데이터 사이언스와 파이썬

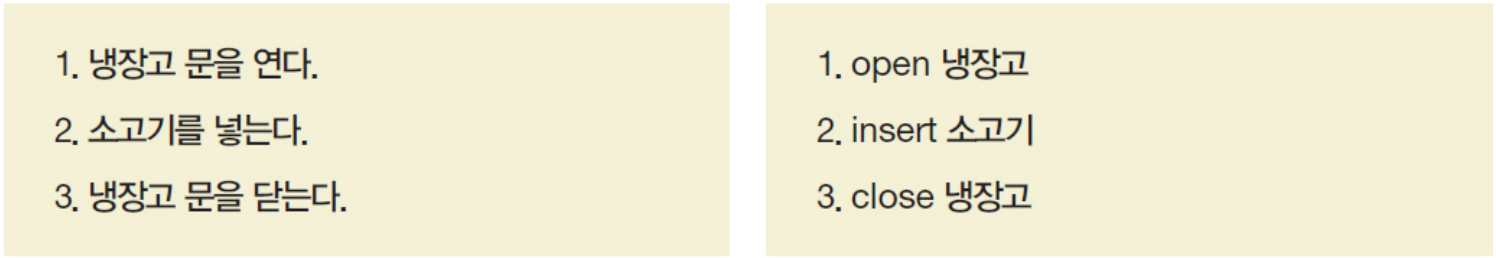
객체와 클래스

2020년 2학기
데이터사이언스융합전공

남세진
jordse@gmail.com

절차 지향 언어

- 절차 지향 언어(procedure oriented language)의 개념
  - 1990년 이전에 나온 대부분의 프로그래밍 언어는 절차 지향 언어
  - 프로그램 코드를 순서대로 작성하여 실행하는 언어
  - 파스칼, 코볼, 포트란, 베이직, C언어 등



(a) 냉장고에 소고기를 넣는 과정 (b) 냉장고에 소고기를 넣는 프로그램

그림 4-18 절차 지향 언어의 프로그래밍 개념

https://www.hanbit.co.kr/store/books/look.php?p_code=B5566366165

구조적 프로그래밍

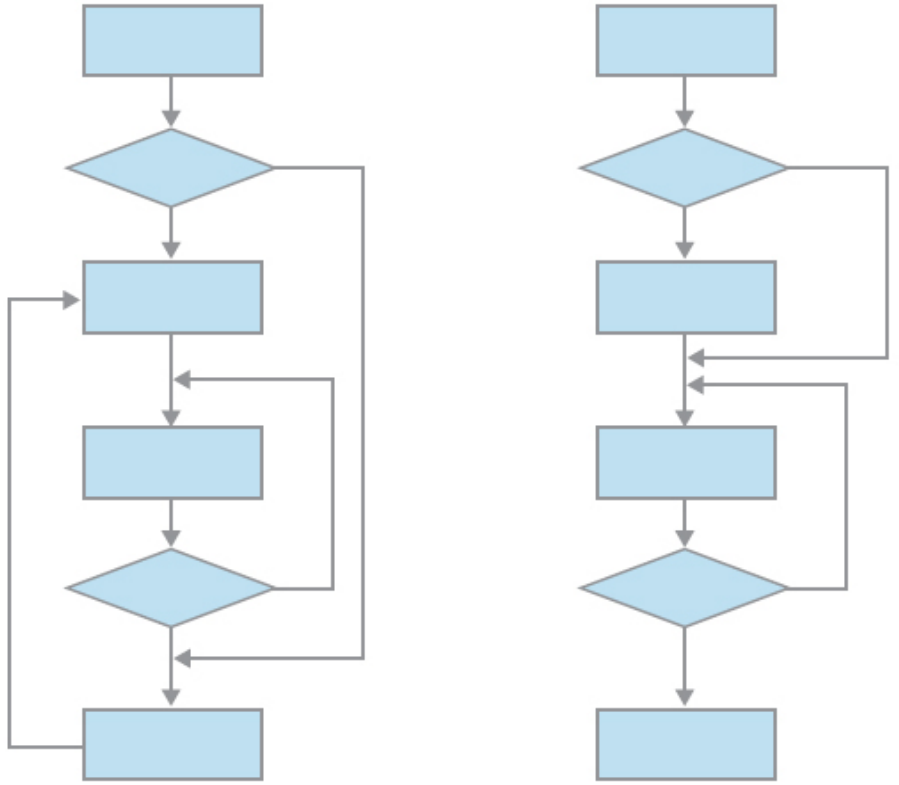
- 구조적 프로그래밍 등장 배경
  - 절차 지향 언어는 명령을 실행하는 순서가 자주 바뀌거나 복잡해지면 운영 및 유지 비용이 많은 발생
  - 명령어 실행 순서가 뒤바뀌거나 제어 구조가 복잡해지지 않도록 몇 가지 규칙에 따라 프로그램을 작성

