

# PYTHON PROPERTY / DESCRIPTOR 이해하기

Moon Yong Joon

# PROPERTY

Moon Yong Joon



# Property 처리

# Property class 이해

a special descriptor object를 생성하는 class로  
내부에 getter, setter, deleter 메소드를 가지고  
있음

```
a = property()  
print(a)  
print(a.getter)  
print(a.setter)  
print(a.deleter)
```

```
<property object at 0x0000000055D2D18>  
<built-in method getter of property object at 0x0000000055D2D18>  
<built-in method setter of property object at 0x0000000055D2D18>  
<built-in method deleter of property object at 0x0000000055D2D18>
```

# Creating Property- 객체 직접 정의

인스턴스 객체의 변수 접근을 메소드로 제약하기 위해서는 Property 객체로 인스턴스 객체의 변수를 Wrapping 해야 함

```
class P:
    def __init__(self,x):
        self._x = x

    def getx(self) :
        return self._x
    def setx(self, x) :
        self._x = x
    def delx(self) :
        del self._x

    x = property(getx,setx,delx," property test ")

p1 = P(1001)
print(id(p1.x))

print(p1.x)
p1.x = -12
print(p1.x)
print(p1.__dict__)
```

property(fget=None,  
fset=None,  
fdel=None,  
doc=None)

---

92626992  
1001  
-12  
{ '\_x': -12 }



# Decorator 처리

# Decorator vs. 직접 실행

property로 직접 실행하거나 decorator를 사용  
하나 동일한 처리를 함

```
class P1:
    def __init__(self, foo) :
        self._foo = foo

    @property
    def foo(self):
        return self._foo

class P2:
    def __init__(self, foo) :
        self._foo = foo

    def foo(self):
        return self._foo
    foo = property(foo)

p1 = P1("foo")
p2 = P2("foo")

print(p1.foo)
print(p2.foo)

foo
foo
```

# Creating Property decorator

인스턴스 객체의 변수 접근을 메소드로 제약하기 위해서는 Property 객체로 인스턴스 객체의 변수를 Wrapping 해야 함

```
class P:

    def __init__(self,x):
        self._x = x

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, x):
        self._x = x

    @x.deleter
    def x(self):
        del self._x

p1 = P(1001)
print(id(p1.x))
print(p1.x)
|
p1.x = -12
print(p1.x)
print(p1.__dict__)
```

```
92627024
1001
-12
{'_x': -12}
```



# DESCRIPTOR

Moon Yong Joon



# Descriptor protocol

# Descriptor란

`__get__`, `__set__`, `__delete__` 인 descriptor protocol를 정의해서 객체를 접근하게 처리하는 방식

```
Class A() :  
    name = descriptor(...)
```

```
Class descriptor() :  
    def __init__(...)  
    def __get__(...)  
    def __set__(...)  
    def __delete__(...)
```

name 속성 접근시 실제 descriptor 내의  
`__get__`/`__set__`/`__delete__` 이 실행되어 처리  
됨

# Descriptor 메소드

Descriptor는 특성 객체의 속성 접근 시 먼저 그 속성의 특징을 체크하여 처리할 수 있는 방법으로 기술자 프로토콜로 제지하면서 처리는 "바인딩 행동"을 가진 메소드들로 구성

```
__get__(self, instance, owner),  
__set__(self, instance, value),  
__delete__(self, instance).
```

# Descriptor 메소드 정의

Descriptor 처리를 위해 별도의 Class를 정의  
시에 추가해야 할 메소드

검색

```
obj.__get__(self, instance, owner)
```

생성/변경

```
obj.__set__(self, instance, value)
```

소멸

```
obj.__delete__(self, instance)
```

# Descriptor 메소드 파라미터

별도의 descriptor 와 실체 class 인스턴스 간의  
실행환경을 연계하여 처리

```
_get_(self, instance, owner),  
_set_(self, instance, value),  
_delete_(self, instance)
```

Self: descriptor 인스턴스  
Instance: 실체 class의 인스턴스  
Owner: 실체 class의 타입  
Value : 실체 class의 인스턴스의 변수  
에 할당되는 값

