연산자 중복에서 인수 구성은 바꿀 수 없음. 내용만 바꿀 수 있음

원래의 연산자 기능과 혼동되지 않는 범위에서 하는게 중요

getitem은 내용을 읽는 것만 가능. 쓰는거 하려면 setitem

repr 다른 문자열이랑 구별할 수 있게끔

class StringRepr:

def \_\_repr\_\_(self):

return 'repr called'

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

return 'str called'

>>> s=StringRepr()

>>> print(s)

str called

>>> str(s)

'str called'

>>> repr(s)

'repr called'

repr이랑 str 둘다 문자열을 정의할 때? 호출 되는거

print, str하면 str이 호출 되고 repr을 호출하면 repr

일반적인 문자열 출력은 str이 호출. repr정의 했을 때만 repr메서드호출

>>> repr('abc')

"'abc'"

>>> str('abc')

'abc'

>>> repr('abc')

"'abc'"

>>> str('abc')

'abc'

>>> repr(2)

'2'

>>> repr('2')

"'2'"

>>> repr([1,2,3])

'[1, 2, 3]'

>>> str(2)

'2'

>>> str('2')

'2'

>>> l=[2,'2']

>>> str(l)

"[2, '2']"

>>> repr(l)

"[2, '2']"

>>> str(l)==repr(l)

True

repr은 좀 더 문자열 보존하는 형태?

str은 원형에 가까운?

컨테이너자료형(리스트와사전등)의\_\_str\_\_() 메서드는내부객체의\_\_repr\_\_()메서드를사용한다.?????????

repr 없을 때 str이 repr을 대신하지 않음.

근데 str이 없을 떄는 repr이 대신함

hash정의 할 떄는 \_\_eq\_\_도 같이 정의해야함

class GetAttr3( object ):

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

print( '\_\_getattr\_\_', x )

raise AttributeError

def \_\_getattribute\_\_( self, x ):

print( '\_\_getattribute\_\_ called..', x )

return object.\_\_getattribute\_\_( self, x )

여기서 마지막에 object.\_\_getaasdsadas은 재귀호출 막을라고 최상의 객체 object를 쓴건데 여기선 왜 되는거지??????????라며 의아해하심

class Attr:

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

print('\_\_setattr\_\_(%s)=%s called'%(name,value))

object.\_\_setattr\_\_(self,name,value)

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

print('\_\_delattr\_\_(%s)called'%name)

object.\_\_delattr\_\_(self,name)

실제 설정, 삭제가 이뤄지게끔 object에서 함

call이 있으면 인스턴스 호출 방식을 함수 호출 방식으로 가능

호출 가능한지 아닌지 알려면 해당 클래스를 열어서 call이 있는지 봐야하는데 그러면 불편하니깐 import colltions 하고 isinstance(s,collections.Callabe) 하면 호출가능하면 true나옴

\_\_new\_\_()

생성과 호출 전에 동작함(\_\_init\_\_()은 생성 이후에)

객체의 생성을 담당하는 메서드

init이라는 생성자 보다도 먼더 동작

class NewTest:

def \_\_new\_\_(cls,\*args,\*\*kw):

print('\_\_new\_\_called',cls)

instance=object.\_\_new\_\_(cls)

return instance

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

print('\_\_init\_\_called',self)

new와 init의 차이는 self.

self는 인스턴스 객체가 생성된 이후부터 사용할 수 있음

인스턴스객체를 생성하는 생성자

슈퍼클래스는 생성자가 있어여 하는데 없는 경우도 있음 대신 new로 연산자 중복해도 초기화 동작을 new 안에서 해주면 되기 때문에 초기화가 가능한 장점이 생긴다

class Singleton:

\_\_instance=None

def \_\_new\_\_(cls,\*args,\*\*kw):

if cls.\_\_instance is None:

cls.\_\_instance=object.\_\_new\_\_(cls)

return cls.\_\_instance

인스턴스 개수가 1개로 제한되어야 할 때 사용하는 singleton

?????????????

데코레이터 – 함수 클로저 ㅣ 정수 리턴? 정보은닉

@부여해서 해당내용이 어떤의미인지 컴파일러에 알려주는(자바-이노테이션)

@래퍼? @이용해서 장식자 만들 수도 있음.

