

17. 클래스와 객체 지향 개념

1. 파이썬의 데이터 객체
2. 클래스 - 객체를 만드는 도구
3. int 클래스 이해하기
4. 컨테이너 자료형 클래스 이해하기 - str, list, tuple, set, dict
5. `__str__`, `__repr__` 메소드 이해하기
6. 정리

1. 파이썬의 데이터 객체

◆ 클래스는 객체를 만드는 도구임

- 파이썬에서는 데이터는 '객체'라고 함.
- 객체는 클래스를 통해서 만들어짐. 정수 객체는 정수 클래스를 통해서 만들고, 실수 객체는 실수 클래스를 이용해서 만듦.
- 파이썬에 없는 객체를 만들고 싶다면, 그 객체를 생성할 수 있는 클래스를 직접 만들어야 함.

```
>>> a = 10  
>>> type(a)  
<class 'int'>
```

클래스는 객체를 만드는 도구예요
그리고 객체의 자료형입니다.

1. 파이썬의 데이터 객체

◆ 9가지 클래스와 객체

<pre>>>> a = 10 >>> type(a) <class 'int'></pre>	<pre>>>> d = True >>> type(d) <class 'bool'></pre>	<pre>>>> g = (1,3,5) >>> type(g) <class 'tuple'></pre>
<pre>>>> b = 3.5 >>> type(b) <class 'float'></pre>	<pre>>>> e = 'hello' >>> type(e) <class 'str'></pre>	<pre>>>> h = {2,4,6,8} >>> type(h) <class 'set'></pre>
<pre>>>> c = 3+7j >>> type(c) <class 'complex'></pre>	<pre>>>> f = [1,2,3,4] >>> type(f) <class 'list'></pre>	<pre>>>> i = {'baseball':9, 'soccer':11} >>> type(i) <class 'dict'></pre>

1. 파이썬의 데이터 객체

◆ 리스트 객체에 적용할 수 있는 속성 목록

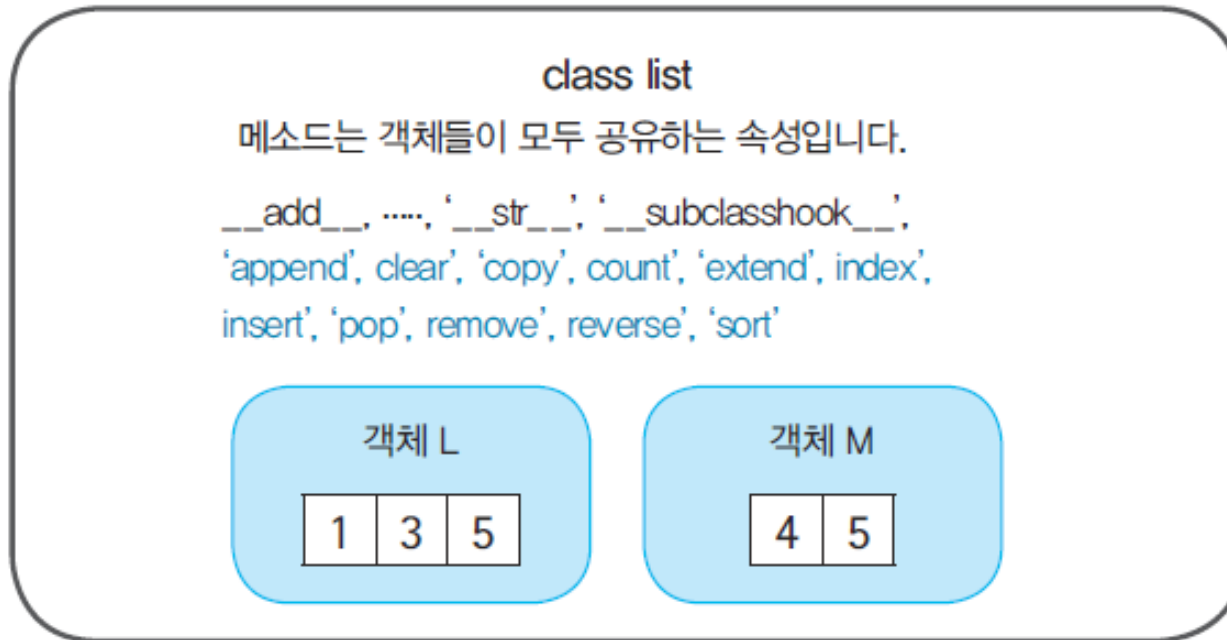
```
>>> dir(list)
['__add__', '.....', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

▪ 리스트 객체에 속성 사용하기

```
>>> L = [1, 3, 5]; M = [4, 5]    # 두 개의 리스트 객체 L과 M을 만듭니다.
>>> L.append(10)
>>> M.append(20)
>>> L, M                        # L과 M을 출력합니다.
([1, 3, 5, 10], [4, 5, 20])
```

1. 파이썬의 데이터 객체

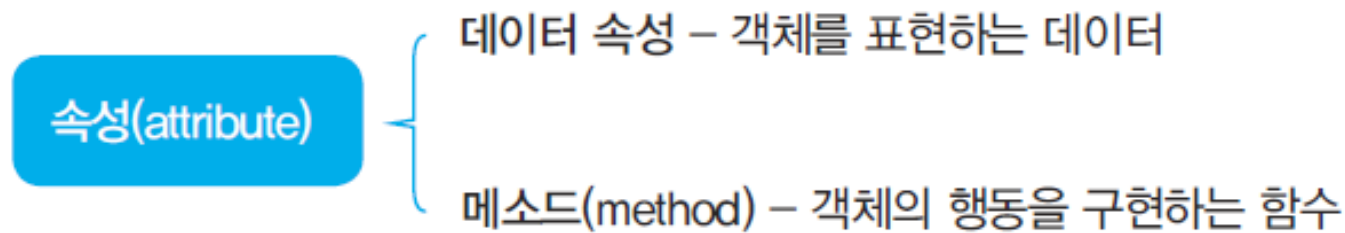
◆ 리스트 객체에 적용할 수 있는 속성 목록



어떤 클래스든지 객체는 얼마든지 만들 수 있고, 모든 객체들이 메소드를 공유합니다. 따라서 메소드를 사용하고자 할 때 '객체.메소드()'라고 호출하여 어떤 객체에 적용하는 메소드임을 분명히 명시해야 합니다.

1. 파이썬의 데이터 객체

◆ 속성 (attribute)



- 데이터 속성은 객체가 각각 갖고 있는 데이터임.
- 메소드는 객체들이 공유하여 사용하는 속성임.
- int 속성 확인하기

