파이썬 프로그래밍

클래스와 연산자 중복 정의

한국기술교육대학교
온라인평생교육원

▣ 연산자 중복

1. 수치 연산자 중복

- 직접 정의하는 클래스 인스턴스에 연산자를 적용하기 위하여 미리 약속되어 있는 메소드들을 정의

메소드(Method)	연산자(Operator)	인스턴스 o에 대한 사용 예
__add__(self, B)	+ (이항)	o + B, o += B
__sub__(self, B)	- (이항)	o - B, o -= B
__mul__(self, B)	*	o * B, o *= B
__div__(self, B)	/	o / B, o /= B
__floordiv__(self, B)	//	o // B, o //= B
__mod__(self, B)	%	o % B, o %= B
__divmod__(self, B)	divmod()	divmod(o, B)
__pow__(self, B)	pow(), **	pow(o, B), o ** B
__lshift__(self, B)	<<	o << B, o <<= B
__rshift__(self, B)	>>	o >> B, o >>= B
__and__(self, B)	&	o & B, o &= B
__xor__(self, B)	^	o ^ B, o ^= B
__or__(self, B)	|	o | B, o |= B
__neg__(self)	- (단항)	-A
__abs__(self)	abs()	abs(o)
__pos__(self)	+ (단항)	+o
__invert__(self)	~	~o

▣ 연산자 중복

1. 수치 연산자 중복

- ▪클래스를 직접 정의
- ▪클래스에서 객체를 인스턴스화하여 활용
- ▪이러한 객체에 연산을 적용 → 어떻게 해야 할까?
- ▪인스턴스 메소드 → 첫 번째 인자에는 self가 옴
- ▪+ 연산자 사용하면 자동으로 add 메소드 불러짐
- ▪+=, -= 처럼 확장연산자 사용 가능

▣ 연산자 중복

1. 수치 연산자 중복

```
class Integer:
    def __init__(self, i):
        self.i = i
    def __str__(self):
        return str(self.i)
    def __add__(self, other):
        return self.i + other

i = Integer(10)
print i
print str(i)

print
i = i + 10
print i

print
i += 10
print i
```

```
10
10

20

30
```

▣ 연산자 중복

1. 수치 연산자 중복

- __str__은 객체를 프린트할 때 호출됨
- i = i + 10을 했을 때 add 메소드 호출됨
- 10 → 두 번째 인자 other에 들어감
- str(i)의 i는 객체, self.i의 i는 i 식별자 값
- self.i + other → self.i 는 정수, other도 정수 → 정수가 반환됨
- print i는 객체가 아닌 정수를 출력
- print i += 10에서 +=는 add를 호출하지 않음
- print self만 하게 되면 정수 i 가 아닌 객체 i
- 객체를 프린트하니까 str이 호출되어 문자화가 된 self.i 값 출력

▣ 연산자 중복

1. 수치 연산자 중복

```
class MyString:
    def __init__(self, str):
        self.str = str
    def __div__(self, sep): # 나누기 연산자 /가 사용되었을 때 호출되는 함수
        return self.str.split(sep) # 문자열 self.str을 sep를 기준으로 분리

m = MyString("abcd_abcd_abcd")
print m / "_"
print m / "_a"

print
print m.__div__("_")
```

```
['abcd', 'abcd', 'abcd']
['abcd', 'bcd', 'bcd']

['abcd', 'abcd', 'abcd']
```

▪self.str.split(sep) → self 객체가 가지고 있는 str을 sep으로 분리

▪'_'를 가지고 있는 문자열을 찾아 sep에 집어 넣음

▪특수 메소드는 / 기호 또는 직접 이름을 넣어 호출 가능

▣ 연산자 중복

1. 수치 연산자 중복

- 연산자 왼쪽에 피연산자, 연산자 오른쪽에 객체가 오는 경우
  - 메소드 이름 앞에 r이 추가된 메소드 정의

```
class MyString:
    def __init__(self, str):
        self.str = str
    def __div__(self, sep):
        return str.split(self.str, sep)
    __rdiv__ = __div__

m = MyString("abcd_abcd_abcd")
print m / "_"
print m / "_a"
print
print "_" / m
print "_a" / m
```

```
['abcd', 'abcd', 'abcd']
['abcd', 'bcd', 'bcd']

['abcd', 'abcd', 'abcd']
['abcd', 'bcd', 'bcd']
```

- __rdiv__ : 약속된 메소드
- __rdiv__ → __div__랑 같으나 객체가 오른쪽에 위치
- O + B를 B+ O로 쓰고 싶으면 __radd__ 사용
- 메소드 이름 앞에 'r'이 붙으면 연산자 오른쪽에 객체가 호출

