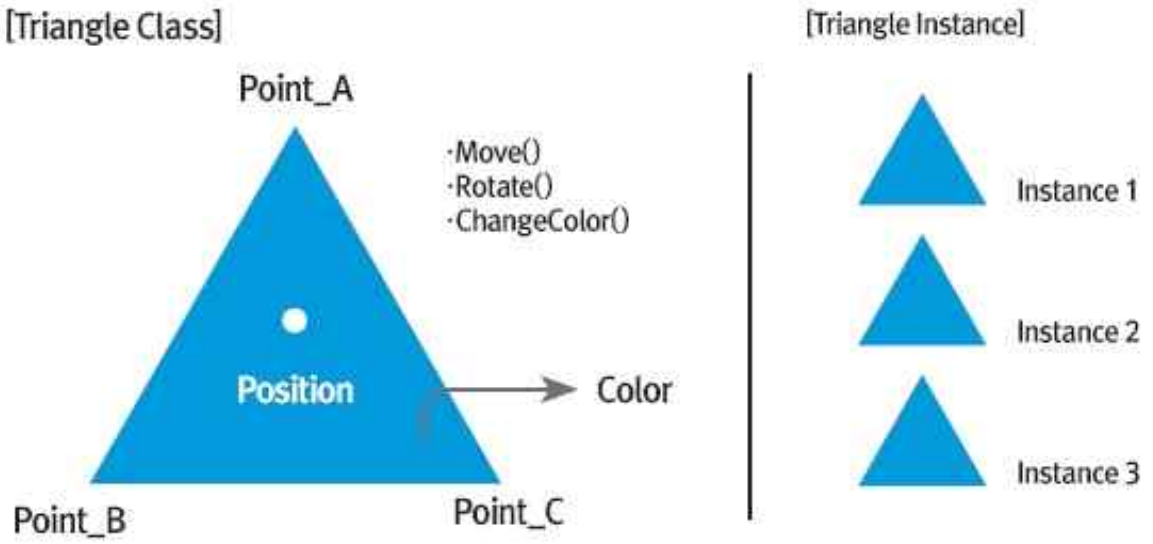
# Chapter 5
# 클래스

# 목차

# 클래스 소개

# 클래스 이야기




- 데이터와 필요한 함수를 클래스로 묶음
- 메서드(Method)
- 인스턴스(Instance)
- 정보은닉(Information Hiding)
  - 클래스 내부에서만 사용하는 데이터와 함수를 외부에서 접근할 수 없게 함
- 추상화(Abstraction)
  - 부모클래스 (공통적인 부분을 추출한 기본 클래스), 자식 클래스, 상속
- 다형성(Polymorphism)
  - 자식 클래스(인스턴스)들이 동일한 메서드 호출에 대해 다른 동작을 수행

# 클래스 선언

- 빈 클래스 선언 : 클래스 객체 생성 → 이름 공간 생성

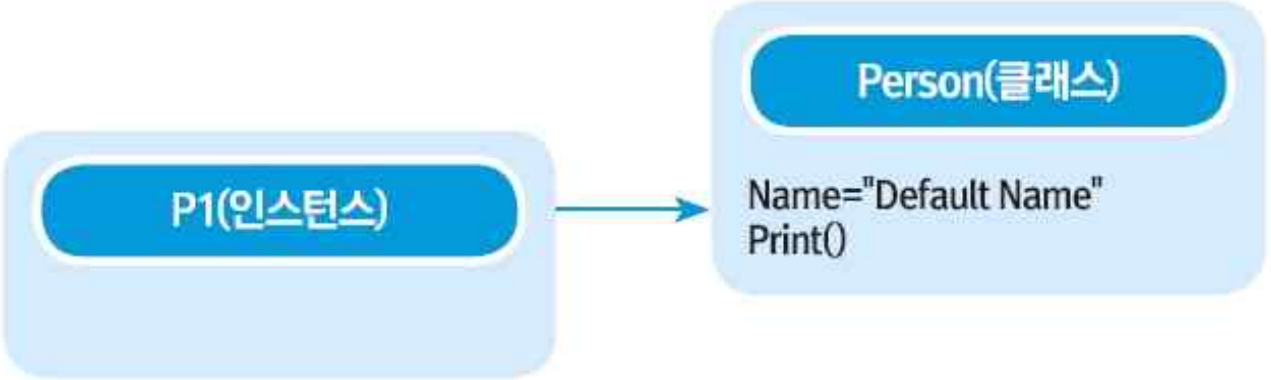
```
>>> class MyClass:
...     """ 아주 간단한 클래스 """
...     pass
...
>>> dir()# 생성된 이름 공간을 확인
['MyClass', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__', '__package__', '__spec__']
```

