

12장 상속

12.2 상속

- 부모 클래스를 상속받아서 자식 클래스를 정의
- 상속(inheritance)은 기존에 존재하는 클래스로부터 코드와 데이터를 이어받고 자신이 필요한 기능을 추가하는 기법이다.



상속



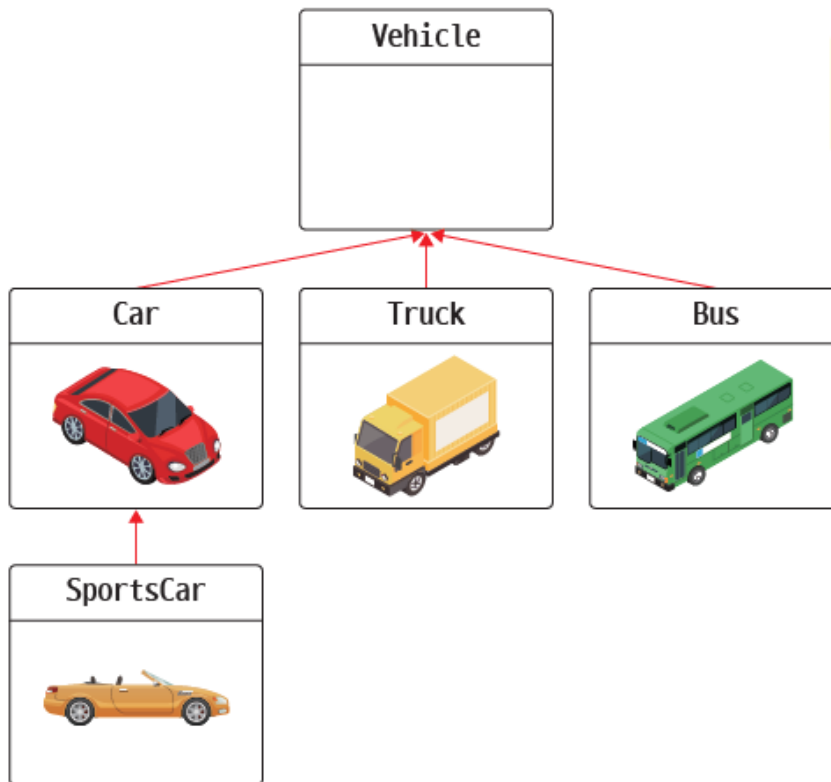
부모의 속성



상속을 이용하면 소프트웨어도
쉽게 개발할 수 있습니다.



상속의 예



상속에서는 자식 클래스에서
부모 클래스로 화살표를
그립니다.



상속과 is-a 관계

- 상속은 클래스 간의 “is-a” 관계를 생성하는데 사용된다.
 - ▣ 푸들은 강아지이다. // Poodle **is a** puppy.
 - ▣ 자동차는 차량이다.
 - ▣ 꽃은 식물이다.
 - ▣ 사각형은 모양이다.

부모 클래스	자식 클래스
Animal(동물)	Lion(사자), Dog(개), Cat(고양이)
Bike(자전거)	MountainBike(산악자전거), RoadBike, TandemBike
Vehicle(탈것)	Car(자동차), Bus(버스), Truck(트럭), Boat(보트), Motorcycle(오토바이), Bicycle(자전거)
Student(학생)	GraduateStudent(대학원생), UnderGraduate(학부생)
Person(사람)	Student(학생), Employee(직원)
Shape(도형)	Rectangle(사각형), Triangle(삼각형), Circle(원)

12.3 상속 구현하기

Syntax: 상속 정의

자식 클래스 또는 서브 클래스라고 한다.