# Descriptor 사용 이유

- 구현 class에 대한 getter/setter/deleter 함수를 별도 관리
- 구현시 메소드 정의 없이 변수로 처리로 대처

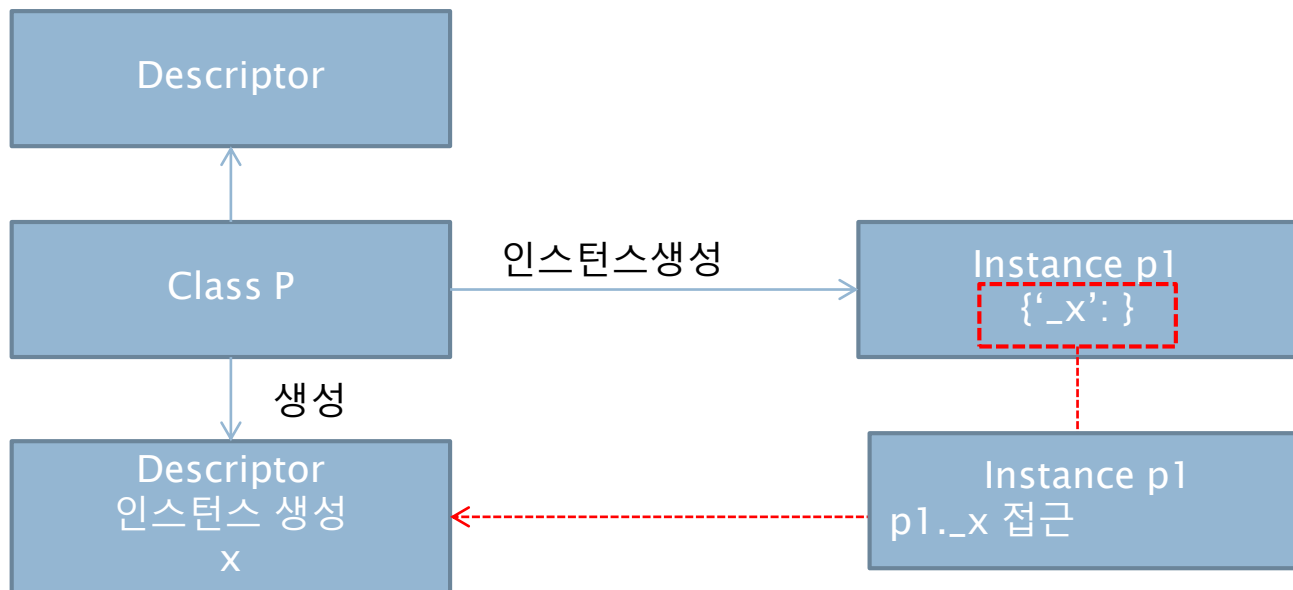


# Descriptor 이해



# Descriptor

Descriptor 클래스를 이용해 인스턴스 객체의 변수 명으로 처리할 수 있도록 만듦



class 내 descriptor 인스턴스의 메소드 호출하여 처리

# Descriptor 처리 방식

Descriptor class를 생성하여 실제 구현 클래스 내부의 속성에 대한 `init(no 변수)/getter/setter/deleter`를 통제할 수 있도록 구조화

Descriptor class 생성

```
Class Descriptor :  
    def __init__  
    def __get__  
    def __set__  
    def __del__
```

구현 class 정의시 속성에  
대한 인스턴스 생성

```
class Person() :  
    name = Descriptor()
```

구현 class에 대한 인스턴스  
생성 및 인스턴스 속성에  
값 세팅

```
user = Person()  
User.name = 'Dahl'
```

# Descriptor class 정의

## 클래스에 생성자 및 get/set/delet 메소드 정의

```
class Descriptor(object) :
    def __init__(self):
        self.name = '_name'

    def __get__(self, instance, owner):
        return instance.__dict__['_name']

    def __set__(self, instance, value):
        instance.__dict__['_name'] = value

    def __delete__(self, instance):
        del instance.__dict__['_name']

class A :
    name = Descriptor()

a = A()
a.name = 'Dahl'
print(a.name)
```

Dahl

# Descriptor : binding behavior

실제 인스턴스에는 `_x`가 생기고 모든 조회 및 변경은 descriptor class를 이용해서 처리

```
class D(object) :
    def __init__(self, name) :
        self._x = "_" + name

    def __get__(self, instance, owner):
        return instance.__dict__['_x']

    def __set__(self, instance, value):
        instance.__dict__['_x'] = value
```

```
class D1() :
    x = D("x")
    print(x)
```

```
d = D1()
d.x = 10
```

```
print(d.__dict__)
print(d.x)
```

```
print(" Class binding call ",D.__get__(D('x'),d,D1))
```

```
<__main__.D object at 0x0000000058ADC18>
{'_x': 10}
10
Class binding call 10
```



# 타 클래스 메소드 처리

# 사용자 디스크립터 정의

사용자 정의 클래스에 `__get__`을 선언해서 타 클래스 메소드를 실행하도록 정의

```
class P(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None):
        self.fget = fget

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)
```