# 클래스 선언

- 변수와 메서드 있는 클래스 선언 : 클래스 객체 생성 → 이름 공간 생성
- 인스턴스 객체 생성 : 독립적인 이름 공간 생성

```
>>> class Person:                          #클래스 정의
        name = "Default Name"              #멤버 변수
        def Print(self):                   #멤버 메서드
            print("My Name is {0}".format(self.name))


>>> p1 = Person()                          #인스턴스 객체 생성
>>> p1.Print()                             #멤버 변수 값을 출력
My Name is Default Name
```

- 클래스 객체와 인스턴스 객체 간의 이름 공간. 인스턴스 객체는 변경 사항이 없는 동안은 클래스 객체와 동일한 데이터와 메서드를 가리킴.



# 클래스 선언

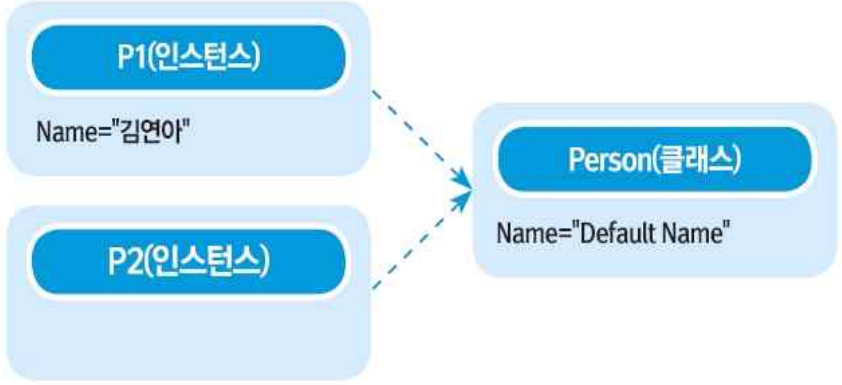
- 인스턴스 객체 이름 공간에 변경된 데이터 저장
- 멤버 변수와 메서드 접근 할 때 : 속성 접근자 ('.')
- 멤버 변수와 메서드 기본 접근 지정자 : public
- 메서드 정의 첫 인자 : self (현재 인스턴스 객체를 가리킴) (예약어도 아니고 관용적인 표현일 뿐이지만 항상 따르도록 하자.)
- 메서드 호출 방식
  - 바운드 메서드 호출 : 인스턴스 객체를 통해 호출. 암묵적으로 첫 인자는 인스턴스 객체가 됨
  - 언바운드 메서드 호출 : 클래스 객체를 통해 호출. 첫 인자로서 명시적으로 인스턴스 객체를 줌

```
>>> p1.name = "내 이름은 김연아!!“  #인스턴스 객체 멤버 변수 값 변경

>>> p1.Print()                    #바운드 메서드 호출
My Name is 내 이름은 김연아!!

>>> Person.Print(p1)              #언바운드 메서드 호출
My Name is 내 이름은 김연아!!
```

# 클래스 객체와 인스턴스 객체의 이름 공간

- 변수나 메서드 이름 찾는 순서
  - 인스턴스 객체 영역 → 클래스 객체 영역 → 전역 영역

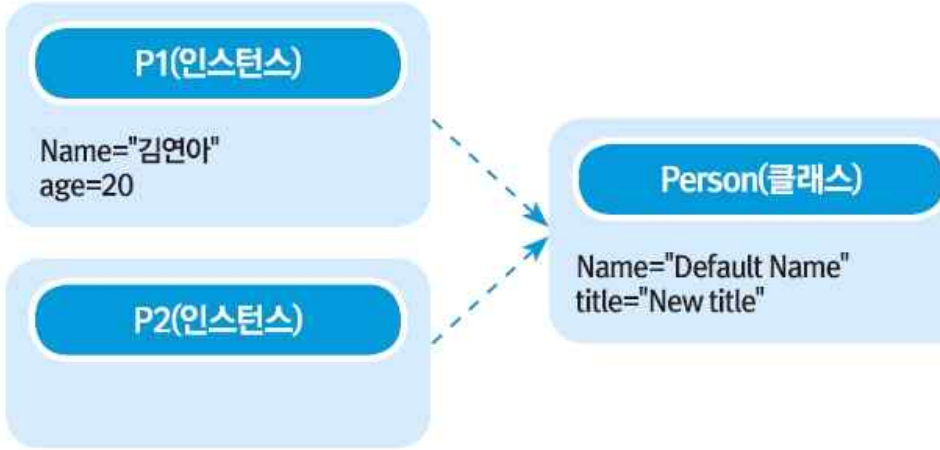
```
>>> class Person:                          #클래스 정의
        name = "Default Name"

>>> p1 = Person()                          #인스턴스 객체 생성
>>> p2 = Person()
>>> print("p1's name: ", p1.name)#p1의 Name 속
p1's name:  Default Name
>>> print("p2's name: ", p2.name)#p2의 Name 속
p2's name:  Default Name
>>>


>>> p1.name = "김연아"                     #p1 Name 속성 변경

>>> print("p1's name: ", p1.name)    #p1의 Name 속성
p1's name:  김연아
>>> print("p2's name: ", p2.name)    #p2의 Name 속성
p2's name:  Default Name
```



