ch.6 객체와 클래스

혜란 발표~

객체

- 파이썬애서는 숫자에서 모듈까지 파이썬의 모든 것은 객체 (변수는 데이터를 라벨링/참조하는 정도)

1. value : 메모리에 기록된 내용. 가변객체는 바꿀 수 있지만, 불변객체는 바꿀 수 없다
2. type : 데이터의 종류로, 유형에 따라 그 값을 어떻게 읽고 다루어야 할지가 결정된다
3. indentify : 객체가 메모리에 위치한 주소값. value와 type이 동일한 데이터가 메모리 공간에 여러 개 존재할 수 있지만, 이들은 서로 별개의 객체이며 identify이 서로 다르다.

```
a = 123

print(a) #value
print(type(a)) # type
print(id(a)) #identify

#output
123
<class 'int'>
4538269632
```

▼ 문제

```
a = 1.0
b = 1
print(type(a)==type(b))
print(id(a)==id(b))
print(a==b)
```

▼ 답

f f t

클래스

- int같은 데이터 유형을 표현하기 위한 데이터 → class

```
print(type(0))
print(type(type(0)))
print(type(type(type(0))))
#output
<class 'int'> # 0의 데이터 유형
<class 'type'> # int의 데이터 유형
<class 'type'> #type의 데이터 dbgud
```

- 클래스와 인스턴스

```
print(isinstance(111,int)) #true
print(isinstance('111',int))#false
```

객체가 어떤 클래스에 속할 때, 그 객체를 그 클래스의 인스턴스라고 부른다.

▼ 인스턴스 맹글기 두가지 방법

1. 리터럴(literal)

```
>>> 1789       # 정수 리터럴: 정수 인스턴스가 만들어진다
1789

>>> '파이썬'  # 문자열 리터럴: 문자열 인스턴스가 만들어진다
'파이썬'

>>> {'year': 1789}    # 사전 식: 사전 인스턴스 만들기
{'year': 1789}

>>> lambda x: x * x   # 람다 식: 함수 인스턴스 만들기
<function <lambda> at 0x7ffa0a43ce18>
```

2. <class name>()

```
>>> int()             # 정수의 기본 인스턴스 만들기
0

>>> int(1789)         # 1789에 대응하는 정수의 인스턴스 만들기
1789

>>> int('1789')       # '1789'에 대응하는 정수의 인스턴스 만들기
1789
```

- 오키 이제 알았고 클래스를 맹글어 보자

```
class madeInHaeran():
    def hi(self): #메서드의 매개변수에 꼭 self가 들어가야 한다 이때 self는 객체 haeran
        print('하용')

haeran = madeInHaeran()
haeran.hi()
```

클래스 상속

언제 씀? : 기존에 있는 클래스에 기능을 추가,변경하고 싶다 → 코드 복붙해서 새 클래스 만들면 가성비 x

→ 상속받아서 코드 복붙을 방지하자!

```
class Book(): #슈퍼 클래스, 베이스 클래스
    def __init__(self,number):
        self.number = number

    def borrow(self,borrow_person):
        self.borrow_person = borrow_person

class customBook(Book): #서브 클래스, 자식 클래스
    def lend(self,lend_person):
        self.lend_person = lend_person

b = customBook(3)
b.lend('haeran')
b.borrow('tt')
print(b.lend_person)
```

```
print(b.borrow_person)
print(b.number)

#output
haeran
tt
3
```

메서드 오버라이드

부모 클래스의 메서드를 재정의 하고싶어!

```
class Book():
    def __init__(self,number):
        self.number = number

    def borrow(self,borrow_person):
        self.borrow_person = borrow_person

class customBook(Book):
    def __init__(self, date): #부호 함수의 __init__을 오버라이드
        self.date = date

    def lend(self,lend_person):
        self.lend_person = lend_person

b1 = customBook(20190607)
print(b1.date)
print(b1.number)

#out
20190607
Traceback (most recent call last):
  File "/Users/shinhaeran/Desktop/test1.py", line 17, in <module>
    print(b1.number)
AttributeError: 'customBook' object has no attribute 'number'
```