# 타 클래스 메소드 처리 : 실행

호출될 클래스 정의 및 하고 descriptor 클래스에 메소드를 전달.

```
class A() :  
    def __init__(self, name) :  
        self._name = name  
  
    def name(self) :  
        return self._name
```

```
a = A('dahl')  
p = P(A.name)  
print(p.__dict__)
```

```
print(p.__get__(a,A))
```

```
{'fget': <function A.name at 0x00000000055DC620>}  
dahl
```

int 타입의 디스크립터 예시



# int.\_\_add\_\_ : descriptor

Int 클래스 내의 \_\_add\_\_ 메소드는 descriptor로 정의 되어 있음

```
type(int.__add__)  
wrapper_descriptor
```

```
dir(int.__add__)  
['__call__',  
 '__class__',  
 '__delattr__',  
 '__doc__',  
 '__format__',  
 '__get__',  
 '__getattr__',  
 '__hash__',  
 '__init__',  
 '__name__',  
 '__new__',  
 '__objclass__',  
 '__reduce__',  
 '__reduce_ex__',  
 '__repr__',  
 '__setattr__',  
 '__sizeof__',  
 '__str__',  
 '__subclasshook__']
```

# int.\_\_add\_\_.\_\_get\_\_

Method descriptor는 `__get__(self, instance, owner)` 가지고 있어 처리됨

```
print(int.__add__)  
  
s = int.__add__.__get__(1,int)  
print(s)  
print(s(0))  
print(s(3))  
  
help(int.__add__.__get__)
```

```
<slot wrapper '__add__' of 'int' objects>  
<method-wrapper '__add__' of int object at 0x0000000061E101F0>  
1  
4  
Help on method-wrapper object:  
  
__get__ = class method-wrapper(object)  
    Methods defined here:
```

# int.\_\_add\_\_.\_\_get\_\_ 예시

get을 통해 인스턴스를 생성된 인스턴스를 가져와서 \_\_add\_\_를 처리

```
p_int = 1

print(p_int.__add__(3))
#특정 인스턴스에서 __get__으로 __add__ 메소드를 접근
print((3).__add__)
print(type(p_int).__add__.__get__(3,int))
# 3+ 1를 더함
print(type(p_int).__add__.__get__(3,int)(p_int))
```

```
4
<slot wrapper '__add__' of 'int' objects>
<method-wrapper '__add__' of int object at 0x0000000061E10230>
<method-wrapper '__add__' of int object at 0x0000000061E10230>
4
```



# 데이터 디스크립터 예시

# 1. Descriptor 클래스 정의

Descriptor 인스턴스는 실제 변수명을 관리하는 것으로 만들고 `__get__`/`__set__`으로 변수명을 접근과 갱신 처리

```
class Des(object) :
    def __init__(self,name) :
        self.name = '_' + name

    def __get__(self,instance,owner) :

        return getattr(instance,self.name,"")

    def __set__(self,instance,value) :
        setattr(instance,self.name, value)
```

## 2. 사용 클래스 정의

사용되는 클래스의 변수에 descriptor 인스턴스 객체를 생성

```
class P(object) :  
    name = Des("name")  
    print("name ",name.__dict__)
```

# 3. 사용 클래스의 인스턴스 생성

인스턴스를 만들고 클래스 변수 `name`을 사용하면 `descriptor`가 작동됨

```
c = P()

print(c.name)
c.name = "dahl"
print(c.name)

('name ', {'name': '_name'})

dahl
```



Descriptor : 직접 멤버 접근



# Descriptor : 상속 구현

descriptor 클래스를 상속해서 처리

```
class D(object) :
    def __init__(self, x) :
        self.x = x

    def __get__(self, instance=None, cls=None) :
        return self.x

    def __set__(self, instance, value) :
        self.x = value

class D1(D) :
    def __init__(self, x) :
        D.__init__(self, x)

d = D(1)
print " d", d.__dict__
print " "
print " dircet call ", d.x
d.x = 2
print " d", d.__dict__
print " instance call", d.__get__()
print " Class binding call ", D.__get__(d, d)
print type(d)
print " class binding", type(d).__get__(d, d)
```

d {'x': 1}

dircet call 1

d {'x': 2}

instance call 2

Class binding call 2

<class '\_\_main\_\_.D'>

class binding 2

\_\_set\_\_을 구현으로 d.x에 갱신이 가능

# Descriptor : 한번 사용

Descriptor 클래스를 생성해서 한 개의 변수 처리만 처리하는 방법

```
class Descriptor(object):
    def __init__(self):
        self._name = ''
    def __get__(self, instance, owner):
        print "Getting: %s" % self._name
        return self._name
    def __set__(self, instance, name):
        print "Setting: %s" % name
        self._name = name.title()
    def __delete__(self, instance):
        print "Deleting: %s" % self._name
        del self._name

class Person(object):
    name = Descriptor()
    |
user = Person()
user.name = 'john smith'
user.name

del user.name
```

```
Setting: john smith
Getting: John Smith
Deleting: John Smith
```



# 내장함수로 타 객체 접근

# getattr 함수

내장함수 (getter )을 이용해서 처리하는 방법  
x.a로 조회하는 것과 같음

```
help(getattr)
```

Help on built-in function getattr in module `__builtin__`:

```
getattr(...)
```

```
    getattr(object, name[, default]) -> value
```