# 클래스 객체와 인스턴스 객체의 이름 공간

- 런타임에 클래스 객체와 인스턴스 객체 이름 영역에 멤버 변수 추가/삭제 가능

```
>>> Person.title = "New title"        #클래스 객체에 새로운 멤버 변수 추가
>>> print("p1's title: ", p1.title)    #두 인스턴스 객체에서 모두 접근 가능
p1's title:  New title
>>> print("p2's title: ", p2.title)
p2's title:  New title
>>> print("Person's title: ", Person.title) #클래스 객체에서도 접근 가능
Person's title:  New title

>>> p1.age = 20 #p1 객체에 age 멤버 변수 추가 (다른 인스턴스에서는 접근 불가)

>>> print("p1's age: ", p1.age)
p1's age:  20
>>> print("p2's age: ", p2.age)
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    print("p2's age: ", p2.age)
AttributeError: 'Person' object has no attrib
```




```
>>> print("class's Name: ", p1.__class__.Name) # 인스턴스 객체를 생성한 클래스에 접근
Default Name      # 인스턴스 객체과 클래스 객체에 같은 이름의 변수가 있을 때 유용
```

# 클래스 객체와 인스턴스 객체의 이름 공간

namespace.py

- 전역 변수와 클래스 변수 이름이 동일할 때
  - 클래스 메서드에서 self를 누락하면 의도치 않은 에러 발생

```
str = "NOT Class Member" #전역 변수
class GString:
        str = ""                        #클래스 멤버 변수
        def Set(self, msg):
                self.str = msg          #인스턴스 멤버 변수
        def Print(self):
                print(str)              #self가 누락되어 전역변수에 접근

g = GString()
g.Set("First Message")      #g 인스턴스의 str 속성 변경
g.Print()
print('g.str=', g.str)
print('GString.str=', GString.str)
```

[실행결과]
NOT Class Member
g.str= First Message
GString.str=

# 클래스 객체와 인스턴스 객체의 관계

- isinstance(인스턴스 객체, 클래스 객체) → Ture/False : 어떤 클래스에서 생성?
  - 자식클래스 인스턴스도 부모클래스 인스턴스로 평가됨
  - 모든 클래스는 object 객체를 상속받음 (버전 3 이후)

```
>>> class Person:
        pass
>>> class Bird:
        pass
>>> class Student(Person):                    #상속 관계
        pass
>>> p, s = Person(), Student()          #인스턴스 생성
>>> print("p is instance of Person: ", isinstance(p, Person))
p is instance of Person:  True
>>> print("s is instance of Person: ", isinstance(s, Person))
s is instance of Person:  True
>>> print("p is instance of object: ", isinstance(p, object)) #버전 3이후
p is instance of object:  True
>>> print("p is instance of Bird: ", isinstance(p, Bird))
p is instance of Bird:  False
>>> print("int is instance of object: ", isinstance(int, object)) # 기본자료형도…
int is instance of object:  True
```

# 생성자, 소멸자 메서드

- 생성자 메서드 : __init__()
  - 인스턴스 객체 생성할 때 초기화 작업
- 소멸자 메서드 : __del__()
  - 인스턴스 객체의 레퍼런스 카운트가 0이 될 때 호출됨. (del 할 때와 다를 수 있음)

앞 뒤에 __가 붙은 이름은 특별한 용도로 미리 정의된 것들임.

```
>>> class MyClass:
        def __init__(self, value):              #생성자
            self.Value = value
            print("Class is created Value = ", value)
        def __del__(self):                      #소멸자
            print("Class is deleted")
>>> def foo():
        d = MyClass(10)                         #함수 foo 안에서만 d가 존재
>>> foo()
Class is created Value =  10
Class is deleted
>>> d = MyClass(10)                             #참조 생성할 때 객체 생성
Class is created Value =  10
>>> del d                                       #참조지울 때 객체 소멸
Class is deleted
```