```
>>> dir(int)
['__abs__', '...', '__xor__', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

- numerator/denominator는 데이터 속성임. 정수를 분수로 표현했을 때 numerator는 분자, denominator는 분모의 값을 갖는다.

1. 파이썬의 데이터 객체

◆ 속성 (attribute)

- int 클래스의 bit_length() 메소드, numerator/denominator 데이터 속성 예제

```
>>> x = 10
>>> x.bit_length()    # 10=1010(2)
4
>>> bin(x)    # 0b는 이진수라는 뜻임.
'0b1010'
>>> x.numerator
10
>>> x.denominator
1
```

```
>>> y = 55
>>> y.bit_length()    # 55=110111(2)
6
>>> bin(y)
'0b110111'
>>> y.numerator    # 데이터 속성에는 괄호없음.
55
>>> y.denominator
1
```

2. 클래스 - 객체를 만드는 도구

◆ 강아지 객체 만들기

강아지 클래스 속성

데이터 속성 : 객체를 표현하는 데이터(예 : 이름, 나이, 색,)

메소드(Method) : 객체의 행동을 구현하는 함수(예: 꼬리흔들기, 짖기, 달리기, 구르기)



2. 클래스 - 객체를 만드는 도구

◆ 메소드만을 갖는 간단한 클래스 구조

```
class 클래스명:
    """ docstring """

    def 메소드(self):
        """ docstring """
```

콜론을 넣습니다.

클래스명은 대문자로 시작하는 것이 좋습니다.
클래스에 대한 설명을 docstring 안에 적어 줍니다.

메소드는 객체의 행동을 구현하는 데 이용합니다.
메소드가 하는 일을 docstring 안에 적어 줍니다.

모든 메소드의 첫 번째 인수는 self여야 합니다.

강아지 클래스 예제

```
class Dog:
    """ 클래스 Dog은 강아지 객체를 표현하는 클래스입니다. """

    def wag_tail(self):
        """ 강아지가 꼬리 흔드는 행동을 나타내는 메소드입니다. """
        print('The dog is wagging its tail.')
```

2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog 선언 및 객체 생성하기

```
class Dog:
    """ 클래스 Dog은 강아지 객체를 표현하는 클래스입니다. """

    def wag_tail(self):
        """ 강아지가 꼬리 흔드는 행동을 나타내는 메소드입니다. """
        print('The dog is wagging its tail.')
```

위의 강아지 클래스 객체 생성하기

```
>>> happy = Dog()      # '객체명 = 클래스명()'으로 happy 객체를 만듭니다.
>>> happy.wag_tail()    # 객체 happy가 wag_tail() 메소드를 호출했어요.
The dog is wagging its tail.
>>> hope = Dog()       # '객체명 = 클래스명()'으로 hope 객체를 만듭니다.
>>> hope.wag_tail()     # 이번에는 객체 hope가 wag_tail() 메소드를 호출했어요.
The dog is wagging its tail.
```

2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog 선언 및 객체 생성하기

id와 type() 함수로 객체 확인하기

```
>>> id(happy)          # 객체 happy의 id값을 출력해 봅니다.  
53907376  
>>> type(happy)        # 객체 happy의 데이터 타입을 알아봅니다.  
<class '__main__.Dog'>  
>>> id(hope)           # 객체 hope의 id값을 출력해 봅니다.  
49366704  
>>> type(hope)         # 객체 hope의 데이터 타입을 알아봅니다.  
<class '__main__.Dog'>
```

2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog 에 메소드 추가하기

```
class Dog:
    """ 클래스 Dog은 강아지 객체를 표현하는 클래스입니다. """

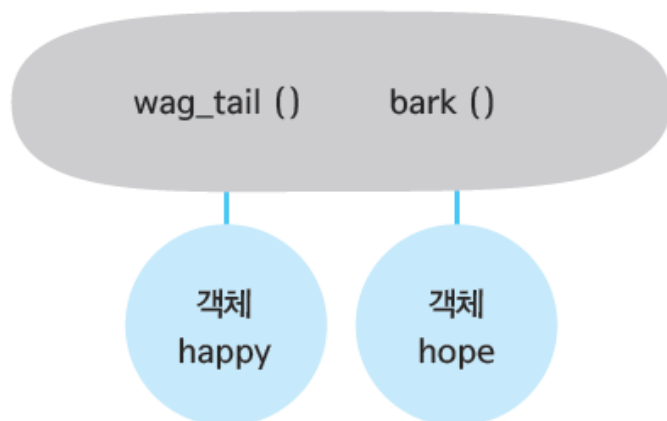
    def wag_tail(self):
        """ 강아지가 꼬리 흔드는 행동을 나타내는 메소드입니다. """
        print('The dog is wagging its tail.')

    def bark(self):
        """ 강아지가 짖는 행동을 나타내는 메소드입니다. """
        print('The dog is barking. woof ~')
```

```
>>> happy = Dog()
>>> hope = Dog()
>>> happy.bark() # 객체 happy가 bark() 메소드를 호출합니다.
The dog is barking. woof ~
>>> hope.bark() # 객체 hope가 bark() 메소드를 호출합니다.
The dog is barking. woof ~
```

2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog 에 메소드 추가하기



객체 happy와 hope는 메소드 `wag_tail()`, `bark()`를 공유합니다.
`happy.bark()`는 객체 happy가 `bark()` 메소드를 호출합니다. 마찬가지로 `hope.wag_tail()`은 `wag_tail()` 메소드를 객체 hope가 호출합니다.