Get a named attribute from an object; getattr(x, 'y') is equivalent to x.y.  
When a default argument is given, it is returned when the attribute doesn't  
exist; without it, an exception is raised in that case.

```
class A() :  
    def __init__(self,name) :  
        self.name = name  
|  
a = A("dahl")  
print(getattr(a, 'name', None))
```

dahl

# setattr 함수

내장함수 (setter )을 이용해서 처리하는 방법  
x.a = 1로 갱신하는 것과 같음

```
help(setattr)
```

Help on built-in function setattr in module `_builtin_`:

```
setattr(...)  
setattr(object, name, value)
```

Set a named attribute on an object; setattr(x, 'y', v) is equivalent to  
`x.y = v`.

```
class A() :  
    def __init__(self,name) :  
        self.name = name  
  
a = A("dahl")  
print(getattr(a,'name',None))  
setattr(a,'age',50)  
print(getattr(a,'age',None))
```

```
dahl  
50
```

# delattr 함수

내장함수 (delattr)을 이용해서 처리하는 방법  
del x.a 으로 삭제하는 것과 같음

```
help(delattr)
```

Help on built-in function delattr in module `__builtin__`:

```
delattr(...)  
    delattr(object, name)
```

Delete a named attribute on an object; `delattr(x, 'y')` is equivalent to `del x.y`.

```
class A() :  
    def __init__(self, name) :  
        self.name = name  
  
a = A("dahl")  
print(getattr(a, 'name', None))  
setattr(a, 'age', 50)  
print(getattr(a, 'age', None))  
print(a.__dict__)  
delattr(a, 'age')  
print(a.__dict__)
```

```
dahl  
50  
{'age': 50, 'name': 'dahl'}  
{'name': 'dahl'}
```

# Descriptor : 여러번 사용 1

Descriptor 클래스 내의 name 변수를 실제 생성되는 변수명 관리로 변경

```
class Descriptor(object):
    def __init__(self):
        self._name = ''
    def __get__(self, instance, owner):
        print "Getting: %s" % self._name
        return self._name
    def __set__(self, instance, name):
        print "Setting: %s" % name
        self._name = name.title()
    def __delete__(self, instance):
        print "Deleting: %s" % self._name
        del self._name
```

```
class Person(object):
    name = Descriptor()
    |
user = Person()
user.name = 'john smith'
user.name

del user.name
```

Setting: john smith  
Getting: John Smith  
Deleting: John Smith



```
class Descriptor(object):
    def __init__(self, name):
        self.name = '_' + name
    def __get__(self, instance, owner):
        x = getattr(instance, self.name)
        print "Getting: %s" % x
        return x
    def __set__(self, instance, value):
        print "Setting: %s" % value
        setattr(instance, self.name, value)
    def __delete__(self, instance):
        print "Deleting: %s" % getattr(instance, self.name)
        delattr(instance, self.name)
```

```
class Person(object):
    name = Descriptor("name")
```

```
user = Person()
print user.__dict__

user.name = 'john smith'
print user.__dict__
user.name

del user.name
```

```
{}
```

Setting: john smith  
{'\_name': 'john smith'}  
Getting: john smith  
Deleting: john smith

# Descriptor : 여러번 사용 2

별도의 descriptor 클래스를 사용해서 구현

```
class D(object) :
    def __init__(self, name) :
        self.name = "_" + name

    def __get__(self, instance, owner) :
        return getattr(instance, self.name)
    def __set__(self, instance, value) :
        setattr(instance, self.name, value)
```

descriptor class를 구현함

```
class D1(object) :
    def __init__(self) :
        D1.x = D("x")

d = D1()
print " d : ", d.__dict__
d.x = 1
print d.__dict__
print " direct call", d.x

d1 = D1()
print " d1 :", d1.__dict__
d1.x = 2
print d1.__dict__
print " direct call", d1.x
```

```
d : {}
{'_x': 1}
direct call 1
d1 : {}
{'_x': 2}
direct call 2
```

getattr/setattr는 실제 D1  
내의 \_\_dict\_\_에서 관리 됨



# Descriptor : 여러번 사용 3

사용자 정의 내의 변수로 getter, setter 처리를  
변수명으로 처리하도록 하는 방법

```
class D(object) :  
    def __init__(self, name) :  
        self.name = '_' + name  
  
    def __get__(self, instance, owner) :  
        return getattr(instance, self.name, None)  
    def __set__(self, instance, value) :  
        setattr(instance, self.name, value)  
  
class Person(object) :  
    name = D('name')  
  
p = Person()  
p.name = 'dahl'  
print p.__dict__  
print "class binding ", Person.__dict__['name'].__get__(p, Person)  
print "direct binding ", p.name  
  
{ '_name': 'dahl' }  
class binding dahl  
direct binding dahl
```

