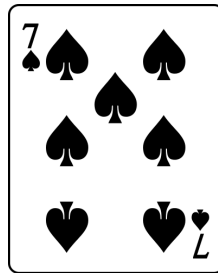


# 객체지향 프로그래밍

## 객체와 클래스

우리의 사고 대상이 되는 모든 데이터를 객체<sup>1</sup>object(오브젝트)로 표현할 수 있다. 객체 고유의 특성 또는 상태를 속성attribute이라고 하며, 객체가 고유로 갖고 있는 기능 또는 능력을 행위behavior라고 한다. 예를 들어, 다음 놀이카드playing card를 객체로 볼 수 있다. 특성이라고 할 수 있는 ‘스페이드’와 ‘7’, 상태라고 할 수 있는 ‘펼침’을 모두 이 객체의 속성으로 볼 수 있다.



놀이카드는 언제든지 뒤집을 수 있어야 한다. 따라서 놀이카드 객체는 ‘카드뒤집기’ 행위를 할 수 있는 기능이 있어야 한다. 이 객체에 ‘카드뒤집기’를 수행하면 ‘펼침’ 상태가 ‘가림’으로 바뀔 것이다.

객체를 중심으로 사고하여 프로그램을 작성하는 방식paradigm을 객체지향 프로그래밍Object-Oriented Programming(OOP)이라고 한다. 객체지향 프로그램은 데이터를 모두 객체로 취급하며 객체 끼리 메시지를 주고 받으며 프로그램이 작동한다. Python에서 객체의 속성은 속성변수attribute<sup>2</sup>로 정의하며, 객체의 행위는 메소드method(함수,프로시저)로 정의한다.

객체 (object)	
속성 (attribute)	행위 (behavior)
특성 / 상태	기능 / 능력
속성변수 (attribute)	메소드 (method)
변수	함수, 프로시저

위 카드의 속성과 행위를 Python 코드로 표현하면 다음과 같다.

속성변수	메소드
<pre>suit = "Spade" rank = "7" face_up = True</pre>	<pre>def flip():     face_up = not face_up</pre>

세 가지 속성은 모두 속성변수 suit, rank, value, face\_up으로 각각 지정할 수 있고, ‘카드뒤집기’ 행위는 flip() 메소드로 정의할 수 있다. flip() 메소드는 face\_up 속성변수의 값을 True에서 False, 또는 False에서 True로 바꾸는 기능을 갖고 있다. 이제 놀이카드 객체를 어떻게 만드는지 알아보자.

<sup>1</sup> object의 우리말로 사용하는 객체(客體)는 ‘생각과 행동의 대상물’로 해석하는게 가장 가깝다.

<sup>2</sup> 다른 객체지향언어에서는 속성변수를 필드field변수라고 하기도 한다.

## 1. 클래스 정의, 객체 생성

객체지향 프로그램에서 객체object는 메모리에 거주하는 실물instance이다. 클래스class는 실물 객체를 만들어내는 일종의 형판template이다. 객체의 모든 것(속성과 행위)을 정해주는 청사진blueprint이라고 할 수 있다. 클래스 하나로 실물 객체를 몇 개든 만들어낼 수 있다.

클래스 (class) 청사진 (blueprint) 형판 (template)	객체 (object) 실물 (instance)
1	n

### 클래스 정의

놀이카드 객체를 만드는 Card 클래스 사례를 가지고 클래스를 정의하는 방법을 공부해보자.

```
class Card:
    pass
```

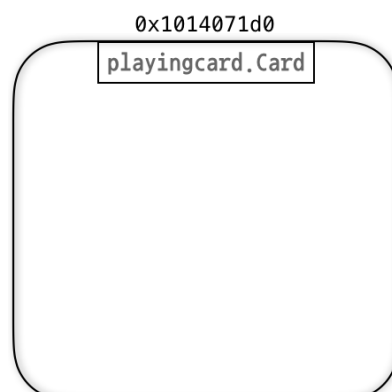
클래스의 정의는 위 코드와 같이 작성한다. 키워드 `class`를 쓰고 이어서 클래스 이름을 정해준다. 클래스 이름은 대문자로 시작하는 것이 관례이다. 두 단어 이상을 나열하고 싶은 경우, 새로운 단어가 시작할 때마다 대문자로 시작하도록 하면서 붙여쓴다. `pass`는 아무 일도 하지 않는 명령이므로 이 클래스는 속성과 행위가 없는 빈 클래스이다. 내부가 비어있긴 하지만 이 클래스로도 객체를 생성할 수는 있다. 이 클래스를 `playingcard.py` 파일에 저장하고 실행창에서 객체를 생성해보자.

### 객체 생성

실물 객체는 클래스 이름을 호출하여 생성할(만들) 수 있다. 실행창에서 객체를 생성해보자.

```
>>> from playingcard import *
>>> Card()
<playingcard.Card object at 0x1014071d0>
```

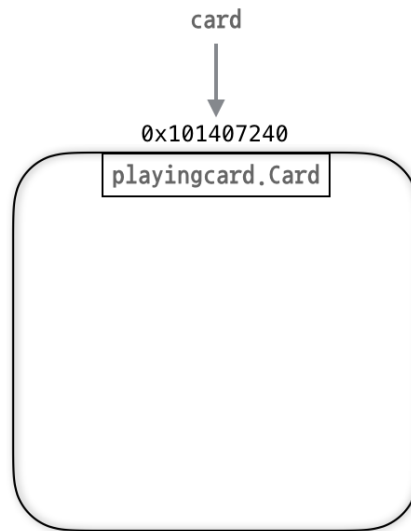
`Card()`를 호출하면 Card 클래스에서 정의한 청사진대로 실물 객체를 하나 만들고, 위와 같은 메시지를 프린트 한다. 이 메시지에는 방금 생성한 객체가 `playingcard` 모듈에 있는 Card 클래스로 생성한 객체임을 알려주는 정보와 객체가 자리잡고 있는 메모리 주소 `0x1014071d0`가 포함되어 있다. 클래스 정의에는 파라미터가 없더라도, 호출할 때는 뒤에 빈 괄호를 반드시 붙여야 함을 명심하자. 생성된 실물 객체를 그림으로 형상화하면 다음과 같다.