2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog 에 메소드 추가하기

- dir(클래스명)이라고 하면, 해당 클래스의 속성 목록을 볼 수 있음.

```
>>> dir(Dog)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', 'bark', 'wag_tail']
```

2. 클래스 - 객체를 만드는 도구

◆ 데이터 속성 추가하기

- 앞의 강아지 클래스에서 bark() 메소드 결과를 보면 두 객체 happy와 hope 중에서 어느 객체가 bark() 메소드를 호출했는지 알 수 없음.
- 데이터 속성을 추가하여 어느 객체가 호출했는지 정보를 넣을 수 있음.

bark() 메소드 호출

The dog is barking. woof ~

이름이 추가된 bark() 메소드 호출

Happy is barking. woof ~

Hope is barking. woof ~

- 특별한 메소드인 '생성자'를 이용하여 위의 작업을 할 수 있음.

2. 클래스 - 객체를 만드는 도구

◆ 데이터 속성 추가하기

- 강아지 클래스에 '이름', '나이' 두 개의 데이터 속성을 추가함 (생성자 이용함)

생성자 `__init__(self)`

```
def __init__(self, name, age):  # __init__()은 객체를 만들 때 자동 호출됩니다.  
    self.name = name           # 만들어진 객체의 name 공간에 name값으로 초기화합니다.  
    self.age = age              # 만들어진 객체의 age 공간에 age값으로 초기화합니다.
```


2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog - 생성자 추가한 코드

```
class Dog:
    """ 클래스 Dog은 강아지 객체를 표현하는 클래스입니다. """
    def __init__(self, name, age): # __init__()은 객체를 만들 때 자동 호출됩니다.
        self.name = name
        self.age = age
    def introduce(self):
        """ 나를 소개하는 메소드입니다. """
        print('My name is {} and {} years old.'.format(self.name, self.age))
    def wag_tail(self):
        """ 강아지가 꼬리 흔드는 행동을 나타내는 메소드입니다. """
        print('{} is wagging its tail.'.format(self.name))
    def bark(self):
        """ 강아지가 짖는 행동을 나타내는 메소드입니다. """
        print('{} is barking. woof ~'.format(self.name))
```

2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog 생성자 통하여 객체 생성하는 코드

```
x = Dog('Happy', 2)
y = Dog('Hope', 3)
x.introduce()    # self는 x입니다.
y.introduce()    # self는 y입니다.
x.bark()          # self는 x입니다.
y.bark()          # self는 y입니다.
x.wag_tail()     # self는 x입니다.
y.wag_tail()     # self는 y입니다.
```

[결과]

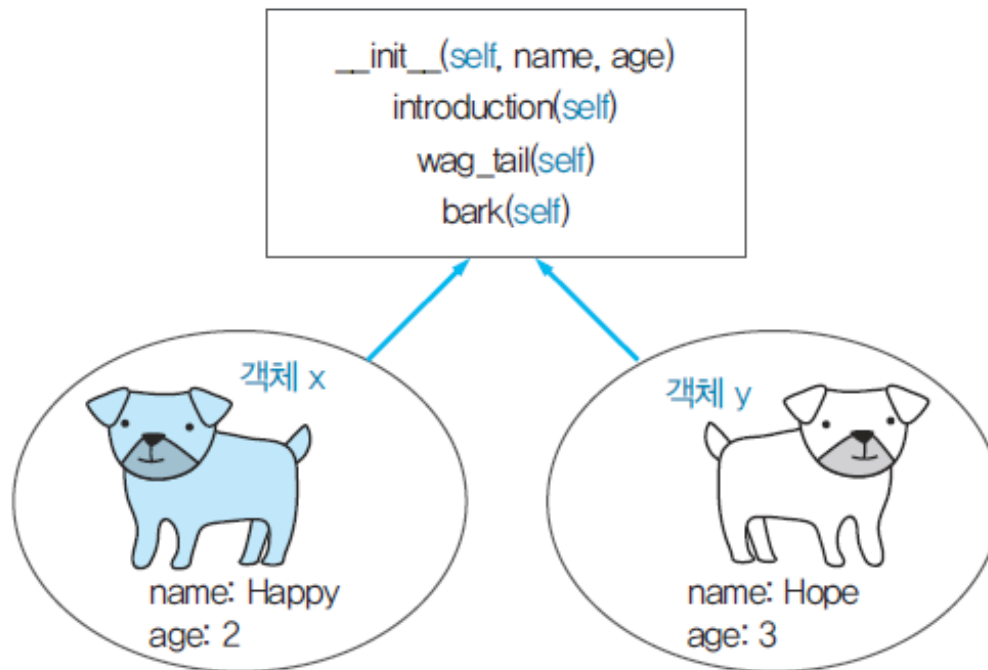
My name is happy and 2 years old.
 My name is hope and 3 years old.
 Happy is barking. woof ~
 Hope is barking. woof ~
 Happy is wagging its tail.
 Hope is wagging its tail.