Descriptor class  
를 정의

Descriptor 인스턴  
스를 통해 \_name  
변수가 생김



# Binding descriptor

# 처리절차: 1.Descriptor 정의

별도의 클래스에 `__get__`/`__set__`/`__delete__` 메소드를 정의하고 class 내에 실제 descriptor로 인스턴스를 생성함

```
import inspect

class TypedProperty(object):


    def __init__(self, name, type, default=None):
        self.name = "_" + name
        self.type = type
        self.default = default if default else type()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Must be a %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        raise AttributeError("Can't delete attribute")
```

```
: class Person(object):
    name = TypedProperty("name", str)
    age = TypedProperty("age", int, 42)
```



# 처리절차 : 2. 세부 정의 및 실행

Class가 관리하는 영역에 name과 age 객체가 생성 되어 있음

```
class Person(object):
    name = TypedProperty("name",str)
    age = TypedProperty("age",int,42)

acct = Person()

print('method descriptor', inspect.isdatadescriptor(TypedProperty))
acct.name = "obi"
acct.age = 1234

print "acct __dict__ ", acct.__dict__
print "Person __dict__ ", Person.__dict__
```

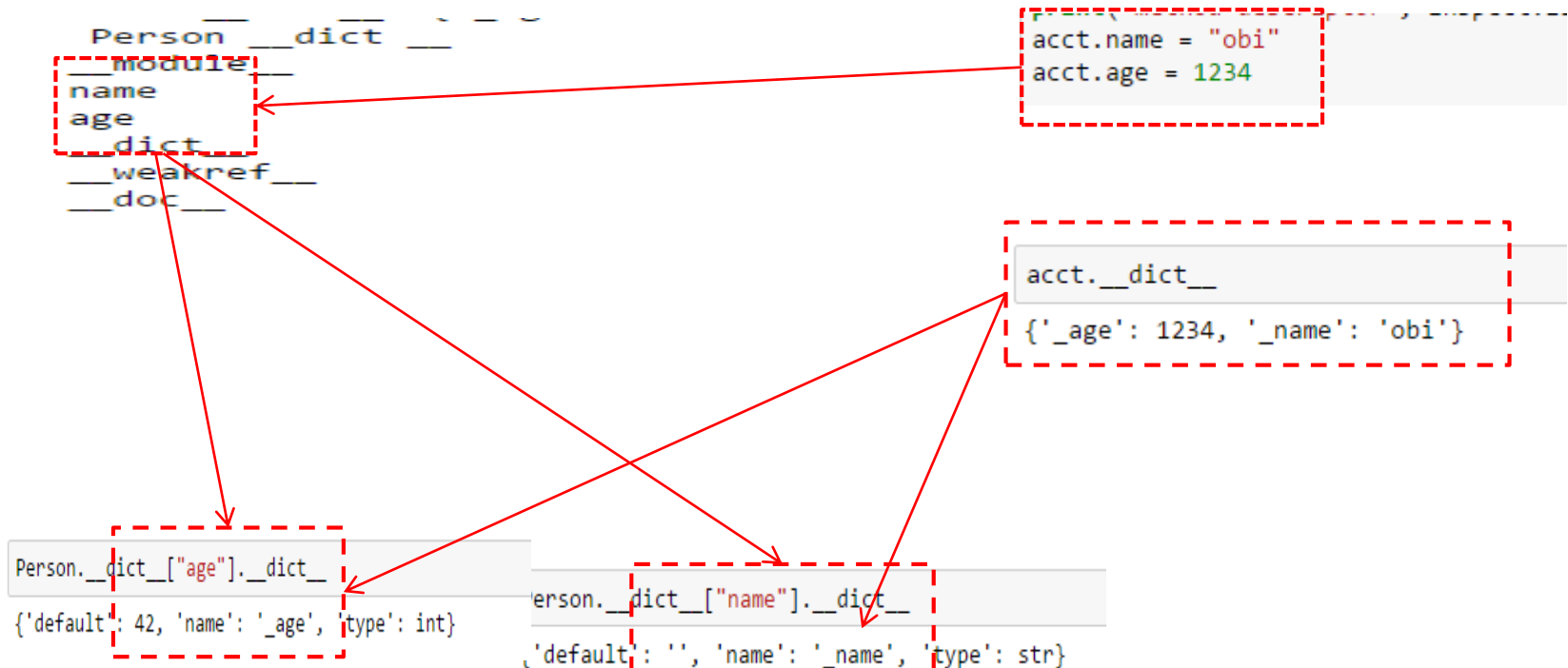
```
('method descriptor', True)
acct __dict__ {'age': 1234, 'name': 'obi'}
Person __dict__ {'__module__': '__main__', 'name': <__main__.TypedProperty object at 0x7fb3a02a54d0>, 'age': <__main__.TypedProperty object at 0x7fb3a02787d0>, '__dict__': <attribute '__dict__' of 'Person' objects>, '__weakref__': <attribute '__weakref__' of 'Person' objects>, '__doc__': None}
```



