컴퓨팅사고와 파이썬 프로그래밍

Ch 7. 객체지향형 프로그래밍, 파이썬 클래스와 인스탄스

교수 김 영 탁

영남대학교 정보통신공학과

(Tel : +82-53-810-2497; E-mail : ytkim@yu.ac.kr)

Outline

- 객체지향형 프로그래밍 개요
- 클래스(class)와 객체 인스탄스(object instance)
- 클래스 속성(class attribute), 인스탄스 속성(instance attribute)
- 클래스 메소드, 인스탄스 메소드, 정적 메소드
- 연산자 (operator), 연산자 오버로딩 (operator overloading)
- 파이썬 클래스의 상속 (inheritance)

객체지향형 프로그래밍
(Object-Oriented Programming)

큰 규모의 소프트웨어 개발에서 필요한 핵심 기능

◆ 큰 규모의 소프트웨어 개발에 필요한 핵심 기능

- **System stability** (시스템 안정성): 시스템이 항상 정상적인 범위/상태에 존재하도록 유지.
- **Software re-usability** (소프트웨어 재사용성): 기존에 개발된 소프트웨어 모듈을 재사용함으로써 쉽게 새로운 시스템 개발. 시스템 개발 기간과 비용을 절감.
- **User-friendly Interface** (사용자 친화형 접속): 사용자가 쉽게 이해하고, 사용할 수 있는 인터페이스
- **System expandability** (시스템 확장성): 새로운 최신 기술이 개발되었을 때, 이를 쉽게 수용할 수 있는 확장성.
- **System Manageability** (시스템 관리 용이성): 시스템을 쉽게 관리할 수 있는 구조

객체 지향형 프로그래밍 (Object-Oriented Programming) 개요

- **데이터 추상화 (Data Abstraction)**
  - 클래스 내부에서 데이터를 처리하는 기능이 어떻게 구현되었는가에 대한 상세한 정보는 제공하지 않고, 단지 사용 방법에 대한 추상적인 정보만 제공
  - 예) 자동차의 엔진 룸 내부가 어떻게 구현되어 있고, 어떻게 동작하는가에 대한 정보 없이도, 자동차 운전석에 있는 기기와 장치를 사용하여 운전할 수 있게 하는 것과 유사함
- **캡슐화 (Encapsulation)**
  - 데이터와 데이터를 처리하는 멤버 함수를 함께 포함하는 캡슐로 만들어 사용
  - 멤버함수는 데이터가 항상 정상적인 범위 내에서 유지될 수 있도록 관리
- **정보 은닉 (Information Hiding)**
  - 클래스 내부에서 데이터를 처리하는 기능을 구현하는 방법은 새로운 기술이 개발됨에 따라 계속 바뀔 수 있으며, 따라서 구현 기술에 대한 상세한 내용은 사용자에게 알려주지 않고, 은닉시킴
  - 새로운 기술의 도입을 쉽게 할 수 있게 하며, 새로운 기술이 도입되어도 사용하는 방법에서는 변경이 없도록 함
- **정보 보호 (Data Protection)**
  - 클래스 내부의 데이터를 외부 사용자가 직접 접속할 수 있도록 허용하지 않으며, 항상 외부로 공개된 멤버함수를 통하여 접속할 수 있도록 관리
  - 정보가 비정상적으로 변경되는 것을 방지하며, 보호함

객체 지향형 프로그래밍에서의 캡슐화 (Encapsulation)

◆ 하나의 캡슐 안에 데이터와 멤버함수를 함께 포함

- 데이터: 사용자 정보를 포함, 각 데이터에는 정상적인 범위 (range of data)가 지정되어 있음
- 멤버 함수 : 데이터를 처리, 데이터의 접근 기능을 구현

◆ 예) 사람에 관련된 정보

- age: 사람의 나이를 표현할 때, 정상적인 범위는 0 ~ 150
- 나이에 관련 데이터 연산(Operations): +,-,*,/,%, logical, etc.

◆ 예) queue

- 데이터: 큐에 저장되는 개체 (숫자, 사용자/서비스 고객 등)
- 멤버함수: enqueue(), dequeue()

객체 지향형 프로그래밍에서의 상속 (inheritance)

◆상속(inheritance)

- 객체 지향형 프로그래밍에서 추상화 개념과 함께 상속(***inheritance***) 기능을 제공
- 소프트웨어 개발에서 공통적으로 사용할 수 있는 기능과 공통적인 기능에 일부 추가 기능을 구현하는 파생된 기능을 구분하여 구현하게 하여, 공통적인 기능 모듈을 다양한 분야에서 활용하게 함
- 소프트웨어 재사용 (**software-reuse**) 과 다형성 (***polymorphism***) 기능을 제공

◆일반화 (generalized, generic)와 특화 (specialized)

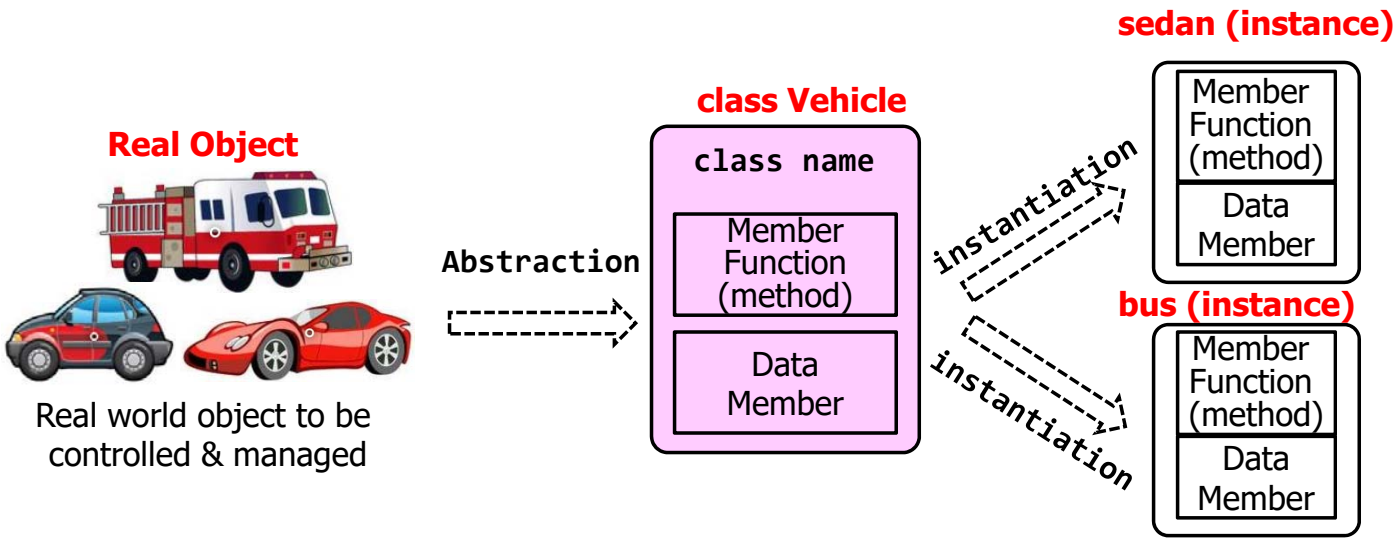
- 먼저 일반화된 소프트웨어 모듈을 설계/구현한 후, 이를 기반으로 필요에 따라 특화 (specialized)된 소프트웨어 모듈을 추가 생성함으로써 소프트웨어 재사용
- 특화된 소프트웨어 모듈은 일반화된 소프트웨어 모듈을 기반으로 특화된 기능을 추가 구현

객체 (object), 클래스 (class), 인스탄스 (instance)

Object, Class, Instance

◆ Object, Class, Instance

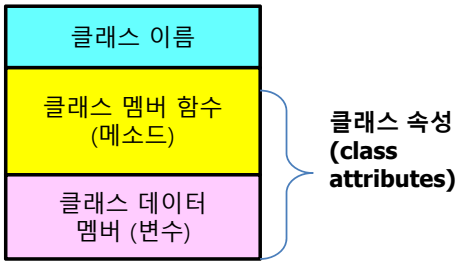
- 실제 생활환경의 객체들이 클래스로 추상화 모델링되어 소프트웨어로 표현되고 구현됨
- C++ 와 파이썬은 객체 지향형 프로그래밍 언어임
- 클래스는 데이터 멤버와 데이터 멤버들을 관리하는 멤버 함수 (메소드)를 포함함
- 클래스를 사용하여 인스탄스들을 생성



Object, Class, Instance

◆ 클래스 속성 (class attribute)과 인스탄스 속성 (instance attribute)

- 클래스 속성 (class attribute)은 그 클래스로 부터 생성된 인스탄스들이 공유함
- 인스탄스 속성 (instance attribute)은 특정 인스탄스에만 국한됨
- 파이썬 클래스는 프로그램 실행 중에 속성이 추가되거나 삭제될 수 있음
- 클래스 데이터 속성 (데이터 멤버)들은 클래스 변수로도 표현됨
- 인스탄스 데이터 속성 (데이터 멤버)들은 인스탄스 변수로도 표현됨
- 클래스에서 정의된 멤버함수를 메소드(method)라고 함
- 파이썬에서 정의된 클래스의 모든 속성들은 외부에서 접근이 가능함



파이썬 클래스 선언의 예 – class Person (1)

◆ Sample class Person (1)

```
# class Person - __init__(), __new__()

class Person():
    "Sample class Person()"
    def __init__(self, name=None, age=None):
        self.name = name
        self.age = age
        print("Person::__init__() is executed with\n name={}, age={}.".format(name, age))
    def __new__(cls, *args, **kwargs):
        print("__new__(constructor) with\n cls= {}, args= {}, kwargs={} "\
            .format(cls, args, kwargs))
        cls.__inst = super(Person, cls).__new__(cls)
        return cls.__inst
    def __str__(self):
        return "Person(name={}, age={})".format(self.name, self.age)
    def __del__(self):
        print("Person::__del__() is executed.")
    def __dict__(self):
        print("Person::__dict__() is executed.")
        return self.name, self.age
```

