14. 연사자의 중복과 장식자
14.1 연산자의 중복
- 수치 연산자 중복
연산자 중복이란 프로그램 언어에서 지원하는 연산자에 대해 클래스가 새로운 동작을 하도록 정의하는 것이다. 파이썬에서 사용하는 모든 연산자는 클래스 내에서 새롭게 정의할수 있다.

1. 이항 연산자
파이썬에서는 내장 자료형에 사용하는 모든 연산을 독자가 정의하는 클래스에서 동작하도록 구현할수 있다. 예를들어, 다음 클래스 MyStr을 보자.

```
>>> class MyStr:
...     def __init__(self, s):
...             self.s = s
...
>>>
```

여기에 현재는 가능하지 않은 나눗셈 연산이 가능하도록 해보자. __truediv__() 메서드만 추가하면 된다. 나누기(/) 연산자는 __truediv__() 메서드로 확장된다. 예를 들면 s1 / ':' 에 의해 s1.__trudiv__(':')가 호출된다.

```
>>> class MyStr:
...     def __init__(self, s):
...             self.s = s
...     def __truediv__(self, b):
...             return self.s.split(b)
...
>>> s1 = MyStr('a:b:c')
>>> s1 / ':'           # 나누기 연산이 가능해졌다. ------------s1.__truediv__(' :' )
['a', 'b', 'c']        # 문자열 리스트
```

이처럼 파이썬은 모든 연산자에 대응하는 적절한 이름의 메서드가 정해져 있어서 연산자가 사용될 때 해당 메서드로 확장된다. 표는 수치 연산에 적용되는 주요 연산자에 대한 메서드 목록이다.

표. 수치 연산자 메서드

메서드	연산자
__add__(self, other)	+
__sub__(self,other)	-
__mul__(self, other)	*
__truediv__(self, other)	/
__floordiv__(self, other)	//
__mod__(self, other)	%
__divmod__(self, other)	divmod()
__pow__(self, other[,modulo])	pwo() **

__lshift__(self, other)	<<
__rshift__(self, other)	>>
__and__(self, other)	&
__xor__(self, other)	^
__or__(self, other)	|

더하기 연산을 추가해 보자. __add__() 메서드를 추가하면 된다.

```
>>> class MyStr:
...     def __init__(self, s):
...             self.s=s
...     def __truediv__(self, b):
...             return self.s.split(b)
...     def __add__(self, b):
...             return self.s + b
...
>>> s1 = MyStr('a:b:c')
>>> s1 + ':d'           # s1.__add__(' :d' )
'a:b:c:d'
```

2. 역 이항 연산자

앞의 예에서 만일 나누기에서 피연자의 순서가 바뀌었다면 어떤 결과가 나올까?

```
>>> 'a:' + s1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not MyStr
```

문자열 'a:' 이 Mystr형 객체와의 연사을 지원하지 않기 때문에 당연히 TpyeError에러가 발생한다. a+b 연산에서 a.__add__(b)를 우선 시도하고, 이것이 구현되어 있지 않으면 b.__radd__(a)를 시도한다. 이제 __radd__() 메서드를 정의하고 실행해보자.

```
>>> class MyStr:
...     def __init__(self, s):
...             self.s = s
...     def __truediv__(self, b):
...             return self.s.split(b)
...     def __add__(self, b):
...             return self.s + b
...     def __radd__(self, b):
...             return b + self.s
...
>>> s1 = MyStr('a:b:c')
```

```
>>> 'z:' + s1   # --------- s1.__radd__( 'z:' )
'z:a:b:c'
```

이처럼 객체가 연산자의 오른편에 있을 때 사용되는 메서드의 이름은 다음과 같다. 메서드 이름앞에 r이 추가되어 있다.

표. 피연산자가 바뀐 경우의 수치 연산자 메서드

메서드	연산자
__radd__(self, other)	+
__rsub__(self, other)	-
__rmul__(self, other)	*
__rtruediv__(self, other)	/
__rfloordiv__(self, other)	//
__rmod__(self, other)	%
__rdivmod__(self, other)	divmod()
__rpow__(self, other[, modulo])	pow() **
__rrshift__(self, other)	<<
__rrshift__(self, other)	>>
__rand__(self, other)	&
__rxor__(self, other)	^
__ror__(self, other)	|

```
s += b   --->  s.__iadd__(b)
```

아래 표에는 확장 산술 연산자에 대한 중복 메서드가 정리되어 있다.

표. 확장 산술 연산자 메서드

메서드	연산자
__iadd__(self, other)	+=
__isub__(self, other)	-=
__imul__(self, other)	*=
__itruediv__(self, other)	/=
__ifloordiv__(self, other)	//=
__imod__(self, other)	%=
__ipow__(self, other)	**=
__ilshift__(self, other)	<<=
__irshift_(self, other)	>>=
__iand__(self, other)	&=
__ixor__(self, other)	^=
__ior__(self, other)	|=

4. 단항 연산자를 위한 메서드로 준비되어 있다. 다음은 단항 연산자의 중복 메서드에 대한 예이다.

```
-s → s.__neg__()
+s → s.__pos__()
abs(s)  →  s.__abs__()
```

표에는 단항 연산자에 대한 중복 메서드가 정리되어 있다.

표. 단항 연산자 메서드

메서드	연산자
__neg__(self)	-
__pos__(self)	+
__abs__(self)	abs()
__invert__(self)	~비트 반전(0은 1로, 1은 0으로)

이 외에도 파이썬은 여러 연산자를 중복할수 있는 메서드를 준비하고 있다. 다음 연산자는 객체를 인수로 하여 함수 현태로 호출되며, 이에따라 적절한 값을 반환해주어야 한다. 예를들어, int(a)에 의해서 a.__int__() 메서드가 호출된다. 다음은 형변환의 중복 메서드에 대한 예이다.

```
comples(s) → s.__complex__()
int(s) → s.__int__()
float(s)  → s.__float__()
round(s)  →  s.__round__()
operator.index(s) → s.__index__()
```

표에는 형변환에 대한 중복 메서드가 정리되어 있다.

표. 기타 형변환 메서드

메서드	연산자
__complex__(self)	complex()
__int__(self)	int()
__float__(self)	float()
__round__(self)	round()
__index__(self)	operator,index()

여기서 a.__index__() 메서드는 poerator.index(a)에 의해서 호출되는데, 이 메서드는 a를 정수로 변환할 수 있는지 확인해서 정수 값을 반환한다. 만일 a가 정수가 아니면 TypeError 에러가 발생한다. 인덱스로 사용할 자료형은 정수로만 표현해야 하는데 이를 검사하기 위해서 __index__() 메서드가 존재한다.

```
>>> import operator
>>> operator.index(5)
5
>>> operator.index(5.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

예를 들어, bin()와 hex(), oct()와 같은 함수들은 정수 인수가 필요하며 다음과 같이 __index__()메서드가 내부에서 사용된다.

```
operator.index(s)  →  s.__index__()
bin(s)  →  bin(s.__index__())
hex(s)  →  hex(s.__index__())
oct(s)  →  oct(s.__index__())
```

다음은 __index__() 메서드를 사용하는 간단한 예이다.