# Descriptor 세부 설명

# Descriptor 실행 구조

Descriptor 생성시 instance 변수가 클래스 내부에 객체로 만들어 실행시 객체의 메소드들이 실행됨



# Descriptor 실행 구조 : 흐름 1

Descriptor 를 이용해서 Person 클래스 내에 name과 age 객체 생성

```
class Person(object):  
    name = TypedProperty("name",str)  
    age = TypedProperty("age",int,42)
```

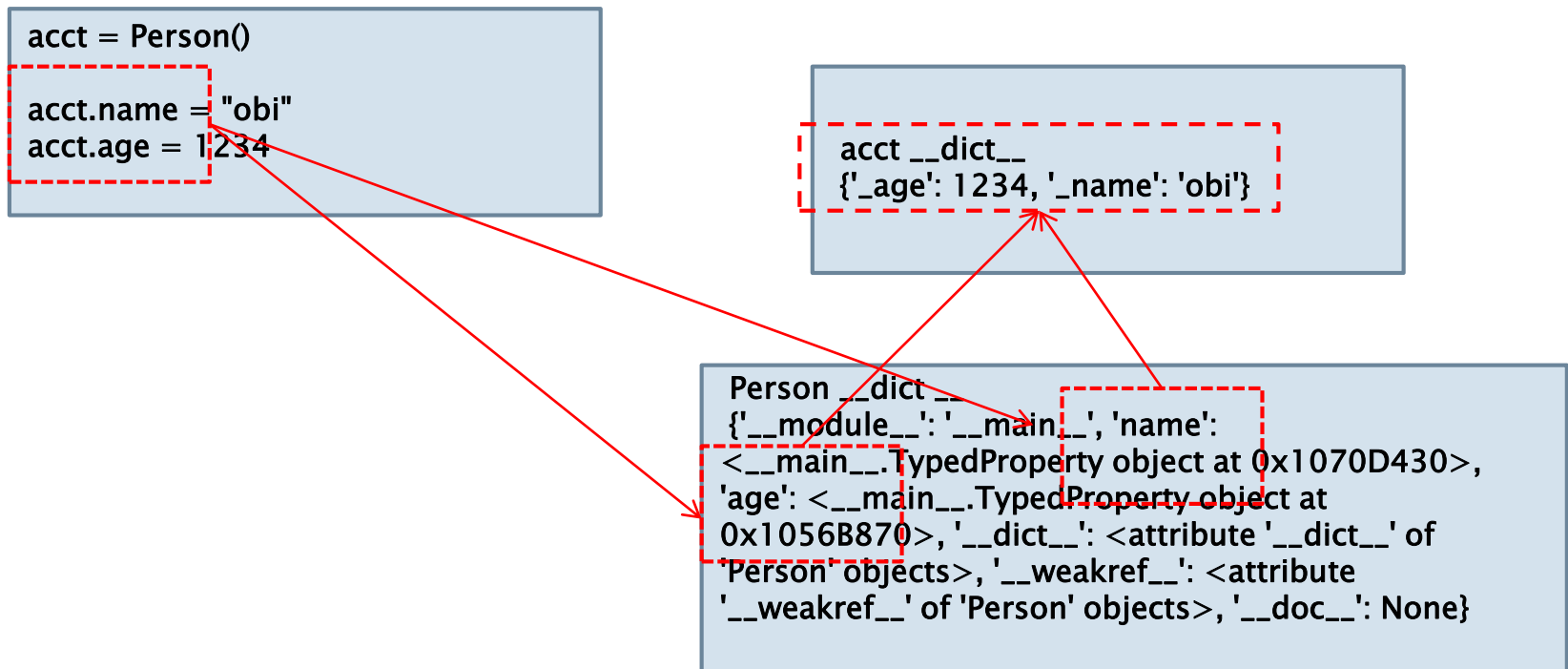
```
acct = Person()
```

```
acct __dict__  
{}
```

```
Person __dict__  
{'__module__': '__main__', 'name':  
<__main__.TypedProperty object at 0x1070D430>,  
'age': <__main__.TypedProperty object at  
0x1056B870>, '__dict__': <attribute '__dict__' of  
'Person' objects>, '__weakref__': <attribute  
'__weakref__' of 'Person' objects>, '__doc__': None}
```