파이썬 클래스 선언의 예 – class Person (2)

```
# class Person – part 2

#------------
p = Person ("Kim", 25)
print("p = ", p)
print("type(p) = {}".format(type(p)))
print("p.__doc__ = ", p.__doc__)
p.name = "Lee"
p.age = 20
print("after updates of p's name and age,\np = {}".format(p))

name, age = p.__dict__()
print("name = {}, age = {}".format(name, age))
print("deleting instance p ... ")
del p
#print("after del p, p = ", p)
```

```
__new__(constructor) with
 cls= <class '__main__.Person'>, args= ('Kim', 25), kwargs={}
Person::__init__() is executed with
 name=Kim, age=25.
p =  Person(name=Kim, age=25)
type(p) = <class '__main__.Person'>
p.__doc__ =  Sample class Person()
after updates of p's name and age,
p = Person(name=Lee, age=20)
Person::__dict__() is executed.
name = Lee, age = 20
deleting instance p ...
```

파이썬 클래스 내장함수

◆ 파이썬 클래스 내장함수

파이썬 클래스 내장함수	설 명
__init__()	새롭게 생성되는 클래스 객체를 위하여 자동적으로 호출되는 초기 설정자 상속된 클래스의 생성자에서는 반드시 부모 클래스 (base class)의 __init__() 메소드를 명시적으로 호출하여야 함 예) "BaseClass.__init__(*self*, [*args*...])".
__new__()	__new__() 메소드는 클래스 객체 생성자이며, __init__()는 초기 설정자 (initiailizer) 기능을 수행함
__del__()	클래스 소멸자 (destructor)이며, 클래스 객체 인스탄스가 소멸될 때 호출됨
__doc__()	파이썬 모듈, 함수, 클래스 및 메소드을 설명하는 문자열을 반환 파이썬 객체 선언에서 첫번째 실행문으로 문자열을 설정하여 객체의 docstring을 설정할 수 있음
__dict__()	속성 (attribute) 이름과 현재 설정된 값 (value)를 매핑시키는 딕셔너리를 반환
__slots__()	__slots__() 메소드는 dict를 사용하지 않도록 하며, 고정된 속성 집합을 위한 공간만을 할당하도록 함
__repr__()	print()를 사용하여 객체를 출력할 때 사용되는 문자열을 제공
__str__(self)	print()가 호출되어 객체를 출력할 때 사용되는 문자열을 제공

클래스 생성자, 소멸자

```
# Class Definition with __init__, __new__, and __del__
class B(object):
    "Sample Class B"
    def __init__(self, data=0, *args, **kwargs): #redefinition of
__init__
        print("__init__(initializer) is executing ...")
        self.data = data
    def __new__(cls, *args, **kwargs): #redefinition of __new__ method
        print("__new__(constructor) with cls= {0}, args= {1},
              kwargs={2} is executing ...".format(cls, args, kwargs))
        instance = object.__new__(cls)
        return instance
    def __del__(self):
        print("__del__(destructor) is executing ...")
    def get(self):
        return self.data
    def set(self, value):
        self.data = value

b = B()
print("b.get() : ", b.get())
c = B(10)
d = B(20, value=100, name='B-3')
print("deleting b ...")
del b
print("deleting c ...")
del c
print("deleting d ...")
del d
```

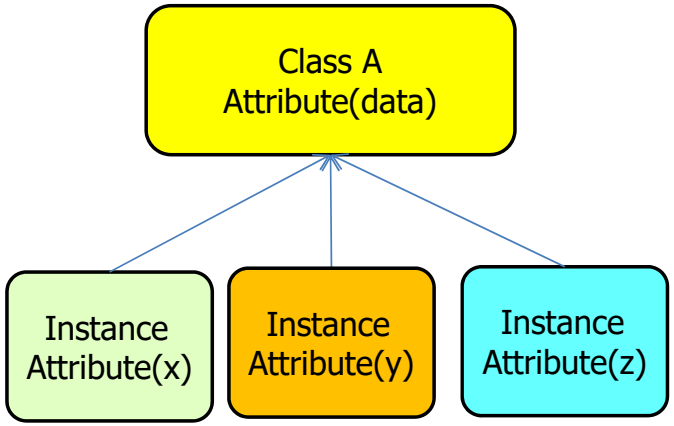
```
>>>
==== RESTART: C:/YTK-Progs/2018 Book (Python)/ch 8 Class, Inheritance/8_2 Class with init, new, d
el, get, set.py ====
__new__(constructor) with cls= <class '__main__.B'>, args= (), kwargs={} is executing ...
__init__(initializer) is executing ...
b.get() :  0
__new__(constructor) with cls= <class '__main__.B'>, args= (10,), kwargs={} is executing ...
__init__(initializer) is executing ...
__new__(constructor) with cls= <class '__main__.B'>, args= (20,), kwargs={'value': 100, 'name': '
B-3'} is executing ...
__init__(initializer) is executing ...
deleting b ...
__del__(destructor) is executing ...
deleting c ...
__del__(destructor) is executing ...
deleting d ...
__del__(destructor) is executing ...
>>>
```

파이썬
클래스 속성 (Class Attributes)
인스탄스 속성 (Instance Attributes)

파이썬 클래스 속성 접근

◆ 파이썬 클래스 속성 (Class Attribute)과 인스탄스 속성 (Instance Attribute)

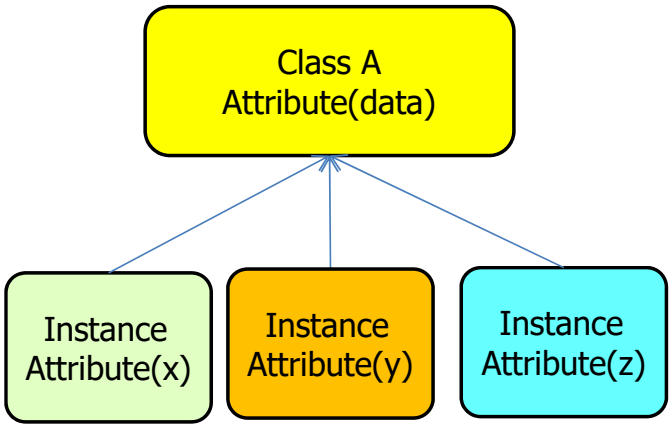
- 파이썬 클래스와 인스탄스의 속성 구분 관리
  - 파이썬 클래스/인스탄스 속성에는 멤버 함수와 멤버 데이터가 있음
  - 클래스 속성은 그 클래스로 생성된 인스탄스들에게 공통적으로 사용됨
  - 인스탄스 속성은 개별 인스탄스별로 구분되어 사용됨
  - 파이썬 클래스와 인스탄스에는 속성을 setattr() 함수를 사용하여 동적으로 추가할 수 있고, delattr()을 사용하여 삭제할 수 있음



파이썬 클래스 속성 접근

◆ 파이썬 클래스 속성 (Class Attribute)과 인스탄스 속성 (Instance Attribute)의 읽기, 쓰기

- 클래스 속성의 읽기 및 쓰기
  - class A:
    data = 10
  - getattr(A, 'data')
  - setattr(A, 'incr', classmethod(incr))
- 인스탄스 속성의 읽기 및 쓰기
  - a = A()
  - getattr(a, 'data')
    print("a.data = ", a.data)
  - setattr(a, 'data', 20)
    a.data = 30
  - setattr(a, 'instance_data', 50)



클래스/인스탄스 속성 관련 함수

◆ 클래스/인스탄스 속성 관련 내장 함수

클래스/인스탄스 속성관련 함수	설 명
getattr(obj, name[, default])	객체의 이름으로 지정된 속성 값을 반환 객체의 이름이 설정되어 있지 않으면 기본값(default value)을 반환
setattr(obj, name, value)	객체의 이름으로 지정된 속성을 전달된 값 (value)로 설정
delattr(obj, name)	객체의 이름으로 지정된 속성을 삭제
hasattr(obj, name)	객체의 이름으로 지정된 속성을 포함하고 있으면 True를 반환; 만약 해당 속성이 포함되어 있지 않으면 False를 반환

클래스 속성 (Class Attributes)과 인스탄스 속성 (Instance Attributes)

```
#class attr vs. instance attr
class A:
    "Sample class A"
    pass # null operation (just place holder)

print("A.__dict__ :", A.__dict__)

a = A() # instantiate an object a
b = A() # instantiate an object b

print("adding attributes to class A : name, value")
A.name = 'A' # create a class attribute
A.value = 10 # create a class attribute
print("A.__dict__ :", A.__dict__)

a.inst_a_attr = 'K' # instance attribute
print("a.__dict__ :", a.__dict__)
print("a.name : ", a.name)
print("a.value : ", a.value)


b.inst_b_attr = 20 # instance attribute
print("b.__dict__ :", b.__dict__)
print("b.name : ", b.name)
print("b.value : ", b.value)
```

```
A.__dict__ : {'__module__': '__main__', '__doc__': 'Sampl
e class A', '__dict__': <attribute '__dict__' of 'A' objects>, '
__weakref__': <attribute '__weakref__' of 'A' objects>}
adding attributes to class A : name, value
A.__dict__ : {'__module__': '__main__', '__doc__': 'Sampl
e class A', '__dict__': <attribute '__dict__' of 'A' objects>, '
__weakref__': <attribute '__weakref__' of 'A' objects>, 'na
me': 'A', 'value': 10}
a.__dict__ : {'inst_a_attr': 'K'}
a.name :  A
a.value :  10
b.__dict__ : {'inst_b_attr': 20}
b.name :  A
b.value :  10
```