```
>>> class Index:
...     def __index__(self):
...             print('__index__ called')
...             return 3
...
>>> L = [1, 2, 3, 4, 5]
>>> i = Index()
>>> L[i:]
__index__ called
[4, 5]
>>> bin(i)
__index__ called
'0b11'
>>> oct(i)
__index__ called
'0o3'
>>> hex(i)
__index__ called
'0x3'
```

- 컨테이너 자료형의 연산자 중복

이 절에서는 컨테이너 자료형(시퀀스 자료형 + 매핑 자료형)에 적용할 메서드를 소개한다. 여기서 소개하는 메서드를 적절하게 잘 정의하면, 컨테이너 자료형을 만들 수 있다. 기본적으로는 __len()와 _contains__(), __getitem__(), __setiem__(), __delitem__() 메서드를 구현해야 한다. 만일 독자가 시퀀스형이면서 변경 가능한 자료형을 만들겠다면 append()와 count(), index(), extend(), insert(), pop(), remove(), reverse(), sort() 메서드 등을 구현하면 된다. 사전과 같은 매핑 자료형은 keys()와 values(), items(), get(), clear(), copy(), setdefault(), opo(), pooitem(), update() 같은 메서드를 구현하면 된다. 산술 연산으로는 __add__()와 __radd__(), __iadd__(), __mul__(), _rmul__(), __imul__() 메서드 등을 구현하면 된다. 반복자를 지원하려면 __iter__() 메서드를 구현한다. 여기서는 간단한 경우에 대해서만 설명하기로 한다. 표 14-6은 시퀀스 자료형에 대한 메서드 목록이다.

표. 시퀀스 자료형의 메서드

메서드	연산자
object.__len__(self)	len(object)
object.__contains__(self, item)	item in object
object.__getitem__(self, key)	object[key]
object.__setitem__(self, key, value)	object[key] = vlaue
object.__delitem__(self, key)	del objecd[key]

1. 인덱싱

인덱싱은 시퀀스 자료형에서 순서에 의해서 데이터에 접근하기 위한 방법을 제공한다. 기본적으로 __getitem__() 메서드를 정의해야 한다. 예를 들어, 다음 클래스 Square는 생성자 인수로 범위를 받아들이며, 해당 범위에서 요구된 인덱스 값의 제곱을 반환하는 시퀀스 자료형이다. 시퀀스 자료형에서의 인덱스는 정수 값으로 전달하며, 만일 인덱스 범위를 초과하면 IndexError 에러를 발생해야 한다. 만일 데이터의 자료형이 맞지 않으면 TypeError 에러를 발생한다.

```
>>> class Square:
...     def __init__(self, end):
...             self.end = end
...     def __len__(self):
...             return self.end
...     def __getitem__(self, k):
...             if type(k) != int:
...                     raise TypeError('...')
...             if k < 0 or self.end <= k:
...                     raise IndexError('index {} out of range'.format(k))
...             return k * k
...
>>> s1 = Square(10)
>>> len(s1)             # s1.__len__()
10
>>> s1[4]               # s1.__getitem__(4)
16
>>> s1.__getitem__(4)   # s1[4]
16
>>> s1[20]              # 범위를 벗어난 참조이다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in __getitem__
IndexError: index 20 out of range
>>> s1['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getitem__
```

TypeError: ...

이렇게 만들어진 클래스는 for 문에 적용할 수 있다.

```
>>> for x in s1:
...     print(x, end = ' ')
...
0 1 4 9 16 25 36 49 64 81
```

for 문은 인스턴스 객체 s1의 __getitem__() 메서드를 0부터 호출하기 시작한다. 이 메서드에 의해서 반환된 값이 x에 전달되고, 내부의 print() 함수가 실행된다. IndexError 에외가 발생하면 반복을 중단한다. 인스턴스 객체 s1은 제한된 범위 내에서 시퀀스형 객체로서의 역할을 충실히 수행한다.

자료형 변환을 적용해 보자. __getitem__() 메서드만 정의되어 있으면, 다른 시퀀스 자료형으로 변환하는 것이 가능하다.

```
>>> list(s1)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> tuple(s1)
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

이와 같은 방식의 연산을 지원하면 내장 자료형과 사용범이 같은 일관된 연산을 적용할 수 있고, 이것은 통일성과 편리함을 가져다준다. 사용자가 정의한 클래스가 기존 자료형의 코딩 스타일을 그대로 따르므로 일관된 코딩 스팅을 유지할수 가 있다.

__getitem__() 메서드가 치환 연사잔 오른쪽에서 인덱싱에 의해서 호출되는 메서드라면 __setitem__() 메서드는 치환 연산자 왼쪽에서 호출되는 메서드이다. 예를 들어, s1[0] = 10을 수행하면 s1.__setitem__(0, 10)이 호출된다. 또한, __delitem__() 메서드는 del s1[0]에 의해서 s1.__delitem__(0)이 호출된다.

2. 슬라이싱

슬라이싱에서는 인덱싱에서와 같이 __getitem__()와 __setitem__(), __delitem__() 메서드를 사용하지만 인수로 정수가 아닌 slice 객체를 전달한다.

우선 slice 객체 자체를 살펴보자. 이 객체는 start와 step, stop 세 개의 멤버를 가지는 단순한 객체로 이해하면 된다. 사용하는 형식은 다음과 같다.

```
slice([start,] stop [, stem])
```

slice 객체를 생성해서 몇 가지를 시험해 보자.

```
>>> s = slice(1, 10, 2)
>>> s
slice(1, 10, 2)
```

```
>>> type(s)             # 무슨 자료형인지 알아본다.
<class 'slice'>
>>> s.start, s.stop, s.step
(1, 10, 2)
```

만일 slice 객체를 전달할 때 인수가 생략되면 None 객체를 기본값으로 가진다.

```
>>> slice(10)
slice(None, 10, None)
>>> slice(1, 10)
slice(1, 10, None)
>>> slice(1, 10, 3)
slice(1, 10, 3)
```

슬라이싱 m[1:5]는 m.__getitem__(slice(1,5))를 호출한다. 즉, 인덱싱의 정수 인덱스 대신에 slice 객체가 범위를 나타내는데 사용된다. 확장 슬라이싱 m[1:10:2]는 m.__getitem__(slice(1, 10, 2))를 호출한다. 이제 예제 클래스를 살표보자.

```
>>> class Square:
...     def __init__(self, end):
...             self.end = end
...     def __len__(self):
...             return self.end
...     def __getitem__(self, k):
...             if type(k) == slice:    # slice 자료형인가
...                     start = k.start or 0
...                     stop = k.stop or self.end
...                     step = k.step or 1
...                     return map(self.__getitem__, range(start, stop, step))
...             elif type(k) == int:    # 인덱싱
...                     if k < 0 or self.end <= k:
...                             raise IndexError(k)
...                     return k * k
...             else:
...                     raise TypeError('...')
...
>>> s = Square(10)
>>> s[4]                # 인덱싱
16
>>> list(s[1:5])        # 슬라이싱
[1, 4, 9, 16]
>>> list(s[1:10:21])    # 간격은 2로
```

```
[1]
>>> list(s[:])          # 전체범위
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

먼저 _getitme_() 메서드의 정의를 살펴보자. 가장 먼저 k의 자료형을 검사한다. k가 slice형이면 슬라이싱을, 아니면 인덱싱을 적용한다. 슬라이싱 부분에서 start와 stop, step을 별도의 지역변수에 치환한 이유는 range() 함수가 정수 인수만을 요구하기 때문이다. 최종적으로 map() 함수에 의해서 각 인덱스 값의 제곱에 대한 리스트를 반환한다.

3. 매핑 자료형

매핑 자료형에서 object._getitem_(self, key) 등의 메서드의 key는 사전의 키로 사용할수 있는 임의의 객체가 될수 있다. 만일 key에 대응하는 값을 찾을수 없으면 KeyError 에러를 발생시킨다. 다음 클래스 MyDict는 치환(d[key]=vlaue)과 참조(a = d[key1]), 크기(len(d)) 정보를 얻을수 있는 간단한 클래스이다.