# Descriptor 실행 구조 : 흐름 2

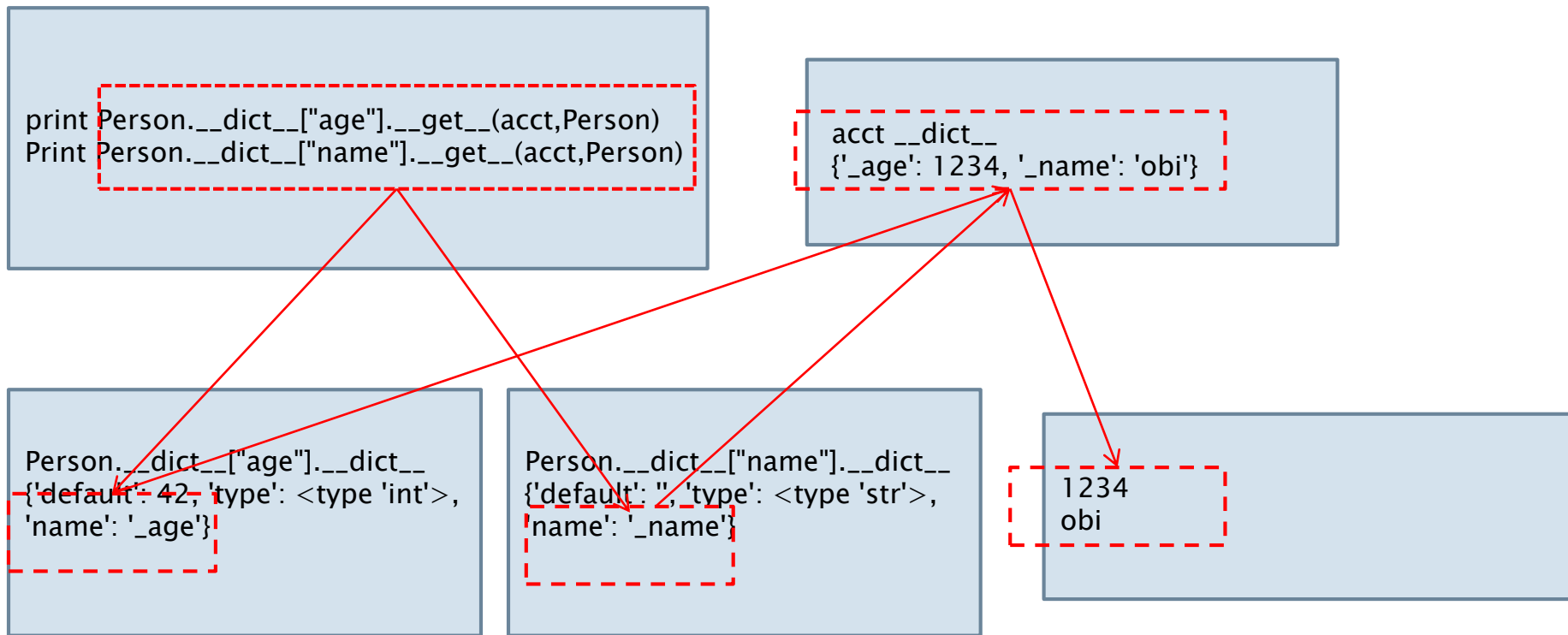
Person 클래스의 인스턴스 내부 변수에 값을 할당하면 `__set__` 메소드를 이용해서 인스턴스 값 생성





# Descriptor 실행 구조 : 흐름 3

acct.age/acct.name 호출하면 Person.\_\_dict\_\_["인스턴스변수명"].\_\_get\_\_(acct,Person) 가 실행되어 결과값을 조회



# 사용자 정의 PROPERTY 만들기

Moon Yong Joon



# Property 생성자

# Property class

파이썬은 Property는 하나의 class이고 이를 데코레이터나 인스턴스로 처리해서 사용

```
help(property)
```

Help on class property in module \_\_builtin\_\_:

```
class property(object)
    property(fget=None, fset=None, fdel=None, doc=None) -> property attribute

    fget is a function to be used for getting an attribute value, and likewise
    fset is a function for setting, and fdel a function for del'ing, an
    attribute. Typical use is to define a managed attribute x:

    class C(object):
        def getx(self): return self._x
        def setx(self, value): self._x = value
        def delx(self): del self._x
        x = property(getx, setx, delx, "I'm the 'x' property.")

    Decorators make defining new properties or modifying existing ones easy:

    class C(object):
        @property
        def x(self):
            "I am the 'x' property."
            return self._x
        @x.setter
        def x(self, value):
            self._x = value
        @x.deleter
        def x(self):
            del self._x
```