Syntax

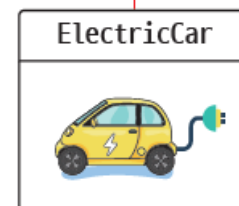
```
class 자식클래스(부모클래스) :  
    def 메소드1(self, ...) :  
        ...  
    def 메소드2(self, ...) :  
        ...
```

부모 클래스 또는 슈퍼 클래스라고 한다.

```
class ElectricCar(Car) :  
    def setBatterySize(self, size):  
        ...  
    def getBatterySize(self):  
        ...
```



부모클래스



자식클래스

예제

일반적인 자동차를 나타내는 클래스이다.

```
class Car:
```

```
    def __init__(self, make, model, color, price):
        self.make = make                # 메이커
        self.model = model              # 모델
        self.color = color              # 자동차의 색상
        self.price = price              # 자동차의 가격
```

```
    def setMake(self, make):            # 설정자 메소드
        self.make = make
```

```
    def getMake(self):                  # 접근자 메소드
        return self.make
```

차량에 대한 정보를 문자열로 요약하여서 반환한다.

```
    def getDesc(self):
        return "차량=(" + str(self.make) + ", " + \
            str(self.model) + ", " + \
            str(self.color) + ", " + \
            str(self.price) + ")"
```

예제

```
class Car:
    def __init__(self, make, model, color, price):
    def setMake(self, make):
    def getMake(self):
    def getDesc(self):
```

```
class ElectricCar(Car) :                                # ①
    def __init__(self, make, model, color, price, batterySize):
        super().__init__(make, model, color, price)      # ②
        self.batterySize=batterySize                    # ③

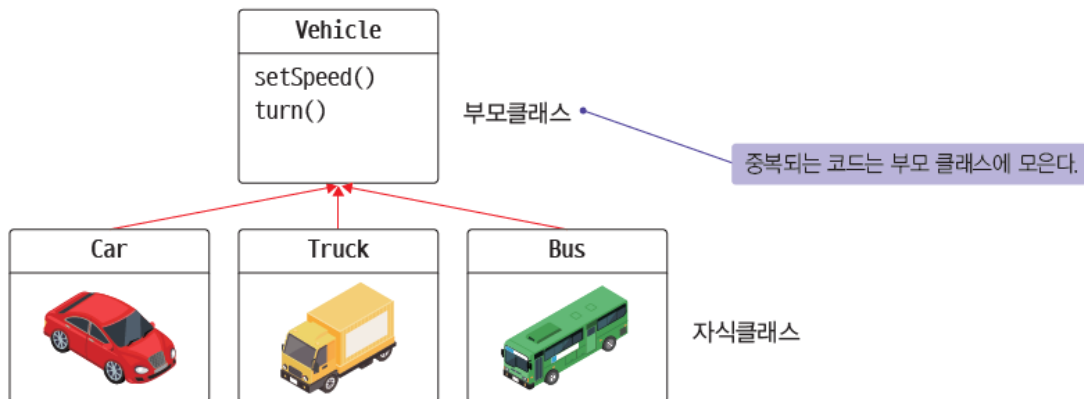
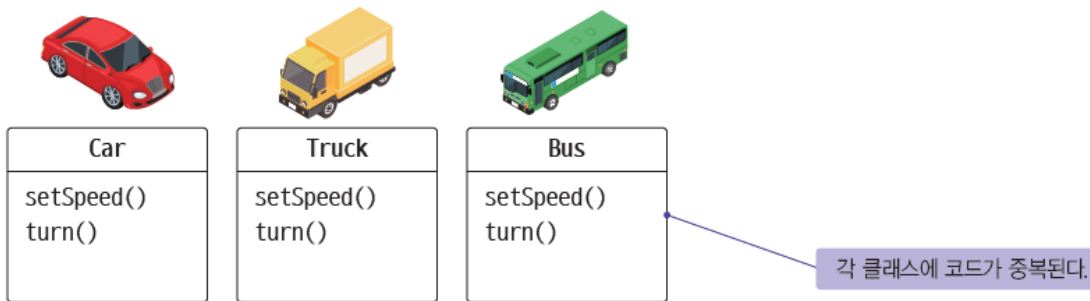
    def setBatterySize(self, batterySize):               # 설정자 메소드
        self.batterySize=batterySize

    def getBtterySize(self):                             # 접근자 메소드
        return self.batterySize

myCar = ElectricCar("Tisla", "Model S", "white", 10000, 0) #
myCar.setMake("Tesla")                                   # 설정자 메소드 호출
myCar.setBatterSize(60)                                  # 설정자 메소드 호출
print(myCar.getDesc())                                   # 전기차 객체를 문자열로 출력
```