클래스/인스탄스 속성 데이터의 추가 및 삭제

```
# class attribute and instance attribute

class A:
    data = 10 # class attribute

print("hasattr(A, 'data') : ", hasattr(A, 'data'))
print("getattr(A, 'data') : ", getattr(A, 'data'))
print("A.__dict__ :", A.__dict__)

a = A() # instantiation of class A
setattr(a, 'data', 20) #instance attribute
print("after setattr(a, 'data', 20)")
print("  => getattr(a, 'data') : ", getattr(a, 'data'))
print("a.__dict__ :", a.__dict__)
delattr(a, 'data')
print("after delattr(a, 'data')")
print("hasattr(a, 'data') : ", hasattr(a, 'data'))
print("  => getattr(a, 'data') : ", getattr(a, 'data'))

print("hasattr(a, 'name') : ", hasattr(a, 'name'))
setattr(a, 'name', 'instance of A')
print("after setattr(a, 'name', 'instance of A')")
print("  => hasattr(a, 'name') : ", hasattr(a, 'name'))

print("a.__dict__ :", a.__dict__)
print("A.__dict__ :", A.__dict__)
```

```
hasattr(A, 'data') :  True
getattr(A, 'data') :  10
A.__dict__ : {'__module__': '__main__', 'data'
: 10, '__dict__': <attribute '__dict__' of 'A'
objects>, '__weakref__': <attribute '__weakref
__' of 'A' objects>, '__doc__': None}
after setattr(a, 'data', 20)
  => getattr(a, 'data') :  20
a.__dict__ : {'data': 20}
after delattr(a, 'data')
hasattr(a, 'data') :  True
  => getattr(a, 'data') :  10
hasattr(a, 'name') :  False
after setattr(a, 'name', 'instance of A')
  => hasattr(a, 'name') :  True
a.__dict__ : {'name': 'instance of A'}
A.__dict__ : {'__module__': '__main__', 'data'
: 10, '__dict__': <attribute '__dict__' of 'A'
objects>, '__weakref__': <attribute '__weakref
__' of 'A' objects>, '__doc__': None}
```

클래스 메소드 (Class Method),
인스탄스 메소드 (Instance Method),
정적 메소드 (Static Method)

Python Class Methods

메소드의 종류	설 명
클래스 메소드 (Class Method)	클래스에서 정의되는 메소드이며, 클래스의 특정 인스탄스에 상관되지 않고 사용됨 클래스 메소드의 첫 번째 인수는 그 클래스 이름으로 설정됨
인스탄스 메소드 (Instance Method)	개별 인스탄스마다 별도로 설정되는 메소드 인스탄스 메소드를 설정할 때 첫 번째 인수는 반드시 self로 설정됨
정적 메소드 (Static Method)	정적 메소드는 그 클래스로부터 생성된 모든 인스탄스 들에게 공통적으로 사용됨

클래스 메소드와 인스탄스 메소드

◆ 클래스/인스탄스 메소드, 클래스/인스탄스 속성

```
#class with instance method, class method, static method

class A():
    data = 0 # class attribute
    @classmethod
    def cls_incr(cls, value):
        A.data += value
        return 'cls_incr invoked', cls
    def inst_incr(self, value): #instance method
        self.data += value
    def __repr__(self):
        return "instance of class A(current self.data = {})"\
               .format(self.data)

print("A.__dict__ = ", A.__dict__)
print("A.data = ", A.data)
A.cls_incr(50)
print("after A.cls_incr(50), A.data = ", A.data) # class attribute

a = A()
print("a.__dict__ = ", a.__dict__) # prints instance variables and their values
print("a = ", a)
a.data = 0 # icnluding instance's attribute
print("after a.data = 0, a.__dict__ = ", a.__dict__)
a.inst_incr(100)
print("after a.inst_incr(100), a.data = ", a.data) # instance attribute
print("after a.inst_incr(100), A.data = ", A.data) # class attribute
```

```
A.__dict__ =  {'__module__': '__main__', 'data': 0, 'cls_inc
r': <classmethod object at 0x02A38688>, 'inst_incr': <functi
on A.inst_incr at 0x02AA4028>, '__repr__': <function A.__rep
r__ at 0x02AA4070>, '__dict__': <attribute '__dict__' of 'A'
objects>, '__weakref__': <attribute '__weakref__' of 'A' obj
ects>, '__doc__': None}
A.data =  0
after A.cls_incr(50), A.data =  50
a.__dict__ =  {}
a =  instance of class A(current self.data = 50)
after a.data = 0, a.__dict__ =  {'data': 0}
after a.inst_incr(100), a.data =  100
after a.inst_incr(100), A.data =  50
```

클래스 메소드의 동적 추가

◆ dynamic inclusions of class method

```
# Dynamic inclusion of class method
def incr(cls, value = 1):
    print("cls = %s!" %cls)
    cls.data += value
    return cls.data

class A(object):
    "class A"
    data = 0 # class attribute

setattr(A, 'incr', classmethod(incr))

print("A.incr() : ", A.incr())
a = A()
print("a.data : ", a.data)
b = A()
print("b.data : ", b.data)
a.incr(10)
print("after a.incr(10) ==> a.data : ", a.data)
print("after a.incr(10) ==> b.data : ", b.data)
print("a.__dict__ : ", a.__dict__)
print("b.__dict__ : ", b.__dict__)
print("A.__dict__ : ", A.__dict__)
a.inst_x = 20
b.inst_y = 30
print("a.__dict__ : ", a.__dict__)
print("b.__dict__ : ", b.__dict__)
print("A.__dict__ : ", A.__dict__)
```

```
cls = <class '__main__.A'>!
A.incr() :  1
a.data :  1
b.data :  1
cls = <class '__main__.A'>!
after a.incr(10) ==> a.data :  11
after a.incr(10) ==> b.data :  11
a.__dict__ :  {}
b.__dict__ :  {}
A.__dict__ :  {'__module__': '__main__',
'__doc__': 'class A', 'data': 11, '__dic
t__': <attribute '__dict__' of 'A' objec
ts>, '__weakref__': <attribute '__weakre
f__' of 'A' objects>, 'incr': <classmeth
od object at 0x03417178>}
a.__dict__ :  {'inst_x': 20}
b.__dict__ :  {'inst_y': 30}
A.__dict__ :  {'__module__': '__main__',
'__doc__': 'class A', 'data': 11, '__dic
t__': <attribute '__dict__' of 'A' objec
ts>, '__weakref__': <attribute '__weakre
f__' of 'A' objects>, 'incr': <classmeth
od object at 0x03417178>}
```

정적 메소드 (static method)

◆ 정적 메소드 (static method)

- 정적 메소드는 클래스로부터 생성된 모든 인스탄스들에 공유됨
- 메소드 정의 앞에 "@staticmethod" 를 명시
- 정적 메소드는 클래스 이름 또는 인스탄스 이름으로 호출될 수 있음
- 정적 메소드는 첫 번째 인수로 인스탄스 객체를 전달하지 않음

정적 메소드의 예

```
# class A with staticmethod incr()
class A:
    data = 0 # class attribute
    @staticmethod
    def incr(value = 1): #static method
        A.data += value # class attribute
        return A.data

print("A : ", A)
print("A.data : ", A.data)
print("A.incr() : ", A.incr())
print("A.incr(10) : ", A.incr(10))

a = A()
b = A()
print("a : ", a)
print("a.data : ", a.data)
print("a.incr(50) : ", a.incr(50))
print("a.data : ", a.data)
print("b : ", b)
print("b.data : ", b.data)
print("b.incr(50) : ", b.incr(50))
print("a.data : ", a.data)
```

```
====== RESTART: C:/YTK-Progs/2018 Book
A :  <class '__main__.A'>
A.data :  0
A.incr() :  1
A.incr(10) :  11
a :  <__main__.A object at 0x0308DBB0>
a.data :  11
a.incr(50) :  61
a.data :  61
b :  <__main__.A object at 0x03334910>
b.data :  61
b.incr(50) :  111
a.data :  111
>>>
```

정적 메소드의 동적 추가

```
# class A with staticmethod incr()

def incr(value = 1): # function to be used as static method
    A.data += value
    return A.data
class A:
    data = 0 # class attribute (data)

setattr(A, 'incr', staticmethod(incr))
print("A : ", A)
print("A.incr() : ", A.incr())
print("A.data : ", A.data)
print("A.incr(10) : ", A.incr(10))

a = A()
b = A()
print("a : ", a)
print("a.data : ", a.data)
print("a.incr(50) : ", a.incr(50))
print("a.data : ", a.data)
print("b : ", b)
print("b.data : ", b.data)
print("b.incr(50) : ", b.incr(50))
print("a.data : ", a.data)
```

```
>>>
 RESTART: C:/YTK-Progs/2018 Book (Pytho
y
A :  <class '__main__.A'>
A.incr() :  1
A.data :  1
A.incr(10) :  11
a :  <__main__.A object at 0x0308DBF0>
a.data :  11
a.incr(50) :  61
a.data :  61
b :  <__main__.A object at 0x030E4670>
b.data :  61
b.incr(50) :  111
a.data :  111
>>>
```

연산자 (Operator)와
연산자 오버로딩 (Operator Overloading)

파이썬 연산자 (1)

Operation	Syntax	Function
Addition	a + b	add(a, b)
Concatenation	seq1 + seq2	concat(seq1, seq2)
Containment Test	obj in seq	contains(seq, obj)
Division	a / b	truediv(a, b)
Division	a // b	floordiv(a, b)
Bitwise And	a & b	and_(a, b)
Bitwise Exclusive Or	a ^ b	xor(a, b)
Bitwise Inversion	~ a	invert(a)
Bitwise Or	a | b	or_(a, b)
Exponentiation	a ** b	pow(a, b)
Identity	a is b	is_(a, b)
Identity	a is not b	is_not(a, b)
Indexed Assignment	obj[k] = v	setitem(obj, k, v)
Indexed Deletion	del obj[k]	delitem(obj, k)
Indexing	obj[k]	getitem(obj, k)
Left Shift	a << b	lshift(a, b)
Modulo	a % b	mod(a, b)
Multiplication	a * b	mul(a, b)
Matrix Multiplication	a @ b	matmul(a, b)

파이썬 연산자 (2)

Operation	Syntax	Function
Matrix Multiplication	a @ b	matmul(a, b)
Negation (Arithmetic)	- a	neg(a)
Negation (Logical)	not a	not_(a)
Positive	+ a	pos(a)
Right Shift	a >> b	rshift(a, b)
Slice Assignment	seq[i:j] = values	setitem(seq, slice(i, j), values)
Slice Deletion	del seq[i:j]	delitem(seq, slice(i, j))
Slicing	seq[i:j]	getitem(seq, slice(i, j))
String Formatting	s % obj	mod(s, obj)
Subtraction	a - b	sub(a, b)
Truth Test	obj	truth(obj)
Ordering	a < b	lt(a, b)
Ordering	a <= b	le(a, b)
Equality	a == b	eq(a, b)
Difference	a != b	ne(a, b)
Ordering	a >= b	ge(a, b)
Ordering	a > b	gt(a, b)