이 메모리 주소로 이 객체를 참조할 수 있지만 실제 메모리 주소를 기억하여 사용하는 작업은 보나 마나 성가신 일이므로, 다음과 같이 생성직후 지정문으로 이름을 붙여두고 사용한다. (작명 규칙은 일반 변수와 동일함)

```
>>> card = Card()
```

이 명령을 실행한 직후를 그림으로 형상화하면 다음과 같다.



이젠 메모리주소 대신 `card` 라는 이름으로 이 객체에 접근할 수 있다.

```
>>> card
<playingcard.Card object at 0x101407240>
```

## 2. 생성메소드 : 속성 정의 및 활용

### 생성 메소드 `__init__`

클래스 내부에 정의해두면 객체를 생성할 때 저절로 호출되어 실행되는 `__init__` 메소드가 있는데, 이를 **생성메소드** `constructor` 라고 한다. 새로 생성하는 객체 속성변수의 초기값을 설정하는 용도로 주로 쓴다. 이와 같이 미리 이름이 지정되어 있어 저절로 호출되어 실행되는 메소드는 이름의 앞뒤 양쪽에 `'__'` 를 붙여서 구별한다.

`Card`가 지니고 있어야 할 속성은 종류 `suit`과 계급 `rank`, 펼침상태 `face up/down`인데, `__init__` 생성메소드로 이 속성변수의 초기값을 다음과 같이 설정할 수 있다.

```
1 class Card:
2     def __init__(self, suit, rank, face_up):
3         self.suit = suit
4         self.rank = rank
5         self.face_up = face_up
```

줄 2~5가 생성메소드 정의이다. 생성메소드의 이름은 `__init__` 로 미리 정해져있다. 클래스 내부에 정의하는 모든 메소드의 첫 파라미터는 무조건 예외없이 `self` 라야 한다. (`self` 인수가 없는 메소드는 객체 소속이 아니라 클래스 소속으로 분류되어, 객체가 없어도 불러 쓸 수 있는 메소드로 뒤에서 따로 공부

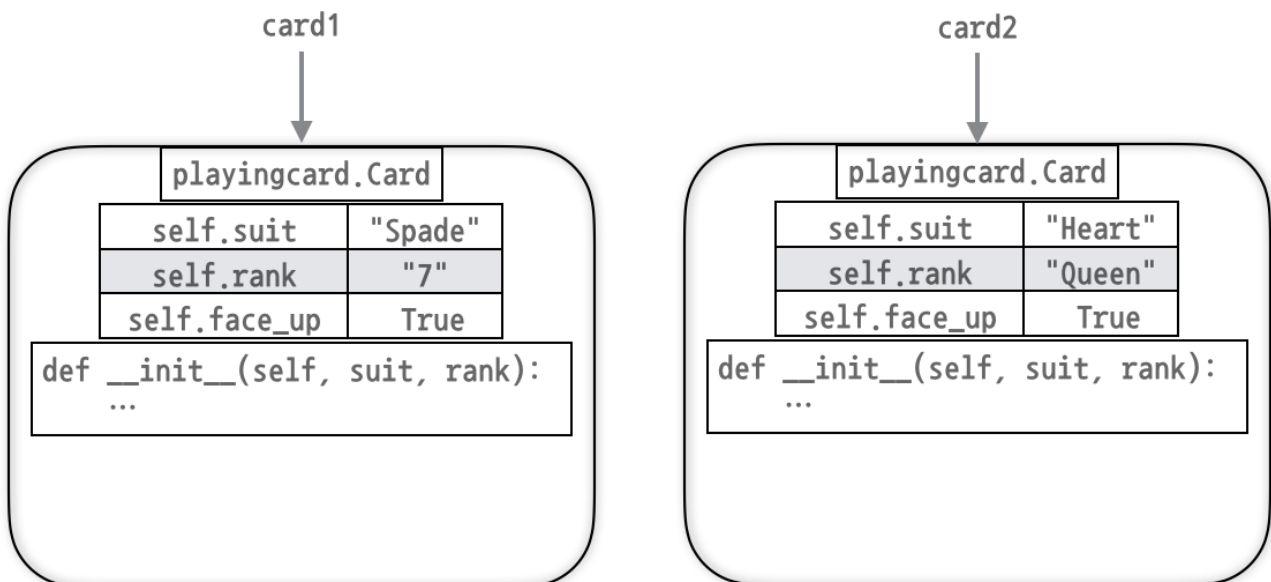
한다.) `self`는 생성될 객체 자신을 가리키는 이름이다. 3개의 파라미터 `suit`, `rank`, `face_up`을 추가로 만들었는데, 이 파라미터 변수들의 값은 객체를 생성하면서 인수로 전달받는다. 줄 3~5의 세 문장은 이 클래스의 속성변수의 초기값을 설정하기 위한 지정문이다. 속성변수는 일반 변수와 구별을 위해서 반드시 앞에 `self.`를 붙여야 한다. 앞에 `self.`가 붙음으로 해서 해당 속성변수의 소속이 명확해진다. 속성변수는 클래스 내부 전역에서 사용할 수 있다. 인수로 전달받은 `suit`, `rank`, `face_up`의 값은 각각 생성되는 객체 자신의 속성변수인 `self.name`, `self.balance`, `self.face_up`으로 각각 지정될 것이다.