# 정적 메서드, 클래스 메서드

- 정적 메서드, 클래스 메서드
  - 메서드 확장 형태
  - 인스턴스 객체를 통하지 않고 클래스를 통해 직접 호출 가능
- 정적 메서드
  - 메서드 정의 시에 self 인자 선언하지 않음
  - 호출 시에 첫 인자 전달 필요 없음.
- 클래스 메서드
  - 메서드 정의 시에 첫 인자로 클래스 cls 정의
  - 호출 시에 암묵적으로 첫 인자로 클래스 객체가 전달됨
- 정적 메서드와 클래스 메서드 등록 방법

아래 2개의 데코레이터 사용
**@staticmethod**
**@classmethod**

# 정적 메서드, 클래스 메서드

static_class_method.py

```
class Shape:
        name = "Shape"
        def __init__(self):
                pass
        def myname():    # 클래스객체로써만 호출 가능. 인스턴스객체에서는 사용불가
                return Shape.name
        @staticmethod
        def s_name():    # 클래스객체와 인스턴스객체에서 모두 사용가능
                return Shape.name
        @classmethod
        def c_name(cls): # 클래스객체와 인스턴스객체에서 모두 사용가능
                return cls.name

s = Shape()
print(s.myname())        # 인스턴스 정보를 받지 않는 함수에 인스턴스를 전달해서 오류!
print(s.s_name())
print(s.c_name())
print(Shape.myname())
print(Shape.s_name())
print(Shape.c_name())
```

# 정적 메서드, 클래스 메서드 차이

static_class_method.py 수정

```
class Shape:
        name = "Shape"
        def __init__(self):
                pass
        def myname():           # 클래스객체로써만 호출 가능. 인스턴스객체에서는 사용불가
                return Shape.name
        @staticmethod
        def s_name():           # 클래스객체와 인스턴스객체에서 모두 사용가능
                return Shape.name
        @classmethod
        def c_name(cls):        # 클래스객체와 인스턴스객체에서 모두 사용가능
                return cls.name
class Rect(Shape):              # 상속받은 클래스에서 다르게 동작.
        name = 'Rectangle'

s = Shape()
#print(s.myname())    # 인스턴스 정보를 받지 않는 함수에 인스턴스를 전달해서 오류!
print(s.s_name())
print(s.c_name())
print(Shape.myname())
print(Shape.s_name())
print(Shape.c_name())
print(Rect.s_name())        # 클래스정보를 받지 않으므로 Shape로 간주됨.
print(Rect.c_name())        # 클래스정보를 받으므로 Rect인 것을 알 수 있음.
```

[실행결과]

```
Shape
Shape
Shape
Shape
Shape
Shape
Shape
Rectangle
```

# name mangling

name_mangling.py

- 변수의 기본 속성은 public이므로 클래스 외부에서 변경 가능
- 이를 해결하기 위해 컴파일러가 임의로 이름 변경을 변경해 주는 것
  - 변수 이름 : '__'으로 시작하면 외부에서 참조할 때는 자동으로 '_클래스이름__ 멤버변수이름' 으로 변경되어 **private**하게(?) 사용 가능.

```
class CounterManager:
        __cnt = 0           # name mangling
        def __init__(self):
                CounterManager.__cnt += 1

c1 = CounterManager()
#print(CounterManager.__cnt)        #외부에서 접근 안됨
#print(C1.__cnt)                    #외부에서 접근 안됨
print(CounterManager._CounterManager__cnt) # 변형된 이름으로 접근이 됨
print(c1._CounterManager__cnt)             # 변형된 이름으로 접근이 됨

print(dir(CounterManager))
```

[실행결과]

```
1
1
['SPrintCount',
'_CounterManager__cnt',
……]
```

# 문자열 보관 클래스

gstring.py

- 문자열로써 초기화
  Remove, Print 메서드 제공.