연산자 오버로딩 개요

◆ 연산자

- 기본 연산자: +, -, *, /, %, ==, != etc.
- 연산자를 함수로 생각할 수 있음

◆ 연산자를 사용한 연산

- 예) x + 7
- "+" 는 x와 7 두 개의 연산수 (operand)를 가지는 이진 연산자
- 수학 공식에서 사용하며, 세계 공통어로 모든 사람이 이해할 수 있음

◆ 연산자를 함수와 같이 생각하면: +(x, 7)

- "+" 는 함수 이름
- x, 7 은 함수에 전달되는 인수 (arguments)
- 함수 "+"는 전달된 인수들에 대한 덧셈 계산을 한 후 그 결과를 반환

연산자 오버로딩

◆ 기본 제공 (built-in) 연산자

- 예) **+, -, *, /, = , %, ==, !=**
- 파이썬 프로그래밍 언어에서 기본적으로 제공되는 자료형에 사용
- 하나의 인수를 사용하는 일진 연산자 (unary operator) 또는 두 개의 인수를 사용하는 이진 연산자 (binary operator)

◆기본 제공 연산자를 일반 클래스에 사용

- 파이썬 프로그램에서 구현하는 클래스에 기본 연산자를 사용하는 것이 가능함
- 예) 복소수를 위한 class Cmplx 객체를 +, -, *, / 연산자를 사용하여 계산을 하도록 할 수 있음
  - 수학 공식에서 익숙한 연산자를 사용하므로 쉽게 이해할 수 있음
  - 사용자 편의성이 증가된 (user friendly) 인터페이스 제공

◆ 연산의 의미를 직관적으로 이해할 수 있는 연산자를 사용하여야 함

산술 연산자의 오버로딩

연산자	연산자 오버로딩 함수	설명
+	__add__(self, other)	v3 = v1 + v2
-	__sub__(self, other)	v3 = v1 - v2
*	__mul__(self, other)	v3 = v1 * v2
@	__matmul__(self, other)	행렬 곱셈 (matrix multiplication)
/	__truediv__(self, other)	실수 나눗셈
//	_floordiv__(self, other)	정수 나눗셈 (소수점 이하 버림)
%	__mod__(self, other)	모듈로 계산 (나눗셈 후 나머지만 사용)
divmod()	__divmod__(self, other)	나눗셈의 몫과 나머지를 튜플로 반환
**, pow()	__pow__(self, other[, modulo])	지수승 계산 (power)
<<	__lshift__(self, other)	비트단위로 왼쪽 자리 옮김
>>	__rshift__(self, other)	비트단위로 오른쪽 자리 옮김
&	__and__(self, other)	비트단위 AND
^	__xor__(self, other)	비트단위 배다적 OR
|	__or__(self, other)	비트단위 OR

class Cmplx의 연산자 오버로딩

```
# class Cmplx with operator overloading (+, -, *)

class Cmplx:
    def __init__(self, real = 0.0, imagin = 0.0):
        self.real = real
        self.imagin = imagin
    def __add__(self, other):
        r = self.real + other.real
        i = self.imagin + other.imagin
        return Cmplx(r, i)
    def __sub__(self, other):
        r = self.real - other.real
        i = self.imagin - other.imagin
        return Cmplx(r, i)
    def __mul__(self, other):
        r = self.real * other.real - self.imagin * other.imagin
        i = self.real * other.imagin + self.imagin * other.real
        return Cmplx(r, i)
    def __str__(self):
        s = "Cmplx({:>8.4}, {:>8.4})".format(self.real, self.imagin)
        return s

c1 = Cmplx(1.51, 2.22)
c2 = Cmplx(3.83, 4.74)
c3 = c1 - c2
c4 = c1 + c2
c5 = c1 * c2

print("c1 : ", c1)
print("c2 : ", c2)
print("c3 : ", c3)
print("c4 : ", c4)
print("c5 : ", c5)
```

```
c1 :  Cmplx(    1.51,     2.22)
c2 :  Cmplx(    3.83,     4.74)
c3 :  Cmplx(   -2.32,    -2.52)
c4 :  Cmplx(    5.34,     6.96)
c5 :  Cmplx(   -4.74,    15.66)
```

복합 산술 대입 연산자 오버로딩

복합 산술 대입 연산자	연산자 오버로딩 함수 (in-place version of operator overloading)	설명
+=	__iadd__(self, other)	v1 += v2
-=	__isub__(self, other)	v1 -= v2
*=	__imul__(self, other)	v1 *= v2
@=	__imatmul__(self, other)	M1 @= M2
/=	__itruediv__(self, other)	a = a / b
//=	__ifloordiv__(self, other)	a = a // b
%=	__imod__(self, other)	n = n % m
**=	__ipow__(self, other[, modulo])	x = x ** n
<<=	__ilshift__(self, other)	b = b << n
>>=	__irshift__(self, other)	b = b >> n
&=	__iand__(self, other)	a = a & b
^=	__ixor__(self, other)	a = a^b
|=	__ior__(self, other)	a = a | b

복합 산술 대입 연산자 오버로딩

◆ class Cmplx의 첨가 산술 대입 연산자 오버로딩

```
# class Cmplx with operator overloading (+=, -=, *=)
class Cmplx:
    def __init__(self, real = 0.0, imag = 0.0):
        self.real = real
        self.imag = imag
    def __iadd__(self, other):
        self.real = self.real + other.real
        self.imag = self.imag + other.imag
        return Cmplx(self.real, self.imag)
    def __isub__(self, other):
        self.real = self.real - other.real
        self.imag = self.imag - other.imag
        return Cmplx(self.real, self.imag)
    def __imul__(self, other):
        r = self.real * other.real - self.imag * other.imag
        im = self.real * other.imag + self.imag * other.real
        self.real, self.imag = r, im
        return Cmplx(self.real, self.imag)
    def __str__(self):
        s = "Cmplx({:>3}, {:>3})".format(self.real, self.imag)
        return s
```

복합 산술 대입 연산자 오버로딩

```
# class Cmplx with operator overloading (+=, -=, *=) (part 2)

c1 = Cmplx(1, 2)
c2 = Cmplx(3, 4)

print("c1 : ", c1)
print("c2 : ", c2)
c3 = c4 = c5 = c1

print("c3 = {}, c4 = {}, c5 = {}".format(c3, c4, c5))

c3 += c2
c4 -= c2
c5 *= c2

print("c3 += c2; c3 = ", c3)
print("c4 -= c2; c4 = ", c4)
print("c5 *= c2; c5 = ", c5)
```

```
c1 :  Cmplx(  1,   2)
c2 :  Cmplx(  3,   4)
c3 = Cmplx(  1,   2), c4 = Cmplx(  1,   2), c5 = Cmplx(  1,   2)
c3 += c2; c3 =  Cmplx(  4,   6)
c4 -= c2; c4 =  Cmplx(  1,   2)
c5 *= c2; c5 =  Cmplx( -5,  10)
```

비교 연산자의 오버로딩

◆ Comparison Operator

비교연산자	연산자 오버로딩	remark
<	__lt__(self, other)	v1 <v2
<=	__le__(self, other)	v1 <= v2
==	__eq__(self, other)	v1 == v2
!=	__ne__(self, other)	if not define, (not)(__eq__(self, other))
>	__gt__(self, other)	if not defined, (not)(__lt__(self, other))
>=	__ge__(self, other)	if not defined, (not)(__le__(self, other))

비교 연산자 오버로딩

◆ 비교 연산자 오버로딩 예 – class Date

```
# class Date with operator overloading (part 1)

class Date:
    def __init__(self, y=1, m=1, d=1):
        self.year = y
        self.month=m
        self.day=d
    def __str__(self):
        s = "Date({:>4}-{:>02d}-{:>02d})"\
            .format(self.year, self.month, self.day)
        return s
    def __eq__(self, other):
        if self.year == other.year and self.month == other.month\
         and self.day == other.day:
            return True
        else:
            return False
    def __lt__(self, other):
        if self.__eq__(other):
            return False
        elif (self.year, self.month, self.day)\
            < (other.year, other.month, other.day):
            return True
        else:
            return False
```

```
# class Date with operator overloading (part 2)

     def __le__(self, other):
        if self.__eq__(other):
            return True
        elif self.__lt__(other):
            return True
        else:
            return False

#------------------------------
d1 = Date(2020, 7, 27)
d2 = Date(2020, 12, 25)
d3 = Date(2020, 12, 31)
d4 = Date(2020, 12, 25)
#print("d1 : ", d1)
#print("d2 : ", d2)
if d1 < d2:
    print(d1, " is less than ", d2)
if d3 > d2:
    print(d3, " is greater than ", d2)
if d2 == d4:
    print(d2, " is equal to ", d4)
```

```
Date(2020-07-27)  is less than  Date(2020-12-25)
Date(2020-12-31)  is greater than  Date(2020-12-25)
Date(2020-12-25)  is equal to  Date(2020-12-25)
```

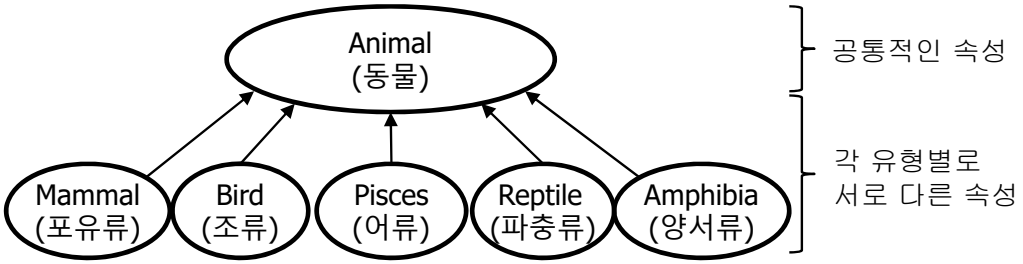
객체지향형 파이썬 클래스의 상속 (Inheritance)

상속 (inheritance) 이란?