## 객체 생성

이제 실행창에서 다음과 같이 객체를 생성해보자.

```
>>> from playingcard import *
>>> card1 = Card("Spade", "7", True)
>>> card2 = Card("Heart", "Queen", True)
```

그러면 `Card` 객체가 2개 생성되면서 생성메소드가 저절로 실행되어 다음 그림과 같은 형상이 된다.

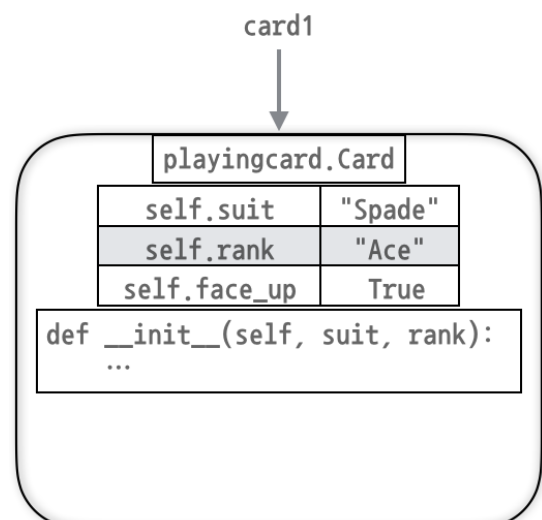


가운데 테이블은 각 객체마다 고유의 속성변수의 값을 지니고 있음을 보여준다.

## 객체 속성의 활용 및 수정

`self.suit`, `self.rank`, `self.face_up`과 같은 속성변수의 값은 클래스 내부에서 자유롭게 설정, 변경, 사용할 수 있다. 그런데 클래스의 바깥에서도 그럴까? 이를 확인하기 위해서 다음 코드를 실행해보자. 오른쪽 그림은 아래 명령을 실행한 후의 객체의 모습이다.

```
>>> card1.suit
'Spade'
>>> card1.rank
'7'
>>> card1.rank = "Ace"
>>> card1.rank
'Ace'
```



이 실행사례에서 볼 수 있듯이 클래스의 바깥에서도 속성변수의 값을 자유롭게 사용하고 변경할 수 있다. 객체 내부의 속성변수가 바깥에 고스란히 노출되어 있어 누구나 객체의 속살을 들여다볼 수 있고 멋대로 고칠 수도 있는 것이다. 사실 이와 같이 객체를 외부에 직접 노출시키는 건 보안상 위험하고 바람직하지 않다. 이를 어떻게 극복할지 뒤에서 배운다.

### 파라미터의 기본값 지정

위에서는 `self.face_up` 속성변수의 경우 다음과 같이 파라미터에서 기본값을 지정할 수도 있다.

```
1 class Card:
2     def __init__(self, suit, rank, face_up=True):
3         self.suit = suit
4         self.rank = rank
5         self.face_up = face_up
```

그러면 `True` 값으로 초기화하는 경우 해당 인수를 다음과 같이 생략해도 된다.

```
>>> from playingcard import *
>>> card = Card("Spade", "7")
>>> card.face_up
True
>>> card = Card("Spade", "7", False)
>>> card.face_up
False
```

### 객체의 간판 문자열

위에서 공부했듯이 객체를 실행창에서 참조하면 미리 정해놓은 형식에 맞추어 클래스 이름과 메모리 주소 정보를 문자열로 만들어 보여 준다. 어떤 객체인지 외부에 보여줄 일종의 간판이라고 할 수 있는데, 실제로 실행창에서 객체를 보거나 프린트하면 다음과 같이 나타난다.

```
>>> card
<playingcard.Card object at 0x101208ef0>
>>> print(card)
<playingcard.Card object at 0x101208ef0>
```

이 간판 문자열의 형식은 원하는 경우 입맛에 맞게 고칠 수 있다. `__str__` 메소드를 정의하면 된다. 호출하지 않아도 상황에 따라 저절로 호출되는 메소드이다. `Card` 객체는 각기 고유한 종류와 계급이 있으므로 이를 간판으로 하는 문자열로 정하면 유용할 것이다. `__str__` 메소드는 다음과 같이 정의할 수 있다.

```
1 class Card:
...     ...
10     def __str__(self):
11         if self.__face_up:
12             return self.__suit + "." + self.__rank
13         else:
14             return "xxxxx" + "." + "xx"
...     ...
```

카드가 펼쳐져 있으면 종류와 계급을 보여주고, 그렇지 않으면 보여주지 않게 정의했다. 이 메소드를 다음과 같이 만들어 추가하여 클래스를 완성하고, 이 메소드가 어떤 역할을 하는지 직접 실행하여 체험해보자.

```
>>> from playingcard import *
>>> card = Card("Spade", "7")
>>> card
<playingcard.Card object at 0x1010f9ef0>
>>> print(card)
Spade.7
>>> card.flip()
>>> print(card)
XXXXX.XX
```

Card 객체를 프린트하면 이제 `__str__` 메소드에서 정의한 대로 문자열을 만들어 내준다. `print` 명령을 실행하면 `__str__` 메소드를 내부적으로 바로 호출했음을 알 수 있다.

