IPA 주관 인공지능센터 기본(fundamental) 과정

- **GitHub link: here**
- **E-Mail: windkyle7@gmail.com**

객체 지향 프로그래밍

OOP (Object-Oriented Programming)

프로퍼티 (Property)

프로퍼티는 `게터(getter)` 혹은 `세터(setter)`를 가지는 멤버를 의미하며, 파이썬에서는 `프로퍼티 어트리뷰트(property attribute)`를 돌려주는 함수이다. **(어트리뷰트는 09. OOP를 참고)**

- `fget`: 어트리뷰트 값을 얻는 함수
- `fset`: 어트리뷰트 값을 설정하는 함수
- `fdel`: 어트리뷰트 값을 삭제하는 함수

In [1]:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

In [2]:

```
c = C()
```

In [3]:

```
c.setx(10)
```

In [4]:

```
c.getx()
```

Out[4]:

```
10
```

In [5]:

```
c.x
```

Out[5]:

```
10
```

In [6]:

```
c.x = 100
```

In [7]:

```
c.x
```

Out[7]:

```
100
```

In [8]:

```
del c.x
```

In [9]:

```
c.x
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-628a1ebb8ea7> in <module>
----> 1 c.x

<ipython-input-1-85074e9574d4> in getx(self)
      4
      5     def getx(self):
----> 6         return self._x
      7
```

```
      8     def setx(self, value):

AttributeError: 'C' object has no attribute '_x'
```

객체 `c`가 클래스 `C`의 인스턴스면, `c.x`는 `게터(getter)`를 호출하고, `c.x = value`는 `세터(setter)`를, `del c.x`는 `딜리터(deleter)`를 호출한다.

아래처럼 `property()`를 `데코레이터`로 사용하여 읽기 전용 프로퍼티를 쉽게 만들 수 있다.

In [10]:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

In [11]:

```
parrot = Parrot()
```

In [12]:

```
parrot.voltage
```

Out[12]:

```
100000
```

파이썬 **3.5**버전부터 프로퍼티 개체의 독스트링이 쓰기 가능하다. 만약 독스트링이 없다면 `fget`의 독스트링(있는 경우)이 복사된다.

In [13]:

```
?parrot.voltage
```

In [14]:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x
```

```
    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

위의 코드는 첫 번째 예제로 작성했던 클래스 `C`와 동일하다. 추가적인 함수들에 원래 프로퍼티(위의 경우 **x**)와 같은 이름을 사용해야 한다.

반환된 프로퍼티 객체는 생성자 인자에 해당하는 `fget`, `fset` 및 `fdel` 어트리뷰트를 가진다.

In [15]:

```
c = C()
```

In [16]:

```
c.x
```

In [17]:

```
c.x = 10
```

In [18]:

```
c.x
```

Out[18]:

```
10
```

In [19]:

```
del c.x
```

In [20]:

```
c.x
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-20-628a1ebb8ea7> in <module>
----> 1 c.x

<ipython-input-14-3195c22c2f74> in x(self)
      6     def x(self):
      7         """I'm the 'x' property."""
----> 8         return self._x
```

```
      9
     10     @x.setter

AttributeError: 'C' object has no attribute '_x'
```

상속

앞에서도 언급했듯, 상속은 상위 객체로부터 하위 객체가 특징(멤버 혹은 어트리뷰트)과 행위(메소드)를 물려받는 것을 의미한다.

In [21]:

```
class A:
    a = 10

    def __init__(self):
        print('A')

    def hello(self):
        print('hello')
```

상속을 받기 위해서는 클래스를 정의할 때, 클래스명 옆에 상속받을 클래스명을 작성한다.

