연산자 오버로딩

1 연산자 오버로딩의 정의

① 오버로딩(Overloading)의 의미

사전적 의미

"과적하다, 과부하가 걸리게 하다"

컴퓨터에서의 의미

- "메소드의 중복정의"
- 즉, 같은 이름의 메소드를 사용하는 것
  - ➡ 자바에서는 매개변수의 타입이나 개수 등을 달리해서 사용

오버로딩의 예시

- 하나의 노래 검색 함수를 두 가지로 활용 가능
  - ➡ 노래 검색(제목) : 제목으로 노래 검색
  - ➡ 노래 검색(가수) : 가수로 노래 검색

연산자 오버로딩

1 연산자 오버로딩의 정의

2 파이썬에서의 오버로딩

**파이썬에서 같은 이름의 메소드를 사용하면
늦게 정의한 메소드로 덮어쓰기 됨**

```
class A:
    def func(self,a):
        return "hello"
    def func(self):
        return "world"

a = A()
print(a.func())
print(a.func(1))
```

```
world

---------------------------
TypeError
<ipython-input-28-cd4b40bc610f
```

연산자 오버로딩

1 연산자 오버로딩의 정의

3 연산자 오버로딩

기존 약속되어 있는(__) add 메소드를 재정의해서 해당 클래스에서 객체 간 덧셈 연산을 가능하게 함

➡ 특수한 메소드

```
a = 1
b = 2
print(a+b)
```

```
3
```

```
class A:
    def __init__(self, i):
        self.i = i

n = A(40)
print(n+1)
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most re
<ipython-input-17-6dcd0f700bdc> in <module>()
      4
      5 n = A(40)
----> 6 print(n+1)

TypeError: unsupported operand type(s) for +: 'A' and 'int'
```

```
class A:
    def __init__(self, i):
        self.i = i
    def __add__(self,other):
        return self.i + other
n = A(40)
print(n+1)
```

```
41
```

연산자 오버로딩

1 연산자 오버로딩의 정의

3 연산자 오버로딩

인스턴스의 사칙연산

```
class MyInteger:
    def __init__(self, i):
        self.i = i

    def __str__(self):
        return str(self.i)

    def __add__(self, other):
        return self.i + other

    def __sub__(self, other):
        return self.i - other
```

```
i = MyInteger(10)
print(str(i))
i = i + 10
print(i)
i += 15
print(i)
```

```
10
20
35
```

```
i = MyInteger(10)
i *= 10
print(i)
```

```
--------------------
TypeError
```

```
def __mul__(self, other):
    return self.i * other
```

연산자 오버로딩을 하지 않으면
인스턴스 간 연산이 되지 않음

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

1 수치 연산자 오버로딩

- 해당 메소드들을 클래스에 오버로딩하면 연산자를 활용해 인스턴스들의 연산이 가능함

  ➡ 주의! 나누기의 경우 파이썬 버전이 2.x 에서 3.x으로 오면서 div에서 truediv로 변경

메소드 (Method)	연산자 (Operator)	인스턴스 객체(O)에 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
__truediv__(self, B)	/	O / B, O /= B

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

1 수치 연산자 오버로딩

- 연산자 왼쪽에 피연산자, 연산자 오른쪽에 객체가 오는 경우

  ➡ 메소드 앞에 r을 붙여야 함

```
class MyInteger:
    def __init__(self, i):
        self.i = i

    def __str__(self):
        return str(self.i)

    def __add__(self, other):
        return self.i + other

    def __sub__(self, other):
        return self.i - other

    def __mul__(self, other):
        return self.i * other

    def __truediv__(self, other):
        return self.i / other
```

```
i = MyInteger(10)
print(i + 10)

i = MyInteger(10)
print(10 + i)
```

```
20

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-97ac4b550484> in <module>()
      3
      4 i = MyInteger(10)
----> 5 print(10 + i)

TypeError: unsupported operand type(s) for +: 'int' and 'MyInteger'
```

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

1 수치 연산자 오버로딩

```
class MyInteger:
    def __init__(self, i):
        self.i = i

    def __str__(self):
        return str(self.i)

    def __radd__(self, other):
        return self.i + other

    def __rsub__(self, other):
        return self.i - other

    def __rmul__(self, other):
        return self.i * other

    def __rtruediv__(self, other):
        return self.i / other
```

```
i = MyInteger(10)
print(10 + i)

i = MyInteger(10)
print(i+10)
```

```
20
---------------------------
TypeError
```

클래스에 필요한 연산이
무엇인지 확인하고
꼼꼼하게 오버로딩해야 함 !