### 3. 메소드 정의 및 호출

이제 `PlayingCard` 클래스의 `flip` 메소드를 다음과 같은 실행의미를 갖도록 작성해보자.

메소드	실행 의미
<code>flip()</code>	<code>PlayingCard</code> 객체의 <code>face_up</code> 속성변수 논리값을 역으로 바꾼다.

`PlayingCard` 클래스에 `flip` 메소드를 다음과 같이 추가할 수 있다.

```
1 class PlayingCard:
2     def __init__(self, suit, rank, face_up=True):
3         self.suit = suit
4         self.rank = rank
5         self.face_up = face_up
6
7     def flip(self):
8         self.face_up = not self.face_up
```

진하게 표시된 줄 7-8이 메소드의 정의이다. 함수/프로시저를 정의하는 것과 같은 요령으로 정의한다. `flip` 메소드도 객체 소속이므로 `self` 파라미터를 반드시 명시해야 한다. 이 메소드 정의 내부에서 `self` 를 사용할 수 있도록 하기 위함이다. 속성변수 `self.face_up`의 논리값이 `True` 이면 `False` 로, `False` 이면 `True` 로 변경한다.

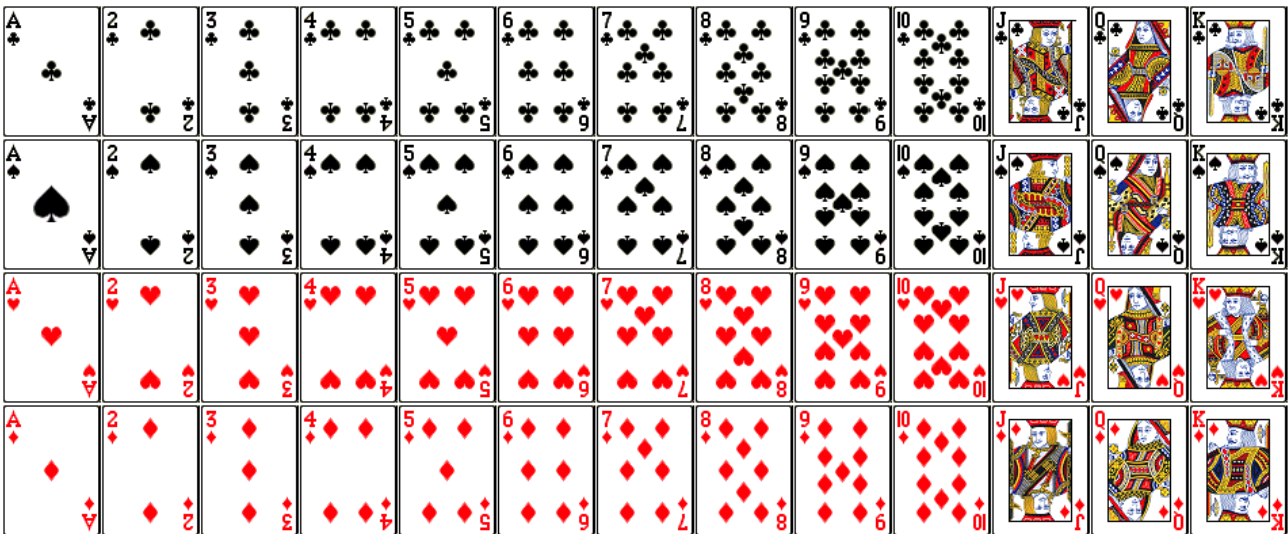
특정 객체의 메소드를 호출하는 방법은 객체이름, 점, 메소드 이름 순으로 `card.flip()`과 같이 나열하면 된다. 다음과 같이 객체를 만들어 이 메소드를 호출하면 `face_up` 속성변수의 값이 바뀔을 확인할 수 있다.

```
>>> from playingcard import *
>>> card = Card("Spade", "7")
>>> card.face_up
True
>>> card.flip()
>>> card.face_up
```

```
False
>>> card.flip()
>>> card.face_up
True
```

#### 4. 클래스 속성

각 객체는 다른 객체와 구별이 되는 고유의 속성이 있다. 위에서 만든 card1과 card2 객체를 보면 종류와 계급이 서로 다르다. 그런데 모든 객체가 공유해야 하는 속성도 있을 수 있다. 놀이카드에서 사용하는 종류와 계급은 정해져 있으며 이들의 종류와 계급은 모든 객체가 공유해야 할 것이다. 놀이카드는 종류가 4 가지, 계급이 13 가지 있다. 카드 52장은 이 종류와 계급의 조합으로 만들어진다.



따라서 종류의 집합과 계급의 집합은 모든 객체가 공유하는 속성으로 분류할 수 있는데, 이와 같이 모든 객체가 공유하는 속성을 클래스 속성(class attribute)이라고 한다. Card 클래스의 속성으로 종류의 집합 suits와 계급의 집합 ranks를 튜플로 정의해보자. Card 객체를 만들 때 참고할 수 있을 것이다. 이 클래스 속성변수 suits와 ranks는 다음의 줄 2~3과 같이 Card 클래스 내부에서 정의할 수 있다. 클래스 속성과 같이 공유하는 데이터는 튜플 같은 수정불가능 데이터를 사용하는게 안전하다.

```
1 class Card:
2     suits = ("Diamond", "Heart", "Spade", "Clover")
3     ranks = ("A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K")
4
5     def __init__(self, suit, rank, face_up=True):
6         self.suit = suit
7         self.rank = rank
8         self.face_up = face_up
9
10    def flip(self):
11        self.face_up = not self.face_up
```