[실행결과]
ABCDEFGabcdefg
BCDEFGabcef

```
class GString:
        def __init__(self, init=None):
                self.content = init #인스턴스 변수 추가
        def Remove(self, str):
                for i in str:
                        self.content = self.content.replace(i,'')
                return GString(self.content)
        def Print(self):
                print(self.content)

g = GString("ABCDEFGabcdefg")           #인스턴스 객체 생성
g.Print()
g.Remove("Adg")
g.Print()
```

# 연산자 중복 (operator overloading)

gstring.py 수정

- 기존 GString 클래스의 Remove 메서드 대신
  더 직관적으로 동일한 기능을 '-' 연산자로써 사용 가능하도록 수정

```
class GString:
        def __init__(self, init=None):
                self.content = init #인스턴스 변수 추가
        def __sub__(self, str):                    #'-' 연산자 중복 정의
                for i in str:
                        self.content = self.content.replace(i,'')
                return GString(self.content)
        def __abs__(self):              #abs() 내장 함수 중복
                return GString(self.content.upper())
        def Print(self):
                print(self.content)

g = GString("aBcdef")
g -= "df"               #'-'연산자 중복된 경우 '-='도 지원함
g.Print()
g = abs(g)
g.Print()
```

[실행결과]
aBce
ABCE

# 연산자 중복 (operator overloading)

- 수치 연산자 (연산자 중복에 주로 사용됨)

| 메소드 | 연산자 | 사용 예 |
|---|---|---|
| __add__(self, other) | + (이항) | A + B, A += B |
| __sub__(self, other) | - (이항) | A - B, A -= B |
| __mul__(self, other) | * | A * B, A *= B |
| __truediv__(self, other) | / | A / B, A /= B (3 이상 지원, 그 이하는 버전에서는 __div__가 사용) |
| __floordiv__(self, other) | // | A // B, A //= B |
| __mod__(self, other) | % | A % B, A %= B |
| __divmod__(self, other) | divmod() | divmod(A, B) |
| __pow__(self, other[, modulo]) | pow(), ** | pow(A, B), A ** B |
| __lshift__(self, other | << | A << B, A <<= B |

# 연산자 중복 (operator overloading)

- 시퀀스 객체를 위한 연산자 중복 정의 메서드

| 메서드 | 연산자 | 사용 예 |
|---|---|---|
| __len__(self) | len() | len(A) |
| __contain__(self) | in | item in A |
| __getitem__(self) | A[key] | [ ] 연산자 이용한 항목 read |
| __setitem__(self) | A[key]=value | [ ] 연산자 이용한 항목 write |
| __delitem__(self) | del A[key] | del A[key] |

# 연산자 중복 (operator overloading)

sequencer.py

- 시퀀스 연산자 중복 예제

```
class Sequencer:
    def __init__(self, maxValue):        #생성자
        self.maxValue = maxValue         #인스턴스 변수 추가
    def __len__(self):                   #len() 내장함수
        return self.maxValue
    def __getitem__(self, index):        #인덱스로 아이템 값 접근
        return index * 10
    def __contains__(self, item):        #T/F 인덱스 포함 여부반환
        return (0 < item <= self.maxValue)

s = Sequencer(5)
print(s[1])
print(s[3])
print([s[i] for i in range(1,6)] )
print(len(s))
print(3 in s)
print(7 in s)
```

# 예외 발생과 예외 처리

sequencer.py 수정

```
class Sequencer:
    def __init__(self, maxValue):   #생성자
        self.maxValue = maxValue    #인스턴스 변수 추가
    def __len__(self):              #len() 내장함수
        return self.maxValue
    def __getitem__(self, index):   #인덱스로 아이템 값 접근
        if(0 < index <= self.maxValue):
            return index * 10
        else:
            raise IndexError("Index out of range") #예외발생
    def __contains__(self, item):   #T/F 인덱스 포함 여부반환
        return (0 < item <= self.maxValue)

s = Sequencer(5)
print(s[1])
print(s[3])
print([s[i] for i in range(1,6)] )
print(len(s))
print(3 in s)
print(7 in s)
try:
    print(s[7])
except IndexError as msg:
    print(msg)
```

# 상속

- 상속이란
  - 부모 클래스의 모든 속성(데이터, 메소드)를 자식 클래스로 물려줌
  - 클래스의 공통된 속성을 부모 클래스에 정의
  - 하위 클래스에서는 특화된 메소드와 데이터를 정의

- 장점
  - 각 클래스마다 동일한 코드가 작성되는 것을 방지
  - 부모 클래스에 공통된 속성을 두어 코드의 유지보수가 용이
  - 다형성(polymorphism) : 각 개별 클래스에 특화된 기능을 공통된 인터페이스로 접근 가능