In [22]:

```
class B(A):
    b = 20

    def __init__(self):
        print('B')

    def hi(self):
        print('hi')
```

In [23]:

```
a = A()
```

```
A
```

In [24]:

```
b = B()
```

```
B
```

In [25]:

```
a.a
```

Out[25]:

```
10
```

In [26]:

```
a.hello()
```

```
hello
```

In [27]:

```
b.a
```

Out[27]:

```
10
```

In [28]:

```
b.b
```

Out[28]:

```
20
```

In [29]:

```
b.hello()
```

```
hello
```

In [30]:

```
b.hi()
```

```
hi
```

오버라이딩 (Overriding)

아래처럼 클래스 **A**를 상속받은 클래스 **B**에서 `hello`라는 메소드를 재정의할 수 있다.

In [31]:

```
class A:
    def __init__(self):
        print('A')

    def hello(self):
        print('hello')
```

In [32]:

```
class B(A):
    def __init__(self):
        print('B')

    def hello(self):
        print('hi')
```

In [33]:

```
a = A()
```

```
A
```

In [34]:

```
b = B()
```

```
B
```

In [35]:

```
a.hello()
```

```
hello
```

In [36]:

```
b.hello()
```

```
hi
```

다중 상속

파이썬에서는 하위 객체가 여러 가지의 상위 객체를 상속받을 수 있는 다중 상속을 지원한다.

In [37]:

```
class A:
```

```
    def __init__(self):
        print('A')


class B:
    def __init__(self):
        print('B')
```

예시로 클래스 **A**와 **B**가 있다고 가정할 때, 이 두 클래스를 상속받는 클래스 **C**를 정의해본다.

In [38]:

```
class C(A, B):
    def __init__(self):
        print('C')
```

In [39]:

```
a = A()
```

```
A
```

In [40]:

```
b = B()
```

```
B
```

In [41]:

```
c = C()
```

```
C
```

In [42]:

```
class D:
    def __init__(self):
        print('D')
```

`isinstance`는 인자값으로 받은 `obj` 객체가 단일 객체 혹은 튜플 안에 존재하는 객체들의 인스턴스이면 `True`, 그렇지 않으면 `False`를 반환해주는 함수이다.

In [43]:

```
isinstance(A, B)
```

Out[43]:

```
False
```

In [44]:

```
isinstance(B, A)
```

Out[44]:

```
False
```

In [45]:

```
isinstance(a, A)
```

Out[45]:

```
True
```

In [46]:

```
isinstance(b, B)
```

Out[46]:

```
True
```

`issubclass` 함수는 인자값으로 받은 `cls` 객체가 단일 객체 혹은 튜플 안에 존재하는 객체들을 상속 받았으면 `True`, 받지 않았으면 `False`를 반환해주는 함수이다.

In [47]:

```
issubclass(C, A)
```

Out[47]:

```
True
```

In [48]:

```
issubclass(C, (A,))
```

Out[48]:

```
True
```

In [49]:

```
issubclass(C, (B,))
```

Out[49]:

```
True
```

In [50]:

```
issubclass(C, (A, B))
```

Out[50]:

```
True
```

In [51]:

```
issubclass(C, (D,))
```

Out[51]:

```
False
```

`mro` 메소드는 클래스가 상속받은 상위 클래스들을 리스트로 반환한다.

In [52]:

```
A.mro()
```

Out[52]:

```
[__main__.A, object]
```

In [53]:

```
B.mro()
```

Out[53]:

```
[__main__.B, object]
```

In [54]:

```
C.mro()
```

Out[54]:

```
[__main__.C, __main__.A, __main__.B, object]
```

다이아몬드 문제 (Diamond Problem)

다중 상속은 `다이아몬드 문제`를 일으킨다. 다이아몬드 문제란, 여러 상위 객체를 상속받아서 멤버 혹은 메소드들이 중복되어 그 정의가 모호해지는 현상을 의미한다.

이 문제를 해결하기 위해 **C++**는 `virtual` 키워드를 사용해서 해결할 수 있으며, 자바는 아예 다중 상속을 지원하지 않고 `인터페이스(Interface)`만 추가로 더 상속할 수 있도록 하였다.

파이썬은 왼쪽에 선언된 클래스부터 오른쪽 순으로 우선순위를 두도록 하였다.