  - 구조적 프로그래밍 특징
    - 프로그램을 읽고 이해하기 쉽다.
    - 프로그램의 개발 및 유지 보수의 효율성이 높다
    - 프로그래밍 규칙이 제공된다.
    - 프로그래밍에 대한 신회성이 높다.
    - 프로그래밍에 소요되는 시간과 노력이 감소된다.

  - 구조적 프로그래밍 규칙
    - 프로그램을 구성하는 각 요소를 작은 규모로 조직화한다.
    - 단일 입,출구 형태로 작성한다.
    - 가능하면 goto문을 쓰지 않는다.
    - 순차 구조, 선택 구조, 반복 구조만 쓴다.

구조적 프로그래밍

- 구조적 프로그래밍 등장 배경
  - goto문의 무분별한 분기 구조를 개선하고 모든 명령문

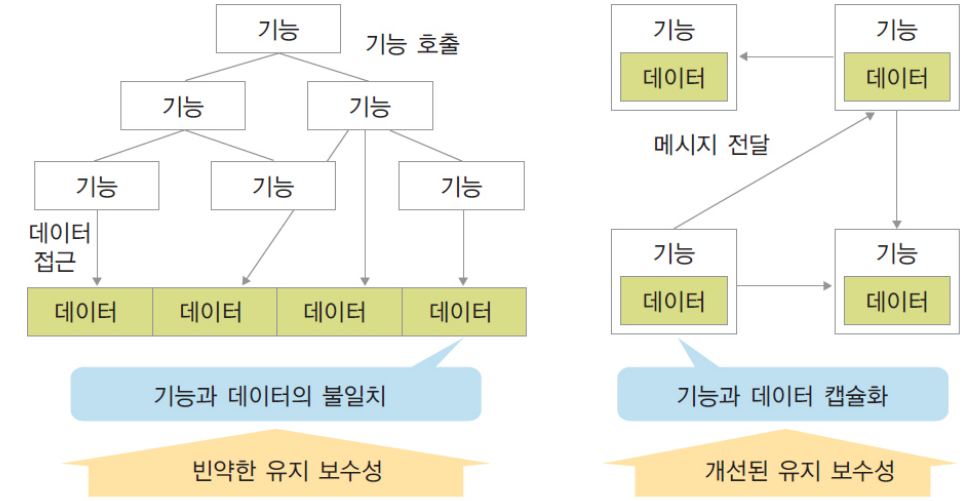


(a) 나쁜 프로그램 (b) 좋은 프로그램

그림 4-19 구조적 프로그래밍의 예

객체 지향 언어

- 객체 지향 언어
  - 객체 단위로 데이터와 가능을 하나로 묶어 쓰는 언어
  - 1990년대에 본격적으로 등장했고, 비주얼베이직, C++, 자바 등
  - 원하는 기능과 데이터를 따로 정의한 후, 필요할 때 마다 묶어 사용하기 때문에 프로그램의 운영 및 유지가 쉬움.



(a) 절차 지향 언어 (b) 객체 지향 언어

그림 4-20 절차 지향 언어와 객체 지향 언어의 차이점

- 절차 지향 언어: 데이터와 데이터를 처리하는 기능이 별도로 관리
- 객체 지향 언어: 데이터와 기능을 묶어 캡슐화시킨 후 메시지를 전달하여 일을 처리

객체 지향 언어의 주요 개념

- 클래스(class)
  - 다른 사물과 구분되는 속성을 가진 객체가 모여, 일반화된 범주로 묶인 것
  - 객체에 대한 일반화된 틀(template)을 제공
  - 예를 들어, 자동차 클래스가 있다면 이 클래스는 내 자동차, 홍길동 자동차, 김갑돌 자동차 등의 객체로 구성될 수 있음.