```
>>> class MyDict:
...     def __init__(self):
...             self.d = {}
...     def __getitem__(self, k):         # key
...             return self.d[k]
...     def __setitem__(self, k, v):
...             self.d[k] = v
...     def __len__(self):
...             return len(self.d)
...
>>> m = MyDict()        # _init_() 메서드 호출
>>> m['day'] = 'light'  # m._setitem_('day','light')
>>> m.__setitem__('night','darkness')   # m['night'] = 'darkness'
>>> m['day']            # m.getitem_('day')
'light'
>>> m['night']          # m._getitem_('night')
'darkness'
>>> len(m)              # _len_() 메서드 호출
2
```

4. 반복자 지원

반복자에 관한 내용은 18장에서 자세하게 다루고 있으므로 18.2절의 예를 참고하기 바란다.

- 문자열 변환 연산

인스턴스 객체를 print() 함수로 출력할 때 내가 원하는 형식으로 출력하거나, 인스턴스 객체를 사람이 읽기 좋은 형태로 변환하려면 문자열로 변환하는 기능이 필요하다. 이 절에서는 인스턴스 객체를 문자열로 변환하는 기능에 대해서 살펴보자.

1. 문자열로의 변환 : __str__()와 __repr__() 메서드

인스턴스 객체를 문자열로 변환하는 메서드는 _str_()와 _repr_() 두 가지이다. 이 두 메서드는 호출되는 시점이 조금 다른데, 언제 이들이 호출되는지는 다음 예를 보면 쉽게 알수 있다.

```
>>> class StringRepr:
...     def __repr__(self):
...             return 'repr called'
...     def _str_(self):
...             return 'str called'
...
>>> s = StringRepr()
>>> print(s)
repr called
>>> str(s)
'repr called'
>>> repr(s)
'repr called'
```

print() 함수와 str() 함수에 의해서 _srt_() 메서드가 호출되며, repr() 함수에 의해서 _repr_() 메서드가 호출된다. _repr_() 메서더의 목적은 객체를 대표해서 유일하게 표현할수 있는 문자열을 만들어 내는 것이다. 즉, 다른 객체의 출력과 혼동되지 않는 모양으로 표현해야 한다는 의미이다.

```
>>> repr(2)
'2'
>>> repr('2')
"'2'"
>>> repr('abc')         # 문자열  'abc '에 대한 repr 문자열 표현
"'abc'"
>>> repr([1, 2, 3])     # 리스트 [1, 2, 3]에 대한 repr 문자열 표현
'[1, 2, 3]'
```

다음으로 _str_() 메서드의 목적은 사용자가 읽기 편한 형태의 표현으로 출력한다.

```
>>> str(2)
'2'
>>> str('2')
'2'
```

재미있는 현상은 컨테이너 자료형(리스트와 사전 등)의 _str_() 메서드는 내부 객체의 _repr_() 메서드를 사용한다.

```
>>> L = [2, '2']
>>> str(L)
"[2, '2']"
```

```
>>> repr(L)
"[2, '2']"
>>> str(L) == repr(L)
True
```

만일 _str_() 메서드를 호출할 상황에서 _str_() 메서드가 정의되어 있지 않으면 _repr_() 메서드가 대신 호출된다.

```
>>> class StringRepr:
...     def __str__(self):
...             return 'str called'
...
>>> s = StringRepr()
>>> str(s)
'str called'
>>> repr(s)
'<__main__.StringRepr object at 0x0000023D7F4C6898>'
```

그러나 __repr__() 메서드가 정의되어 있지 않은 경우에 __str__() 메서드가 __repr__() 메서드를 대신하지는 않는다.

2. 바이트로의 변환 :__bytes__() 메서드
문자열이 아닌 바이트 자료형으로 변환하려면 _bytes_() 메서드를 사용한다. b._bytes_() 메서더는 bytes(b) 함수에 의해서 호출된다.

```
>>> class BytesRepr:
...     def __bytes__(self):
...             return 'bytes called'.encode('utf-8')
...
>>> b = BytesRepr()
>>> bytes(b)
b'bytes called'
```

3. 서식 기호 새로 지정하기: __format__() 메서드
__format__() 메서드는 format() 함수나 문자열의 format() 메서드에 의해서 호출된다.
예를 들어, 다음과 같은 경우에 호출된다.

```
>>> format(x, "o")      # x.__format__("o") 호출
'121'
>>> "x:{:o}".format(x)  # x._format_("o") 호출
'x:121'
```

__format__() 메서드에 전달되는 변환 기호는 사용자가 새로 정의할 수도 있다.
다음은 대문자 변환을 위한 새로운 변환 기호 u와 소문자 변환을 위한 새로운 변환 기호 1을 정의하는 예이다.
변환 기호가 요구될때는 __format__() 메서드가 호출된다.

```
>>> class MyStr:
...     def _init_(self, s):
...             self.s=s
...     def _format_(self, fmt):
...             print(fmt)              # 서식 문자열을 확인한다.
...             if fmt[0] == 'u':       # u이면 대문자로 변환한다.
...                     s = self.s.upper()
...                     fmt = fmt[1:]
...             else:
...                     s = str(self.s)
...             return s.__format__(fmt)
...
```

- 진릿값과 비교 연산
1. __bool__() 메서드
클래스 인스턴스의 진릿값은 __bool__() 메서드의 반환 값으로 결정된다. 만일 이 메서드가 정의되어 있지 않으면 __len__() 메서드를 호출한 결고가 0이면 False로 간주하고 아니면 True 로 간주한다. 만일, __len__() 과 __bool__() 메서드 모두가 정의되어 있지 않으면 모든 인스턴스는 True가 된다.

```
>>> class Truth:
...     def __init__(self, num):
...             self.num = num
...     def __bool__(self):
...             return self.num != 0
...
>>> bool(Truth(0))
False
>>> bool(Truth(3))
True
```

3. 비교연산
파이썬의 모든 비교연산은 중복이 가능하도록 메서드 이름이 준비되어 있다.

표 비교 연산을 위한 메서드

연산자	메서드
<	object._lt_(self, other)
<=	object._le_(self, other)
>	object._gt_(self, other)
>=	object._ge_(self, other)
==	object._eq_(self, other)
!=	object._ne_(self, other)

x<y는 x._lt_(y) 메서드로 확장되며, x<=y는 x._le_(y) 메서드로 확장된다. 다른 연산자도 같은 방식이 적용된다. _eq_()와 _ne_() 메서드는 각자의 논리로 적용될수 있다. 즉, o == other이 참이라고 해서 o != other이 거짓이 아닐수도 있다는 것이다. 하지만, _ne_() 메서드가 정의되어 있지 않고, _eq_() 메서드만 정의되어 있을 경우 o != other는 not(o == other)의 논리가 적용된다. 다음 예를 참고하자.