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

2 비교 연산자 오버로딩

- 비교 연산자 오버로딩을 통해 객체 간 비교 연산 가능

메소드	연산자
__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

```
class MyInteger:
    def __init__(self, i):
        self.i = i

    def __gt__(self, y):
        return self.i > y

    def __ge__(self, y):
        return self.i >= y

    def __lt__(self, y):
        return self.i < y

    def __le__(self, y):
        return self.i <= y

    def __eq__(self, y):
        return self.i == y

    def __ne__(self, y):
        return self.i != y
```

```
c = MyInteger(10)
print(c > 1)
print(c >= 1)
print(c < 1)
print(c <= 1)
print(c == 1)
print(c != 1)
```

```
True
True
False
False
False
True
```

```
c = MyInteger(10)
d = MyInteger(20)
print(c > d)
print(c >= d)
print(c < d)
print(c <= d)
print(c == d)
print(c != d)
```

```
False
False
True
True
False
True
```

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

3 시퀀스/매핑 자료형의 연산자 오버로딩

- 클래스에 다음 연산자들을 활용해 자신만의 시퀀스 자료형을 만들 수 있음

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(k)

```
class MyClass:
    def __init__(self, end):
        self.end = end

    def __len__(self):
        return self.end

s1 = MyClass(10)
print(len(s1))

s2 = 10
print(len(s2))
```

```
10

---------------------------------------------
TypeError                                 Tra
<ipython-input-58-d9ff78e9328b> in <module>()
     10
     11 s2 = 10
---> 12 print(len(s2))

TypeError: object of type 'int' has no len()
```

len() 내장 함수는
시퀀스 자료형이 아닌 숫자는
오류가 발생하지만
연산자 오버로딩을 활용해
기능을 변경할 수 있음

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

3 시퀀스/매핑 자료형의 연산자 오버로딩

- 내장 자료형과 개발자가 정의한 자료형에 대해 일관된 연산 적용이 가능

  ➡ 일관된 코딩 스타일을 유지할 수 있음

IndexError란
시퀀스 자료형이 범위를 벗어난
인덱스를 참조할 때 발생하는 오류

```
class MyClass:
    def __init__(self, end):
        self.end = end

    def __getitem__(self, k):
        if k < 0 or self.end <= k:
            raise IndexError("Out of Index - " + str(k))
        return k * k

s1 = MyClass(10)
print(s1[0])
print(s1[1])
print(s1[4])
print(s1[9])

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

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

사용자가 정의한 MyClass 객체를
리스트, 튜플 객체로 변경할 수 있음

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

올바른 연산자 오버로딩

isinstance (인스턴스, 클래스)

해당 객체가 어떤 클래스로부터 만들어졌는지 확인할 수 있음

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

```
i = A(10)
i = i+1
print(i)
print(type(i))
print(isinstance(i,A))
```

```
11
<class 'int'>
False
```

```
i = i - 10
print(i)
```

```
1
```

연산자 오버로딩

2 연산자 오버로딩의 종류와 활용

올바른 연산자 오버로딩

isinstance (인스턴스, 클래스)

해당 객체가 어떤 클래스로부터 만들어졌는지 확인할 수 있음

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

```
i = A(10)
i = i+1
print(i)
print(type(i))
print(isinstance(i,A))
```

```
<__main__.A object at 0x7fbd2828a198>
<class '__main__.A'>
True
```

```
i = i - 10
print(i)
```

```
---------------
TypeError
```

상속과 다형성

1 상속

① 상속(Inheritance)의 의미

사전적 의미	“뒤를 이음, 친족관계가 있는 사람이 재산을 물려줌”
컴퓨터에서의 의미	“클래스의 속성과 메소드를 부모가 자식에게 물려주는 것”

- class 부모 클래스:
  내용

  class 자식 클래스(부모 클래스):
  내용
- 상속받은 자식 클래스는 부모 클래스의 속성과 메소드를 사용할 수 있음

  ➡ 자식 클래스의 이름공간에 부모 클래스의 이름공간이 포함됨

상속과 다형성

1 상속

2 상속의 이유

- 코드를 재사용할 수 있음
- 상속받은 자식 클래스는 부모 클래스의 모든 기능을 그대로 사용 가능
- 자식 클래스는 필요한 기능만 정의하거나, 기존의 기능을 변경(Overriding)할 수 있음