부모에게 도움받기: super

자식클래스에서 부모 클래스의 메서드를 호출하고 싶어! → super() 사용

```
class Book():
    def __init__(self,number):
        self.number = number

    def borrow(self,borrow_person):
        self.borrow_person = borrow_person

#init을 부모에 있는 number 그대로 쓰고싶어!
class customBook(Book):
    def __init__(self, number, date):
        super().__init__(number) #book.__init__호출 -> self 인자를 슈퍼 클래스로 전달
        self.date = date

    def lend(self,lend_person):
        self.lend_person = lend_person

b1 = customBook(3,20190607)
print(b1.date)
print(b1.number)
```

자신: self

파이썬은 인스턴스 메서드의 첫 번째 인자로 self를 포함해야 한다.

적절한 객체의 속성과 메서드를 찾기 위해 self인자를 사용한다.

```
class Book():
    def __init__(self,number):
        self.number = number

    def borrow(self,borrow_person):
        self.borrow_person = borrow_person

    def aa(self):
        print('얍')


b1 = Book(3)
```

```
b1.aa()
Book.aa(b1)
```

get/set 속성값과 프로퍼티

java : private 객체에 접근하기 위해 getter(), setter() 써야함

python: 필요x, 접근제어자 키워드 x니까! → 그래도 속성에 직접 접근하는 것이 부담스러우면(왜,,?) 메서드를 만들어 보자

▼ python의 접근제어자

다른언어와 달리 private, public 등의 접근제어자 키워드가 존재하지 않고 작명법(naming)으로 접근제어를 합니다. public, private, protected에 대한 규칙은 다음과 같습니다.

public	private	protected
아무 밑줄이 접두사에 없어야 함 ex) num	두개의 밑줄 __이 접두사여야 함 ex) __num	한 개의 밑줄 _이 접두사여야 함 ex) _num
	접미사는 밑줄이 한 개까지만 허용 ex) __num_	
	접미사의 밑줄이 두 개 이상이면 public으로 간주 ex) __num__	

```
class Duck():
    def __init__(self, input_name):
        self.hiddem_name = input_name #hidden_name: 은닉된 멤버변수; 현재 블락에서만 접근 가능

    def get_name(self):
        print('inside th getter')
        return self.hiddem_name
    def set_name(self, input_name):
        print('inside th setter')
```

```
        self.hiddem_name = input_name

    name = property(get_name,set_name)

d1 = Duck('haeran')
print(d1.name)
print(d1.get_name())
d1.set_name('aaa')
print(d1.get_name())
```

- get_name, set_name 메서드를 name이라는 속성의 프로퍼티로 정의한다.
- property(getter메서드, setter메서드)

→ private 변수마다 하나하나 getter, setter함수 만드는건 너무 가성비가 떨어진다.

▼ → decorator를 사용하자!

- getter 메서드 앞에 @property 데커레이터를 쓴다
- setter 메서드 앞에 @name.setter 데커레이터를 쓴다

```
class Duck():
    def __init__(self, input_name):
        self.hiddem_name = input_name

    @property
    def name(self):
        print('inside th getter')
        return self.hiddem_name
    @name.setter
    def name(self, input_name):
        print('inside th setter')
        self.hiddem_name = input_name

    # name = property(get_name,set_name)

d1 = Duck('haeran')
print(d1.name)
d1.name = 'aaa'
print(d1.name)
```

메서드 타입

1. 인스턴스 메서드: 메서드의 첫번째 인자가 self인 경우. 일반적인 클래스를 생성할 때의 메서드 타입. 인스턴스 메서드의 첫번째 매개변수는 self고, 파이썬은 이 메서드를 호출할 때 객체를 전달한다
2. 클래스 메서드: 클래스 전체에 영향을 미침. @classmethod사용/메서드의 첫번째 매개변수는 클래스 자신 ; cls

```
class A():
    count = 0
    def __init__(self):
        A.count +=1
    def exclaim(self):
        print('나는 A클래스야')
    @classmethod
    def kids(cls):
        print('A has',cls.count,'littls objects') #A.count로 접근 가능

a1 = A()
a2 = A()
a2 = A()
A.kids()

#ouput
A has 3 littls objects
```