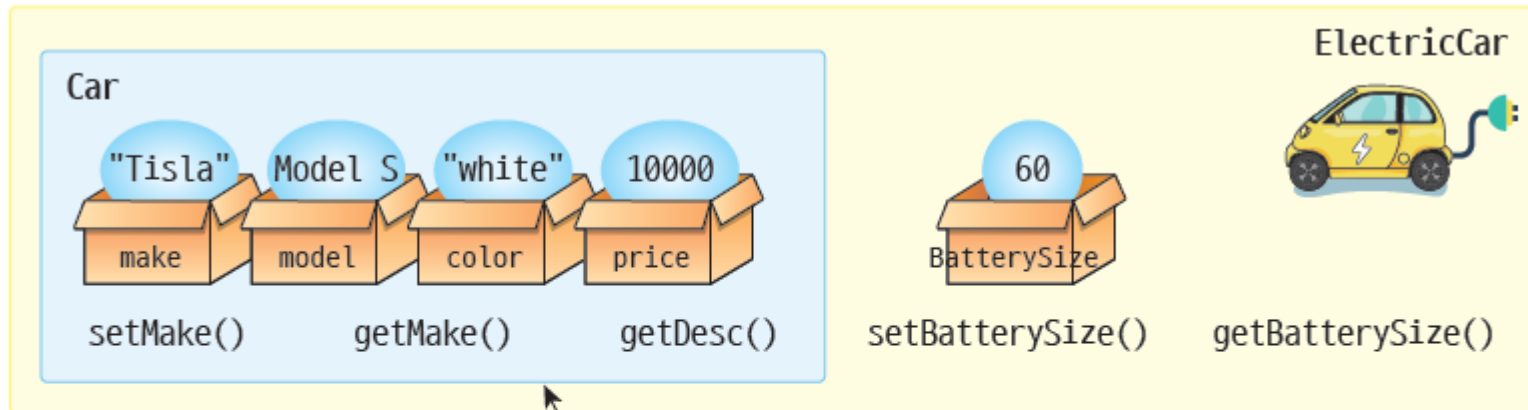
```
차량 =(Tesla,Model S,white,10000)
```

왜 상속을 사용하는가?



부모 클래스의 생성자 호출

```
class ElectricCar(Car):  
    def __init__(self, make, model, color, price, batterySize):  
        super().__init__(make, model, color, price)  
        self.batterySize=batterySize
```



부모 클래스의 변수는
누가 초기화 하나요?

생성자를 호출하지 않으면 오류

```
class Animal:
    def __init__(self, age=0):
        self.age=age

    def eat(self):
        print("동물이 먹고 있습니다. ")

# 부모 클래스의 생성자를 호출하지 않았다!
class Dog(Animal):
    def __init__(self, age=0, name=""):
        self.name=name

# 부모 클래스의 생성자가 호출되지 않아서 age 변수가 생성되지 않았다.

d = Dog();
print(d.age)
```

type()과 isinstance() 함수

```
...  
x = Animal();  
y = Dog();  
print(type(x), type(y))  
print(isinstance(x, Animal), isinstance(y, Animal))
```

```
<class '__main__.Animal'> <class '__main__.Dog'>  
True True
```

부모 클래스의 private 멤버

```
class Parent(object):
    def __init__(self):
        self.__money__ = 10 # __로 시작, __로 끝남
        self.__money = 100 # private: __로 시작, __로 끝나지 않음

class Child(Parent):
    def __init__(self):
        super().__init__()

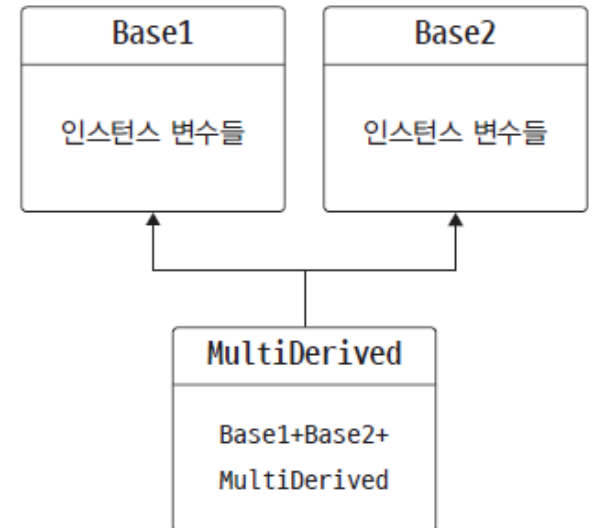
obj = Child()
print(obj.__money__)
print(obj.__money) # 오류
```