```
>>> class Compare:
...     def __init__(self, n):
...             self.n = n
...     def __eq__(self, o):
...             print('__eq__ called')
...             return self.n == o
...     def __lt__(self, o):
...             print('__lt__ called')
...             return self.n < o
...     def __le__(self, o):
...             print('__le__ called')
...             return self.n <= o
...
>>> c = Compare(10)
>>> c < 10              # __lt__
__lt__ called
False
>>> c <= 10             # __le__
__le__ called
True
>>> c > 10              # 에러가 발생한다. __gt__ 정의가 안 된다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'Compare' and 'int'
>>> c == 10             # __eq__() 메서드
__eq__ called
True
>>> c != 10             # __ne__() 메서드나 not __eq__() 메서드 결과
__eq__ called
```

```
False
```

- 해시 값에 접근하기 : __hash__() 메서드

해시 값을 돌려주는 내장 함수 hash(m)가 호출될 때 m._hash_() 메서드가 호출된다. hash() 함수를 사용한 예로, 사전은 (키, 값) 쌍을 저장할 때 키에대한 hash() 함수의 호출 결과를 값을 저장하기 위한 해시키로 사용한다. _hash_() 메서드는 정수를 반환해야 한다. 이 메서드를 정의한 클래스는 _eq_() 메서드도 함께 정의해야 해시가 가능한 객체로 취급된다.

```
>>> class Obj:
...     def __init__(self, a, b):
...             self.a = a
...             self.b = b
...     def _key(self):
...             return (self.a, self.b)
...     def __eq__(self, o):
...             return self._key() == o.key()
...     def __hash__(self):
...             return hash(self._key())
...
>>> o1 = Obj(1, 2)
>>> o2 = Obj(3, 4)
>>> hash(o1)
3713081631934410656
>>> hash(o2)
3713083796997400956
>>> d = {o1:1, o2:2}
```

해시 키는 변경이 가능해서는 안된다. 만일 변경 가능한 자료형으로 클래스를 정의하면 hash() 함수를 호출했을 때 TypeError 에러를 반환해서 해시키로 사용할수 없도록 해야 한다.

```
>>> class Obj2:
...     def __init__(self, a, b):
...             self.a = a
...             self.b = b
...     def __hash__(self):
...             raise TypeError('not proper type')
...
>>> o1 = Obj2(1, 2)
>>> d = {o1 : 1}                # 키로 사용할 수 없다.
Traceback (most recent call last):
...
TypeError: not proper type
```

- 속성값에 접근하기

파이썬에서는 인스턴스 객체의 속성을 다루는 메서드가 네 개 준비되어 있다. 다음은 이 메서드를 정리한 표이다.

표) 인스턴스 객체의 속성을 다루는 메서드

메서드	속성
getattr(self, name)	정의되어 있지 않은 속성을 참조할 때, 이 메서드가 호출된다. 속성 이름 name은 문자열이다.
getattribute(self, name)	_getattr_() 메서드와 같으나 속성이 정의되어 있어도 호출된다.
setattr(self, name, value)	self.name = value와 같이 속성에 치환(대입)이 일어날 때 호출된다.
delattr(self, name)	del sell.name에 의해서 호출된다.

1. __getattr__()와 __getattribute__() 메서드

인스턴스 객체에 대한 일반적인 접근 방법인 obj.attr() 메서드는 getattr(obj, 'attr')로 수행된다. 우선 _getattr_()과 _getattribute_() 메서드의 차이를 살펴보자. _getattr_() 메서드는 이름 공간에 정의되지 않는 이름에 접근할 때 호출되며 이에 대해서 처리를 할수 있다.

```
>>> class GetAttr1(object):
...     def __getattr__(self, x):
...             print('__getattr__',x)
...             if x == 'test':
...                     return 10
...             raise AttributeError
...
>>> g1 = GetAttr1()
>>> g1.c = 10
>>> g1.c                # 정의된 이름을 호출한다.
10
>>> g1.a                # 정의되지 않은 이름을 호출한다.
__getattr__ a                   # ------------------ __getattr__() 메서드가 호출된다.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in __getattr__
AttributeError
>>> g1.test             # 정의되지 않았지만 준비된 이름이다.
__getattr__ test
10
```

getattribute() 메서드는 이름 정의 여부에 관계없이 모든 속성에 접근하면 호출된다.

```
>>> class GetAttr2(object):
...     def __getattribute__(self, x):
...             print('__getattribute__ called..', x)
```

```
...             return object.__getattribute__(self, x)
...
>>> g2 = GetAttr2()
>>> g2.c = 10
>>> g2.c                # 정의된 이름을 호출한다.
__getattribute__ called.. c
10
>>> g2.a                # 정의되지 않은 이름을 호출한다.
__getattribute__ called.. a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __getattribute__
AttributeError: 'GetAttr2' object has no attribute 'a'
```

따라서 _getattribute_() 메서드는 호출되는 이름 전체에 대한 제어권을 얻어 낸다. 두 메서드가 모두 정의되어 있는 예를 보자. 정의되어 있는 이름에 접근할때는 _getattribuate_() 메서드를 호출하고, 정의되어 있지 않은 이름에 접근할때는 _getattribute_()와 _getattr_() 메서드 모두를 호출한다.

```
>>> class GetAttr3(object):
...     def __getattr__(self, x):
...             print('a_getattr_', x)
...             raise AttributeError
...     def __getattribute__(self, x):
...             print('__getattribute__ called..',x)
...             return object.__getattribute__(self, x)
...
>>> g3 = GetAttr3()
>>> g3.c = 10
>>> g3.c                # 정의된 속성에 접근한다.
__getattribute__ called.. c
10
>>> g3.a                # 정의되지 않은 속성에 접근한다.
__getattribute__ called.. a
a_getattr_ a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __getattr__
AttributeError
```

주의할점은, self._getattribute_(x)와 같이 호출해서는 안된다. 재귀적으로 자기 자신을 무한히 호출하게 되므로 상위 클래스를 통해서 object._getattribute_(self, x)와 같은식으로 접근해야 한다.

2. __setattr__()와 __delattr__() 메서드

인스턴스 객체에서 속성을 설정할때는 _setattr_() 메서드를 속성을 삭제할때는 _delattr_ 메서드를 사용한다. obj.x = o는 setattr(obj, 'x', o)로 수행되며 obj._setattr_('x', o)를 호출하고 del obj.x는 delattr(obj, 'x')로 수행되며 obj._delattr_('x')를 호출한다.