- 객체(object)
  - 개별적으로 식별되는 사물을 지칭하며, 속성과 기능을 가짐.

  - 속성(attribute): 각 객체가 가진 고유한 특징
  - 기능(function): 행동 패턴
  - 속성과 기능을 캡슐화 함
  - 예를 들어, '홍길동 차 ' 라는 객체는 색, 차종, 크기, 모양, 최고 속도 등의 속성을 가지고, 전진과 후진, 정지, 가속과 감속 등의 기능을 한다.

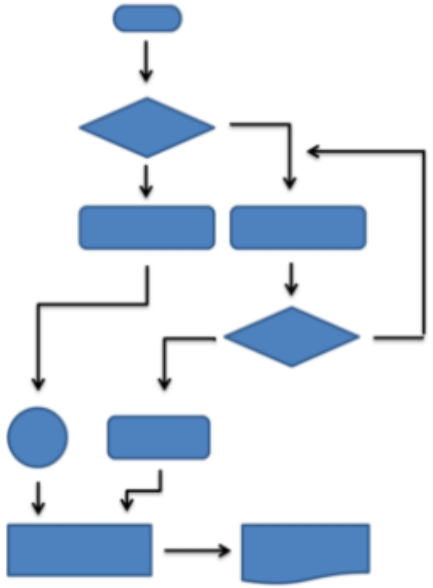
객체 지향 언어의 주요 개념

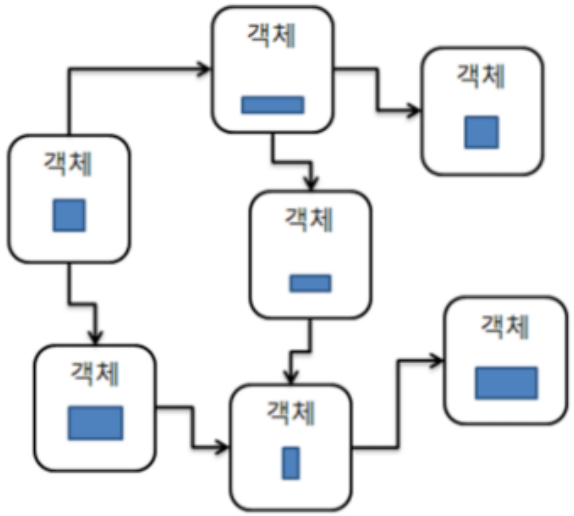
- 상속(inheritance)
  - 클래스는 필요할 경우 더 세분해서 각각의 특성 별로 관리할 수 있는데, 이것을 하위 클래스라도 함.
  - 하위 클래스는 상위 클래스가 가지는 속성과 기능을 모두 이어받을 수 있는데 이를 상속이라고 함
  - 상속은 객체 지향언어에서 재사용 수단으로 **다형성**과 밀접한 관계가 있음
  - 하위클래스는 상속을 통해 상위 클래스의 속성과 기능을 그대로 재사용

```
class Car():
    def hello(self):
        print('hello world')

class Avante(Car):
    pass

carA = Car()
carB = Avante()
carA.hello()
carB.hello()
```

객체 지향 언어의 주요 개념

- 메시지(message)
  - 객체 간에 전달되는 명령 단위
  - 객체는 자발적으로 행위를 수행하지 않으므로 객체가 특정 기능을 수행하게 하려면 메시지가 전달되어야 함
  - 자동차를 움직이려면 운전자가 가속 페달을 밟아 직진하라는 신호를 줘야 하는 것과 같은 맥락
  - 메시지가 다듬어져서 프로그램화되면 메소드(method)로 진화





4.2 객체 지향 언어의 주요 개념

- 추상화(abstraction)
  - 어떤 객체가 상대하는 다른 객체에 대해, 꼭 필요한 부분만 알고 나머지 세부적인 사항은 감추는 것
  - 공통의 속성이나 기능을 묶어 이름을 붙이는 것
  - 객체 지향적 관점에서 클래스를 정의하는 것