메소드를 호출할 때는 self 자리에는 인수를 넘기지 않습니다.

2. 클래스 - 객체를 만드는 도구

◆ 클래스 Dog 생성자 통하여 객체 생성하는 코드



메소드를 부르는 객체가 `self`가 됩니다.

NOTE `__init__()` 메소드를 포함해서 클래스에 정의되는 모든 메소드는 첫 매개 변수 자리에 `self`를 넣어야 합니다. 하지만 객체를 만들어서 메소드를 호출할 때는 일반적으로 `self` 자리에는 인수를 넘기지 않습니다.

2. 클래스 - 객체를 만드는 도구

CODE 91 학생들의 성적 관리를 위해서 학생 클래스를 만들어 볼게요. 데이터 속성으로는 학번, 이름, 성적을 넣을 거예요. 그리고 세 명의 학생 객체를 만들어서 가장 좋은 성적을 출력하는 코드를 작성해 볼게요.

```
class Student:
    """ 학생들의 성적을 관리하기 위한 Student 클래스입니다.
    학생들의 학번, 이름, 성적을 데이터로 저장합니다. """
    def __init__(self, no, name, score):
        self.no = no
        self.name = name
        self.score = score
    def info(self):
        print('학번 - {}, 이름 - {}, 성적 - {}'.format(self.no, self.name, self.score))
    def getScore(self):
        return self.score

# main
s1 = Student(201812345, '홍길동', 90)
s2 = Student(201811111, '이철수', 85)
s3 = Student(201800100, '김영희', 93)
s1.info()
s2.info()
s3.info()
print('가장 좋은 성적 : {}'.format(max(s1.getScore(), s2.getScore(), s3.getScore())))
```

[결과]

```
학번 - 201812345, 이름 - 홍길동, 성적 - 90
학번 - 201811111, 이름 - 이철수, 성적 - 85
학번 - 201800100, 이름 - 김영희, 성적 - 93
가장 좋은 성적 : 93
```

2. 클래스 - 객체를 만드는 도구

CODE 92 동전 지갑 클래스를 만들어 볼게요. 동전 지갑에는 500원짜리, 100원짜리, 50원짜리, 10원짜리 동전이 들어 있어요. 동전 지갑 클래스의 각 동전의 개수를 데이터 속성으로 갖습니다. 지갑에 들어 있는 동전이 얼마인지 출력하는 클래스를 작성합니다

```
class Purse:
    """ 현재 지갑에 돈이 얼마 있는지를 저장하는 클래스입니다.
        데이터 속성은 다음과 같이 4개입니다.
        c500 - 500원짜리 동전 개수
        c100 - 100원짜리 동전 개수
        c50 - 50원짜리 동전 개수
        c10 - 10원짜리 동전 개수 """
    def __init__(self, c500, c100, c50, c10):
        self.c500 = c500
        self.c100 = c100
        self.c50 = c50
        self.c10 = c10
    def total(self):      # 지갑에 있는 동전의 총 액수를 반환합니다.
        return self.c500 * 500 + self.c100 * 100 + self.c50 * 50 + self.c10 * 10

# main
p1 = Purse(3, 2, 5, 1)      # __init__() 메소드 자동 호출
p2 = Purse(4, 1, 1, 2)      # __init__() 메소드 자동 호출
print('지갑 p1에는 {}원 있습니다.'.format(p1.total()))
print('지갑 p2에는 {}원 있습니다.'.format(p2.total()))
```

[결과]

지갑 p1에는 1960원 있습니다.
지갑 p2에는 2170원 있습니다.

2. 클래스 - 객체를 만드는 도구

◆ self 변수

- Self는 클래스 안에 있는 모든 메소드의 첫 번째 매개 변수 자리에 넣어야 함. 생략할 수 없음.
- 일반적으로 self 매개 변수에는 인수를 넘기지 않음.
- self 매개 변수에 인수를 넘기는 경우도 있음.

2. 클래스 - 객체를 만드는 도구

◆ self 변수

```
class Dog:
    """ Dog 객체의 데이터 속성으로 이름과 색을 갖습니다. """
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def introduce(self):
        print('나의 이름은 {}이고, {} 색이에요.'.format(self.name, self.color))
```

self에 인수를 넘기지 않는 경우

```
a = Dog('Luna', 'white')
b = Dog('Terry', 'black')
a.introduce()
b.introduce()
```

self에 인수를 넘기는 경우

```
a = Dog('Luna', 'white')
b = Dog('Terry', 'black')
Dog.introduce(a) # self 에 객체 a 넘김
Dog.introduce(b) # self 에 객체 b 넘김
```

2. 클래스 - 객체를 만드는 도구

◆ self 변수

- 리스트 클래스에 self 이용하기

<pre>>>> L = list([1, 3, 5, 7]) >>> L.append(9) >>> L.pop(2) 5 >>> L [1, 3, 7, 9]</pre>	<pre>>>> L = list([1, 3, 5, 7]) >>> list.append(L, 9) # self에 L을 넘깁니다. >>> list.pop(L, 2) # self에 L을 넘깁니다. 5 >>> L [1, 3, 7, 9]</pre>
---	---

`L = [1, 3, 5, 7]`

`L.append(9)`

`list.append(L, 9)`

이 둘은 같습니다.

L이 self로 넘어갑니다.