그러면 클래스 속성변수는 어디서든 클래스의 이름으로 다음과 같은 형식으로 가져다 쓸 수 있다.

```
>>> from playingcard import *
>>> Card.suits
('Diamond', 'Heart', 'Spade', 'Clover')
```

객체의 이름으로 볼 수도 있다.

```
>>> card = Card("Spade", "7")
>>> card.suits
('Diamond', 'Heart', 'Spade', 'Clover')
```

더구나 어디서든 클래스 이름으로 다음과 같이 값을 바꿀 수도 있다.

```
>>> Card.suits = ('D', 'H', 'S', 'C')
>>> Card.suits
('D', 'H', 'S', 'C')
>>> card.suits
('D', 'H', 'S', 'C')
```

그러나 객체의 이름으로 바꾸지는 못한다.

```
>>> card.suits = ('Diamond', 'Heart', 'Spade', 'Clover')
>>> Card.suits
('D', 'H', 'S', 'C')
>>> card.suits
('Diamond', 'Heart', 'Spade', 'Clover')
```

## 5. 객체 캡슐화

함수를 공부하면서 캡슐화에 대해서 설명하였다. 함수의 정의도 일종의 캡슐화이다. 함수가 내부적으로 어떻게 작동하는지 외부에서는 알 필요 없도록 감출 수 있기 때문이다. 완벽하게 캡슐화를 하려면 함수를 호출하는 측에서 봤을 때 인수를 제공하고 결과를 받을 뿐 다른 부작용(부수효과)은 없도록 작성해야 한다.

객체도 마찬가지다. 앞에서 봤듯이 객체의 속성변수를 외부에서 직접 접근하여 임의로 수정하는 건 보안상 좋지 않다. 실물 놀이카드의 모양이나 계급을 누구나 쉽게 고칠 수 있다고 상상해보자. 아무도 그런 놀이카드는 쓰지 않을 것이다. 따라서 우리가 만드는 놀이카드 객체도 모양이나 그림을 외부에서 볼 수는 있되 수정은 하지 못하도록 해야 한다. 이를 **데이터 캡슐화** data encapsulation 라고 한다. Python 3에서 제공하는 방식으로 Card 클래스의 데이터 캡슐화를 어떻게 하는지 공부해보자.

### 비공개 속성

객체의 속성변수와 메소드는 그냥 두면 누구나 접근이 가능하도록 공개 public 된다. 외부로부터 속성변수에 접근을 차단하고 내부에서만 쓰고 싶으면, 속성변수 이름에 앞에 다음과 같은 형식으로 \_\_(밑줄 두 개를 나란히 나열)를 붙이면 된다.

```
self.__suit = suit
```

그러면 이 속성변수는 비공개 private 속성변수가 된다. 비공개 속성변수는 클래스 내부에서는 어디에서든 접근 가능하지만, 클래스 외부에서는 직접 접근이 불가능하다. 다음은 Card 클래스의 속성변수를 모두 비공개로 만들면 다음과 같다.



```

1 class Card:
2     __suits = ("Diamond", "Heart", "Spade", "Clover")
3     __ranks = ("A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K")
4
5     def __init__(self, suit, rank, face_up=True):
6         self.__suit = suit
7         self.__rank = rank
8         self.__face_up = face_up
9
10    def flip(self):
11        self.__face_up = not self.__face_up

```

실제로 속성변수의 비공개가 실현되었는지 직접 실행하여 확인해보자.

```

>>> from playingcard import *
>>> Card.__suits
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Card' has no attribute '__suits'
>>> Card.__ranks
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Card' has no attribute '__ranks'
>>> card = Card("Spade", "7")
>>> card.__suit
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Card' object has no attribute '__suit'
>>> card.__rank
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Card' object has no attribute '__rank'
>>> card.__face_up
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Card' object has no attribute '__face_up'

```

속성변수가 모두 외부로부터 완전히 차단되었음이 확인되었다. 이 중에서 객체 속성인 `self.__suit`, `self.__rank`, `self.__face_up` 값을 볼 수는 있게 하려면 공개용 메소드로 다음과 같이 작성하여 추가하면 된다.

```

1 class Card:
...     ...
13    def suit(self):
14        return self.__suit
15
16    def rank(self):
17        return self.__rank
18
19    def face_up(self):
20        return self.__face_up

```

이렇게 하면 비공개 속성변수를 직접 읽거나 쓸 수 없지만, 공개 메소드를 호출하여 읽을 수는 있게 되었다. 다음과 같이 메소드를 실행하여 이를 확인해보자.

```
>>> from playingcard import *
>>> card = Card("Spade", "7")
>>> card.suit()
'Spade'
>>> card.rank()
'7'
>>> card.face_up()
True
>>> card.flip()
>>> card.face_up()
False
>>> card.flip()
>>> card.face_up()
True
```

일반적으로 메소드는 객체의 기능이나 행위를 정의한다. 그런데 이와 같이 객체의 특성이나 상태를 나타내는 속성변수의 값까지 메소드 호출로 알아내는 것은 표현상 좀 어색하다. 즉, 코드에서 속성변수 참조와 메소드 호출은 명확히 구별되는게 좋다. Python 3에서는 비공개 속성변수 참조를 메소드 호출과 구별할 수 있게 해주는 기능을 제공한다. 비공개 속성변수를 참조하는 메소드 정의에 프로퍼티 장식자 `@property` 를 달면, ()를 뒤에 붙이지 않고 속성변수 참조가 가능하다. 예를 들어, `card.suit()` 라고 하는 대신, `card.suit` 로 기술할 수 있게 해주는 것이다. `Card` 클래스에서 객체 속성변수를 모두 프로퍼티로 바꾸려면 장식자를 다음과 같이 달면 된다.

```
1 class Card:
...     ...
12     @property
13     def suit(self):
14         return self.__suit
15
16     @property
17     def rank(self):
18         return self.__rank
19
20     @property
21     def face_up(self):
22         return self.__face_up
```