```
>>> 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)
...
>>> a = Attr()
>>> a.x = 10
_setattr_(x)=10 called
>>> a.x
10
>>> del a.x
_delattr_(x) called
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Attr' object has no attribute 'x'
>>>
```

- 인스턴스 객체를 호출하기 : __call__() 메서드

1. __call__() 메서드

어떤 클래스 인스턴스가 _call_() 메서드를 가지고 있으면, 해당 인스턴스 객체는 함수와 같은 모양으로 호출할 수 있다. 인스턴스 객체 x에 대해 다음과 같이 확장된다.

```
x(a1, a2, a3) -> x._call_(a1, a2, a3)
```

다음 클래스 Factorial은 고속 처리를 위하여 기억 기법을 사용한다. 한번 계산된 팩토리얼 값은 인스턴스 객체의 cache 멤버에 저장되어 있다가 필요할 때 다시 사용한다. 팩토리얼 계산은 인스턴스 객체의 _call_() 메서드를 호출하여 이루어진다.

```
>>> class Factorial:
...     def __init__(self):
...             self.cache = {}
...     def __call__(self, n):
```

```
...             if n not in self.cache:
...                     if n == 0:
...                             self.cache[n] = 1
...                     else:
...                             self.cache[n] = n * self.__call__(n-1)
...             return self.cache[n]
>>> fact = Factorial()
>>> for i in range(10):                         )
...     print('{}! = {}'.format(i, fact(i))) )
...
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
```

2. 호출 가능한지 확인하기

어떤 객체가 호출가능한지 알아보려면 collections.Callable의 인스턴스 객체인지 확인한다.

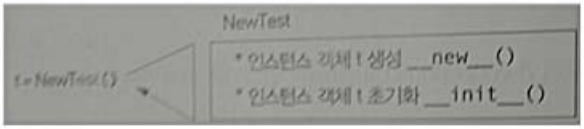
```
>>> import collections
>>> def f():
...     pass
...
>>> isinstance(f, collections.Callable)         # 함수 객체를 확인한다.
True
>>>
>>> fact = Factorial()                          # 인스턴스 객체를 확인한다.
>>> isinstance(fact, collections.Callable)
True
```

- 인스턴스 객체를 생성하기: __new__()메서드

클래스의 __init__() 메서드는 객체가 생성된 이후에 객체를 초기화하기 위해 호출되는 메서드이다. 반면에 정적 메서드(14.3절 참고) __new__()는 객체의 생성을 담당하는 메서드로 __new__() 메서드에 의해서 생성된 객체가 __init__() 메서드에 의해서 초기화된다.

__new__() 메서드는 object 클래스의 __new__() 메서드를 통해서 인스턴스 객체를 생성해야 한다.

```
>>> class NewTest:
...     def __new__(cls, *args, **kw):          # cls는 NewTest
...             print("_new_ called", cls)
...             instance = object.__new__(cls)  # 인스턴스 객체를 생성한다.
...             return instance
...     def __init__(self, *args, **kw):        # self는 생성된 인스턴스 객체이다.
...             print("__init__ called", self)
...
>>> t = NewTest()
_new_ called <class '__main__.NewTest'>
__init__ called <__main__.NewTest object at 0x0000017C98E39E10>
```



만일 _new_() 메서드가 인스턴스 객체를 반환하면 _init_() 메서드가 호출되지만, 그렇지 않으면 _init_() 메서드는 호출되지 않는다.

다음 예는 __new__() 메서드를 사용하여 멤버값을 초기화하는 예이다. 멤버값의 초기화는 일반적으로 __init__() 메서드에서 이루어지지만 상위 클래스의 __init__() 메서드를 명시적으로 호출하지 않으면 상위 클래스의 __init__()메서드는 실행되지 않는다. __new__() 메서드를 사용하여 상위 클래스의 __init__() 메서드 호출 여부와 관계없이 멤버 값의 초기화를 수행하는 예를보자.

```
>>> class Super:
...     def __new__(cls, *args, **kw):
...             obj = object.__new__(cls, *args, **kw)
...             obj.data = []
...             return obj
...
>>> class Sub(Super):
...     def __init__(self, name):
...             self.name = name        # 자기 멤버만 초기화한다.
...

>>> s = Sub( '이강성' )
>>> s.name
 '이강성'
>>> s. data                    # Super._new_()에서 초기화된 멤버이다.
```

다음은 싱글톤 예이다. 싱글톤이란 인스턴스 객체를 오직 하나만 생성해 내는 클래스를 의미한다. 유일하게 하나만 시스템에 존재해야 하는 객체를 정의할 때 유용하다.

```
>>> class Singleton:
...     __instance = None       # 유일한 객체를 지정하기 위한 클래스 변수이다.
...     def __new__(cls, *args, **kwargs):
...             if cls.__instance is None:
...                     # 새로운 객체를 생성한다.
...                     cls.__instance = object.__new__(cls)
...             return cls.__instance
...
>>> class Sub(Singleton):       # 싱글톤으로부터 상속받는다.
...     pass
...
>>> s1 = Sub()
>>> s2 = Sub()                  # 같은 객체를 반환한다.
>>> s1 is s2                    # 같은 객체인지 비교한다.
True
```

- With 문 구현하기

클래스에 _enter_() 메서드와 _exit_() 메서드가 구현되어 있으면 with 문에 사용할수 있다. 우선 wioth문이 동작하는 원리를 이해해 보자. 다음과 같은 with문이 주어져 있다.

```
with ctrled _exec() as var:
       # 코드 블록
```

이때 ctrled_exec()는 다음과 같이 정의되어 있다고 하자.

```
class ctrled_exec:
..
       edf _enter_(self):
              # 준비 작업을 한다.
              return thing    # as 키워드 다음의 변수명에 치환될 객체를 반환한다.
       def _exit_(self, type, value, traceback):
       # 정리작업을 한다.
```

앞서 with 문에 주어진 ctrled_exec()는 문맥관리 객체를 돌려준다. 이객체의 생성 이후에는 바로 _enter_() 메서드가 호출괴며, 이 메서드의 반환 값이 var에 치환된다.
그 다음에 코드 블록이 실행되며 with문을 빠져나올 시점에서 최종적으로는 문맥 관리 객체의 _exit_() 메서드가 실행되고 with 블록을 빠져나온다. 물론 예외가 중간에 발생해도 _exit_() 메서드는 실행된다. 파일 객체의 예를 보자.

```
with open( 'etst.tst' ) as f:
       f.read(1)
```

우선 open('test.txt')를 실행한다. 이때 생성된 객체를 o라 하자. o의 _enter_() 메서드를 호출한 결과를 f에 치환한다. 그리고 f.read(1)를 실행하며 with 문을 나오기 전에 f._exit_(0을 실행한다. _exit_() 메서드는 파일을 닫고 정리하여 더 이상 입출력이 가능하지 않게 한다.