2. 클래스 - 객체를 만드는 도구

◆ 객체의 데이터 속성 이해하기

- 데이터 속성은 각 객체가 갖는 자기만의 데이터임.
- 데이터 속성을 클래스 외부에서도 사용 가능함.

```
a = Dog('Luna', 'white')  
b = Dog('Terry', 'black')  
print('{} is {}'.format(a.name, a.color)) # 데이터 속성을 직접 사용  
print('{} is {}'.format(b.name, b.color)) # 데이터 속성을 직접 사용
```

[결과]

Luna is white.
Terry is black.

2. 클래스 - 객체를 만드는 도구

◆ 객체 출력하기

- print(객체)로 출력하기

```
>>> happy = Dog('Happy', 'white')    # 이름 'Happy', 색 'white'인 Dog 객체 생성
>>> print(happy)                     # 객체를 출력하면 객체의 위치 정보가 나옵니다.
<__main__.Dog object at 0x021C8690>
>>> id(happy)                        # Dog 객체 happy의 위치 id를 출력합니다.
35423888
>>> hex(id(happy))                   # 16진수로 id를 출력합니다.
'0x21c8690'
```

- print(객체)로 출력하면, 객체의 참조값(id)을 출력함. 즉, 객체의 위치 정보를 출력함.

2. 클래스 - 객체를 만드는 도구

◆ 객체 출력하기

- print(객체)를 수행하면 객체의 데이터 속성을 출력해 주지 않고 객체의 참조값을 출력하기 때문에, 객체의 데이터 속성을 출력하려면 따로 메소드를 만들어 주어야 함.

```
def print_dog(self):  
    print('name : {}, color : {}'.format(self.name, self.color))
```

```
>>> a = Dog('Luna', 'white')  
>>> a.print_dog()      # print(객체명)으로 사용할 수 없어서 메소드를 만들었어요.  
name : Luna, color : white
```

2. 클래스 - 객체를 만드는 도구

◆ 객체 출력하기

- `print(객체)`를 수행하면 `print_dog()` 메소드처럼 수행되면 편리함.
- 이를 위해 특별한 메소드 `__str__`, `__repr__`가 제공됨. (이 장 끝에서 설명함)

3. int 클래스 이해하기

◆ int 클래스 목록 보기

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> dir(int)           # int 클래스의 속성들을 알아 봅니다.
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__',
'__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__
repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__
rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real',
'to_bytes']
```

3. int 클래스 이해하기

◆ __init__() 생성자를 이용한 객체 생성

- 다음 두 가지 방식으로 객체 생성 가능.

```
>>> a = 10  
>>> type(a)  
<class 'int'>
```

```
>>> a = int(10) # '객체명 = 클래스()'   
>>> type(a)  
<class 'int'>
```

- 위의 왼쪽과 같이 `a = 10` 이라고 해도 `__init__()` 생성자를 호출함.
- 파이썬에서는 정수 객체를 만드는 일이 많기 때문에 간단하게 왼쪽과 같이 작성해도 정수 객체를 생성하도록 설계되었음.
- 정수뿐 아니라 다른 아홉 가지 자료형도 마찬가지임.

```
>>> b = 3.5  
>>> type(b)  
<class 'float'>
```

```
>>> b = float(3.5)  
>>> type(b)  
<class 'float'>
```

3. int 클래스 이해하기

◆ 연산자 중복

- 메소드에 +, - 같은 연산자 개념이 중복되어 있는 것을 말함.

메소드	연산 기호
<code>__add__(self, other)</code>	+
<code>__sub__(self, other)</code>	-
<code>__mul__(self, other)</code>	*
<code>__truediv__(self, other)</code>	/
<code>__floordiv__(self, other)</code>	//
<code>__mod__(self, other)</code>	%
<code>__pow__(self, other[, modulo])</code>	**
<code>__gt__(self, other)</code>	>
<code>__ge__(self, other)</code>	>=
<code>__lt__(self, other)</code>	<
<code>__le__(self, other)</code>	<=
<code>__eq__(self, other)</code>	==
<code>__ne__(self, other)</code>	!=

3. int 클래스 이해하기

◆ 연산자 중복

- 아래 두 코드는 같은 일을 하는 코드임.

연산 기호 이용	연산자 중복 메소드 이용
<pre> >>> a = 30; b = 7 >>> c = a + b >>> d = a - b >>> e = a * b >>> f = a / b >>> g = a // b >>> h = a % b >>> i = a ** b >>> c, d, e, f, g, h, i (37, 23, 210, 4.285714285714286, 4, 2, 21870000000) </pre>	<pre> >>> a = int(30); b = int(7) >>> c = a.__add__(b) # c = int.__add__(a,b) >>> d = a.__sub__(b) # d = int.__sub__(a,b) >>> e = a.__mul__(b) # e = int.__mul__(a, b) >>> f = a.__truediv__(b) # f=int.__truediv__(a, b) >>> g = a.__floordiv__(b) # g=int.__floordiv__(a, b) >>> h = a.__mod__(b) # h=int.__mod__(a, b) >>> i = a.__pow__(b) # i=int.__pow__(a, b) >>> c, d, e, f, g, h, i (37, 23, 210, 4.285714285714286, 4, 2, 21870000000) </pre> <p>클래스명 self 자리</p>

3. int 클래스 이해하기

◆ 연산자 중복

연산과 중복 메소드들은 간단히 연산자를 사용할 수 있도록 합니다.