# Property : 생성자

생성자에 fget, fset, fdel, doc 파라미터를 받고 처리하도록 정의

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

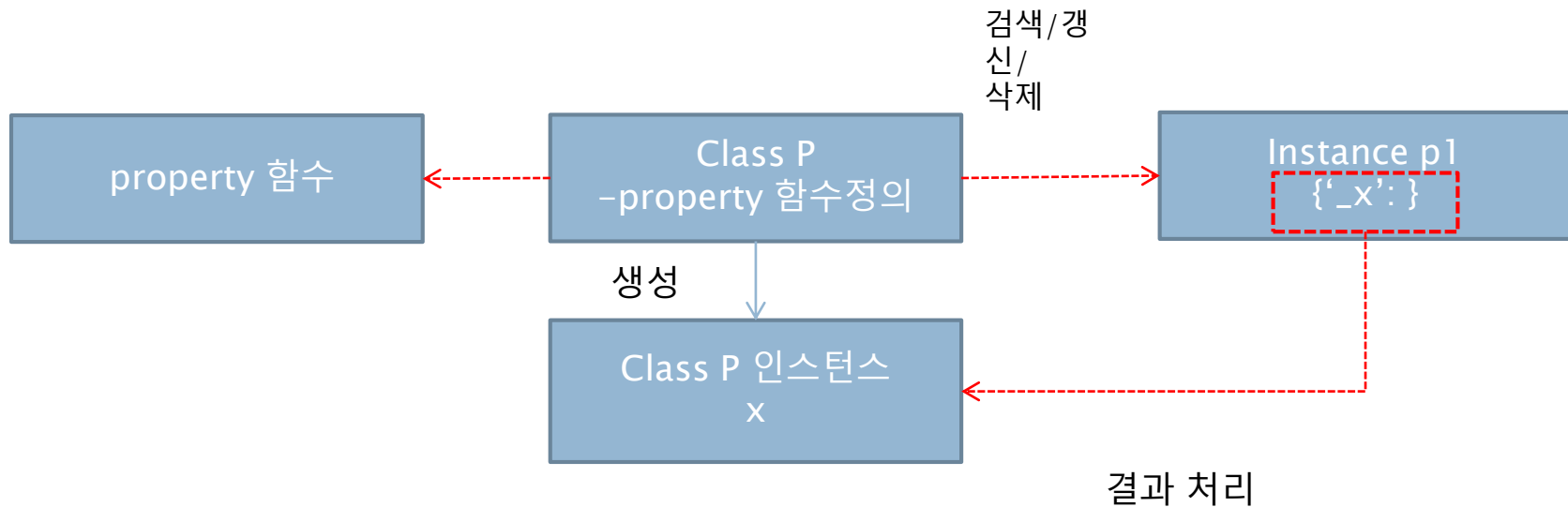
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc
```



# Descriptor 메소드

# Property : descriptor 흐름

파이썬은 Property 함수를 이용해서 인스턴스 객체의 변수 명과 동일하도록 처리



# Property : descriptor 메소드 정의

생성자에 `__get__`, `__set__`, `__delete__` 메소드 정의

```
def __get__(self, obj, objtype=None):
    if obj is None:
        return self
    if self.fget is None:
        raise AttributeError("unreadable attribute")
    return self.fget(obj)

def __set__(self, obj, value):
    if self.fset is None:
        raise AttributeError("can't set attribute")
    self.fset(obj, value)

def __delete__(self, obj):
    if self.fdel is None:
        raise AttributeError("can't delete attribute")
    self.fdel(obj)
```





# Property 내장 메소드

# Property class 내구 구조 이해

메소드 정의 후에 property에 등록하면 fget, fset, fdel 속성에 메소드가 할당됨

```
def getter(self):
    print('Get!')
def setter(self, value):
    print('Set to {!r}!'.format(value))
def deleter(self):
    print('Delete!')

prop = property(getter, setter, deleter)

print(prop.fget is getter)
print(prop.fset is setter)
print(prop.fdel is deleter)

prop1 = property()
prop1 = prop1.getter(getter)
print(prop1.fget is getter)
prop1 = prop1.setter(setter)
print(prop1.fset is setter)
prop1 = prop1.deleter(deleter)
print(prop1.fdel is deleter)
```

True  
True  
True  
True  
True  
True

# Property : 내부 메소드 정의

생성자에 getter, setter, deleter 메소드 정의해서 Property 생성을 의한 메소드 정의

```
def getter(self, fget):  
    return type(self)(fget, self.fset, self.fdel, self.__doc__)  
  
def setter(self, fset):  
    return type(self)(self.fget, fset, self.fdel, self.__doc__)  
  
def deleter(self, fdel):  
    return type(self)(self.fget, self.fset, fdel, self.__doc__)
```



# 실행 예시

# Property 로 사용자 클래스 정의

사용자 클래스 A를 선언하고 name에 데코레이터로 정의해서 실행

```
class A() :  
    def __init__(self, name) :  
        self._name = name  
  
    @Property  
    def name(self) :  
        return self._name  
  
a = A('dahl')  
print(a.name)
```

dahl