10

AttributeError: 'Child' object has no attribute '__money'

다중 상속

```
class Base1:  
    pass  
  
class Base2:  
    pass  
  
class MultiDerived(Base1, Base2):  
    pass
```



다중 상속의 예

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

    def show(self):
        print(self.name, self.age)

class Student:
    def __init__(self, id):
        self.id = id

    def getId(self):
        return self.id

class CollegeStudent(Person, Student):
    def __init__(self, name, age, id):
        Person.__init__(self, name, age)
        Student.__init__(self, id)

obj = CollegeStudent('Kim', 22, '100036')
obj.show()
print(obj.getId())
```

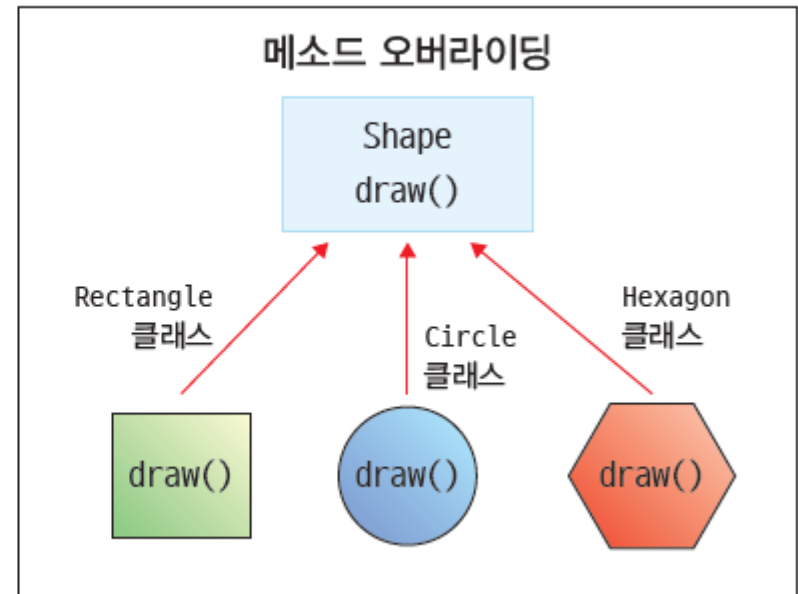
Kim 22
100036

12.4 메소드 오버라이딩 (Method overriding)

- 자식 클래스는 부모 클래스의 모든 메소드를 상속 받는다. 필요에 따라 자식 클래스에서 부모 클래스의 메소드를 다시 정의 가능하다.
- “자식 클래스의 메소드가 부모 클래스의 메소드를 오버라이드(재정의)한다”고 말한다.



메소드 오버라이딩은 부모 클래스의 메소드는 자식 클래스가 자신의 필요에 맞추어서 변경하는 것입니다.



예제: 메소드 오버라이딩

```
import math

class Shape:
    def __init__(self):
        pass

    def draw(self):
        print("draw()가 호출됨")

    def get_area(self):
        print("get_area()가 호출됨")

class Circle(Shape):
    def __init__(self, radius=0):
        super().__init__()
        self.radius = radius

    def draw(self):
        print("원을 그립니다.")

    def get_area(self):
        return math.pi * self.radius ** 2

c = Circle(10)
c.draw()
print("원의 면적:", c.get_area())
```

원을 그립니다.
원의 면적: 314.1592653589793

예제: 부모 클래스의 메서드 호출