```
a = 30; b = 7
```

```
c = a + b
```

↑
연산자

```
c = a.__add__(b)
```

↑
__add__() 메소드에는 연산자 '+' 개념이 중복되어 있어요.



3. int 클래스 이해하기

◆ 연산자 중복 (관계 연산자 : <, <=, >, >=, ==, !=)

```
>>> x = 5; y = 5; z = 7
>>> x > y
False
>>> x >= y
True
>>> x < z
True
>>> x <= y
True
>>> x == y
True
>>> x == z
False
>>> x != z
True
```

```
>>> x = 5; y = 5; z = 7
>>> x.__gt__(y)
False
>>> x.__ge__(y)
True
>>> x.__lt__(z)
True
>>> x.__le__(y)
True
>>> x.__eq__(y)
True
>>> x.__eq__(z)
False
>>> x.__ne__(z)
True
```

3. int 클래스 이해하기

- ◆ r로 시작하는 메소드 - 피연산자를 바꾸어 계산함.

메소드	연산 기호	예제
__radd__(self, other)	+	<pre>>>> a = int(100) >>> b = int(15) >>> a.__radd__(b) # b + a 115</pre>
__rsub__(self, other)	-	<pre>>>> a.__rsub__(b) # b - a -85</pre>
__rmul__(self, other)	*	<pre>>>> a.__rmul__(b) # b * a 1500</pre>
__rtruediv__(self, other)	/	<pre>>>> a.__rtruediv__(b) # b / a 0.15</pre>
__rfloordiv__(self, other)	//	<pre>>>> a.__rfloordiv__(b) # b // a 0</pre>
__rmod__(self, other)	%	<pre>>>> a.__rmod__(b) # b % a 15</pre>
__rpow__(self, other)	**	<pre>>>> a.__rpow__(b) # b ** a 406561177535215237397279707567041671010387 890632379763429051769878756383196170137717 1181093217455781996250152587890625</pre>

3. int 클래스 이해하기

CODE 93 좌표평면 클래스 작성하기. +, - 연산자 중복 이용하기

point.py

Point 객체 생성 및 연산

```
class Point:
```

```
    """ 좌표 평면의 좌표를 나타내는 클래스 """
```

```
    def __init__(self, x = 0, y = 0): # 기본값 갖는 매개변수
        self.x = x
        self.y = y
```

```
    def __add__(self, other):
        newX = self.x + other.x
        newY = self.y + other.y
        return Point(newX, newY)
```

‘+’ 연산에 대해서는 __add__() 이름으로 메소드를 만들어야 합니다.

```
    def __sub__(self, other):
        newX = self.x - other.x
        newY = self.y - other.y
        return Point(newX, newY)
```

‘-’ 연산에 대해서는 __sub__() 이름으로 메소드를 만들어야 합니다.

```
    def print_point(self):
        print('({}, {})'.format(self.x, self.y))
```

```
>>> p1 = Point(3, 5)
>>> p2 = Point(7, 8)
>>> p3 = p1 + p2
>>> p4 = p1 - p2
>>> p1.print_point()
(3,5)
>>> p2.print_point()
(7,8)
>>> p3.print_point()
(10,13)
>>> p4.print_point()
(-4,-3)
```

4. 컨테이너 자료형 클래스 이해하기

◆ 컨테이너 자료형과 __ 메소드들 존재 여부

	<code>__add__</code> <code>__mul__</code>	<code>__contains__</code>	<code>__len__</code>	<code>__getitem__</code>	<code>__setitem__</code>	<code>__delitem__</code>	<code>__iter__</code>
str	O	O	O	O	X	X	O
list	O	O	O	O	O	O	O
tuple	O	O	O	O	X	X	O
set	X	O	O	X	X	X	O
dict	X	O	O	O	O	O	O

집합과 사전에 +, * 연산자를 사용할 수 없는 이유가
`__add__()`, `__mul__()` 메소드가 없기 때문입니다.

4. 컨테이너 자료형 클래스 이해하기

- ◆ `__add__`의 '+' 연산자 중복, `__mul__`의 '*' 연산자 중복
 - `__add__()` 메소드가 있는 자료형은 '+' 연산이 가능함.
 - `__mul__()` 메소드가 있는 자료형은 '*' 연산이 가능함.
 - 집합과 사전 자료형은 `__add__()`, `__mul__()` 메소드가 없기 때문에 '+', '*' 연산이 불가능함.

자료형	<code>__add__</code> ('+'연산)	<code>__mul__</code> ('*'연산)
문자열	<pre>>>> A = 'python'; B = 'coding' >>> A + B 'pythoncoding' >>> A.__add__(B) 'pythoncoding'</pre>	<pre>>>> A * 3 'pythonpythonpython' >>> A.__mul__(3) 'pythonpythonpython'</pre>

4. 컨테이너 자료형 클래스 이해하기

◆ __add__의 '+' 연산자 중복, __mul__의 '*' 연산자 중복

자료형	__add__ ('+'연산)		__mul__ ('*'연산)	
리스트	>>> L1 = [1,2,3]; L2 = [5, 8, 9, 10]			
	>>> L1 + L2 [1, 2, 3, 5, 8, 9, 10] >>> L1.__add__(L2) [1, 2, 3, 5, 8, 9, 10]		>>> L1 * 2 [1, 2, 3, 1, 2, 3] >>> L1.__mul__(2) [1, 2, 3, 1, 2, 3]	
튜플	>>> T1 = (3, 5); T2 = (8, 9, 13)			
	>>> T1 + T2 (3, 5, 8, 9, 13) >>> T1.__add__(T2) (3, 5, 8, 9, 13)		>>> T1 * 3 (3, 5, 3, 5, 3, 5) >>> T1.__mul__(3) (3, 5, 3, 5, 3, 5)	

4. 컨테이너 자료형 클래스 이해하기

- ◆ 집합과 사전은 `__add__()`, `__mul__()`이 없음.

자료형	<code>__add__</code> ('+'연산)	<code>__mul__</code> ('*'연산)
집합	>>> S1 = {11, 22, 33}; S2 = {44, 55}	
	>>> S1 + S2 TypeError: unsupported operand type(s) for +: 'set' and 'set'	>>> S1 * 2 TypeError: unsupported operand type(s) for *: 'set' and 'int'
사전	D1 = {11:'eleven', 12:'twelve'}; D2 = {100:'hundred', 1000:'thousand'}	
	>>> D1 + D2 TypeError: unsupported operand type(s) for +: 'dict' and 'dict'	>>> D1 * 3 TypeError: unsupported operand type(s) for *: 'dict' and 'int'

4. 컨테이너 자료형 클래스 이해하기

◆ `__contains__()`, `__len__()` 메소드

- 컨테이너 자료형은 모두 `__contains__()` 메소드를 갖고 있음.
`__contains__()` 메소드는 `in`, `not in` 연산자를 지원함.
- 컨테이너 자료형 모두 `len()` 내장함수를 이용하여 원소의 개수를 구할 수 있음. 이것이 가능한 이유는 각 자료형에 `__len__()` 메소드가 정의되어 있기 때문임.

자료형	<code>__contains__</code> (<code>in</code> , <code>not in</code> 연산)	<code>__len__</code> (<code>len()</code> 함수)
문자열	<pre>>>> A = 'hello world' >>> 'w' in A # A.__contains__('w') True >>> 't' not in A True</pre>	<pre>>>> len(A) 11 >>> A.__len__() 11</pre>

4. 컨테이너 자료형 클래스 이해하기

◆ `__contains__()`, `__len__()` 메소드

자료형	<code>__contains__</code> (in, not in 연산)	<code>__len__</code> (len() 함수)
리스트	<pre>>>> L = [1,3,5,7,9] >>> 7 in L # L.__contains__(7) True >>> L.__contains__(10) # 10 in L False</pre>	<pre>>>> len(L) 5 >>> L.__len__() 5</pre>
튜플	<pre>>>> T = (2,4,6,8) >>> 8 in T # T.__contains__(8) True >>> 2 not in T False</pre>	<pre>>>> len(T) 4 >>> T.__len__() 4</pre>

4. 컨테이너 자료형 클래스 이해하기

◆ __contains__(), __len__() 메소드

자료형	__contains__ (in, not in 연산)		__len__ (len() 함수)	
집합	>>> S = {4, 7, 1, 2, 9}			
	>>> 7 in S # S.__contains__(7) True >>> 10 not in S True		>>> len(S) 5 >>> S.__len__() 5	
사전	>>> D = {1:'one', 2:'two', 3:'three'}			
	>>> 2 in D # D.__contains__(2) True >>> 5 in D False		>>> len(D) 3 >>> D.__len__() 3	

4. 컨테이너 자료형 클래스 이해하기

◆ range() 반환값과 reversed() 반환값 비교

- 아래 두 코드의 차이점은?