객체 지향 언어의 주요 개념

- 캡슐화(encapsulation)
  - 객체의 속성과 기능을 하나로 묶되, 추상화하여 객체의 세부 내용을 사용자가 보지 못하도록 은폐하는 것
  - 예를 들어, 사용자에게 자동차 엔진(객체)을 설명할 때 세부 동작원리를 설명하지 않고 어떤 기능을 하는지 정도만 알려주는 것과 같음.
  - 캡슐화는 제3자가 객체 내부 데이터와 기능을 변조하는 것을 막아주므로 프로그램의 재사용성과 유지보수성을 향상시킴

```
class IntArray
{
private:
   int m_array[10]; // user can not access this directly any more

public:
   void setValue(int index, int value)
   {
      // If the index is invalid, do nothing
      if (index < 0 || index >= 10)
         return;

      m_array[index] = value;
   }
};
```

https://boycoding.tistory.com/243

객체 지향 언어의 주요 개념

- 다형성(polymorphism)
  - 오버라이딩(Overriding) : 슈퍼 클래스를 상속받은 서브 클래스에서 슈퍼 클래스의 메서드를 같은 이름, 같은 반환 값, 같은 인자로 메소드 내의 로직들을 새롭게 정의하는 것
  - 오버로딩(Overloading) : 하나의 클래스에서 같은 이름의 메서드(단, 매서드 파라미터는 달라야 함)들을 여러 개 가질 수 있도록 함

```java
class People{

    public void printInfo() {
        System.out.println("나는 사람입니다.");
    }
}
```

```java
class Man extends People{
    @Override
    public void printInfo() {
        super.printInfo();
        System.out.println("그리고 나는 남자입니다.");
    }

}
class Woman extends People{
    @Override
    public void printInfo() {
        super.printInfo();
        System.out.println("그리고 나는 여자입니다.");
    }
}
```

객체 지향 언어의 주요 개념

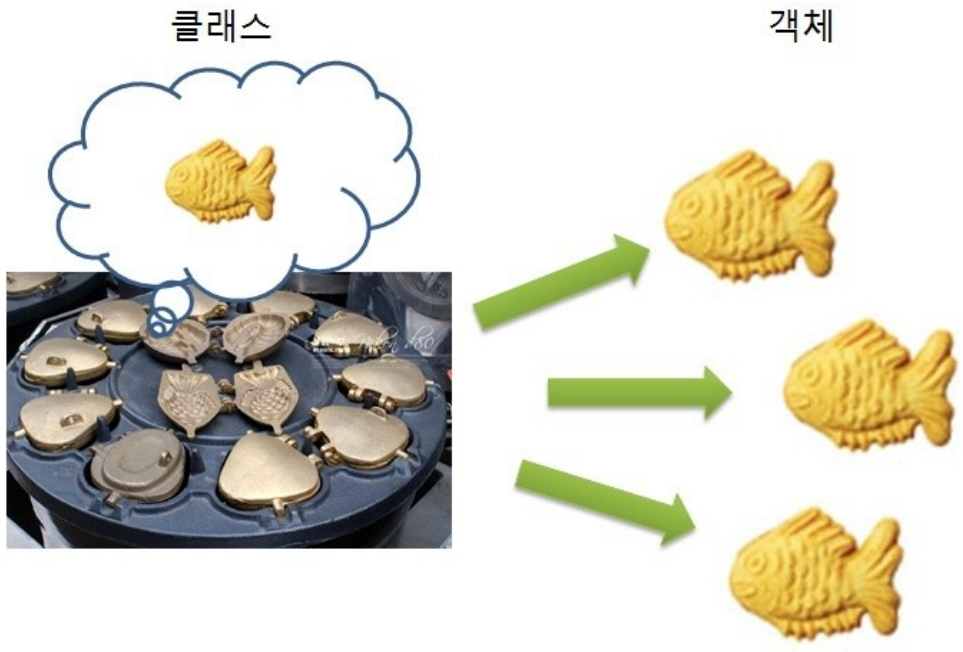
- 오버로딩
  - '동일한 이름'으로 다양한 매개변수와 다양한 리턴 타입의 여러 메소드를 정의하는 것
- Java 언어에서의 오버로딩
- Python에서는?

```java
class MyMath {

  int first, second, third;

  public void a(int first, int second){
     System.out.println("a(int first, int second)");
     this.first = first;
     this.second = second;
  }

  public void a(int first, int second, int third){
     System.out.println("a(int first, int second, int third)");
     this.first = first;
     this.second = second;
     this.third = third;
  }
```