◆ 상속 (inheritance)

- 기존에 존재하는 클래스를 기반으로 사용하며, 새로운 속성을 추가하여 새로운 클래스를 생성하는 기법
- 공통적인 속성을 가지는 클래스를 먼저 설계 및 구현하고, 이를 상속받아 다양한 파생 클래스를 생성할 수 있음
- 이미 잘 검증된 소프트웨어 모듈을 활용함으로써 새로운 소프트웨어 개발의 기간과 비용을 줄일 수 있음

◆ 상속의 예

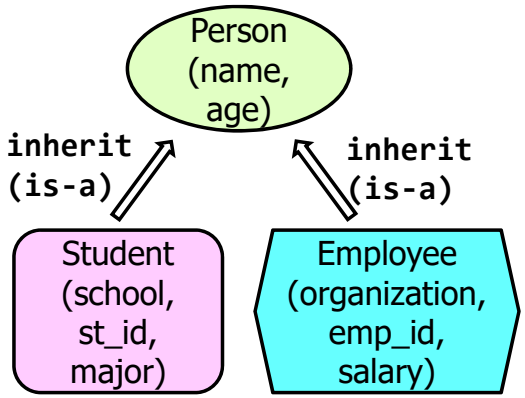


상속 관련 용어

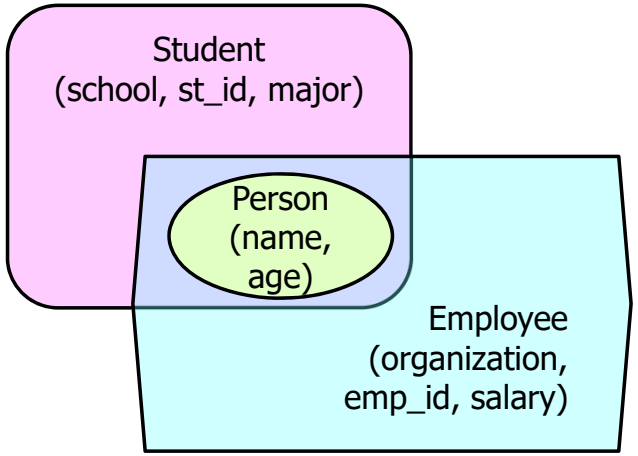
- **부모 클래스 (parent class), 기반클래스, 수퍼클래스**
  - 상속을 해 주는 기존 기반 클래스 (base class)
  - Super class라고도 함
- **자식 클래스 (child class), 파생클래스, 서브클래스**
  - 부모 클래스의 속성을 상속받고, 추가 속성을 정의하는 새로운 파생 클래스 (derived class)
  - Sub-class라고도 함
- **조상 클래스들 (ancestor classes)**
  - 부모 클래스, 부모의 부모인 조부모 및 상속 관계에 있는 직계 선대 클래스들
- **후손 클래스들 (descendant classes)**
  - 자식 클래스, 손주 클래스 및 상속 관계에 있는 직계 후대(후손) 클래스들

클래스의 상속

◆ Example of inheritance



(Inheritance of class)

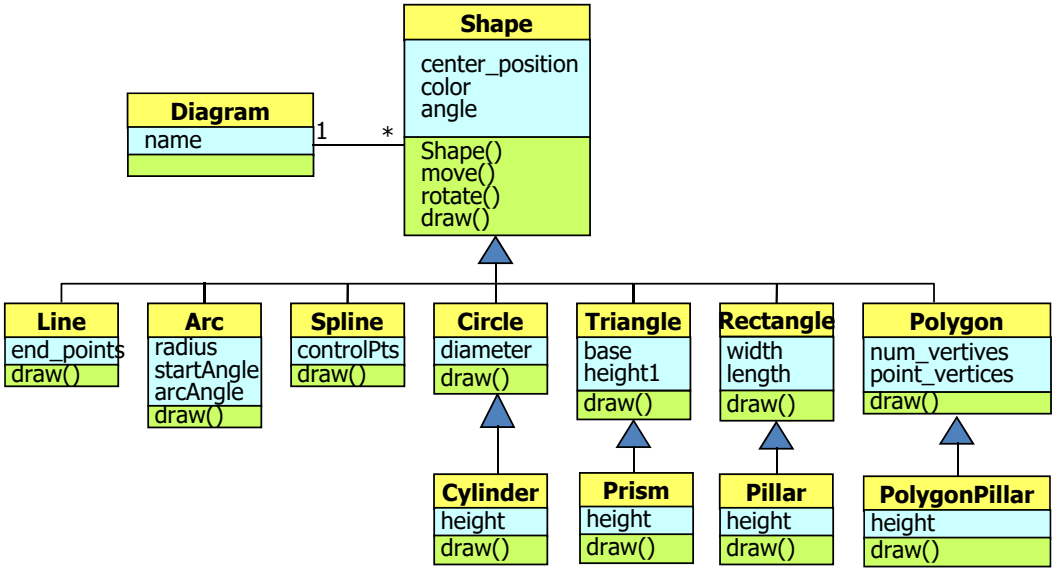


(Attributes in each class)

다단계 상속과 소프트웨어 재사용

◆ 다단계 상속과 소프트웨어 재사용

- 공통적인 속성을 가지며, 이미 검증된 부모 클래스를 다단계에 걸쳐 상속받으며 새로운 클래스 생성
- 자식 클래스는 부모 클래스의 속성을 모두 전달 받음
- 공통적인 속성에 해당하는 부분은 부모 클래스만 수정 및 보완하면 되므로 소프트웨어 유지 보수가 쉬움



클래스 상속의 예 – Person, Student, Employee

```
#Example of class inheritance in Python (part 1)

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return "Person(name={}, age={})".format(self.name, self.age)

class Student(Person):
    def __init__(self, name, age, school, st_id, major):
        Person.__init__(self, name, age)
        self.school = school
        self.st_id = st_id
        self.major = major
    def __str__(self):
        return "Student({}, {}, {}, {}, {})".\
            format(self.name, self.age, self.school, self.st_id, self.major)

class Employee(Person):
    def __init__(self, name, age, company, salary):
        Person.__init__(self, name, age)
        self.company = company
        self.salary = salary
    def __str__(self):
        return "Employee({}, {}, {}, {})".\
            format(self.name, self.age, self.company, self.salary)

p1 = Person("Kim", 20)
print(p1)

p2 = Student("Park", 22, "Yeungnam Univ.", 12345, "Info & Comm Eng")
print(p2)

p3 = Employee("Hong", 25, "Samsung", 2500)
```

클래스 상속의 예 – Person, Student, Employee

```
#Example of class inheritance in Python (part 2)


p1 = Person("Kim", 20)
print(p1)

p2 = Student("Park", 22, "Yeungnam Univ.", 12345, "Info & Comm Eng")
print(p2)

p3 = Employee("Hong", 25, "Samsung", 2500)
```

```
Person(name=Kim, age=20)
Student(Park, 22, Yeungnam Univ., 12345, Info & Comm Eng)
Employee(Hong, 25, Samsung, 2500)
```

클래스 상속의 예 – Shape, Circle, Rectangle, Triangle

```
#Python classes with inheritance  (part 1)

class Shape:
    def __init__(self, stype, x, y, color):
        self.stype = stype #type
        self.x = x #coordinate x of position
        self.y = y #coordinate y of position
        self.color = color

class Circle(Shape):
    def __init__(self, x, y, radius, color='black'):
        Shape.__init__(self, "Circle", x, y, color)
        self.radius = radius #radius of circle
    def __str__(self):
        return "a {} circle: [center=({}, {}), radius ={}]".\
               format(self.color, self.x, self.y, self.radius)

class Rectangle(Shape):
    def __init__(self, x, y, width, length, color = 'black'):
        Shape.__init__(self, "Rectangle", x, y, color)
        self.width = width # width of rectangle
        self.length = length # length of rectangle
    def __str__(self):
        return "a {} rectangle: [center=({}, {}), width ={}, length = {}]".\
               format(self.color, self.x, self.y, self.width, self.length)
```

클래스 상속의 예 – Shape, Circle, Rectangle, Triangle

```
#Python classes with inheritance  (part 2)

class Triangle(Shape):
    def __init__(self, x, y, base, height, color = 'black'):
        Shape.__init__(self, "Triangle", x, y, color)
        self.base = base # base of triangle
        self.height = height # height of triangle
    def __str__(self):
        return "a {} triangle: [center=({}, {}), base ={}, height = {}]".\
               format(self.color, self.x, self.y, self.base, self.height)

s1 = Circle(0, 0, 100)
print(s1)

s2 = Rectangle(3, 4, 10, 20, 'Red')
print(s2)

s3 = Triangle(6, 7, 50, 25, 'Blue')
print(s3)
```

```
a black circle: [center=(0, 0), radius =100]
a Red rectangle: [center=(3, 4), width =10, length = 20]
a Blue triangle: [center=(6, 7), base =50, height = 25]
```

- 객체 지향형 프로그래밍 기반의
시스템 안정성 향상
- class의 생성자 (initiator)
- class의 접근자 (accessor)
- class의 변경자 (mutator)

객체 지향형 프로그래밍기반의 시스템 안정성 향상

◆ System Stability (시스템 안정성)

- 시스템이 항상 정상적인 범위/상태를 유지하도록 관리
- 시스템의 각 속성 데이터 값이 정상적인 범위를 유지하도록 관리
- 예:
  - 온실 (green house) 속의 기온: 14° ~ 25 °
  - 사람의 나이: 0 ~ 130
  - 시간의 시, 분, 초: hour (0 ~ 23), minute (0 ~ 59), second (0 ~ 59)
  - 날짜 (월별로 다른 최대값):
    1/3/5/7/8/10/12월 (1~31), 2월 (1~28/29), 4/6/9/11월 (1~30)

객체 지향형 프로그래밍에서의 시스템 안정성 유지

◆ 클래스/인스탄스 속성의 설정 및 변경

- 생성자(initiator): 최초 클래스/인스탄스 속성의 초기값 설정
- 변경자(mutator): 클래스/인스탄스 속성의 값 변경
- 접근자(accessor): 클래스/인스탄스 속성의 값을 반환