```
>>> len(range(5))  
5
```

```
>>> len(reversed([1,2,3,4]))  
Traceback (most recent call last):  
.....  
TypeError: object of type 'list_reverseiterator' has  
no len()
```

- range() 함수의 반환값 객체는 `__len__()` 메소드를 갖고 있음.
- reversed() 함수의 반환값 객체는 `__len__()` 메소드가 없음.

4. 컨테이너 자료형 클래스 이해하기

◆ range() 반환값과 reversed() 반환값 비교

```
>>> dir(range(5))
['__bool__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'count', 'index', 'start', 'step', 'stop']
```

```
>>> dir(reversed([1,2,3,4])) # __len__() 메소드가 없습니다.
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
 '__length_hint__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__']
```

4. 컨테이너 자료형 클래스 이해하기

◆ __getitem__() 메소드

- 시퀀스 자료형과 사전 자료형에서 [] 기호를 이용하여 원소에 접근할 수 있는 것이 __getitem__() 메소드가 지원되기 때문임.

자료형	__getitem__ 메소드	
문자열	<pre>>>> A = 'computer' >>> A[2] 'm'</pre>	<pre>>>> B = 'python' >>> B.__getitem__(-1) # B[-1] 'n'</pre> <p>문자열 객체 인덱스</p>
리스트	<pre>>>> L = [3,4,7,9] >>> L[1] 4</pre>	<pre>>>> L = [30, 22, 15, 90] >>> L.__getitem__(2) # L[2] 15</pre> <p>리스트 객체 인덱스</p>
튜플	<pre>>>> T = (10, 2, 8, 5) >>> T[0] 10</pre>	<pre>>>> T = (2, 4, 6, 8, 10) >>> T.__getitem__(-2) # T[-2] 8</pre> <p>튜플 객체 인덱스</p>

4. 컨테이너 자료형 클래스 이해하기





◆ __getitem__() 메소드

자료형	__getitem__ 메소드
집합	<pre>>>> S = 1, 3, 5, 7 >>> S[2] <----- 집합은 __getitem__() 메소드가 없어서 인덱스를 사용할 수 없습니다. TypeError: 'set' object does not support indexing >>> S.__getitem__(2) AttributeError: 'set' object has no attribute '__getitem__'</pre>
사전	<pre>>>> D = {'name':'Alice', 'age':10, 'height':135} >>> D['age'] 10 >>> D.__getitem__('height') 135 <----- 사전 객체</pre> <p>• '키'가 인덱스처럼 사용됩니다.</p>

4. 컨테이너 자료형 클래스 이해하기

◆ __setitem__(), __delitem__() 메소드

- 리스트와 사전에만 있는 메소드 (인덱스 개념이 있는 mutable 자료형)
- 리스트에서 아이템을 수정하거나 삭제하는 메소드

자료형	__setitem__	__delitem__
리스트	<pre>>>> L = [1, 3, 5, 8, 9] >>> L.__setitem__(3, 7) # L[3] = 7 >>> L [1, 3, 5, 7, 9]</pre> 	<pre>>>> L = [10, 20, 30] >>> L.__delitem__(1) # del L[1] >>> L [10, 30]</pre> <p>__delitem__() 메소드는 del 연산자에 해당합니다.</p> 
사전	<pre>>>> D = {'name':'Alice', 'age':8, 'grade':1} >>> D.__setitem__('grade', 2) # D['grade'] = 2 >>> D {'name': 'Alice', 'age': 8, 'grade': 2} >>> D.__delitem__('age') # del D['age'] >>> {'name': 'Alice', 'grade': 2}</pre> 	<p>'age':8이 삭제되었습니다.</p> 

4. 컨테이너 자료형 클래스 이해하기

◆ `__iter__()` 메소드

- 어떤 객체가 iterable인지 판단하는 메소드