# 상속

person_student.py

- 부모 클래스 Person / 자식 클래스 Student : 상속 관계
- __dict__ : 클래스 객체의 속성 정보를 dictionary 형태로 가짐.

```
class  Person:  #부모 클래스
   def __init__(self, name, phoneNumber):
      self.Name = name
      self.PhoneNumber = phoneNumber
   def PrintInfo(self):
      print("Info(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))
   def PrintPersonData(self):
      print("Person(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

class  Student(Person): #자식클래스 (Person에서 상속받음)
   def __init__(self, name, phoneNumber, subject, studentID):
      self.Name = name
      self.PhoneNumber = phoneNumber
      self.Subject = subject
      self.StudentID = studentID

p = Person("Derick", "010-123-4567")
s = Student("Marry", "010-654-1234", "Computer Science", "990999")
print(p.__dict__)
print(s.__dict__)
```

[실행결과]

```
{'Name': 'Derick', 'PhoneNumber': '010-123-4567'}
{'Name': 'Marry', 'PhoneNumber': '010-654-1234',
'Subject': 'Computer Science', 'StudentID':
'990999'}
```

# 상속

person_student.py에 추가

- Person / Student 클래스 상속관계 알아보기.
  - **issubclass(자식 클래스, 부모 클래스)** : 클래스 간의 관계 확인
  - 클래스 객체의 **__bases__** 속성 : 직계 부모 클래스를 튜플로 반환.

```
print(issubclass(Student, Person))
print(issubclass(Person, Student))
print(issubclass(Person, Person))    #자기 자신은 항상 True
print(issubclass(Person, object))    #모든 클래스는 object의 자식클래스
print('Student.__bases__=', Student.__bases__)
print('Person.__bases__=',Person.__bases__)
```

[실행결과]

```
True
False
True
True
Student.__bases__= (<class '__main__.Person'>,)
Person.__bases__= (<class 'object'>,)
```

# 상속 - 부모 클래스 생성자 호출

person_student.py 수정

- 중복된 코드 해결

```
class Person:  #부모 클래스
    def __init__(self, name, phoneNumber):
        self.Name = name
        self.PhoneNumber = phoneNumber
    def PrintInfo(self):
        print("Info(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))
    def PrintPersonData(self):
        print("Person(Name:{0}, Phone Number: {1})".format(self.Name, self.PhoneNumber))

class Student(Person): #자식클래스 (Person에서 상속받음)
    def __init__(self, name, phoneNumber, subject, studentID):
        # 클래스 객체를 이용한 언바운드 메서드 호출 방식이므로 self 를 명시
        Person.__init__( self, name, phoneNumber )        #부모 클래스의 생성자 호출
        self.Subject = subject
        self.StudentID = studentID
```

# 상속 - 자식 클래스에 메서드 추가

person_student.py 수정

```
class Person:   #부모 클래스
   중략
class Student(Person): #자식클래스 (Person에서 상속받음)
   중략
   def PrintStudentData(self): #새로운 메서드 추가
      print("Student(Subject: {0},Student ID: {1})".format(self.Subject,self.StudentID))

p = Person("Derick", "010-123-4567")
s = Student("Marry", "010-654-1234", "Computer Science", "990999")
s.PrintPersonData() #Person으로부터 상속받은 메서드 호출
s.PrintStudentData() #Student에 추가된 메서드 호출
print('dir(s)=', dir(s))
print(p.__dict__)
print(s.__dict__)

print(issubclass(Student, Pe
print(issubclass(Person, Stu
print(issubclass(Person, Pers
print(issubclass(Person, obje
print('Student.__bases__=', 
print('Person.__bases__=',Pe
```

[실행결과]

```
Person(Name:Marry, Phone Number: 010-654-1234)
Student(Subject: Computer Science,Student ID: 990999)
dir(s)= […, 'PrintStudentData', …]
{'Name': 'Derick', 'PhoneNumber': '010-123-4567'}
{'Name': 'Marry', 'PhoneNumber': '010-654-1234', 'Subject': 'Computer Science', 'StudentID': '990999'}
True
False
True
True
Student.__bases__= (<class '__main__.Person'>,)
Person.__bases__= (<class 'object'>,)
```