```
file : test.txt
ABCDEFGHIJKLMNOPQRSTUVWXYZ

>>> o = open('test.txt')
>>> o
<_io.TextIOWrapper name='test.txt' mode='r' encoding='cp949'>
>>> f = o.__enter__()
>>> f
<_io.TextIOWrapper name='test.txt' mode='r' encoding='cp949'>
>>> f.read(1)
'A'
>>> f.__exit__(None, None, None)
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

다음은 임계영역을 구현하는 Locked 클래스의 예이다. with문 안에 사용된 코드 블록은 lock객체의 임계 영역이며 배타적 실행을 보장한다.

```
# file :  with_sample.py

import threading

class Locked:
   def __init__(self, lock):
       self.lock = lock
   def __enter__(self):
       self.lock.acquire()
   def __exit__(self, type, value, tb):
       self.lock.release()


lock = threading.Lock()
with Locked(lock):
    print ('I am in C.S.')
```

장식자와 생성자를 아직 설명하지는 않았지만, 장식자와 생성자를 이용하여 문맥관리 객체를 구현하는 방법도 있다. 다음은 앞에서 보인 예와 동일한 예이다. tield문이전이 _enter_() 메서드에 해당하며 yield 문 이후는 __exit__() 메서드에 해당한다.

```
# file : with_sample2.py

from contextlib import contextmanager
import threading

@contextmanager
def Locked2(lock):
    lock.acquire()
    yield lock
    lock.release()

lock = threading.Lock()
with Locked2(lock):
    print ('I am in C.S.')
```

14.2 장식자

- 장식자의 이해

장식자는 함수를 인수로 받는 함수클로저이다. 함수클로저를 간단히 말하면 함수코드와 그함수의 변수 참조 영역을 묶은 객체라고 할수 있다. 함수를 인스턴스화하는데 유용하다. 따라서 장식자에 유용하게 활용된다. 다음예를 살펴보자.

```
>>> def wrapper(func):          # 장식자는 함수 객체를 인수로 받는다.
        def wrapped_func():     # 내부에서는 wrapper() 함수를 정의한다.
                print('before..')       # 원래 함수 이전에 실행되어야 할 코드이다.
                func()                  # 원래 함수이다.
                print('after..')        # 원래 함수이후에 실행될 코드이다.
```

```
        return wrapped_func
>>>
>>> def myfunc():               # 원해 함수를 실행한 결과이다.
        print('I am here')
>>> myfunc()
I am here
>>> myfunc = wrapper(myfunc)    # 장식자에 함수를 전달한다.
>>> myfunc()                    # 장식된 함수를 실행한다.
before..
I am here
after..
```

앞의 예서 wrapper() 함수는 장식자이다. 함수를 인수로 받으며 함수 클로저를 반환한다.
myfunc = wrapper(myfunc)로 반환된(장식된)함수는 실행할때마다 실제로는 wrapped_func() 함수가 실행된다. 이것이 장식자이다.

장식자를 좀더 간단히 표현하는 방법이 있는데 @wrapper 형식의 선언을 함수 앞에 하는 것이다. 그러면 다음과 같이 변환이 이루어진다.

```
@wapper            def f():
def f():    --->     ~생략~
~ 생략 ~           f = wrapper(f)
```

다음은 이 방법을 사용한 예이다.

```
>>> @wrapper            # 장식자이다.
... def myfunc2():      # myfunc2 = wrapper(myfunc2)
...     print('I am here 2..')
>>>
>>> myfunc2()           # 장식된 함수를 실행한다.
before..
I am here 2..
after..
```

장식자를 사용하는 대표적인 예는 정적 메서드와 클래스 메서드이다. 다음 두 클래스는 동일한 결과를 만든다.

```
>>> class D:
        @staticmethod           # add를 static method로
        def add(x, y):
                return x + y
```

```
>>> class D:
...     def add(x, y):
...         return x + y
...     add = staticmethod(add)
>>> D.add(2, 4)
6
```

장식자는 함수는 다시 한번 감싸서 호출하는 것이므로 호출에 부가적인 시간이 걸린다는 것에 유의해야 한다.

- 연결된 장식자
장식자는 연결해서 사용할수 있다.

```
@A @B @C
def f():
        ~ 생략~
```

혹은 다음과 같이 여러줄에 걸쳐 사용할수도 있다.

```
@A
@B
@C
def f():
        ~생략~
```

앞의 코드는 다음 코드와 동일하다.

```
def f():
        ~ 생략 ~
f = A(B(C(F)))
```

다음은 연결된 장식자의 간단한 예이다. 장식의 목표는 장싲자에 따라서 HTML 태그가 붙어서 나오게 하는 것이다.

```
@makebold
@makeitalic
def say():
        return "hello:
print(say())            # 출력 : <b><i>hello</i></b>
```

장식자를 완성해 보자.

```
>>> def makebold(fn):          # 굵게 만드는 장식자이다.
...     def wrapper():
```

```
...         return "<b>" + fn() + "</b>"
...     return wrapper
>>>
>>> def makeitalic(fn):          # 기울임꼴로 만드는 장식자이다.
...     def wrapper():
...         return "<i>" + fn() + "</i>"
...     return wrapper
>>>
>>> @makebold           # 장식자를 중첩해서 표현한다.
... @makeitalic
... def say():
...     return "hollo"
>>> say()               # 기울임꼴로 그리고 굵게 만드는 출력을 얻는다.
'<b><i>hollo</i></b>'
```

- 장식된 함수에 인수를 전달하기

인수를 갖는 일반 함수를 장식하려면 장식 함수도 인수를 받아서 처리해야 한다.

```
>>> def debug(fn):              # 장식자를 정의한다.
...     def wrapper(a, b):      # fn과 동일한 인수를 받는다.
...         print('debug', a, b)
...         return fn(a, b)     # 함수를 호출한다.
...     return wrapper
>>>
>>> @debug
... def add(a, b):
...     return a + b
>>> add(1, 2)
debug 1 2
3
```

인수 전달을 일반화하려면 다음과 같이 가변 인수와 키워드 인수를 함께 사용해야 한다.

```
>>> def debug(fn):
...     def wrapper(*args, **kw):               # 가변인수, 키워드 인수
...         print('calling', fn.__name__,'args=', args, 'kw=',kw)
...         result = fn(*args, **kw)     # 그대로 전달한다.
...         print('₩tresult=',result)
...         return result
...     return wrapper
>>> @debug
... def add(a, b):
```

```
...     return a + b
>>> add(1, 2)
calling add args= (1, 2) kw= {}
        result= 3
3
```

- 인수를 갖는 장식자
장식자는 인수를 가질수 있다. 다음은 그예이다.

```
@A @B @C(args)
def f():
        ~ 생략 ~
```
이것은 다음 코드와 동일하다.

```
def f():
        ~ 생략 ~
f = A(B(C(args)(f)))
```

즉, c(args)는 새로운 장식자를 반환하며, 해당 장식자의 인수로 f를 전달하는 것이다. 다음은 함수 인수의 타입을 확인하는 예이다.

```
>>> def accepts(*types):
...     def check_accepts(f):
...         def new_f(*args, **kw):
...             for(a, t) in zip(args, types):
...                 assert isinstance(a, t), "arg {} does not match {}".format(a, t)
...             return f(*args, **kw)           # 여기서 실제 함수가 호출된다.
...         return new_f
...     return check_accepts
```