```
import math
class Shape:
    def __init__(self):
        pass

    def draw(self):
        print("draw()가 호출됨")

    def get_area(self):
        print("get_area()가 호출됨")

class Circle(Shape):
    def __init__(self, radius=0):
        super().__init__()
        self.radius = radius

    def draw(self):
        super().draw()
        print("원을 그립니다.")

    def get_area(self):
        return math.pi * self.radius ** 2

c = Circle(10)
c.draw()
```

draw()가 호출됨
원을 그립니다.

Lab: 직원과 매니저

- 회사에 직원(Employee)과 매니저(Manager)가 있고, 직원은 월급만 있지만 매니저는 월급 외에 보너스가 있다고 하자.

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
    def getSalary(self):
        return self.salary
    def __str__(self):
        return f"이름: {self.name}, 월급: {self.getSalary()}"

class Manager(Employee):
    def __init__(self, name, salary, bonus):
        super().__init__(name, salary)
        self.bonus = bonus
    def getSalary(self):
        return super().getSalary() + self.bonus

kim = Manager("김철수", 2000000, 1000000)
print(kim)
```

이름: 김철수, 월급: 3000000

Lab: 은행 계좌

- BankAccount 클래스는 다음 인스턴스 변수와 메소드를 가진다.
 - ▣ balance – 잔액(정수형)
 - ▣ name – 소유자의 이름(문자열)
 - ▣ number – 통장 번호(정수형)
 - ▣ withdraw() – 출금 메소드
 - ▣ deposit() – 입금 메소드
- SavingsAccount 클래스 (저축예금)는 BankAccount 클래스를 상속 받으며, 추가로 다음 인스턴스 변수와 메소드를 가진다.
 - ▣ interest_rate – 이자율(실수형)
 - ▣ add_interest() - 호출될 때마다 예금에 이자를 더하는 메소드
- CheckingAccount 클래스 (당좌예금)는 BankAccount 클래스를 상속 받으며, 추가로 다음 인스턴스 변수와 메소드를 가진다.
 - ▣ withdraw_charge – 수표를 1회 발행할 때 수수료(정수형)
 - ▣ withdraw() - 찾을 금액에 수수료를 더해서 출금한다.

Solution

```
class BankAccount:  
    def __init__(self, name, number, balance):
```

```
        self.balance = balance  
        self.name = name  
        self.number = number
```

```
    def withdraw(self, amount):  
        self.balance -= amount  
        return self.balance
```

```
    def deposit(self, amount):  
        self.balance += amount  
        return self.balance
```

```
class CheckingAccount(BankAccount) :  
    def __init__(self, name, number, balance):  
        super().__init__(name, number, balance)  
        self.withdraw_charge = 10000      # 수표 발행 수수료
```

```
    def withdraw(self, amount):  
        return BankAccount.withdraw(self, amount + self.withdraw_charge)
```

BankAccount 클래스:

balance – 잔액(정수형)

name – 소유자의 이름(문자열)

number – 통장 번호(정수형)

withdraw() – 출금 메소드

deposit() – 입금 메소드

Solution

```
class SavingsAccount(BankAccount) :
    def __init__(self, name, number, balance, interest_rate):
        super().__init__(name, number, balance)
        self.interest_rate = interest_rate

    def set_interest_rate(self, interest_rate):
        self.interest_rate = interest_rate

    def get_interest_rate(self):
        return self.interest_rate

    def add_interest(self):                # 예금에 이자를 더한다.
        self.balance += self.balance*self.interest_rate

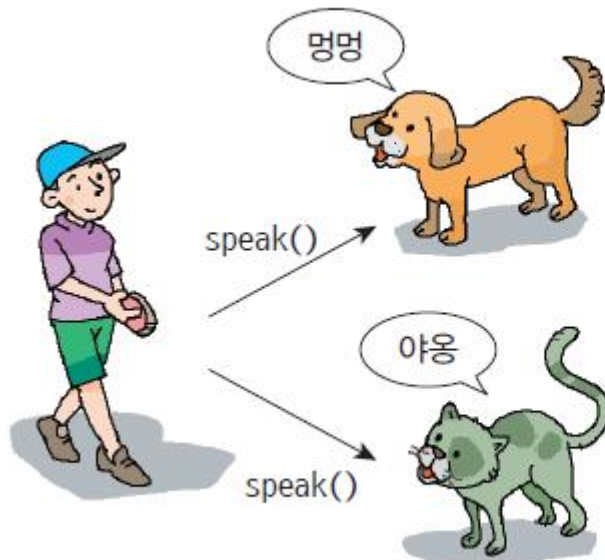
a1 = SavingsAccount("홍길동", 123456, 10000, 0.05)
a1.add_interest()
print("저축예금의 잔액=", a1.balance)

a2 = CheckingAccount("김철수", 123457, 2000000)
a2.withdraw(100000)
print("당좌예금의 잔액=", a2.balance)
```

