, dict,str

#CounterList,ConterDict,MutliString

class CounterList(list):

def \_\_init\_\_(self,\*args):

super().\_\_init\_\_(\*args)

self.counter=0

def \_\_getitem\_\_(self, item):

self.counter +=1

return super().\_\_getitem\_\_(item)

c=CounterList(range(10))

print(c)

c.reverse()

print(c)

del c[2:6]

print(c)

print(c[1]\*c[2])

print(c.counter)

print("\*"\*16)

class CounterDict(dict):

def \_\_init\_\_(self,\*args):

super().\_\_init\_\_(\*args)

self.couter = 0

print("CounterDict init 被调用")

def \_\_getitem\_\_(self, item):

self.couter +=1

return super().\_\_getitem\_\_(item)

d=CounterDict({'name':'bill'})

print(d['name'])

print(d.get('age'))

#print(d['age']) #'[]'会报错，get返回None

print(d.couter)

class MultiString(str):

def \_\_new\_\_(self,\*args,sep=' '):

s=''

for arg in args:

s += arg + sep

index = -len(sep)

if index == 0:

index = len(s)

return super().\_\_new\_\_(self,s[:index])

def \_\_init\_\_(self,\*args,sep=' '):

pass

s1=MultiString("abc","hello")

print(s1)

s2=MultiString("a","b","c",sep='#2')

print(s2)

4.属性方法

class Rectangle:

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

self.left=0

self.top = 0

def setLeft(self,left):

self.left = left

def setPosition(self, position):

print("setPosition called")

self.left,self.top = position

def getPosition(self):

print('getPosition called')

return self.left,self.top

def deletePosition(self):

print("del Posrition called")

self.left,self.top = 0,0

position = property(getPosition,setPosition,deletePosition)

r = Rectangle()

r.left=100

print(r.left)

#r.setPosition([300,400])

#print(r.getPosition())

r.position = 12,34

print(r.position)

del r.position

print(r.position)

5.监控所有属性

'''

\_\_getattr\_\_(self,name): 监控所有的属性读操作

\_\_setattr\_\_(self,name,value):用来监控所有的属性写操作

\_\_delattr\_\_(self,name)：监控所有属性的删除操作

'''

class Rectangle:

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

self.left=0

self.top = 0

self.width = 0

self.height=0

def \_\_setattr\_\_(self, key, value):

print("{} is setted,new is {}".format(key,value))

if key == "size":

self.width,self.height = value

elif key == 'position':

self.left,self.top = value

else:

self.\_\_dict\_\_[key] = value

def \_\_getattr\_\_(self, item):

print("{} is getted.".format(item))

if name == 'size':

return self.width,self.height

elif name == 'position':

return self.left,self.top

def \_\_delattr\_\_(self, item):

if name == 'size':

self.width,self.height = 0,0

elif name == 'position':

self.left,self.top = 0,0

r = Rectangle()

r.size = 12,34

print("-"\*20)

r.position = 100,200

print("-"\*40)

class MyClass:

def \_\_setattr\_\_(self, key, value):

if key == 'value':

if value > 0:

self.\_\_dict\_\_[key] = value

else:

print("{} must be bigger than 0".format(key))

else:

self.\_\_dict\_\_[key] = value

c=MyClass()

c.value = 20

print("c.value=",c.value)

c.value = -12

print("c.value=",c.value)

6.静态方法和类方法

#实例方法、静态方法、类方法

#静态方法和类方法中不能调用实例方法，而在实例方法中可以调用静态方法和类方法

class MyClass:

name = 'Bill' #静态变量

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

print("MyClass 的构造方法被调用")

self.value = 20

#静态方法

@staticmethod

def run():

print("\*",MyClass.name,"\*")

print("MyClass的静态方法run被调用")

@classmethod

def do(self):

print(self)

print('[',self.name,']')

self.run()

#print(self.value)

#实例方法

def do1(self):

print(self.value)

print('<',self.name,'>')

print(self)

MyClass.run()

print("-"\*40)

c=MyClass()

print("-"\*40)

c.do()

print("-"\*40)

c.do1()

print("-"\*40)

c.run()

7.自定义可迭代的类

'''

可迭代的类

for value in values:

\_\_iter\_\_(self):

\_\_next\_\_(self):

'''

class RightTriangle:

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

self.n = 1

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

result = '\*'\*(2\*self.n-1)

self.n +=1

return result

rt=RightTriangle()

for e in rt:

if len(e)>20:

break

print(e)

#可无限迭代的菲波那切数列

class Fibonacci:

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

self.a = 0

self.b = 1

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

result = self.a

self.a,self.b = self.b,self.a+self.b

return result

fibs = Fibonacci()

for e in fibs:

if e > 1000:

break

print(e)

8.将迭代器转换为列表

#可无限迭代的菲波那切数列

class Fibonacci:

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

self.a = 0

self.b = 1

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

result = self.a

self.a,self.b = self.b,self.a+self.b

if result >500: raise StopIteration

return result

fibs = Fibonacci()

print(list(fibs))

9.生成器(generator)

#生成器generator,也是用于迭代的

def myGenerator():

numList = [1,2,3,4,5,6,7,8]

for num in numList:

yield num

for num in myGenerator():

print(num,end=' ')

nestedList=[[1,2,3],[4,3,2],[4,5,6,7,8]]

def enumList(nestedList):

for subList in nestedList:

for element in subList:

yield element

print("="\*10)

for num in enumList(nestedList):

print(num,end=' ')

print("="\*10)

print(list(enumList(nestedList)))

10.递归生成器

#递归生成器

#终止条件：

nestedList=[[1,2,3],[4,3,2],5,[4,5,6,[6,7,8],7,8]]

def enumList(nestedList):

try:

for subList in nestedList:

for element in enumList(subList):

yield element

except TypeError:

yield nestedList #递归抛出异常点nestedList

for num in enumList(nestedList):

print(num,end=' ')

11.生成器中阻止字符串被迭代

#"10"-->字符串符合迭代,单个字符也符合迭代,所以会造成无限递归

nestedList=[[1,2,3],[4,3,2],"10",[4,5,6,[6,7,8],"hello world",7,8]]

def enumList(nestedList):

try:

try:nestedList + ' ' #非字符串抛出异常

except:

pass #异常pass

else:

raise TypeError #字符串，主动抛出异常

for subList in nestedList:

for element in enumList(subList):

yield element

except TypeError:

yield nestedList #递归抛出异常点nestedList

for num in enumList(nestedList):

print(num,end=' ')