함수가 세 개나 중첩되어 있다. 장식자 내부에 떠 다른 장식자가 정의되어 있다. @accepts(int.int)에 의해서 다시 장식자 check_accepts() 함수가 반환된다. accept(*types)에 의해서 전달되는 types 인수는 (int, float)와 같은 자료형이다. 이 값들은 new_f() 함수 안의 isinstance(a, t)를 통하여 입력 인수 a가 t 타입인지를 확인한다. assert 문은 진릿값을 확인하고 거짓이면 AssertionError 에러를 발생시킨다.

```
>>> def accepts(*types):
...     def check_accepts(f):
...         def new_f(*args, **kw):
...             for(a, t) in zip(args, types):
...                 assert isinstance(a, t), "arg {} does not match {}".format(a, t)
...             return f(*args, **kw)           # 여기서 실제 함수가 호출된다.
...         return new_f
```

```
...     return check_accepts
```

함수가 세 개나 중첩되어 있다. 장식자 내부에 떠 다른 장식자가 정의되어 있다. @accepts(int,int)에 의해서 다시 장식자 check_accepts() 함수가 반환된다. accept(*types)에 의해서 전달되는 types 인수는 (int, float)와 같은 자료형이다. 이 값들은 new_f() 함수 안의 isinstance(a, t)를 통하여 입력 인수 a가 t 타입인지를 확인한다. assert 문은 진릿값을 확인하고 거짓이면 AssertionError 에러를 발생시킨다.

```
>>> isinstance(2, int)
True
>>> assert isinstance(2, int)
>>> assert isinstance(2.5, int)
AssertionError                            Traceback (most recent call last)
<ipython-input-53-562962327ad9> in <module>
----> 1 assert isinstance(2.5, int)

AssertionError:
```

에러가 발생하지 않으면 f(*args, **kw)를 통해 원래 함수를 호출한다.

```
>>> accepts(int, int)                   # 장식자
<function __main__.accepts.<locals>.check_accepts(f)>
>>> accepts(int, int)(add)              # add를 check_accepts로 장식한다.
<function __main__.accepts.<locals>.check_accepts.<locals>.new_f(*args, **kw)>
```

다음은 사용하는 예이다.

```
>>> @accepts(int, int)           #add = accepts(int, int)(add)
... def add(a, b):
...     return a+b
>>> add(1, 2)           # 타입이 맞지 경우
3
>>> add(3.4, 6)         # 타입이 맞지 않는 경우
AssertionError                            Traceback (most recent call last)
<ipython-input-76-5ae470abee69> in <module>
----> 1 add(3.4, 6)             # 타입이 맞지 않는 경우

<ipython-input-70-2e2818bae0c7> in new_f(*args, **kw)
      3         def new_f(*args, **kw):
      4             for(a, t) in zip(args, types):
----> 5                 assert isinstance(a, t), "arg {} does not match {}".format(a, t)
      6             return f(*args, **kw)               # 여기서 실제 함수가 호출된다.
      7         return new_f
```

```
AssertionError: arg 3.4 does not match <class 'int'>
```

- 메서드 장식하기

메서드는 함수와 다르지 않다. 함수와 같은 방법으로 클래스의 메서드를 장식할 수 있다. accepts() 함수를 다시 정의하고 메서드에 적용해 보자. 유일한 차이점은 메서드를 장식할 때는 첫 인수로 self를 제외한다는 것이다. 다음 예에서 accepts() 장식자는 add() 메서드의 첫인수 self를 제외한 두 번째부터의 인수들을 accepts()에 사용한다.

```
>>> def accepts(*types):
...     def check_accepts(f):
...         def new_f(self, *args, **kw):                # self를 고혀해야 한다.
...             for(a, t) in zip(args, types):
...                 assert isinstance(a, t), "arg {} does not match {} ".format(a, t)
...             return f(self, *args, **kw)      # 실제 함수가 호출된다.
...         return new_f
...     return check_accepts
>>>
>>> class Sori:
...     @accepts(int, int)              # self를 제외한 나머지 타입을 선언한다.
...     def add(self, a, b):
...         return a+b
>>> s= Sori()
>>> s.add(2,3)
5
```

- @functools.wraps

다음예에서 s.add_name_을 확인해 보자.

```
>>> def debug(fn):
...     def wrapper(*args, **kw):                # 가변 인수, 키워드 인수
...         result = fn(*args, **kw)     # 그대로 전달한다.
...     return wrapper

>>> class Sori:
...     @debug
...     def add(self, a, b):
...         return a+b
>>> s = Sori()
>>> s.add.__name__
'wrapper' ---------------- 원래는 add이어야 하는데 장식자 함수이름으로 바뀌었다.
```

이렇게 바뀐 이유는 장식자 debug에서 wrapper() 함수를 반환하기 때문에 생긴일이다. 이문제에 대한 해결법은 @functools.wraps 장식자를 사용하는 것이다. 이 장식자는 장식된 함수의 _name_과 _doc_가 원래 함수의 것을 갖도록 해준다.

```
>>> import functools
>>> def debug(fn):
...     @functools.wraps(fn)            # 감쌀 함수 fn을 넘겨준다.
...     def wrapper(*args, **kw):       # 가변인수, 키워드인수
...         return fn(*args, **kw)      # 그대로 전달한다.
...     return wrapper
>>> class Sori:
...     @debug
...     def add(self, a, b):
...         '''doc string'''
...         return a+b
>>> s = Sori()
>>> s.add.__name__
'add'
>>> s.add.__doc__
'doc string'
```

- 클래스 장식자

함수 클로저를 이용하지 않고 클래스를 이용하여 장식자를 만들 수 있다. _call_() 메서드에 함수를 전달하고 클래스 인스턴스를 생성한다. 인스턴스 객체는 _call_() 메서드를 통하여 마치 장식된 함수와 같이 동작할수 있다.

```
>>> from functools import wraps
>>> class class_decorator:      # 클래스로 만들어진 장식자이다.
...     def __init__(self, view_func):
...         self.view_func = view_func
...         wraps(view_func)(self)
...     def __call__(self, request, *args, **kwargs):
...         print('호출전에 수행할 코드들...')
...         response = self.view_func(request, *args, **kwargs)
...         print('호출 이후에 수행할 코드들 ...')
...         return response
>>> @class_decorator
... def add(a, b):
...     return a + b
>>> add(1, 2)
호출전에 수행할 코드들...
호출 이후에 수행할 코드들 ...
```

- 유용한 장식자들

장식자를 어디에 사용할가? 장식자는 다양한 경우에 유용하게 활용된다. 예를들어, 외부에서 제공되는 수정할수 없느느 라이브러리의 행동 특성을 변경하기 위해서, 소스를 건드리지 않고 디버깅 정보를 출력하기 위해서, 같은 방식으로 여러 함수를 확장하기 위해서 등등이 될수 있다.

파이썬에서 표준으로 제공하는 장식자들은 거의 없다. 파이썬에서는 단지 기능만을 제공할뿐 무한한 사용 가능성에 대해서는 사용자에게 일임하고 있다. 마치 클래스나 함수를 만드는 기능을 제공하는 것과 마찬가지이다. 직접 만들어 보거나 여러 유용한 장식자가 인터넷에 공개되고 있으니 필요에 따라 검색해 보면 좋을 것이다.