이제 속성변수 참조는 다음과 같이 할 수 있다.

```
>>> from playingcard import *
>>> card = Card("Spade", "7")
>>> card.suit
'Spade'
>>> card.rank
'7'
>>> card.face_up
```

```

True
>>> card.flip()
>>> card.face_up
False
>>> card.flip()
>>> card.face_up
True

```

이제 메소드 호출 식의 참조는 더 이상 허용하지 않는다.

```

>>> card.suit()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
>>> card.rank()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
>>> card.face_up()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not callable

```

객체지향 프로그래밍에서 캡슐화를 잘 하자는 원칙이 있다. 클래스를 만들 때 외부에 공개하기로 정한 메소드만 공개하고 나머지는 모두 비공개로 설정하는 것이다. 이렇게 함으로써 외부 사용자는 공개된 메소드 호출을 통해서만 객체를 사용할 수 있다. 즉, 객체들끼리 대화와 소통은 메소드를 주고 받으면서 이루어지도록 함으로써, 객체 내부의 사유 데이터는 보호하면서 객체를 사용하게 하자는 것이다.

## 잠깐!

지금까지 속성변수와 메소드 이름을 어떻게 지었는지 둘러보자. 속성변수는 주로 데이터를 나타내므로 명사로 이름지었고, 메소드는 행위를 나타내므로 동사로 이름지었다. 이 원칙을 잘 따르면 프로그램의 가독성을 높일 수 있어서 좋다.

## 6. 클래스 소속 메소드

### 정적 메소드

Card 클래스는 Card 객체를 만드는 역할을 주로 한다. 카드놀이를 하기 위해서는 잘 섞인 카드가 별 단위로 필요한데 카드 한벌은 누가 만들게 해야하나? 카드에 대해서 가장 잘 알고 있는 클래스가 바로 Card 클래스이므로 Card 클래스의 고유 기능으로 갖추면 가장 좋겠다. 이와 같은 클래스 소속 메소드를 정적 메소드 static method라고 하고, 메소드 정의 앞에 다음과 같은 장식을 붙여서 구별한다.

```
@staticmethod
```

정적메소드는 객체에 속해있지 않으므로 self 파라미터가 필요없다. 나머지는 일반 메소드를 정의하는 방식과 같다. Card 객체 한 벌을 잘 섞어서 리스트로 만들어 내주는 클래스 소속 정적메소드는 다음과 같이 정의할 수 있다. 카드는 모두 보이지 않게 덮어서 리스트로 만들어 모으며, random 모듈의 shuffle 메소드를 써서 무작위로 섞어서 내준다.

```

1 class Card:
...     ...
56     @staticmethod
57     def fresh_deck():
58         cards = []
59         for s in Card.__suits:
60             for r in Card.__ranks:
61                 cards.append(Card(s,r,False))
62         random.shuffle(cards)
63         return cards

```

정적메소드 호출은 일반 메소드를 호출하는 방식과 같고, 객체가 전혀 없어도 클래스 이름으로 호출할 수 있다는 점이 객체 소속 메소드와 다르다.

## Docstring

코드의 가독성을 높이기 위해서 코드를 설명하는 문서를 코드 내부에 삽입하는게 좋다. 주석comment으로 코드의 일부분을 설명하는 문서를 달 수 있다. 주석은 코드를 실행할 때는 무시된다. 현대 프로그래밍 언어는 일반적으로 코드의 문서를 체계화할 수 있는 부가 기능을 제공한다. Python의 Docstring(문서 문자열)은 모듈, 함수, 클래스, 메소드 정의의 맨 앞에 기술하여 해당 정의를 설명하는 문자열로서, 다음 코드와 같이 `"""`로 둘러싸아 표현한다. 한 줄이나 여러 줄에 걸쳐서 표현할 수 있다.

Docstring은 따로 언급하지 않아도 해당 정의의 속성으로 귀속되어 `__doc__`이라는 이름으로 지정된다. 실행기로 직접 다음과 같이 확인할 수 있다.

```

>>> from playingcard import *
>>> card = Card("Spade", "7")
>>> print(card.__doc__)
defines Card class
>>> print(card.__init__.__doc__)
initializes a playing card object
arguments:
suit -- must be in suits
rank -- must be in ranks
face_up -- True or False (default True)

>>> print(card.__str__.__doc__)
returns its string representation
>>> print(card.flip.__doc__)
flips itself