# 상속 - 메서드 재정의(오버라이딩)

- C++의 메서드 오버라이딩: 부모클래스와 자식클래스 메서드 이름, 매개변수, 반환값이 완전히 일치해야 함

- 파이썬의 메서드 오버라이딩: 두 메서드의 이름만 같으면 됨

# 상속 - 메서드 재정의(오버라이딩)

person_student.py 수정

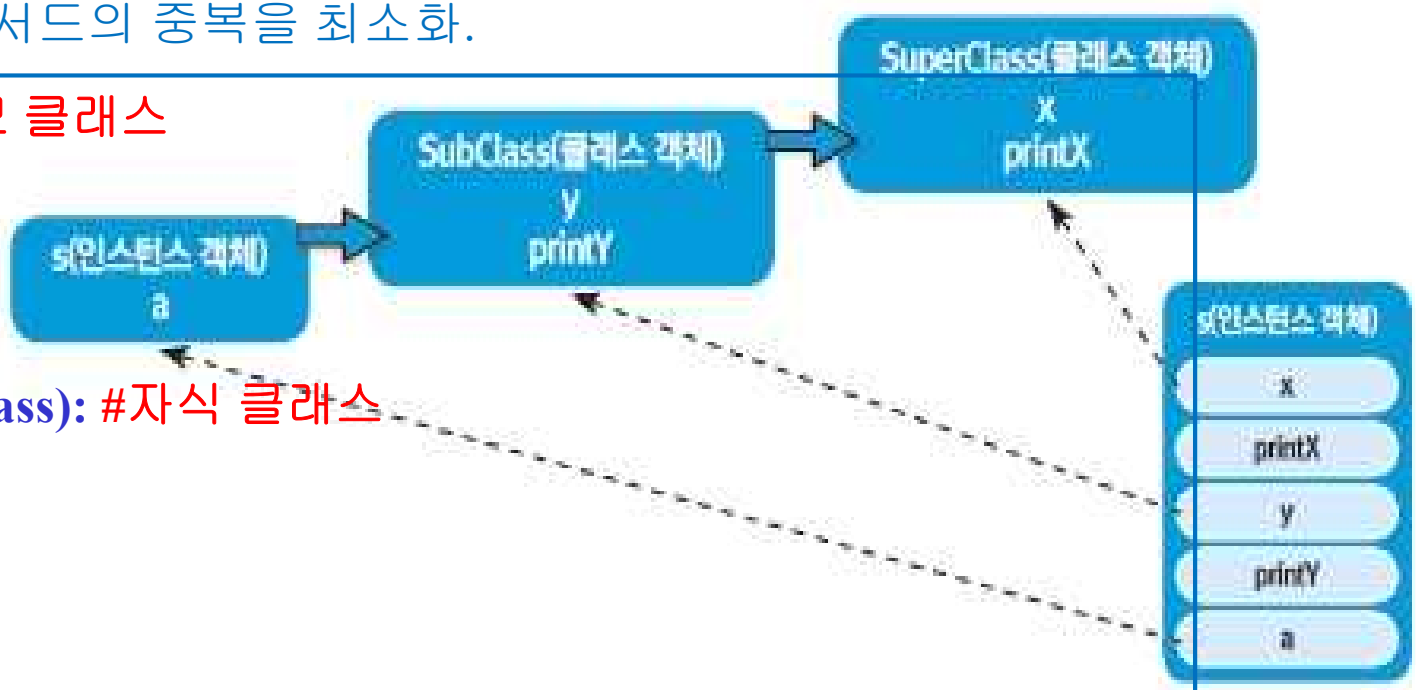
[실행결과]
Info(Name:Derick, Phone Number: 010-123-4567)
Info(Name:Marry, Phone Number: 010-654-1234)
Info(Subject:Computer Science, Student ID:990999)

```
class Person:   #부모 클래스
   중략

class Student(Person): #자식클래스 (Person에서 상속받음)
   중략

   def PrintInfo(self):   # Person의 PrintInfo()재정의 (메서드 오버라이딩)
      #print("Info(Name:{0}, Phone Number:{1})".format(self.Name,self.PhoneNumber))
      # Person의 PrintInfo 메서드를 언바운드 호출
      Person.PrintInfo(self)
      print("Info(Subject:{0}, Student ID:{1})".format(self.Subject,self.StudentID))

p = Person("Derick", "010-123-4567")
s = Student("Marry", "010-654-1234", "Computer Science", "990999")
PersonList = [p, s]

for item in PersonList:
   item.PrintInfo()              #동일 인터페이스인 PrintInfo()호출
```

# 클래스 상속과 이름 공간

- 상속 관계가 포함되었을 때 이름을 찾는 규칙
  - 인스턴스 객체 영역
    → 클래스 객체간 상속을 통한 영역(자식 클래스 영역 -> 부모 클래스 영역)
    → 전역 영역
  - 자식 클래스에서 재정의 하지 않은 메서드나 데이터는 자식 클래스의 이름 공간에 복사해 두지 않고 부모 클래스의 이름 공간의 것을 참조함
    → 데이터와 메서드의 중복을 최소화.

```
class SuperClass:  #부모 클래스
    x = 10
    def printX(self):
        print(self.x)
class SubClass(SuperClass): #자식 클래스
    y = 20
    def printY(self):
        print(self.y)
s = SubClass()
s.a = 30
```