◆ 객체 지향형 프로그래밍에서의 시스템 안정성 유지

- 클래스의 각 속성 데이터의 변경자를 구현할 때, 그 속성 데이터의 정상적인 범위이내의 값으로 설정되는지 확인
- 클래스의 생성자는 속성 변경자를 사용
- 필요한 경우, 클래스 속성 접근자에서도 정상적인 범위의 확인 기능 추가

class Person

```
#Example of class design
class Person:
    def __init__(self, name, age):
        self.setName(name) # use mutator in setting attribute with validation checking
        self.setAge(age)
    def getName(self):
        return self.name
    def getAge(self):
        return self.age
    def setName(self, nm):
        self.name = nm
    def setAge(self, ag):
        if 0 <= ag < 250: # check the correctness of attribute value
            self.age = ag
        else:
            print("*** Error in setting age (name:{}, age:{})".format(self.name, ag))
            self.age = 0 # default value
    def __str__(self):
        return "Person(name={}, age={})".format(self.getName(), self.getAge())
```

class Student

```
class Student(Person):
    def __init__(self, name, age, st_id, major, gpa):
        Person.__init__(self, name, age)
        self.setMajor(major)
        self.setSTID(st_id)
        self.setGPA(gpa)
    def getMajor(self):
        return self.major
    def getSTID(self):
        return self.st_id
    def getGPA(self):
        return self.GPA
    def setMajor(self, major):
        # checking available major
        set_majors = {"EE", "ICE", "ME", "CE"}
        if major in set_majors:
            self.major = major
        else:
            print("*** Error in setting major (name:{}, age:{})".format(self.name, major))
            self.major = None # default value
    def setSTID(self, st_id):
        # include checking correctness of ST_ID here
        self.st_id = st_id
    def setGPA(self, gpa):
        #include checking correct range of GPA
        self.GPA = gpa
    def __str__(self):
        return "Student({}, {}, {}, {}, {})".\
            format(self.getName(), self.getAge(), self.getSTID(),\
             self.getMajor(), self.getGPA())
```

class Employee

```
class Employee(Person):
    def __init__(self, name, age, company, salary):
        Person.__init__(self, name, age)
        self.company = company
        self.salary = salary
    def __str__(self):
        return "Employee({}, {}, {}, {})".\
            format(self.name, self.age, self.company, self.salary)
```

Application of class Student

```
def compareStudent(st1, st2, compare):
    if compare == "st_id":
        if st1.st_id < st2.st_id:
            return True
        else:
            return False
    elif compare == "name":
        if st1.name < st2.name:
            return True
        else:
            return False
    elif compare == "GPA":
        if st1.GPA > st2.GPA: # GPA in decreasing order
            return True
        else:
            return False
    else:
        return None

def sortStudent(L_st, compare):
    for i in range(0, len(L_st)):
        min_idx = i
        for j in range(i+1, len(L_st)):
            if compareStudent(L_st[j], L_st[min_idx], compare):
                min_idx = j
        if min_idx != i:
            L_st[i], L_st[min_idx] = L_st[min_idx], L_st[i]

def printStudents(L_st):
    for s in range(len(L_st)):
        print(L_st[s])
```

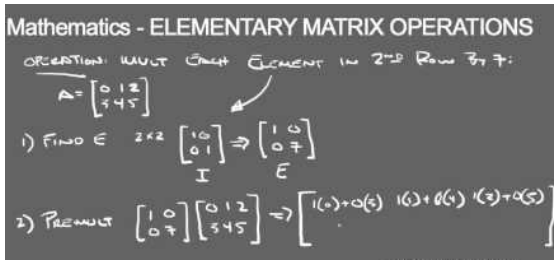
사용자 정의 클래스 예제

2차원 리스트와 행렬 (Matrix)

◆ 행렬(matrix)는 많은 문제를 해결하는데 사용

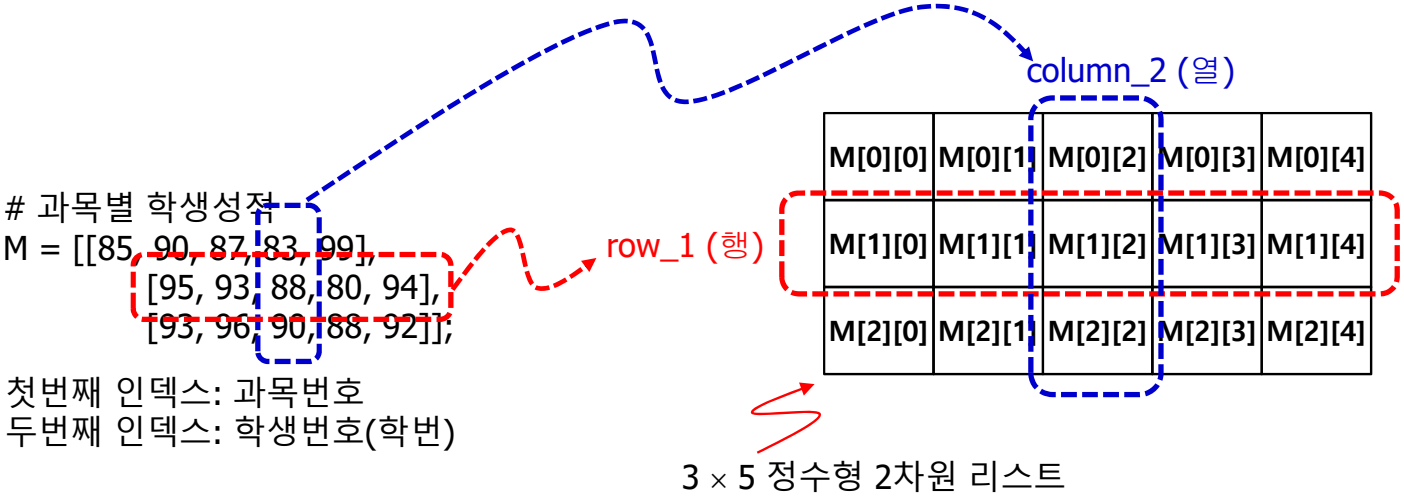
- 공학분야의 많은 문제 (예: 선형 방정식 등)를 행렬 형식으로 정리하고, 다양한 알고리즘으로 처리
- 데이터의 통계 분석에서도 많이 활용됨

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$



2차원 리스트와 행렬 (Matrix)

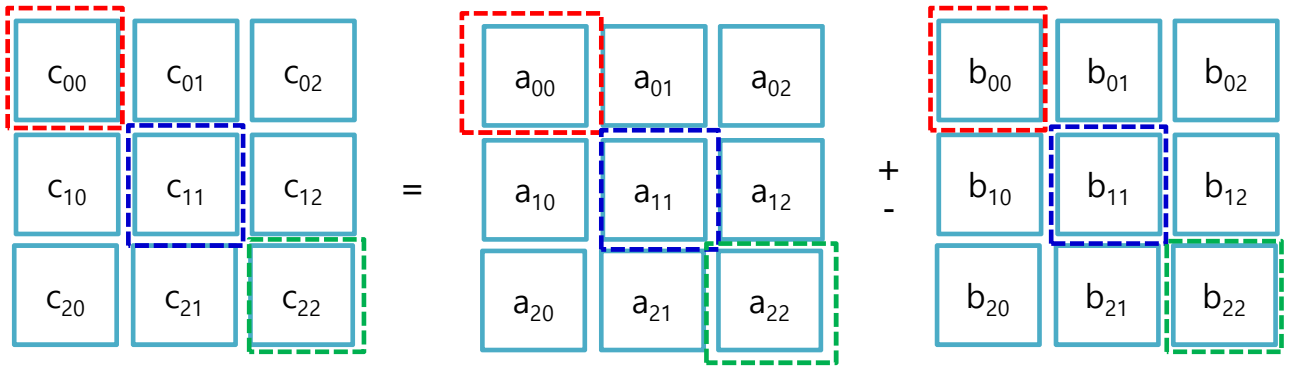
```
A = [1, 2, 3, 4, 5]   # 1차원 리스트
B = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]    # 2차원 리스트
C = [[[1, 2], [3, 4]], # 3차원 리스트
      [[5, 6], [7, 8]],
      [[9, 10], [11, 12]]]
```



행렬의 덧셈과 뺄셈

◆ 행렬의 덧셈과 뺄셈

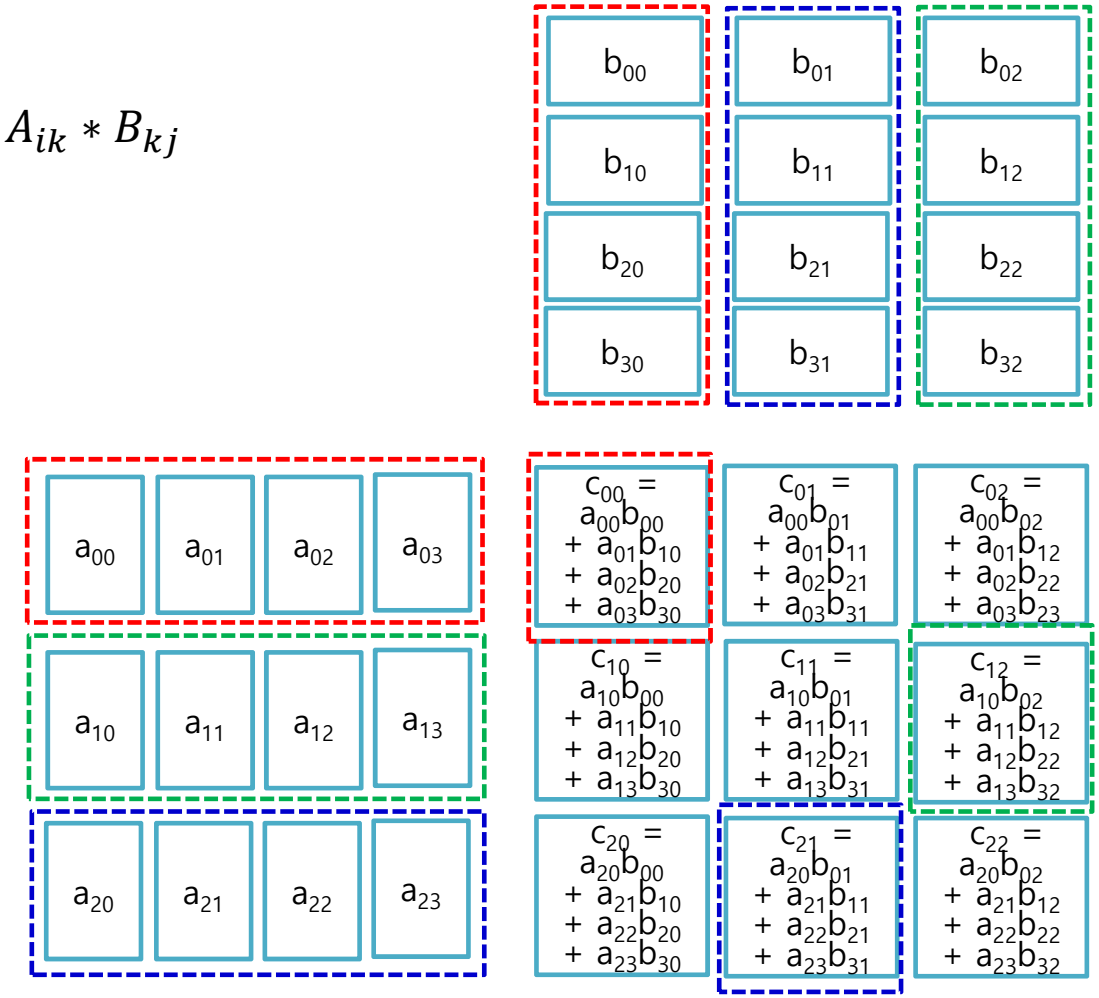
- 각 행과 열에 해당하는 항목끼리 차례로 덧셈 또는 뺄셈
- 계산 공식: $C_{ij} = A_{ij} \pm B_{ij}$
- 2중 반복문을 사용