▣ 연산자 중복

1. 수치 연산자 중복

```
class MyString:
    def __init__(self, str):
        self.str = str
    def __div__(self, sep):
        return str.split(self.str, sep)
    __rdiv__ = __div__
    def __neg__(self):
        t = list(self.str)
        t.reverse()
        return ''.join(t)
    __invert__ = __neg__

m = MyString("abcdef")
print -m
print ~m
```

```
fedcba
fedcba
```

▪~ 있는 것 → invert 연산

▣ 연산자 중복

2. 비교 연산자 중복

- 각각의 비교 연산에 대응되는 메소드 이름이 정해져 있지만 그러한 메소드가 별도로 정의되어 있지 않으면 cmp가 호출됨

메소드	연산자	비고
__cmp__(self, other)	아래 메소드가 부재한 상황에 호출되는 메소드	
__lt__(self, other)	self < other	
__le__(self, other)	self <= other	
__eq__(self, other)	self == other	
__ne__(self, other)	self != other	
__gt__(self, other)	self > other	
__ge__(self, other)	self >= other	

▪객체, 연산자(<), other 쓰면 lt 실행됨
▪lt → less than의 약자
▪메소드 le → less than or equal의 약자
▪== → 동등 연산 → equal의 약자
▪!= → not equal → ne 사용
▪> → greater than 의 약자 → gt
▪>= → ge → greater than or equal
▪__cmp__는 아래 메소드가 부재한 상황에서 호출되는 메소드

▣ 연산자 중복

2. 비교 연산자 중복

- 객체 c에 대한 c > 1연산의 행동 방식
  - c.__gt__()가 있다면 호출 결과을 그대로 반환
  - 정의된 c.__gt__()가 없고, __cmp__() 함수가 있을 경우
    * c.__cmp__() 호출 결과가 양수이면 True 반환, 아니면 False 반환

```
class MyCmp:
    def __cmp__(self, y):
        return 1 - y

c = MyCmp()
print c > 1 # c.__cmp__(1)을 호출, 반환값이 양수이어야 True
print c < 1 # c.__cmp__(1)을 호출, 반환값이 음수이어야 True
print c == 1 # c.__cmp__(1)을 호출, 반환값이 0이어야 True
```

```
False
False
True
```

▪1이 __cmp(self, y)에서 y에 들어감
▪c > 1 → cmp가 돌려주는 값이 양수여야 true 됨
▪c < 1 → cmp가 돌려주는 값이 음수여야 true 됨
▪c == 1 → cmp가 돌려주는 값이 0이여야 true 됨

▣ 연산자 중복

2. 비교 연산자 중복

- 객체 m에 대한 m < 10연산의 행동 방식
  - m.__lt__()가 있다면 호출 결과을 그대로 반환
  - 정의된 m.__lt__()가 없고, __cmp__() 함수가 있을 경우
    * m.__cmp__() 호출 결과가 음수이면 True 반환, 아니면 False 반환

```
class MyCmp2:
    def __lt__(self, y):
        return 1 < y

m = MyCmp2()
print m < 10 # m.__lt__(10)을 호출
print m < 2
print m < 1
```

```
True
True
False
```

▣ 연산자 중복

2. 비교 연산자 중복

- 객체 m에 대한 m == 10연산의 행동 방식
  - m.__eq__()가 있다면 호출 결과을 그대로 반환
  - 정의된 m.__eq__()가 없고, __cmp__() 함수가 있을 경우
    * m.__cmp__() 호출 결과가 0이면 True 반환, 아니면 False 반환

```
class MyCmp3:
    def __eq__(self, y):
        return 1 == y

m = MyCmp3()
print m == 10 # m.__eq__(10)을 호출
m1 = MyCmp3()
print m == 1

class MyCmp4:
    def __init__(self, value):
        self.value = value
    def __cmp__(self, other):
        if self.value == other:
            return 0
m2 = MyCmp4(10)
print m2 == 10
```

```
False
True
True
```

▪value = 10

파이썬 프로그래밍

클래스와 연산자 중복 정의

한국기술교육대학교
온라인평생교육원

▣ 시퀀스/매핑 자료형의 연산자 중복

- 클래스를 개발할 때 다음 메소드들을 적절하게 구현하면 자신만의 시퀀스 자료형을 만들 수 있음
- 변경불가능한 (Immutable) 시퀀스 자료형 및 매핑 자료형을 위해 구현이 필요한 메소드

메소드	연산자
__len__(self)	len()
__contains__(self, item)	item in self
__getItem__(self, key)	self[key]
__setItem__(self, key, value)	self[key] = value
__delItem__(self, key)	del self(key)

▪item 인자가 contains의 두 번째 item 인자로 들어옴
▪itme이 self 안에 존재하는지 알아보는 멤버쉽 테스트 연산자와 매핑
▪[key] → 인덱스 연산
▪매핑 자료형이면 검색 연산이 됨
▪튜플, 문자열 같은 변경 불가능한 것 → setitem 구현 X