```

코드는 개발자들끼리 서로 리뷰 해주면서 발전한다. 따라서 다른 개발자가 읽기 좋게 적절한 주석과 Docstring을 달아주어 깔끔하게 작성해야 한다. 특히 Docstring은 클래스의 속성으로 저절로 저장되므로 잘 작성해두면 양질의 사용매뉴얼을 자동으로 생성할 수도 있다. 다음은 Card 클래스에 Docstring을 추가한 사례이다.

```
1 class Card:
2     """defines Card class"""
3     __suits = ("Diamond", "Heart", "Spade", "Clover")
4     __ranks = ("A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K")
5
6     def __init__(self, suit, rank, face_up=True):
7         """initializes a playing card object
8         arguments:
9         suit -- must be in suits
10        rank -- must be in ranks
11        face_up -- True or False (default True)
12        """
13        self.__suit = suit
14        self.__rank = rank
15        self.__face_up = face_up
16
17    def __str__(self):
18        """returns its string representation"""
19        if self.__face_up:
20            return self.__suit + "." + self.__rank
21        else:
22            return "xxxxx" + "." + "xx"
23
24    def flip(self):
25        """flips itself"""
26        self.__face_up = not self.__face_up
27
28    @property
29    def suit(self):
30        """returns its suit value"""
31        return self.__suit
32
33    @property
34    def rank(self):
35        """returns its rank value"""
36        return self.__rank
37
38    @property
39    def face_up(self):
40        """returns its face_up value"""
41        return self.__face_up
```

## 실습. BankAccount 클래스 만들기

은행계좌 객체를 만드는 BankAccount 클래스를 작성해보자.

### 1단계 - 빈 클래스 만들고 객체 생성 확인하기

bank.py 파일을 만들어 다음 코드를 저장한 뒤,

```
1 class BankAccount:
2     pass
```

실행기에서 다음과 같이 객체를 생성하고 확인해보자.

```
>>> from bank import *
>>> acct = BankAccount()
>>> acct
<bank.BankAccount object at 0x10195b9b0>
>>> print(acct)
<bank.BankAccount object at 0x10195b9b0>
```

### 2단계 - 생성 메소드 만들기

은행계좌 객체는 소유자이름과 예금잔고 정보를 갖고 있어야 한다.

속성변수	내용
name	소유자 이름 (문자열)
balance	은행 잔고, 단위: 원 (정수), 반드시 0 이상이어야 함

은행계좌 객체를 생성하면서 이름과 초기 예금금액을 인수로 받아 설정하도록 \_\_init\_\_ 메소드를 추가하자. 그런데 balance 인수는 생략할 수 있고, 생략한 경우에 0으로 설정한다. balance 인수는 절대 음수가 되지 않아야 한다. 음수 인수가 들어오는 경우 모두 0으로 설정하도록 해야 한다. 실행기에서 다음과 같이 작동해야 한다.

```
>>> from bank import *
>>> acct1 = BankAccount("Yebbuni", 100000)
A bank account for Yebbuni is open.
Your current balance is 100000 won.
>>> print(acct1)
<bank.BankAccount object at 0x10115b9e8>
>>> acct1.name
'Yebbuni'
>>> acct1.balance
100000
>>> acct1.balance += 100000000
>>> acct1.balance
100100000
>>> acct1.name = "Doseonsaeng"
>>> acct1.name
'Doseonsaeng'
>>> acct2 = BankAccount("Gomdori", 50000)
A bank account for Gomdori is open.
Your current balance is 50000 won.
```

```
>>> print(acct2)
<bank.BankAccount object at 0x101213860>
>>> acct3 = BankAccount("Monnani")
A bank account for Monnani is open.
Your current balance is 0 won.
>>> acct4 = BankAccount("Bbanjiri", -1000000)
A bank account for Bbanjiri is open.
Your current balance is 0 won.
```

확인이 끝나면 Docstring을 적절하게 작성하자. 앞으로 만드는 메소드는 모두 특별한 언급이 없어도 Docstring을 작성하도록 한다.

### 3단계 - 간판 문자열 메소드 만들기

`__str__` 메소드를 추가하여 다음과 같이 작동하도록 하자.

```
>>> from bank import *
>>> acct1 = BankAccount("Yebbuni", 100000)
A bank account for Yebbuni is open.
Your current balance is 100000 won.
>>> print(acct1)
Yebbuni's BankAccount object
>>> acct3 = BankAccount("Monnani")
A bank account for Monnani is open.
Your current balance is 0 won.
>>> print(acct3)
Monnani's BankAccount object
```

### 4단계 - 메소드 만들기

BankAccount 클래스에 입출금 기능을 하는 메소드를 다음의 요구사항에 맞추어 작성하여 추가하자.

메소드	의미
<code>show_balance()</code>	자신의 balance를 실행창에 프린트한다.
<code>deposit(amount)</code>	amount가 0 이상의 정수이면, 자신의 balance를 amount 만큼 증가시키고 실행창에 프린트한다. amount가 음수이면, 입금이 불가하다는 메시지를 실행창에 프린트한다.
<code>withdraw(amount)</code>	amount가 음수가 아니고 balance를 초과하지 않으면, 자신의 balance를 amount 만큼 감소시키고 실행창에 프린트한다. 그렇지 않으면, 입금이 불가하다는 메시지를 실행창에 프린트한다.