객체와 클래스

- 객 체
  - Object = 데이터 + 관련 연산들(메쏘드 혹은 동작들)

- 클래스
  - Class = 객체의 정의 혹은 타입
  - (a type of Object)



https://cremazer.github.io/java-Class-and-Object/

클래스

- 객체에 대한 정의
  - 데이터(필드 변수들) 정의
  - 관련 연산들(메쏘드들) 정의

- 클래스는 객체에 대한 타입
  - 객체는 어떤 클래스 타입으로 선언
  - 객체는 어떤 클래스의 실체(instance)이다.

- 실체화(Instantiation)
  - 클래스로부터 객체를 생성하는 것

클래스의 예

- Java의 클래스
  - 필드 변수 정의
  - 메쏘드 정의

- C++의 클래스
  - 데이터 멤버 정의
  - 멤버 함수 정의

Java 클래스 정의

- 클래스 정의 구문

```
class class-name {
    필드 변수 선언
    구성자 선언
    메서드 선언
}
```

변수, 구성자, 메서드를 클래스의 멤버라고 부름

```
class Account {
    int account_number;
    double balance;

    Account (int account, double initial) {
        account_number = account;
        balance = initial;
    } // constructor Account

    void deposit (double amount) {
        balance = balance + amount;
    } // method deposit
} // class Account

Account saving = new Account ();
```

객체 생성

- **new 연산자는 클래스로부터 객체를 생성한다.**

```
Account saving = new Account ();
```

- **saving는 Account 객체를 참조하는 변수**
  - **new** 연산자에 의해 생성된 객체로 초기화 됨.

C++ 클래스

- **C++의 Class**
  - 데이터 멤버와 멤버 함수들의 모음에 대한 타입
  - 클래스는 변수를 선언하고 객체를 생성하는데 사용.

C++ 객체

- 클래스의 실체(instance)

- s는 Stack의 객체를 위한 변수

  Stack s;

```
class Stack {
public:
        char pop();
        void push(char);
        Stack() { top = 0;}
private:
        int top;
        char elements[101];
};
```

Python 객체와 클래스

객체란 무엇인가?

- 숫자에서 모듈까지 파이썬의 모든 것은 객체임.
  - 파이썬은 특수 구문을 이용하여 대부분의 객체를 숨김.
  - 객체는 데이터(변수, 속성attribute이라고 부름)와 코드(함수, 메서드method라고 부름)를 모두 포함함.
  - 객체는 어떤 구체적인 것의 유일한 인스턴스를 나타냄.
  - 모듈과 달리, 객체는 각자 다른 값을 가진 속성의 객체를 동시에 여러 개 생성할 수 있음.
  - 객체는 마치 코드를 덧붙인 슈퍼 자료구조와 같음.

클래스

- 사물의 속성은 변수로, 동작은 함수로 표현

- **멤버**
  - 클래스 구성하는 변수와 함수

- **메서드**
  - 클래스에 소속된 함수

클래스 선언하기

- 생성자
  - 클래스 선언 형식
  - __init__ 생성자
    - 통상 객체 초기화
  - self
    - 객체 자신을 나타냄

```
class 이름:
    def __init__(self, 초기값):
        멤버 초기화
    메서드 정의
```

```
class Person:
    def __init__(self):
        print('hello')

nam = Person()
```

```
hello
```

```
class Person:
    def __init__(self, name):
        print('hello', name)

nam = Person('name')
```