▣ 시퀀스/매핑 자료형의 연산자 중복

1. 인덱싱

- len(s1) --> s1.__len__() 메소드 호출
- sl[4] --> s1.__getitem__(4) 호출
- IndexError
  - 시퀀스 자료형이 범위를 벗어난 인덱스 참조 요구시에 발생됨
  - 리스트, 튜플, 문자열등에서도 동일한 조건에서 발생됨

▪sequare → 어떤 숫자의 제곱에 해당하는 것을 반환
▪s1 = Square(10) → 10은 end에, s1은 10(end값)이 할당되어 있음
▪len() → __len__()
▪s1[1] → __getitem__(self, k)
▪len() → __len__()
▪s1[1] → __getitem__(self, k)

▣ 시퀀스/매핑 자료형의 연산자 중복

1. 인덱싱

```
class Square:
    def __init__(self, end):
        self.end = end
    def __len__(self):
        return self.end
    def __getitem__(self, k):
        if k < 0 or self.end <= k:
            raise IndexError, k
        return k * k

s1 = Square(10)
print len(s1) # s1.__len__()
print s1[1] #s1.__getitem__(1)
print s1[4]
print s1[20]
```

```
10
1
16
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-78c6c0117c4f> in <module>()
     13 print s1[1] #s1.__getitem__(1)
     14 print s1[4]
---> 15 print s1[20]

<ipython-input-3-78c6c0117c4f> in __getitem__(self, k)
      6     def __getitem__(self, k):
      7         if k < 0 or self.end <= k:
----> 8             raise IndexError, k
      9         return k * k
     10

IndexError: 20
```

▣ 시퀀스/매핑 자료형의 연산자 중복

1. 인덱싱

- 다음 for 문은 s1에 대해 __getitem()__ 메소드를 0부터 호출하여 IndexError가 발생하면 루프를 중단한다.

```
for x in s1:
    print x,
```

```
0 1 4 9 16 25 36 49 64 81
```

▪s1 안에서 객체 x를 꺼내는 것
▪for x in s1: → __getitem__ 호출하여 나온 결과값을 하나씩 x에 삽입
▪인덱스 error가 발생하면 for ~ in 구문 멈춤
▪if 구문을 만족시키지 않으면 계속 진행
▪k가 10이면 if 절을 만족하여 IndexError
▪__getitem__ 메소드가 호출
▪getitem에 반드시 error를 발생시켜 for~in 구문 마침

▣ 시퀀스/매핑 자료형의 연산자 중복

1. 인덱싱

- __getitem__() 메소드가 정의되어 있다면 다른 시퀀스 자료형으로 변환이 가능

```
print list(s1)
print tuple(s1)
```

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

▪list, tuple → __getitem__을 호출

- 위에서 알 수 있듯이 파이썬은 내장 자료형과 개발자가 정의한 자료형에 대해 일관된 연산 적용이 가능
  - 파이썬 언어의 장점: 일관된 코딩 스타일 유지

▪s1 객체는 Square(10) 객체이지만 여러 연산자, 메소드 사용 가능

▣ 시퀀스/매핑 자료형의 연산자 중복

2. 매핑 자료형의 연산자 중복

```
class MyDict:
    def __init__(self, d = None):
        if d == None: d = {}
        self.d = 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__호출
m['day'] = 'light'      #__setitem__호출
m['night'] = 'darkness' #__setitem__호출
print m
print m['day']          #__getitem__호출
print m['night']        #__getitem__호출
print len(m)            #__len__호출
```

```
<__main__.MyDict instance at 0x10bb37638>
light
darkness
2
```

▣ 시퀀스/매핑 자료형의 연산자 중복

2. 매핑 자료형의 연산자 중복

▪MyDic : 스스로 dictionary를 정의
▪none 이 디폴트 인수 → 기본인수가 none
▪어떤 값이 존재하면 self.d에 들어감 → 만들려는 인스턴스에 식별자 생성
▪setitem의 두 번째 인자 k : key 값, 세 번째 인자 v : value
▪len(self.d) → self가 들고 있는 d의 길이
▪m이 현재 가지고 있는 d에 setitem이 불려짐
▪MyDict() 인자가 없으므로 빈 문자열이 들어감
▪k → 'day', v → 'light'
▪사전 안에는 아이템이 2개 존재
▪len(m) → m이 가지고 있는 __len__ 호출 → 가지고 있는 사전의 길이