저축예금의 잔액= 10500.0
당좌예금의 잔액= 1890000

12.5 다형성

- 다형성(polymorphism)은 “많은(poly)+모양(morph)”이라는 의미로서 주로 프로그래밍 언어에서 하나의 식별자로 다양한 타입(클래스)을 처리하는 것을 의미한다.

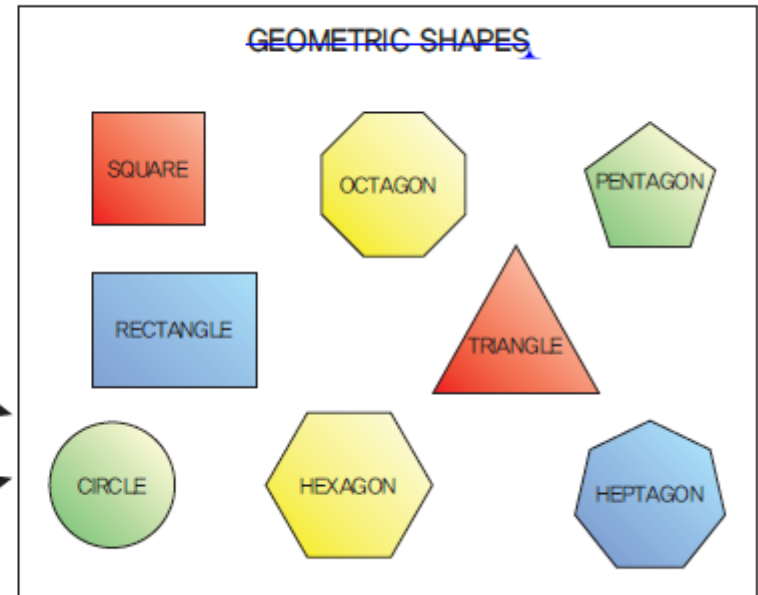
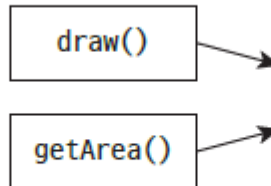


다형성은 동일한 코드로 다양한 타입의 객체를 처리할 수 있는 기법입니다.



다형성의 예

도형의 타입에 상관없이 도형을 그리려면
무조건 `draw()`를 호출하고 도형의 면적을
계산하려면 무조건 `getArea()`를 호출하면
됩니다.



상속과 다형성

```
class Shape:
    def __init__(self, name):
        self.name = name
    def getArea(self):
        raise NotImplementedError("이것은 추상메소드입니다. ")

class Circle(Shape):
    def __init__(self, name, radius):
        super().__init__(name)
        self.radius = radius

    def getArea(self):
        return 3.141592*self.radius**2

class Rectangle(Shape):
    def __init__(self, name, width, height):
        super().__init__(name)
        self.width = width
        self.height = height

    def getArea(self):
        return self.width*self.height

for s in [ Circle("c1", 10), Rectangle("r1", 10, 10) ]:
    print(s.getArea())
```