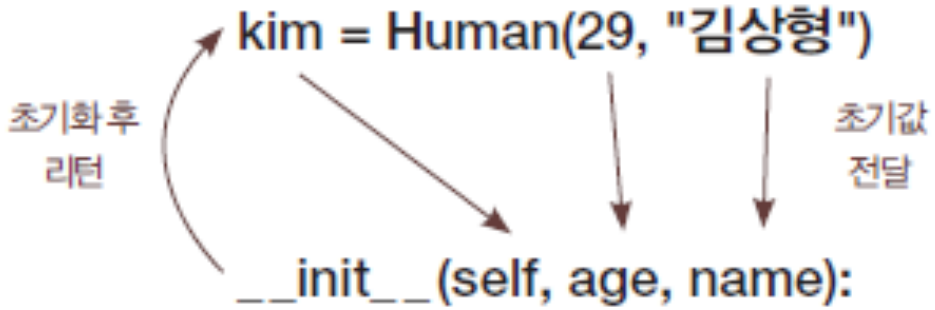
```
hello name
```

클래스와 객체

◦ 객체 생성 구문

객체 = 클래스명(인수)

- 객체를 __init__의 첫 번째 인수 self로 전달
- 생성문에서 전달한 인수를 두 번째 이후의 인수로 전달
- 새로 생성되는 객체 멤버에 대입



클래스와 객체

- 메서드는 필요한 만큼 선언할 수 있음
  - 객체.메서드()

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

    def speak(self):
        print('Ah~~')

    def get_name(self):
        return self.name

    def get_age(self):
        return self.age

nam = Person('nam', 20)
print(nam.get_name())
print(nam.get_age())
print(nam.name)
print(nam.age)
```

```
nam
20
nam
20
```

생성자(초기화 함수)의 파라미터에 성별을 위한 sex를 추가하고, 값을 가져오기 위한 메서드를 추가하시오

상속

- 상속을 이용하면 새로운 클래스는 기존 클래스를 복사하지 않고, 기존 클래스의 모든 코드를 쓸 수 있음
  - 필요한 것만 추가/변경하여 새 클래스를 정의함.
  - 기존 클래스의 행동을 재정의(오버라이드) 함.
  - 기존 클래스는 부모 클래스, 슈퍼 클래스, 베이스 클래스라고 부름.
  - 새 클래스는 자식 클래스, 서브 클래스, 파생된 클래스라고 부름.
  - 이 용어들은 객체 지향 프로그래밍에서 다르게 사용될 수 있음
  - 클래스 정의시 이름 다음의 괄호 안에 부모 클래스 이름 지정

```
class 이름(부모):
    ....
```

```
class Car():
    def hello(self):
        print('hello world')

class Avante(Car):
    pass

carA = Car()
carB = Avante()
carA.hello()
carB.hello()
```

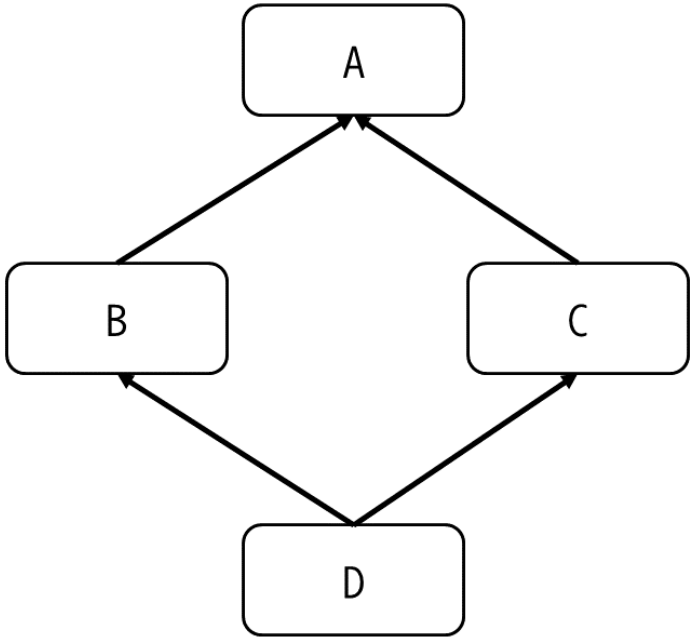
다중 상속

- Python에서는 여러 개의 부모 클래스로부터 상속(다중 상속)을 받을 수 있음
  - 클래스를 정의할 때 상속받을 클래스명들을 괄호 안에 쉼표로 구분하여 나열