```
class Person:
    def __init__(self, name, phone=None):
        self.name = name
        self.phone = phone

    def __str__(self):
        return '<Person {0} {1}>'.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
```

```
p1 = Person('홍길동', 1498)
print(p1.name)
print(p1)
print()
m1 = Employee('손창희', 5564, '대리', 200)
print(m1.name, m1.position)
print(m1)
```

```
홍길동
<Person 홍길동 1498>

손창희 대리
<Person 손창희 5564>
```

상속과 다형성

1 상속

③ 메소드 오버라이딩(Overriding)

사전적 의미	“최우선시되는, 기각하다”
컴퓨터에서의 의미	“메소드의 재정의”

상속과 다형성

1 상속

3 메소드 오버라이딩(Overriding)

자식 클래스에서 부모 클래스에 정의된 메소드를 재정의하여 대치하는 것

```
class Person:
    def __init__(self, name, phone=None):
        self.name = name
        self.phone = phone

    def __str__(self):
        return '<Person {0} {1}>'.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

    def __str__(self):
        return '<Person {0} {1} {2} {3}>'.format(self.name, self.phone, self.position, self.salary)
```

오버라이딩 X

```
p1 = Person('홍길동', 1498)
print(p1.name)
print(p1)
print()
m1 = Employee('손창희', 5564, '대리', 200)
print(m1.name, m1.position)
print(m1)
```

```
홍길동
<Person 홍길동 1498>

손창희 대리
<Person 손창희 5564>
```

오버라이딩 O

```
p1 = Person('홍길동', 1498)
print(p1.name)
print(p1)
print()
m1 = Employee('손창희', 5564, '대리', 200)
print(m1.name, m1.position)
print(m1)
```

```
홍길동
<Person 홍길동 1498>

손창희 대리
<Person 손창희 5564 대리 200>
```

상속과 다형성

2 다형성

① 다형성(Polymorphism)의 의미

다른 클래스에 속한 같은 이름의 인스턴스들이
동일한 메소드 이름으로 호출할 경우
동적으로 선택되어 수행

'다양한 형태를 가질 수 있다'

상속과 다형성

2 다형성

② 다형성의 장점

1. 다른 클래스에 속한 같은 이름의 다양한 메소드들에게 유사한 작업을 수행시킬 수 있음

2. 추상클래스를 상속하는 다른 서브클래스 내에 작성된 같은 이름의 메소드를 다른 목적으로 사용할 수 있음

상속과 다형성

2 다형성

③ 파이썬에서 다형성의 장점

자료형의 타입이 동적으로 결정되므로
다형성을 적용하기에 훨씬 용이함

```
class Animal:
    def cry(self):
        return "엉엉"

class Dog(Animal):
    def cry(self,a,b):
        return a * b

class Duck(Animal):
    def cry(self,a,b):
        return a + b

class Fish(Animal):
    pass
```

```
a = Dog()
b = Duck()
c = Fish()

print(a.cry("멍멍",3))
print(b.cry(4,3))
print(c.cry())
```

```
멍멍멍멍멍멍
7
엉엉
```

상속과 다형성

3 상속 관계 알아내기

isinstance (인스턴스, 클래스)	issubclass (A 클래스, B 클래스)
해당 인스턴스가 해당 클래스에 속하면 True, 속하지 않으면 False 반환	A 클래스가 B 클래스와 같거나 자식 클래스이면 True, 아니라면 False 반환

```
class A:
    pass

class B:
    def f(self):
        pass

class C(B):
    pass
```

```
a = A()

print(isinstance(a, A))
print(issubclass(A, B))
print(issubclass(C, B))
```

```
True
False
True
```

Run! 프로그래밍

Mission 1

클래스의 연산자 오버로딩을 활용해
더하기 연산이 실제로는 뺄셈이 되도록 코딩

```
class A():
    def __init__(self,i):
        self.i = I
    def __str__(self):
        return str(self.i)
    def __add__(self,other):
        return A(self.i - other)

a = A(10)
print(a + 5)
```

Run! 프로그래밍

Mission 2 | 상속을 활용한 메소드 코딩

```
class Asia:
    def __init__(self,name):
        self.name = name
    def show(self):
        return "해당 국가는 아시아에 있습니다.“
```

정답

```
class Korea(Asia):
    def __init__(self,name,population,capital):
        Asia.__init__(self, name)
        self.population = population
        self.capital = capital

            def show_name(self):
        return '국가 이름은 : ', self.name

a = Asia('대한민국’)
b = Korea('대한민국','5천만','서울’)
print(a.show())
print(b.show_name())
```