정적 메서드- 인스턴스 생성 안하고 호출 할 수 있는

static메서드는 self가 없음

>>> class D:

@staticmethod

def add(x,y):

return x+y

D.add(1,2)

클래스 메서드- 첫 인수로 클래스 객체를 전달 받는 것 빼고는 static메서드랑 같음

@classmethod

instance생성과 무관하게 클래스에서 공통적으로 사용되어야할 때? class method나 staticmethod 사용

property쓰면 속성에 접근하는 것처럼 메서드에 접근 가능

>>> class PropertyClass:

defget\_deg(self):

return self.\_\_deg

defset\_deg(self,d ):

returnself.\_\_deg=d%360

deg= property( get\_deg, set\_deg)

>>>p=PropertyClass()

>>> p.deg= 390; p.deg

>>>p.deg= -370; p.deg

장식자를이용한방법

>>> class PropertyClass:

@property#getter메서드를등록한다.

def get\_deg( self ):

return self.\_\_deg

@property# deg의setter 메서드를등록한다.

def set\_deg( self, d ):

return self.\_\_deg= d % 360

>>> p = PropertyClass()

>>> p.deg= 390; p.deg

>>> p.deg= -370; p.deg

클래스 관계

연관 관계, 일반화 관계

연관 관계는 지금까지 클래스들 정의하고 크래스들 간에 관계성.

일반화 관계는 is-a관계 추상적인 개념을 공통적으로 가지고 있는 것 플러스 나만의 뭔가가 있는거?????????/ 나는 사람이다. 일반화 관계- 상속

직원-사무직, 영업직,생산직 공통 정복 k있다 공통적인것들 묶어서 일반화를 시켜주는거임

일관성을 유지할 수 있음. 상속은 재사용성에 있다

추가 기능만을 정의하거나 기존 기능을 변경해서 새로운 클래스 만들면됨

has-a관계(aggregation 관계)는 상속으로 표현하면 안됨

상속 안해도 object는 무조건 자동 상속

person과 employee의 인스턴스는 따로 생성됨

따로따로 만들어지는데 연결되어있다고 생각

p=Person() #Person생성

e=Emplyee() #Person먼저 생성하고 Employee객체 생성

>>> class Person:

def \_\_init\_\_(self,name,phone=None):

self.name=name

self.phone=phone

def \_\_repr\_\_(self):

return '<Person {}{}>'.format(self.name, self.phone)

>>> class Employee(Person):

def \_\_init\_\_(self,name,phone,position,salary):

Person.\_\_init\_\_(self,name,phone)

self.position=position

self.salary=salary

>>> m1=Employee('hong',5567,'daeri',200)

>>> m2=Employee('king',1237,'hwo',300)

>>> print(m1.name,m1.position)

hong daeri

>>> print(m2.name,m2.position)

king hwo

>>> print(m1)

<Person hong5567>

>>> print(m2)

<Person king1237>

print하면 person꺼가 나오는게 repr이 person에 정의되어있어서 print하는게

>>> class Employee(Person):

def \_\_init\_\_(self,name,phone,position,salary):

Person.\_\_init\_\_(self,name,phone)

self.position=position

self.salary=salary

def \_\_repr\_\_(self):

return '<Employee {}{}{}{}>'.format(self.name,self.phone,self.position,self.salary)

이렇게하면 employee꺼가 나옴 method overriding(재정의). repr을 overriding

>>> class Employee(Person):

def \_\_init\_\_(self,name,phone,position,salary):

Person.\_\_init\_\_(self,name,phone)

self.position=position

self.salary=salary

def \_\_repr\_\_(self):

#return '<Employee {}{}{}{}>'.format(self.name,self.phone,self.position,self.salary)

s=super().\_\_repr\_\_()

return s+'<Employee {} {}>'.format(self.position, self.salary)

>>> m1=Employee('hong',5567,'daeri',200)

>>> print(m1)

<Person hong5567><Employee daeri 200>이렇게 할 수도 있음

중복정의는 overloading(여러 개 존재 가능)

재정의는 overriding(한 개만 존재 가능)

오버라이드 하면 파생클래스꺼 사용

추상 클래스는 인스턴스 생성이 안되는 클래스

추상 매서드는 구현이 없는 매서드

추상 매서드는 반드시 상속 받은 쪽에서 구현을 해줘야함

파이썬 클래스의 모든 메서드는 가상함수.

가상함수: 실행시에 호출할 함수 결정 (Dynamic Binding) 동적으로 함수 호출을 연결하겠다

abstract가 안붙으면 static

붙으면 dynamic binding

자바에서 강제하고 싶을 때 쓰는게 인터페이스

예를 들면 스피커 달리면 볼륨키 달아야해 라고 무조건 결정하고 싶으면 인터페이스로 임플리면트 받아야함

파이썬은 오버라이드 하고 있다면 자동 호출 없으면 슈퍼 클래스꺼 호출

>>> class BAse:

def f(self):

self.g()

def g(self):

print('Base')

>>> class Derived(BAse):

def g(self):

print('Derived')

>>> b=BAse()

>>> b.f()

Base

>>> a=Derived()

>>> a.f

<bound method BAse.f of <\_\_main\_\_.Derived object at 0x000000000365D390>>

>>> a.f()

Derived