다음예는 함수 호출횟수를 세는 counter(0 , 함수 호출내용을 로깅하는 logging(), 함수 실행시간을 측정하는 benchmark() 장식자이다.

```
>>> import functools
>>> def counter(func):
...     """
...     함수 호출횟수를 센다.
...     """
...     @functools.wraps(func)
...     def wrapper(*args, **kwargs):
...         wrapper.count = wrapper.count +1
...         res = func(*args, **kwargs)
...         print("{0} : {1} 호출".format(func.__name__, wrapper.count))
...         return res
...     wrapper.count = 0
...     return wrapper
... def logging(func):
...     """
...     함수 호출 내용을 로깅(프린터)하는 장식자이다.
...     """
...     @functools.wraps(func)
...     def wrapper(*args, **kwargs):
...         res = func(*args, **kwargs)
...         print('{}({},{}) => {}'.format(func.__name__, args, kwargs, res))
...         return res
...     return wrapper
... def benchmark(func):
...     """
...     실행시간을 출력하는 장식자이다.
...     """
...     import time
```

```
...     @functools.wraps(func)
...     def wrapper(*args, **kwargs):
...         t = time.clock()
...         res = func(*args, **kwargs)
...         print(func.__name__, time.clock() - t)
...         return res
...     return wrapper
```

다음은 사용하는 예이다.

```
>>> @counter
... @benchmark
... @logging
... def add(a, b):
...     return a + b
>>> add(1, 2)
add((1, 2),{}) => 3
add 0.0001449999999749707
add : 1 호출
3
>>>
>>> add(2, 3)
add((2, 3),{}) => 5
add 9.200000022246968e-05
add : 2 호출
5
```

14.3 정적 메서드와 클래스 메서드

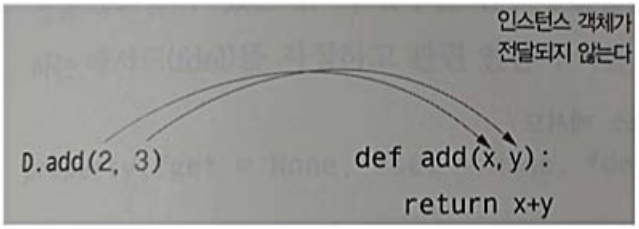
- 정적메서드

정적메서드란 인스턴스 객체를 생성하지 않고도, 혹은 인스턴스 객체를 이용하지 않고도 클래스를 이용하여 직접 호출할수 있는 메서드이다. 일반 메서드는 첫 번째 인수로 인스턴스 객체를 반드시 전달해야 하지만 정적 메서드는 일반함수와 동일한 방식으로 호출된다.

정적메서드는 일반 메서드와는 달리 첫 인수로 self를 받지 않는다. 필요한 만큼의 인수를 선언하면 된다. @staticmethod를 메서드 앞에 장식하면 정적 메서드가 된다.

```
>>> class D:
...     @staticmethod           # add를 정적 메서드로
...     def add(x, y):
...         return x + y
>>> D.add(2, 3)         # 인스턴스객체없이 클래스에서 직접호출한다.
5
>>> d = D()
>>> d.add(2, 3)         # 물론 인스턴스 객체를 통해서도 호출할 수 있다.
5
```

(그림) 정적 메서드의 호출



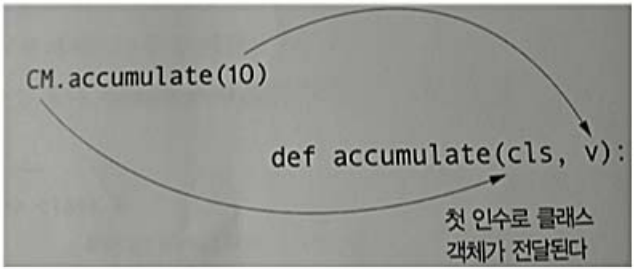
정적 메서드는 인스턴스 객체와 관계없이 실행되어야 하므로 인스턴스 변수를 참조할수 업삳.
대신에 클래스멤버는 참조할수 있다.

```
>>> class E:
...     acc = 0
...     @staticmethod
...     def accumulate(v):
...         E.acc += v
...         return E.acc        # 클래스 멤버
>>> E.accumulate(10)
10
>>> E.accumulate(20)
20
>>> E.acc
```

- 클래스 메서드

클래스메서드는 첫인수로 클래스 객체를 전달받는다. 정적메서드와 같이 클래스를 통하여 직접 호출하는 것이 일반적이지만 첫 인수로 클래스 객체가 전달되는 것이 다르다. 클래스 메서드는 @classmethod 장식자에 의해서 선언된다.

```
>>> class CM:
...     acc = 0
...     @classmethod
...     def accumulate(cls, v):          # 클래스 메서드
...         cls.acc += v
...         return cls.acc
>>> CM.accumulate(10)
10
>>> CM.accumulate(20)
30
>>> CM.acc
30
>>>
>>> c = CM()
>>> c.accumulate(5)              # 인스턴스 객체를 통한 호출이 가능하다.
35
>>> CM.acc
35
>>> c.__class__ is CM            # c._class_와 CM이 같은 객체인가?
True
```



14.4 property 속성만들기

property 속성이란 멤버 변수와 같은 접근 방식을 사용하지만 실제로는 메서드의 호출로 처리되는 속성을 말한다. 즉, 메서드로 정의되어 있지만 호출은 멤버 변수를 사용하는 것처럼 한다는 의미이다. 예를들어, x.a = 1이 실제로는 x.a_set(1) 이란 함수를 통해서 이루어지고, b = x.a이 실제로는 b = x.a_get()을 통해서 이루어지게 할수 있다. property 함수를 통해서 속성을 정의할수 있는데, 이 함수는 변수에 값을 저장하는 메서드(fset), 읽는 메서드(fget), 삭제하는 메서드(fdel)를 지정하고 관련 연산이 이들 메서드를 통해서 이루어지는 객체를 생성한다.

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

여기서 fget은 값을 읽을 때, fset은 값을 쓸 때, fdel은 값을 삭제할 때 자동으로호출되는 메서드이다.

다음예는 degree란 변수에 각도를 저장한다. 하지만, 각도를 0도에서 360도 미만의 범위내에서 정규화한다. 즉, 370도를 저장하면 자동으로 10도를 정규화된다.

```
>>> class PropertyClass:
...     def get_deg(self):
...             return self.__deg
...     def set_deg(self, d):
...             self.__deg = d % 360
...     deg = property(get_deg, set_deg)
>>> p = PropertyClass()
>>> p.deg = 390
>>> p.deg
30
>>> p.deg = -370
>>> p.deg
350
```

장식자를 이용한 방법으로 다음과 같은 방법으로도 코딩이 가능하다. getter와 setter의 이름을 다르게 하지 않고도 메서드를 정의할수 있다.

```
>>> class PropertyClass:
...    @property               # getter 메서드를 등록한다.
...    def deg(self):
...        return self.__deg
...    @deg.setter      # deg의 setter메서드를 등록한다.
...    def deg(self, d):
...        self.__deg = d % 360
>>> p = PropertyClass()
>>> p.deg = 390
>>> p.deg
```

```
30
>>> p.deg = -370
>>> p.deg
350
```