# 클래스 상속과 이름 공간

super_subclass.py

- 부모 클래스 메서드 오버라이딩하고 멤버 데이터 값을 할당

```
class SuperClass:  #부모 클래스
    x = 10
    def printX(self):
        print(self.x)
class SubClass(SuperClass): #자식 클래스
    y = 20
    def printX(self):   #메서드 오버라이딩
        print("SubClass:", self.x)
    def printY(self):
        print(self.y)
s = SubClass()
s.a = 30 #새 인스턴스 변수 추가
s.x = 50 #SuperClass 멤버 변수에 값을 할당
print('SuperClass.__dict__=',SuperClass.__dict__)
print('SubClass.__dict__=',SubClass.__dict__)
print('s.__dict__=', s.__dict__)
```

[실행결과]

**SuperClass.__dict__=** mappingproxy({'__doc__': None, '__weakref__': <attribute '__weakref__' of 'SuperClass' objects>, '__module__': '__main__', **'x': 10,** '__dict__': <attribute '__dict__' of 'SuperClass' objects>, **'printX':** <function SuperClass.printX at 0x000001B525270620>})

**SubClass.__dict__=** mappingproxy({'__doc__': None, '__module__': '__main__', **'y': 20, 'printX':** <function SubClass.printX at 0x000001B5252706A8>, **'printY':** <function SubClass.printY at 0x000001B525270730>})

**s.__dict__= {'x': 50, 'a': 30}**

# 다중 상속

liger.py

- 2개 이상의 클래스로부터 상속받는 경우.

```
class Tiger:
    def Jump(self):
        print("호랑이처럼 멀리 점프하기")

class Lion:
    def Bite(self):
        print("사자처럼 한입에 꿀꺽하기")

class Liger(Tiger, Lion):   #다중 상속
    def Play(self):
        print(
          "라이거만의 사육사와 재미있게 놀기")

l = Liger()
l.Bite() #Lion 메서드
l.Jump() #Tiger 메서드
l.Play() #Liger 메서드
```

[실행결과]

```
사자처럼 한입에 꿀꺽하기
호랑이처럼 멀리 점프하기
라이거만의 사육사와 재미있게 놀기
```

# 상속

liger.py 수정

- 예제 5-8-7 다중 상속시 메서드 이름 검색 순서

```
class Tiger:
    def Jump(self):
        print("호랑이처럼 멀리 점프하기")
    def Cry(self):
        print("호랑이 어흥")
class Lion:
    def Bite(self):
        print("사자처럼 한입에 꿀꺽하기")
    def Cry(self):
        print("사자 으르렁")
class Liger(Tiger, Lion):  #Tiger, Lion 순서대로
    def Play(self):
        print("라이거만의 사육사와 재미있게 놀기")
l = Liger()
l.Bite() #Lion 메서드
l.Jump() #Tiger 메서드
l.Play() #Liger 메서드
l.Cry() #Tiger 먼저 검색
print('Liger.__mro__',Liger.__mro__) #method resolution order
```

```
>>> l = Liger()
>>> l.Cry() #Tiger 먼저 검색
호랑이 어흥
>>> Liger.__mro__ #method resolution order
(<class '__main__.Liger'>, <class '__main__.Tiger'>, <class '__main__.Lion'>, <class 'object'>)
```

__mro__ 속성 : 메서드 이름을 찾는 순서가 튜플로 저장되어 있음.

# super() 를 이용한 상위 클래스 메서드 호출

init_derive.py

```
class Animal:
    def __init__(self):
        print("Animal __init__()")
class Tiger(Animal):
    def __init__(self):
        super().__init__() #부모 클래스 생성자 호출
        print("Tiger __init__()")
class Lion(Animal):
    def __init__(self):
        super().__init__() #부모 클래스 생성자 호출
        print("Lion __init__()")
class Liger(Tiger, Lion):
    def __init__(self):
        super().__init__() #부모 클래스 생성자 호출
        print("Liger __init__()")
l = Liger()
```

```
Animal __init__()  # 한번만 호출됨
Lion __init__()
Tiger __init__()
Liger __init__()
```

Animal.__init__(self) 대신 super().__init__() 사용함으로써
(1) 코드 유지보수가 쉽다
(2) 파이썬 인터프리터가 클래스 간에 다중 상속 관계를 파악하여 Animal.__init__()가 1회만 호출되도록 함.