행렬의 곱셈

◆ 행렬의 곱셈 계산

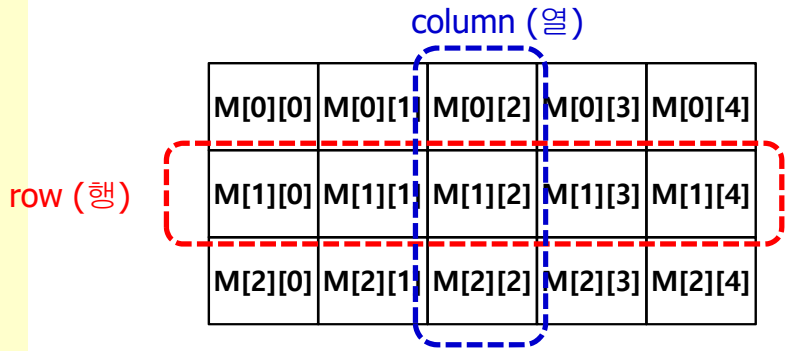
- 계산공식: $C_{ij} = \sum_{k=0}^{N-1} A_{ik} * B_{kj}$
- 3중 반복문을 사용



class Mtrx

```
# class Mtrx and Application Program (1)
class Mtrx:
    def __init__(self, name, n_row, n_col, lst_data):
        self.n_row = n_row
        self.n_col = n_col
        lst_row = []
        self.rows = []
        index = 0
        for i in range(0, self.n_row):
            for j in range(0, self.n_col):
                lst_row.append(lst_data[index])
                index = index + 1
            self.rows.append(lst_row)
            lst_row = []

    def __str__(self):
        s = "\n"
        for i in range(0, self.n_row):
            for j in range(0, self.n_col):
                s += "{:3d}".format(self.rows[i][j])
            s += "\n"
        return s
```




```
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
```

class Mtrx and Application Program (2)

```
    def __add__(self, other): # operator overloading of '+'
        lst_res = []
        for i in range(0, self.n_row):
            for j in range(0, self.n_col):
                r_ij= self.rows[i][j] + other.rows[i][j]
                lst_res.append(r_ij)
        return Mtrx("R", self.n_row, self.n_col, lst_res)

    def __sub__(self, other): # operator overloading of '+'
        lst_res = []
        for i in range(0, self.n_row):
            for j in range(0, self.n_col):
                r_ij= self.rows[i][j] - other.rows[i][j]
                lst_res.append(r_ij)
        return Mtrx("R", self.n_row, self.n_col, lst_res)

    def __mul__(self, other): # operator overloading of '*'
        lst_res = []
        for i in range(0, self.n_row):
            for j in range(0, other.n_col):
                r_ij= 0
                for k in range(0, self.n_col):
                    r_ij = r_ij + self.rows[i][k] * other.rows[k][j]
                lst_res.append(r_ij)
        return Mtrx("R", self.n_row, other.n_col, lst_res)
```

```
#-------------------------
if __name__ == "__main__":
    LA = [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15]
    LB = [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
    LC = [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

    mA = Mtrx("mA", 3, 5, LA)
    print(mA)

    mB = Mtrx("mB", 3, 5, LB)
    print(mB)

    mC = Mtrx("mC", 5, 3, LC)
    print(mC)

    mD = mA + mB
    mD.setName("mD = mA + mB")
    print(mD)

    mE = mA - mB
    mE.setName("mE = mA - mB")
    print(mE)

    mF = mA * mC
    mF.setName("mF = mA * mB")
    print(mF)
```

```
mA =
  1  2  3  4  5
  6  7  8  9 10
 11 12 13 14 15

mB =
  1  0  0  0  0
  0  1  0  0  0
  0  0  1  0  0

mC =
  0  0  0
  1  0  0
  0  1  0
  0  0  1
  0  0  0

mD = mA + mB =
  2  2  3  4  5
  6  8  8  9 10
 11 12 14 14 15

mE = mA - mB =
  0  2  3  4  5
  6  6  8  9 10
 11 12 12 14 15

mF = mA * mB =
  2  3  4
  7  8  9
 12 13 14
```

Homework 7

Homework 7.1

7.1 class Person

- 문자열 자료형의 속성인 이름 (name), 정수형 속성인 주민등록번호 (reg_id)와 정수형 속성인 나이 (age)를 가지는 class Person을 구현하라.
- class Person의 생성자 (initiator), name, reg_id와 age의 접근자 (accessor) 및 변경자 (mutator)를 구현하라. 변경자에서는 데이터 멤버의 값이 설정/변경될 때 정상적인 범위의 값으로 설정/변경되는지 확인하는 기능을 포함하도록 하고, 생성자 __init__() 함수에서는 각 데이터 멤버의 변경자를 사용하여 초기값 설정을 하도록 하라.
- class Person 객체에 대하여 print() 함수로 출력할 때 사용되는 문자열을 제공하기 위한 __str__() 함수로 구현하라.
- class Person을 사용하여 객체를 생성하고, 그 객체의 속성을 설정하며, 출력하는 시험 프로그램을 if __name__ == "__main__" 조건을 사용하여 실행하도록 구현하고, 실행 결과를 확인하라.

```
#Example of class Python
class Person:
    # 이 부분은 직접 구현 할 것

def printPersons(name, L_persons):
    print("{} :".format(name))
    for p in L_persons:
        print("  ", p)
##########################################################################
#  Application

def main():
    persons = [
        Person("Kim AB", 990101, 21),
        Person("Lee BC", 980715, 22),
        Person("Park CD", 101225, 20),
        Person("Hong EF", 110315, 19),
        Person("Yoon FG", 971005, 23),
        Person("Wrong KK", 100000, 350),
        Person("Choi AA", 960101, 24),
        Person("Yeo BB", 210715, 20),
        Person("Park CC", 111235, 21),
        Person("Hong DD", 130325, 19),
    ]
    printPersons("Persons", persons)

if __name__ == "__main__":
    main()
```

```
*** Error in setting age (name:Wrong KK, age:350)
Persons :
   Person(Kim AB, 990101, 21)
   Person(Lee BC, 980715, 22)
   Person(Park CD, 101225, 20)
   Person(Hong EF, 110315, 19)
   Person(Yoon FG, 971005, 23)
   Person(Wrong KK, 100000, 0)
   Person(Choi AA, 960101, 24)
   Person(Yeo BB, 210715, 20)
   Person(Park CC, 111235, 21)
   Person(Hong DD, 130325, 19)
```

Homework 7.2

7.2 class Student

- 위 문제 7.1에서 구현한 class Person을 상속받는 class Student를 구현하라.
- class Student는 추가적인 속성으로 정수형의 학번 (student_id), 문자열의 전공명 (major), 실수형 (float)의 평균성적 (GPA)를 가진다.
- class Student의 생성자 (initiator) 및 student_id, major, GPA의 접근자 (accessor)와 변경자 (mutator)를 구현하라. 변경자에서는 데이터 멤버의 값이 설정/변경될 때 정상적인 범위의 값으로 설정/변경되는지 확인하는 기능을 포함하도록 하고, 생성자 __init__() 함수에서는 각 데이터 멤버의 변경자를 사용하여 초기값 설정을 하도록 하라.
- class Student 객체에 대하여 print() 함수로 출력할 때 사용되는 문자열을 제공하기 위한 __str__() 함수로 구현하라.
  10명의 학생 정보를 class Student로 생성하여 리스트에 포함시키고, 이 학생들을 이름 순, 학번 순, 성적 순으로 정렬하여 출력하는 시험 프로그램을 if __name__ == "__main__" 조건을 사용하여 실행하도록 구현하고, 실행 결과를 확인하라. 이름, 학번을 기준으로한 정렬은 오름차순으로, 성적을 기준으로한 정렬은 내림차순으로 할 것.

```
#Example of class Person, class Student inheritance in Python (1)
class Person:
    #이 부분은 직접 구현할 것

class Student(Person):
    def __init__(self, name, age, st_id, major, gpa):
    # 이 부분 및 나머지 부분은 직접 구현할 것

def compareStudent(st1, st2, key):
    if key == "st_id":
        if st1.st_id < st2.st_id:
            return True
        else:
            return False
    # 나머지 부분은 직접 구현할 것

def sortStudent(L_st, key):
    for i in range(0, len(L_st)):
        min_idx = i
        for j in range(i+1, len(L_st)):
            if compareStudent(L_st[j], L_st[min_idx], key):
                min_idx = j
        if min_idx != i:
            L_st[i], L_st[min_idx] = L_st[min_idx], L_st[i]

def printStudents(L_st):
    for s in range(len(L_st)):
        print(L_st[s])
```

```
#Example of class Person, class Student inheritance in Python (2)

############################################################
#  Application

if __name__ == "__main__":
    students = [
        Student("Kim", 990101, 21, 12345, "EE", 4.0),
        Student("Lee", 980715, 22, 11234, "ME", 4.2),
        Student("Park", 101225, 20, 10234, "ICE", 4.3),
        Student("Hong", 110315, 19, 13123, "CE", 4.1),
        Student("Yoon", 971005, 23, 11321, "ICE", 4.2),
        Student("Wrong", 100000, 23, 15321, "??", 3.2)    ]
    print("students before sorting : ")
    printStudents(students)
    #
    sortStudent(students, "name")
    print("\nstudents after sorting by name : ")
    printStudents(students)
    #
    sortStudent(students, "st_id")
    print("\nstudents after sorting by student_id : ")
    printStudents(students)
    #
    sortStudent(students, "GPA")
    print("\nstudents after sorting by GPA in decreasing order : ")
    printStudents(students)
```

```
*** Error in setting major (name:Wrong, major:??)
students before sorting :
Student(Kim, 990101, 21, 12345, EE, 4.0)
Student(Lee, 980715, 22, 11234, ME, 4.2)
Student(Park, 101225, 20, 10234, ICE, 4.3)
Student(Hong, 110315, 19, 13123, CE, 4.1)
Student(Yoon, 971005, 23, 11321, ICE, 4.2)
Student(Wrong, 100000, 23, 15321, None, 3.2)

students after sorting by name :
Student(Hong, 110315, 19, 13123, CE, 4.1)
Student(Kim, 990101, 21, 12345, EE, 4.0)
Student(Lee, 980715, 22, 11234, ME, 4.2)
Student(Park, 101225, 20, 10234, ICE, 4.3)
Student(Wrong, 100000, 23, 15321, None, 3.2)
Student(Yoon, 971005, 23, 11321, ICE, 4.2)

students after sorting by student_id :
Student(Park, 101225, 20, 10234, ICE, 4.3)
Student(Lee, 980715, 22, 11234, ME, 4.2)
Student(Yoon, 971005, 23, 11321, ICE, 4.2)
Student(Kim, 990101, 21, 12345, EE, 4.0)
Student(Hong, 110315, 19, 13123, CE, 4.1)
Student(Wrong, 100000, 23, 15321, None, 3.2)

students after sorting by GPA in decreasing order :
Student(Park, 101225, 20, 10234, ICE, 4.3)
Student(Lee, 980715, 22, 11234, ME, 4.2)
Student(Yoon, 971005, 23, 11321, ICE, 4.2)
Student(Hong, 110315, 19, 13123, CE, 4.1)
Student(Kim, 990101, 21, 12345, EE, 4.0)
Student(Wrong, 100000, 23, 15321, None, 3.2)
```