```
>>> a = 100
>>> for i in a:
    print(i)
```

정수는 iterable 객체가 아니기 때문에 for 반복문을 이용할 수 없습니다.

Traceback (most recent call last):
File "<pyshell#11>", line 1, in <module>
for i in a:
TypeError: 'int' object is not iterable

다음과 같이 int 자료형에는 `__iter__()` 메소드가 없습니다.

```
>>> dir(int)
['__abs__', '__add__', ....., '__int__',
'__invert__', '__le__', ....., '__xor__',
'bit_length', ....., 'to_bytes']
```

`__iter__`이 없습니다.

5. __str__, __repr__ 메소드 이해하기

◆ 아래 두 코드의 차이점은?

```
>>> L = [1, 3, 5]
>>> print(L)
[1, 3, 5]
```

리스트 객체 출력

```
>>> S = {'red', 'blue', 'yellow'}
>>> print(S)
{'yellow', 'blue', 'red'}
```

집합 객체 출력

```
>>> class Dog:
    def __init__(self, name):
        self.name = name

>>> happy = Dog('Happy')
```

프로그래머가 만든 클래스 Dog의 객체 출력

```
>>> print(happy)
<__main__.Dog object at 0x033B59B0>
```

- print(리스트 객체), print(집합 객체)는 객체의 데이터 자체가 출력됨.
- print(프로그래머가 만든 객체)라고 하면 객체의 참조값인 id가 출력됨.
- __str__(), __repr__() 메소드를 이용하면 프로그래머가 만든 객체도 print() 함수를 이용하여 참조값이 아닌 데이터 자체를 출력할 수 있음.

5. __str__, __repr__ 메소드 이해하기



__repr__(), __str__() 메소드는 둘 다 작성해도 되고 하나만 있어도 됩니다. print() 함수를 만나면 __str__() 메소드가 있는지 보고, 있으면 __str__() 메소드를 실행합니다. 만약에 __str__() 메소드가 없으면 __repr__() 메소드가 있는지 보고, 있으면 __repr__() 메소드를 실행해 줍니다.

```
def __str__(self):
```

```
    return
```

```
def __repr__(self):
```

```
    return
```

```
return 옆에 값이 print (
```

```
) 함수를 수행할 때 인수로 들어갑니다.
```

5. __str__, __repr__ 메소드 이해하기

코드 1

기본 __repr__(), __str__() 사용

```
class Dog:
    def __init__(self, name):
        self.name = name

# main

happy = Dog('Happy')
hope = Dog('Hope')
print(happy) # __repr__(), __str__() 없으면 id 출력
print(hope)  # __repr__(), __str__() 없으면 id 출력
```

결과 1

```
<__main__.Dog object at 0x012E0AB0>
<__main__.Dog object at 0x025C3F10>
```

코드 2

수정된 __repr__(), __str__() 사용

```
class Dog:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return self.name

    def __str__(self):
        return self.name

# main

happy = Dog('Happy')
hope = Dog('Hope')
print(happy)
print(hope)
```

print() 함수의 인수가 됩니다.

__str__() 메소드를 자동 호출합니다.

__str__() 메소드를 자동 호출합니다.

결과 2

```
Happy
Hope
```

5. __str__, __repr__ 메소드 이해하기

CODE 94 비행기 예약 클래스 작성하기. 비행기 예약 정보는 출발지, 도착지, 비행기편 명으로 하고 간단하게 생성자와 __str__() 메소드만 작성함.

```
class Plane:
    """ 비행 정보를 갖고 있는 클래스입니다.
        출발지, 도착지, 비행기편명을 데이터 속성으로 갖습니다. """
    def __init__(self, depart, dest, number):
        self.depart = depart
        self.dest = dest
        self.number = number
    def __str__(self):
        return '{} is from {} to {}'.format(self.number, self.depart, self.dest)
```

```
>>> asiana = Plane('Seoul', 'LA', 'OZ202')
>>> kal = Plane('Seoul', 'NY', 'KE0085')
>>> print(asiana)
OZ202 is from Seoul to LA
>>> print(kal)
KE0085 is from Seoul to NY
```

6. 정리

- ◆ 프로그래머가 자신만의 객체를 만들고 싶다면 클래스를 정의하여야 함.
- ◆ 클래스는 속성 (데이터 속성, 메소드)로 구성됨.
- ◆ 클래스에는 객체를 생성하는 `__init__()` 생성자와 객체 출력에 이용할 수 있는 `__str__()`, `__repr__()` 메소드 등으로 구성됨.
- ◆ 파이썬의 모든 자료형은 클래스로 구현되어 있음.
- ◆ 각 자료형 클래스에 정의된 메소드들을 잘 이해하여야 함.