kids() 함수를 접근하기 위해 굳이 객체를 생성할 필요가 없다 ~~ 우왕 신기방기

덕타이핑

```
class Parrot:
      def fly(self):
          print("Parrot flying")

  class Airplane:
      def fly(self):
          print("Airplane flying")

  class Whale:
```

```
        def swim(self):
            print("Whale swimming")

    def lift_off(entity):
        entity.fly()

    parrot = Parrot()
    airplane = Airplane()
    whale = Whale()

    lift_off(parrot) # prints `Parrot flying`
    lift_off(airplane) # prints `Airplane flying`
    lift_off(whale) # Throws the error `'Whale' object has no attribute 'fly'`
```

- Parrot 클래스와 Airplane 클래스는 분명 서로 상속되거나 하는 그런 관계는 없습니다만, 내부에 동일한 메소드의 fly()메소드가 있는 것만으로 호출하는 `lift_off(entity)' 함수에서 fly가 정상적으로 실행된다
- 마지막 Whale 클래스는 해당 fly()메소드가 없기 때문에, AttributeError가 발생한다
- **속성과 메소드 존재에 의해 객체의 적합성이 결정된다.**

private 네임 맹글링

- 맹글링: 프로그래밍 언어 자체적으로 일정한 규칙에 의해 변수나 함수의 이름을 변경하는 것 →맹글링을 호출하는 것이 언더스코어(_)
- phthon 에서의 맹글링 : _Class + [name]
- private 네임 맹글링 : 클래스 정의 외부에서 볼 수 없도록 하는 속성에 대한 네이밍 컨벤션
- 속성 이름 앞에 __를 붙이자!
- 메서드에 __를 붙이면 다른 곳에서 이 모듈을 inport 불가

```
class Duck():
    def __init__(self,input_name):
        self.__name = input_name
    @property
    def name(self):
        print('inside the getter')
        return self.__name
    @name.setter
    def name(self, input_name):
        print('insider the setter')
```

```
        self.__name = input_name

d = Duck('ㅇ-<-<')
print(d.name) # 이거는 __name이 아니고 name의 getter메서드다 ~~
print(d.__name)

#output
inside the getter
ㅇ-<-<
Traceback (most recent call last):
  File "/Users/shinhaeran/Desktop/myProject/code/test.py", line 15, in <module>
    print(d.__name)
AttributeError: 'Duck' object has no attribute '__name'
```

▼ 사실 속성을 렬루 private으로 만든게 아니라서 접근은 가능하다

```
class Duck():
    def __init__(self,input_name):
        self.__name = input_name
    @property
    def name(self):
        print('inside the getter')
        return self.__name
    @name.setter
    def name(self, input_name):
        print('insider the setter')
        self.__name = input_name

d = Duck('ㅇ-<-<')
print(d.name) # 이거는 __name이 아니고 name의 getter메서드다 ~~
print(d._Duck__name)

#output
inside the getter
ㅇ-<-<
ㅇ-<-<
```

엥 d._Duck__name하니까 getter 메서드를 들가지 않았다 → __name에 직접 접근했다는 뜻

특수 메서드 (magic method)

- __<method name>__
- 특수한 용도에 사용하서 magic 같은 효과

디버깅 할 때 유용한 메서드

Notes	You Want...	So You Write...	And Python Calls...
#1	개체를 초기화	x = MyClass()	x.__init__()
#2	문자열의 "공식적인" 표현	repr(x)	x.__repr__()
#3	문자열의 "비공식적인" 값	str(x)	x.__str__()
#4	바이트 배열의 "비공식적인" 값	bytes(x)	x.__bytes__()
#5	서식화된 문자열 값	format(x, format_spec)	x.__format__(format_spec)

1. __init__() 메소드는 개체가 생성된 후에 호출됩니다. 실제 생성하는 과정을 제어하고 싶다면 __new__() 메소드를 사용할 수 있습니다.
2. 관습적으로, __repr__() 메소드는 유효한 파이썬 표현식인 문자열을 반환해야 합니다.
3. __str__() 메소드는 print(x)를 사용할 때 호출됩니다.
4. 바이트 형은 파이썬 3에만 있으므로 __bytes__()도 파이썬 3에만 있습니다.
5. 관습적으로 format_spec은 서식 명세를 위한 작은 언어에 부합해야 합니다. 파이썬 표준 라이브러리의 decimal.py은 자신의 __format__() 메소드를 제공합니다.