▣ 시퀀스/매핑 자료형의 연산자 중복

2. 매핑 자료형의 연산자 중복

```
class MyDict:
    def __init__(self, d=None):
        if d == None: d = {}
        self.d = d
    def __getitem__(self, k):
        return self.d[k]
    def __setitem__(self, k, v):
        self.d[k] = v
    def __len__(self):
        return len(self.d)
    def keys(self):
        return self.d.keys()
    def values(self):
        return self.d.values()
    def items(self):
        return self.d.items()

m = MyDict({'one':1, 'two':2, 'three':3})
print m.keys()
print m.values()
print m.items()
```

```
['three', 'two', 'one']
[3, 2, 1]
[('three', 3), ('two', 2), ('one', 1)]
```

▪d에는 사전 위치가 할당됨

파이썬 프로그래밍

클래스와 연산자 중복 정의

한국기술교육대학교
온라인평생교육원

▣ 문자열 변환과 호출 가능 객체

1. 문자열로 변환하기

1) __repr__
- 객체를 대표하여 유일하게 표현할 수 있는 공식적인 문자열
- eval() 함수에 의하여 같은 객체로 재생성 될 수 있는 문자열 표현

2) __str__
- 객체의 비공식적인 문자열 표현
- 사용자가 보기 편한 형태로 자유롭게 표현될 수 있음

```
class StringRepr:
    def __repr__(self):
        return 'repr called'
    def __str__(self):
        return 'str called'

s = StringRepr()
print s
print str(s)
print repr(s)
print `s`
```

```
str called
str called
repr called
repr called
```

▪__repr__에 대응되는 역함수가 존재 → eval() 함수
▪단순하게 s 프린트하면 __str__이 호출
▪repr(s) 내장함수 활용 시 __repr__이 호출
▪`s`로 프린트하면 __repr__이 호출
▪일반적으로 __str__이 가장 많이 사용됨

▣ 문자열 변환과 호출 가능 객체

1. 문자열로 변환하기

- __str__() 호출시
  - __str__()가 정의되어 있지 않으면 __repr__()이 대신 호출됨

```
class StringRepr:
    def __repr__(self):
        return 'repr called'

s = StringRepr()
print s
print repr(s)
print str(s)
print `s`
```

```
repr called
repr called
repr called
repr called
```

▪s와 str함수를 프린트할 때, __str__ 정의가 없으면 __repr__활용

▣ 문자열 변환과 호출 가능 객체

1. 문자열로 변환하기

- __repr__() 호출시
- __repr__()이 정의되어 있지 않으면 객체 식별자가 출력됨
- 대신하여 __str__()이 호출되지 않음

```
class StringRepr:
    def __str__(self):
        return 'str called'

s = StringRepr()
print s
print repr(s)
print str(s)
print `s`
```

```
str called
<__main__.StringRepr instance at 0x101d3f908>
str called
<__main__.StringRepr instance at 0x101d3f908>
```

▪repr()함수는 __str__을 찾지 않음

▪__repr__ 없으면 그대로 디폴트 객체 표현 양식이 출력됨

▣ 문자열 변환과 호출 가능 객체

2. 호출 가능한 클래스 인스턴스 만들기

- 클래스 인스턴스에 __call__ 메소드가 구현되어 있다면 해당 인스턴스는 함수와 같이 호출될 수 있다.

 - 인스턴스 x에 대해 x(a1, a2, a3)와 같이 호출된다면 x.__call(a1, a2, a3)__ 가 호출된다.

```
class Accumulator:
    def __init__(self):
        self.sum = 0
    def __call__(self, *args):
        self.sum += sum(args)
        return self.sum

acc = Accumulator()
print acc(1,2,3,4,5)
print acc(6)
print acc(7,8,9)
print acc.sum
```

```
15
21
45
45
```

▪함수는 호출가능한 객체

▪새로 만든 객체를 호출 가능한 객체로 만들려면 __call__사용

▪acc 인스턴스를 호출함

▪*args → 가변 인수 (여러 개의 인수를 튜플로 받아낼 수 있음)

▪self.sum에는 누적이 되어 있는 상태

▪acc라는 객체에 가로를 사용하여 호출이 가능하다고 명명함

▣ 문자열 변환과 호출 가능 객체

2. 호출 가능한 클래스 인스턴스 만들기

- 호출 가능 객체인지 알아보기

```
def check(func):
    if callable(func):
        print 'callable'
    else:
        print 'not callable'

class B:
    def func(self, v):
        return v
class A:
    def __call__(self, v):
        return v

a = A()
b = B()
check(a)
check(b)
print
print callable(a)
print callable(b)
```

```
callable
not callable

True
False
```

▣ 문자열 변환과 호출 가능 객체

2. 호출 가능한 클래스 인스턴스 만들기

- class A는 __call__이 존재 → a() 사용하면 __call__이 호출됨
- class B는 __call__이 없으므로 b() 형태로 사용 X
- check 내장함수 없이 바로 callable() 함수 사용 가능
- callable()함수의 인자로 class 이름 사용 가능
- class A, B 모두 callable 함 → 모든 클래스는 callable 함