314.1592
100

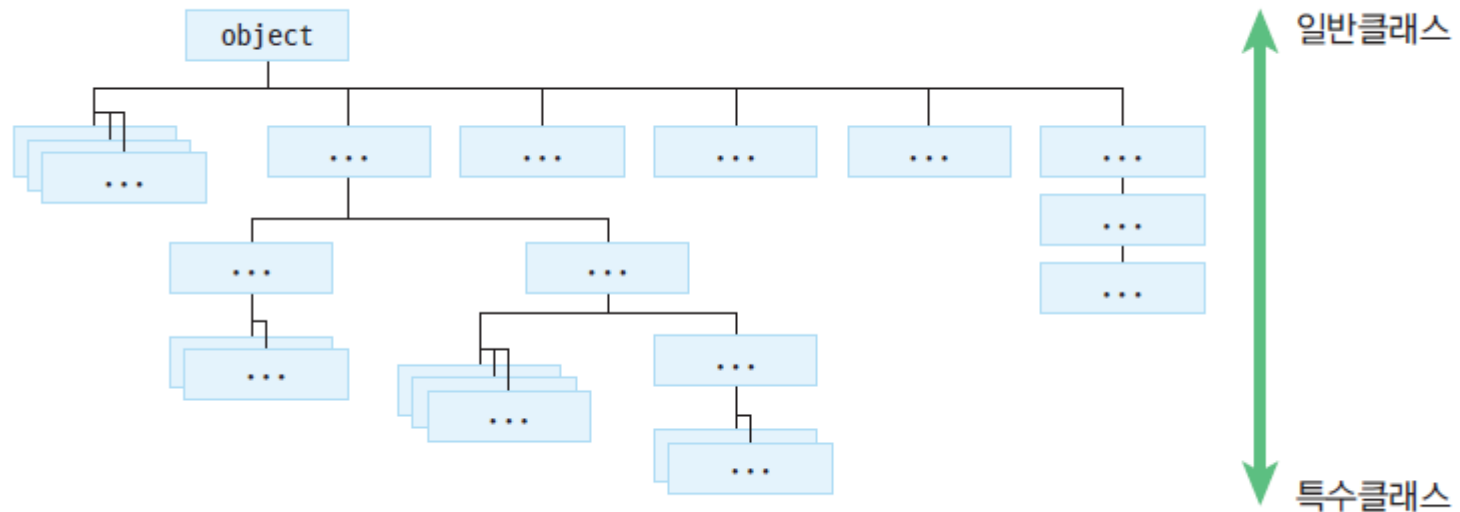
내장 함수와 다형성

```
mylist = [1, 2, 3]          # 리스트  
print("리스트의 길이=", len(mylist))  
  
s = "This is a sentence"    # 문자열  
print("문자열의 길이=", len(s))  
  
d = {'aaa': 1, 'bbb': 2}    # 딕셔너리  
print("딕셔너리의 길이=", len(d))
```

```
리스트의 길이= 3  
문자열의 길이= 18  
딕셔너리의 길이= 2
```

12.6 object 클래스

- 모든 클래스의 맨 위에는 object 클래스가 있다고 생각하면 된다.



object 클래스의 메소드

메소드	
<code>__init__ (self [,args...])</code>	생성자 예 <code>obj = className(args)</code>
<code>__del__(self)</code>	소멸자 예 <code>del obj</code>
<code>__repr__(self)</code>	객체 표현 문자열 반환 예 <code>repr(obj)</code>
<code>__str__(self)</code>	문자열 표현 반환 예 <code>str(obj)</code>
<code>__cmp__ (self, x)</code>	객체 비교 예 <code>cmp(obj, x)</code>

`__repr__()` 메소드

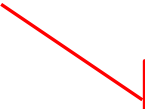
```
class Book(object):  
    def __init__(self, title, isbn):  
        self.__title = title  
        self.__isbn = isbn  
    def __repr__(self):  
        return "ISBN: " + self.__isbn + "; TITLE: " + self.__title  
  
book = Book("The Python Tutorial", "0123456")  
print(book)
```

ISBN: 0123456; TITLE: The Python Tutorial

__str__() 메소드

```
class MyTime:
    def __init__(self, hour, minute, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
    def __str__(self):
        return '%02d:%02d:%02d' % (self.hour, self.minute, self.second)

time = MyTime(10, 5)
print(time)
```



string modulo operator

10:05:00

12.7 상속과 구성

- is-a 관계: 상속
 - ▣ 승용차는 차량의 일종이다(Car is a Vehicle).
 - ▣ 강아지는 동물의 일종이다(Dog is an animal).
 - ▣ 원은 도형의 일종이다(Circle is a shape)
- has-a 관계: 구성
 - ▣ 도서관은 책을 가지고 있다(Library has a book).
 - ▣ 거실은 소파를 가지고 있다(Living room has a sofa).