Homework 7.3

7.3 class Date

- 연(year), 월(month), 일(day)의 데이터 멤버를 가지는 class Date를 구현하라. class Date의 생성자 (initiator) 및 데이터 멤버에 대한 접근자 (accessor)와 변경자 (mutator)를 구현하라. 변경자에서는 데이터 멤버의 값이 설정/변경될 때 정상적인 범위의 값으로 설정/변경되는지 확인하는 기능을 포함하도록 하고, 생성자 __init__() 함수에서는 각 데이터 멤버의 변경자를 사용하여 초기값 설정을 하도록 하라.
- class Date 객체에 대하여 print() 함수로 출력할 때 사용되는 문자열을 제공하기 위한 __str__() 함수로 구현하라. 10개의 class Date 인스탄스들을 임의로 생성하여 리스트에 포함시킨 후, 이들을 오름차순으로 정렬하는 파이썬 프로그램을 작성하라.
- class Date를 사용하여 객체를 생성하고, 그 객체의 속성을 설정하며, 출력하는 시험 프로그램을 if __name__ == "__main__" 조건을 사용하여 실행하도록 구현하고, 실행 결과를 확인하라.

```
#Example of class Date
class Date:
    def __init__(self, yr, mt, dy): # 직접 구현
    def __lt__(self, other): # 직접 구현
    def __str__(self): # 직접 구현
    def getYear(): # 직접 구현
    def getMonth(): # 직접 구현
    def getDay(): # 직접 구현
    def setYear(): # 설정 값의 검사 기능 포함, 직접 구현
    def setMonth(): # 설정 값의 검사 기능 포함, 직접 구현
    def setDay(): # 설정 값의 검사 기능 포함, 직접 구현
#############################################################
#  Application
#---------------------------
if __name__ == "__main__":
    dates = [
        Date(2000, 9, 15),
        Date(1997, 2, 20),
        Date(2001, 5, 2),
        Date(2001, 5, 1),
        Date(1999, 3, 1)
        ]
    print("dates before sorting : ")
    for d in dates:
        print(d)
    #
    dates.sort()
    print("\nstudents after sorting : ")
    for d in dates:
        print(d)
```

```
dates before sorting :
(2020- 9-24)
(2000- 9-15)
(2020- 2-29)
(2020- 1-31)
(1997- 2-20)
(2001- 5- 2)
(2001- 5- 1)
(1999- 3- 1)

dates after sorting :
(1997- 2-20)
(1999- 3- 1)
(2000- 9-15)
(2001- 5- 1)
(2001- 5- 2)
(2020- 1-31)
(2020- 2-29)
(2020- 9-24)
```

Homework 7.4

7.4 class Time

- 시(hour), 분(min), 초(sec)의 데이터 멤버를 가지는 class Time을 구현하라. class Time의 생성자 (initiator) 및 데이터 멤버에 대한 접근자 (accessor)와 변경자 (mutator)를 구현하라. 변경자에서는 데이터 멤버의 값이 설정/변경될 때 정상적인 범위의 값으로 설정/변경되는지 확인하는 기능을 포함하도록 하고, 생성자 __init__() 함수에서는 각 데이터 멤버의 변경자를 사용하여 초기값 설정을 하도록 하라.
- class Time 객체에 대하여 print() 함수로 출력할 때 사용되는 문자열을 제공하기 위한 __str__() 함수로 구현하라. 10개의 class Time 인스탄스들을 임의로 생성하여 리스트에 포함시킨 후, 이들을 오름차순으로 정렬하는 파이썬 프로그램을 작성하라.
- class Time을 사용하여 객체를 생성하고, 그 객체의 속성을 설정하며, 출력하는 시험 프로그램을 if __name__ == "__main__" 조건을 사용하여 실행하도록 구현하고, 실행 결과를 확인하라.

```
#Example of class inheritance in Python
class Time:
    def __init__(self, hr, mn, sec):
        # 직접구현
    def __lt__(self, other): # 직접구현
    def __str__(self): # 직접구현
    def getHour(): # 직접구현
    def getMinute(): # 직접구현
    def getSecond(): # 직접구현
    def setHour(): # 설정되는 값의 검사 기능 포함, 직접구현
    def setMinute(): # 설정되는 값의 검사 기능 포함, 직접구현
    def setSecond(): # 설정되는 값의 검사 기능 포함, 직접구현
#############################################################
#  Application

times = [
    Time(23, 59, 59),
    Time(9, 0, 5),
    Time(13, 30, 0),
    Time(3, 59, 59),
    Time(0, 0, 0),
    ]
print("times before sorting : ")
for t in times:
    print(t)
#
times.sort()
print("\ntimes after sorting : ")
for t in times:
    print(t)
```

```
*** Error in setting time (hour: 25)
*** Error in setting time (minute: 60)
*** Error in setting time (second: 100)
times before sorting :
(23:59:59)
( 9: 0: 5)
(13:30: 0)
( 3:59:59)
( 0: 0: 0)
( 1: 1: 1)

times after sorting :
( 0: 0: 0)
( 1: 1: 1)
( 3:59:59)
( 9: 0: 5)
(13:30: 0)
(23:59:59)
```

Homework 7.5

7.5 class Mtrx

- 행렬 (matrix)의 덧셈, 뺄셈, 곱셈 연산을 연산자 오버로딩 기능으로 사용할 수 있게 하는 class Mtrx를 구현하라. class Mtrx의 생성자 (initiator) 및 덧셈, 뺄셈, 곱셈 연산을 위한 연산자 오버로딩 멤버함수를 구현하고, class Mtrx 객체에 대하여 print() 함수로 출력할 때 사용되는 문자열을 제공하기 위한 __str__() 함수로 구현하라.
- 2개의 3 x 5 class Mtrx 인스탄스 mA과 mB, 1개의 5 x 3 class Mtrx 인스탄스 mC를 임의로 생성하고, mD = mA + mB; mE = mA – mB; mF = mA * mC를 계산하여 시험 프로그램을 if __name__ == "__main__" 조건을 사용하여 실행하도록 구현하고, 실행 결과를 확인하라.

```
# class Mtrx and Application Program (1)

class Mtrx:
    def __init__(self, name, n_row, n_col, L_data): #직접 구현
    def setName(self, name): # 직접 구현
    def __str__(self): #직접 구현
    def __add__(self, other): # 직접 구현, operator overloading of '+'
    def __sub__(self, other): # 직접 구현, operator overloading of '-'
    def __mul__(self, other): # 직접 구현, operator overloading of '*'
```

```
# class Mtrx and Application Program (2)

#--------------------------
if __name__ == "__main__":
    LA = [ 1,  2,  3,  4,  5,\
           6,  7,  8,  9, 10,\
          11, 12, 13, 14, 15]
    LB = [1, 0, 0, 0, 0,\
          0, 1, 0, 0, 0,\
          0, 0, 1, 0, 0]
    LC = [0, 0, 0,\
          1, 0, 0,\
          0, 1, 0,\
          0, 0, 1,\
          0, 0, 0]

    mA = Mtrx("mA", 3, 5, LA)
    print(mA)

    mB = Mtrx("mB", 3, 5, LB)
    print(mB)

    mC = Mtrx("mC", 5, 3, LC)
    print(mC)

    mD = mA + mB
    mD.setName("mD = mA + mB")
    print(mD)

    mE = mA - mB
    mE.setName("mE = mA - mB")
    print(mE)

    mF = mA * mC
    mF.setName("mF = mA * mC")
    print(mF)
```

```
mA =
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15

mB =
 1  0  0  0  0
 0  1  0  0  0
 0  0  1  0  0

mC =
 0  0  0
 1  0  0
 0  1  0
 0  0  1
 0  0  0

mD = mA + mB =
 2  2  3  4  5
 6  8  8  9 10
11 12 14 14 15

mE = mA - mB =
 0  2  3  4  5
 6  6  8  9 10
11 12 12 14 15

mF = mA * mC =
 2  3  4
 7  8  9
12 13 14
```