In [55]:

```
C.mro()
```

Out[55]:

```
[__main__.C, __main__.A, __main__.B, object]
```

In [56]:

```
c = C()
```

```
C
```

In [57]:

```
C.__base__
```

Out[57]:

```
__main__.A
```

In [58]:

```
C.__bases__
```

Out[58]:

```
(__main__.A, __main__.B)
```

In [59]:

```
class A:
    def __init__(self):
        print('A')
```

In [60]:

```
class B:
    def __init__(self):
        print('B')
```

`super`는 상위 클래스를 의미한다.

In [61]:

```
class C(A, B):
    def __init__(self):
```

```
        super().__init__()
        print('C')
```

In [62]:

```
C()
```

```
A
C
```

Out[62]:

```
<__main__.C at 0x7fef81c23898>
```

메타 클래스 (Meta Class)

개념적으로, 메타 클래스는 클래스의 클래스이다. 기존에 존재하는 클래스를 확장시키기 위해 주로 사용한다. 앞에서 `type` 함수로 새로운 타입 즉, 클래스를 정의할 수 있는 것을 확인했었다.

In [63]:

```
MyA = type('A', (), {'name': 'My A Class', 'age': 1})
```

분명 `type` 함수는 객체의 타입을 알아내기 위해 사용했던 함수로 사용했었지만, 사실 `type` 또한 메타 클래스이다.

기본적으로, 클래스는 `type()` 을 사용해서 만들어진다. 클래스의 구현체는 새 이름 공간**(namespace)**에서 실행된다.

In [64]:

```
MyB = type('B', (MyA,), {'name': 'My B Class', 'age': 2})
```

`type` 의 독스트링에는 다음과 같이 정의되어있다.

In [65]:

```
?type
```

- **type(object_or_name, bases, dict)**
- **type(object) -> the object's type**
- **type(name, bases, dict) -> a new type**

인자값으로 객체만 넣어주었을 경우, 그 객체의 타입을 반환하지만 인자값으로 객체의 이름, 상위 객체들, 기본 파라미터를 넣어주면 새로운 타입을 반환해준다.

In [66]:

```
MyA
```

Out[66]:

```
__main__.A
```

In [67]:

```
my_a = MyA()
```

In [68]:

```
my_a
```

Out[68]:

```
<__main__.A at 0x7fef81be98d0>
```

In [69]:

```
dir(my_a)
```

Out[69]:

```
['__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__',
 'age',
 'name']
```

In [70]:

```
my_a.name
```

Out[70]:

```
'My A Class'
```

In [71]:

```
my_a.age
```

Out[71]:

```
1
```

In [72]:

```
my_b = MyB()
```

In [73]:

```
my_b
```

Out[73]:

```
<__main__.B at 0x7fef81bb5c18>
```

In [74]:

```
dir(my_b)
```

Out[74]:

```
['__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__',
 'age',
 'name']
```

In [75]:

```
my_b.name
```

Out[75]:

```
'My B Class'
```

In [76]:

```
my_b.age
```

Out[76]:

```
2
```

메타 클래스를 만들 때는 다음과 같이 `type`을 상속받는다.

In [77]:

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        if name == 'Model':
            print('ModelMetaclass')
            return type.__new__(cls, name, bases, attrs)
        return type.__new__(cls, name, bases, attrs)
```

클래스 이름은 `type(name, bases, namespace)`의 결과에 지역적으로 연결된다. 위의 예에서는 클래스명이 `Model`일 경우, `ModelMetaclass`라는 메시지를 출력하도록 하였다.

다음으로 메타 클래스를 사용하여 기능을 확장시키는 클래스를 구현한다. 클래스를 만드는 과정은 클래스 정의 줄에 `metaclass` 키워드 인자를 전달하거나, 그런 인자를 포함한 이미 존재하는 클래스를 상속함으로써 커스터마이즈될 수 있다. 다음과 같이 `metaclass=클래스명`를 넣어준다.

In [78]:

```
class Model(metaclass=ModelMetaclass):
    def __init__(self):
        print('call __init__')
```

```
ModelMetaclass
```

In [79]:

```
class MyModel(metaclass=ModelMetaclass):
    def __init__(self):
        print('MyClass')
```

추상 클래스 (Abstract Class)

추상 클래스는 미완성 메소드(추상 메소드)를 포함하고 있는 클래스를 의미하며, 일반적인 객체지향 언어에서 추상 클래스는 인스턴스화 할 수 없으며, 상속을 통해 하위 클래스에 의해서 완성될 수 있다.

추상 클래스를 구현할 때 자바같은 경우, 클래스 앞에 `abstract` 키워드를 붙여서 사용한다. 파이썬에서는 추상 클래스를 구현할 때는 `ABC`를 사용한다.

ABC

메타 클래스를 사용하지 않고 추상 베이스 클래스를 간단히 `ABC`에서 파생시켜서 만들 수 있다.

In [80]:

```
from abc import ABC
```

In [81]:

```
class MyABC(ABC):
    pass
```

`ABC`의 형은 여전히 `ABCMeta`이므로, `ABC`를 상속할 때는 메타 클래스 사용에 관한 일반적인 주의가 필요한데, 다중 상속이 메타 클래스 충돌을 일으킬 수 있기 때문이다. `metaclass` 키워드를 전달하고 `ABCMeta`를 직접 사용해서 추상 베이스 클래스를 정의할 수도 있다.

In [82]:

```
from abc import ABCMeta
```

In [83]:

```
class MyABC(metaclass=ABCMeta):
    pass
```

ABCMeta

`ABCMeta`는 `추상 베이스 클래스 (ABC)`를 정의하기 위한 메타 클래스이다.

In [84]:

```
class MyABC(ABC):
    pass
```

`ABCMeta`를 메타 클래스로 생성된 클래스는 다음과 같이 `register`라는 메서드를 가진다. `register` 메소드를 통해 서브 클래스를 가질 수 있다.

In [85]:

```
MyABC.register(tuple)
```

Out[85]:

```
tuple
```

In [86]:

```
assert issubclass(tuple, MyABC)
```

In [87]:

```
assert isinstance((), MyABC)
```

`__subclasshook__` 메소드는 `subclass`를 이 `ABC`의 서브 클래스로 간주할지를 체크한다. `ABC`의 서브 클래스로 취급하고 싶은 클래스마다 `register()`를 호출할 필요 없이 `issubclass`처럼 사용할 수 있음을 의미한다. **(**이 클래스 메소드는 `ABC`의 `__subclasscheck__()` 메소드에서 호출된다**.)**

이 메서드는 `True`, `False` 또는 `NotImplemented`를 반환해야 한다. `True`를 반환하면 `subclass`를 **ABC**의 서브 클래스로 간주하며, `False`를 반환하면 `subclass`를 **ABC**의 서브 클래스로 간주하지 않는다. `NotImplemented`를 반환하면 서브 클래스 체크가 동일한 순서로 진행된다.

In [88]:

```
class Foo:
    def __getitem__(self, index):
        pass

    def __len__(self):
        pass

    def get_iterator(self):
```

```
        return iter(self)
```

`abstractmethod`는 추상 메소드를 나타내는 데코레이터이다.

In [89]:

```
from abc import abstractmethod
```

In [90]:

```
class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

In [91]:

```
MyIterable.register(Foo)
```

Out[91]:

```
__main__.Foo
```

만약 `staticmethod`와 `classmethod`를 추상 메소드로 만들고 싶다면 다음과 같이 사용할 수 있다.

In [92]:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass

    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls):
        pass
```

```
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod():
        pass

    @property
    @abstractmethod
    def my_abstract_property(self):
        pass

    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        pass

    @abstractmethod
    def _get_x(self):
        pass

    @abstractmethod
    def _set_x(self, val):
        pass

    x = property(_get_x, _set_x)
```