예

```
class Animal:
    pass

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

class Person:
    def __init__(self, name):
        self.name = name
        self.pet = None

dog1 = Dog("dog1")
person1 = Person("홍길동")
person1.pet = dog1
```

Lab: Card와 Deck

- 카드를 나타내는 Card 클래스를 작성하고,
52개의 Card 객체를 가지고 있는 Deck 클래스를 작성한다.
각 클래스의 `__str__()` 메소드로 덱 안에 들어 있는 카드를 출력한다.

```
['♣A', '♣2', '♣3', '♣4', '♣5', '♣6', '♣7', '♣8', '♣9', '♣10', '♣J', '♣Q', '♣K',  
'♦A', '♦2', '♦3', '♦4', '♦5', '♦6', '♦7', '♦8', '♦9', '♦10', '♦J', '♦Q',  
'♦K', '♥A', '♥2', '♥3', '♥4', '♥5', '♥6', '♥7', '♥8', '♥9', '♥10', '♥J', '♥Q',  
'♥K', '♠A', '♠2', '♠3', '♠4', '♠5', '♠6', '♠7', '♠8', '♠9', '♠10', '♠J', '♠Q',  
'♠K']
```


Solution

```
class Card:
```

```
suitNames = ['♣', '♦', '♥', '♠']
```

```
rankNames = [None, 'A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
```

```
def __init__(self, suit, rank):
```

```
self.suit = suit
```

```
self.rank = rank
```

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

```
return Card.suitNames[self.suit]+ Card.rankNames[self.rank]
```

```
class Deck:
```

```
def __init__(self):
```

```
self.cards = [Card(suit, rank) for suit in range(4) for rank in range(1, 14)]
```

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

```
lst = [str(card) for card in self.cards]
```

```
return str(lst)
```

```
deck = Deck()
```

```
print(deck)
```

#덱 객체를 출력한다. str()이 호출된다.

Lab: 학생과 강사

- 일반적인 사람을 나타내는 Person 클래스를 정의한다.
Person 클래스를 상속받아서 학생을 나타내는 클래스 Student와
선생님을 나타내는 클래스 Teacher를 정의한다.

```
이름=홍길동  
주민번호=12345678  
수강과목=['자료구조']  
평점=0
```

```
이름=김철수  
주민번호=123456790  
강의과목=['Python']  
월급=3000000
```

Solution

```
class Person:
    def __init__(self, name, number):
        self.name = name
        self.number = number
    def __str__(self):
        return "\n이름="+self.name + "\n주민번호="+self.number

class Student(Person):
    UNDERGRADUATE=0
    POSTGRADUATE = 1

    def __init__(self, name, number, studentType ):
        super().__init__(name, number)
        self.studentType = studentType
        self.gpa=0
        self.classes = []
    def enrollCourse(self, course):
        self.classes.append(course)
    def __str__(self):
        return f"{super().__str__()}\\n수강과목={self.classes}\\n평점={self.gpa}"

hong = Student("홍길동", "12345678", Student.UNDERGRADUATE )
hong.enrollCourse("자료구조")
print(hong)
```

이름=홍길동
주민번호=12345678
수강과목=['자료구조']
평점=0

Solution

```
class Person:
    def __init__(self, name, number):
        self.name = name
        self.number = number
    def __str__(self):
        return "\n이름="+self.name + "\n주민번호="+self.number
```

```
class Teacher(Person):
    def __init__(self, name, number):
        super().__init__(name, number)
        self.courses = []
        self.salary=3000000

    def assignTeaching(self, course):
        self.courses.append(course)

    def __str__(self):
        return f"{super().__str__()} \n강의과목={self.courses} \n월급={self.salary}"

kim = Teacher("김철수", "123456790")
kim.assignTeaching("Python")
print(kim)
```

```
이름=김철수
주민번호=123456790
강의과목=['Python']
월급=3000000
```