출력의 형식은 아래의 실행사례와 같아야 한다.

```
>>> from bank import *
>>> acct1 = BankAccount("Yebbuni", 100000)
A bank account for Yebbuni is open.
Your current balance is 100000 won.
>>> acct1.deposit(50000)
50000 won has been successfully deposited.
Yebbuni's balance is 150000 won.
>>> acct1.deposit(-30000)
Deposit failed.
Yebbuni's balance is 150000 won.
>>> acct1.withdraw(80000)
```

```

80000 won has been successfully withdrawn.
Yebbuni's balance is 70000 won.
>>> acct1.withdraw(-30000)
Withdraw failed.
Yebbuni's balance is 70000 won.
>>> acct1.withdraw(100000)
Withdraw failed.
Yebbuni's balance is 70000 won.
>>> acct1.show_balance()
Yebbuni's balance is 70000 won.
>>> acct1.balance
70000

```

## 5단계 - 객체 캡슐화 : 비공개 속성변수

BankAccount의로 만든 객체의 속성변수는 외부에서 다음과 같이 임의로 수정할 수 있다.

```

>>> from bank import *
>>> acct2 = BankAccount("Gomdori", 50000)
A bank account for Gomdori is open.
Your current balance is 50000 won.
>>> acct2.name = "Pooh"
>>> acct2.name
'Pooh'
>>> acct2.balance = 100000000
>>> acct2.balance
100000000

```

BankAccount의 보안을 확보하려면 self.name과 self.balance 속성변수를 외부에서 값을 고칠 수 없도록 해야한다. 이 두 속성변수를 비공개 속성변수로 고쳐서 외부에서 수정불가능하게 만들자. 그러면 객체를 생성하면서 만들어진 self.name 속성변수는 영원히 수정불가능해진다. 그리고 self.balance 속성변수 값을 deposit과 withdraw 메소드 호출을 통해서만 수정이 가능하다. 다른 방법은 없다. 이 두 속성변수를 비공개로 만들어 코드를 수정한 후, 다음의 실행사례와 동일한 결과가 나오는지 확인하자.

```

>>> from bank import *
>>> acct2 = BankAccount("Gomdori", 50000)
A bank account for Gomdori is open.
Your current balance is 50000 won.
>>> acct2.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'BankAccount' object has no attribute 'name'
>>> acct2.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'BankAccount' object has no attribute '__name'
>>> acct2.__name = 'Pooh'
>>> acct2.show_balance()
Gomdori's balance is 50000 won.
>>> acct2.__balance = 100000000
>>> acct2.show_balance()
Gomdori's balance is 50000 won.
>>> acct2.balance = 100000000
>>> acct2.show_balance()
Gomdori's balance is 50000 won.

```



```
>>> acct2.deposit(20000)
20000 won has been successfully deposited.
Gomdori's balance is 70000 won.
>>> acct2.withdraw(5000)
5000 won has been successfully withdrawn.
Gomdori's balance is 65000 won.
```

## 6단계 - 클래스 속성변수

개설한 계좌가 총 몇개인지 기억하려면 어떻게 할까? 클래스 속성변수를 사용하여 개설한 계좌의 수를 기억하도록 BankAccount 클래스 코드를 수정하자. no\_of\_accounts 라는 클래스 변수를 정의하고, 초기 값을 0으로 설정한다. 객체가 하나 생성될 때마다 이 클래스 변수 값이 1씩 증가되도록 하면 된다. 수정을 완료한 다음, 다음과 같이 실행되는지 확인하자.

```
>>> from bank import *
>>> BankAccount.no_of_accounts
0
>>> acct1 = BankAccount("Yebbuni",100000)
A bank account for Yebbuni is open.
Your current balance is 100000 won.
>>> BankAccount.no_of_accounts
1
>>> acct2 = BankAccount("Gomdori",50000)
A bank account for Gomdori is open.
Your current balance is 50000 won.
>>> BankAccount.no_of_accounts
2
>>> acct3 = BankAccount("Monnani")
A bank account for Monnani is open.
Your current balance is 0 won.
>>> BankAccount.no_of_accounts
3
>>> acct4 = BankAccount("Bbanjiri",-1000000)
A bank account for Bbanjiri is open.
Your current balance is 0 won.
>>> BankAccount.no_of_accounts
4
```

## 7단계 - 정적 메소드 [자습]

클래스 속성변수도 외부에서 얼마든지 수정가능하다. 누가 다음과 같이 개설 계좌의 수를 임의로 수정해 버리면 곤란하다.

```
>>> BankAccount.no_of_accounts
4
>>> BankAccount.no_of_accounts = 10000
>>> acct4.no_of_accounts
10000
```

객체의 비공개 속성변수와 마찬가지로 클래스 속성변수 앞에 다음과 같이 \_\_를 붙이면 외부에서 참고가 불가능해진다.

```
__no_of_accounts = 0
```

직접 고쳐서 다음과 같이 더 이상 접근 및 수정이 불가능해짐을 확인해보자.

```
>>> from bank import *
>>> BankAccount.no_of_accounts
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'BankAccount' has no attribute 'no_of_accounts'
>>> BankAccount.__no_of_accounts
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'BankAccount' has no attribute '__no_of_accounts'
```

이제 이 클래스 속성변수의 값은 객체가 생성될 때마다 1씩 증가될 뿐, 임의로 수정할 수는 없게 되었다. 하지만 이 값이 궁금한 경우 알려주는 기능은 있어야 한다. 이 속성변수는 (객체에 속하지 않고) 클래스에 속해 있으므로, 이 값을 읽어주는 메소드도 클래스 소속이어야 한다. 다음의 정적메소드 정의를 BankAccount 클래스에 추가하자.

```
@staticmethod
def count_accounts():
    return BankAccount.__no_of_accounts
```

그리고 아래 실행 사례 대로 실행하면서 실행 결과를 확인하면서 정적메소드를 이해하자.

```
>>> from bank import *
>>> BankAccount.count_accounts()
0
>>> acct4 = BankAccount("Bbanjiri",-1000000)
A bank account for Bbanjiri is open.
Your current balance is 0 won.
>>> BankAccount.count_accounts()
1
>>> acct4.count_accounts()
1
>>> BankAccount.__no_of_accounts
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'BankAccount' has no attribute '__no_of_accounts'
```