반복자처럼 작동하는 메서드

Notes	You Want...	So You Write...	And Python Calls...
#1	to iterate through a sequence	iter(seq)	seq.__iter__()
#2	to get the next value from an iterator	next(seq)	seq.__next__()
#3	to create an iterator in reverse order	reversed(seq)	seq.__reversed__()

1. __iter__() 메소드는 새로운 반복자를 만들 때마다 호출됩니다. 초기값을 이용해 반복자를 초기화하기 좋은 장소입니다.
2. __next__() 메소드는 반복자에서 다음 값을 받아오려고 할 때 호출됩니다. → for 문에서 stopIteration발생하면 멈춤
3. __reversed__() 메소드는 자주 사용되지 않습니다. 이미 존재하는 서열(sequence)을 받아 항목을 역순(마지막에서 처음으로)으로 yield하는 반복자를 반환합니다.

비교연산 특수 메서드

Notes	You Want...	So You Write...	And Python Calls...
Untitled	equality	x == y	x.__eq__(y)
Untitled	inequality	x != y	x.__ne__(y)
Untitled	less than	x < y	x.__lt__(y)
Untitled	less than or equal to	x <= y	x.__le__(y)
Untitled	greater than	x > y	x.__gt__(y)
Untitled	greater than or equal to	x >= y	x.__ge__(y)
Untitled	truth value in a boolean context	if x:	x.__bool__()

산술연산 메서드

Notes	You Want...	So You Write...	And Python Calls...
Untitled	addition	x + y	x.__add__(y)
Untitled	subtraction	x - y	x.__sub__(y)
Untitled	multiplication	x * y	x.__mul__(y)
Untitled	division	x / y	x.__truediv__(y)
Untitled	floor division	x // y	x.__floordiv__(y)
Untitled	modulo (remainder)	x % y	x.__mod__(y)
Untitled	floor division & modulo	divmod(x, y)	x.__divmod__(y)
Untitled	raise to power	x ** y	x.__pow__(y)
Untitled	left bit-shift	x << y	x.__lshift__(y)
Untitled	right bit-shift	x >> y	x.__rshift__(y)
Untitled	bitwise and	x & y	x.__and__(y)
Untitled	bitwise xor	x ^ y	x.__xor__(y)
Untitled	bitwise or	x | y	x.__or__(y)

컴포지션 composition

- 자식 클래스가 부모 클래스처럼 행동하고 싶을 때 상속/(composition = aggrigation) 사용
- → 근데 다른 클래스의 일부 기능을 쓰고 싶은데 전체 기능까지는 상속 받을 필요 x→ composition

```
class Calc:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        return self.x + self.y

    def subtract(self):
        return self.x - self.y

class Calc2:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self):
        return self.x + self.y

    def multiply(self):
        return self.x * self.y
```

▼ 여기서 Clac에서 Cal2의 multiply만 가져오고 싶으면 어케 바꿔야 할까?

```
class Calc:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.calc2 = Calc2(x, y) # 해당 클래스의 객체를 명시적으로 가져옴

    def add(self):
        return self.x + self.y

    def subtract(self):
        return self.x - self.y

    def multiply(self):
        return self.calc2.multiply() # 해당 클래스의 객체에 있는 메서드를 명시적으로 활용함



calc1 = Calc(1, 2)
print(calc1.multiply())
calc2 = Calc(x=2, y=3)  # 함수 인자와 마찬가지로 직접 인자명과 값을 써도 됨
print(calc2.multiply())
```