```python
class Person:
    def hello(self):
        print('hello Person')


class Employee:
    def work(self):
        print('hello Employee')


class TeamManager(Person, Employee):
    def manage(self):
        print('hello Team Manager')


nam = TeamManager()
nam.hello()
nam.work()
nam.manage()
```

```
hello Person
hello Employee
hello Team Manager
```

다중상속시 메서드를 찾는 순서




```python
class Person:
    def hello(self):
        print('hello Person')


class Employee(Person):
    def hello(self):
        print('hello Employee')

class Manager(Person):
    def hello(self):
        print('hello Manager')

class TeamManager(Employee, Person):
    pass

nam = TeamManager()
nam.hello()
```

다중상속시 메서드를 찾는 순서

```python
class Person:
    def hello(self):
        print('hello Person')


class Employee(Person):
    def hello(self):
        print('hello Employee')

class Manager(Person):
    def hello(self):
        print('hello Manager')

class TeamManager(Employee, Person):
    pass

nam = TeamManager()
nam.hello()
print(TeamManager.mro())
```

- 메서드 탐색 순서(Method Resolution Order, MRO)에 따라 메서드를 결정
  - 다중 상속을 한다면 class TeamManager(Employee, Person):의 클래스 목록 중 왼쪽에서 오른쪽 순서로 메서드를 찾음

```
hello Employee
[<class '__main__.TeamManager'>, <class '__main__.Employee'>, <class '__main__.Person'>, <class 'object'>]
```

메서드 오버라이드

- 부모 클래스에서 정의한 함수를 오버라이드(재정의)할 수 있음
  - __init__ 함수를 포함한 모든 메서드를 오버라이드 할 수 있음

```python
class Person:
    def hello(self):
        print('hello Person')


class Employee(Person):
    def hello(self):
        print('hello Employee')

class Manager(Person):
    def hello(self):
        print('hello Manager')
```

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

class Employee(Person):
    def __init__(self, name):
        self.name = 'employee ' + name

class Manager(Person):
    def __init__(self, name):
        self.name = 'manager ' + name

person = Person('nam')
employee = Employee('nam')
manager = Manager('nam')

print(person.name)
print(employee.name)
print(manager.name)
```

super

- 자식 클래스에서 부모 클래스의 메서드를 호출하고 싶다면 super( ) 메서드를 사용하면 됨.

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

    def about(self):
        print('name', self.name)

class Employee(Person):
    def __init__(self, name, age, empID):
        # Person.__init__(self,name, age)
        super().__init__(name,age)
        self.empID = empID

    def get_empID(self):
        return self.empIDs

    def about(self):
        super().about()
        print('empID', self.empID)

nam = Employee('nam', 20, 1234)
nam.about()
```

만약 Employee의 초기화 함수를 아래와 같이 고치면?

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

self

- 메서드의 첫번째 인자 self는 인스턴스 자체임
  - 파이썬은 적절한 객체의 속성과 메서드를 찾기 위해 self 인자를 사용함

```python
class Foo:
    def func1(self):
        print("function 2")

foo = Foo()
foo.func1()
```

```python
class Foo:
    def func1(self):
        print("function 2")

foo = Foo()
# foo.func1()
Foo.func1(foo)
```

```python
class Foo:
    def func1():
        print("function 1")

    def func2(self):
        print("function 2")

foo = Foo()
foo.func1() # == Foo.func1(foo)
```

```
Traceback (most recent call last):
  File "/Users/rtdatum/PycharmProjects/lecture6/code7.py", line 9, in <module>
    foo.func1()
TypeError: func1() takes 0 positional arguments but 1 was given
```

self

```
class Foo:
    def func1():
        print("function 1")

    def func2(self):
        print("function 2")

# foo = Foo()
# foo.func1() # == Foo.func1(foo)
Foo.func1()
```

```
function 1
```

```
class Foo:
    def func1(self):
        print("function 2")

foo = Foo()
foo.func1()
```

```
class Foo:
    def func1(self):
        print("function 2")

foo = Foo()
# foo.func1()
Foo.func1(foo)
```

- 인스턴스 foo를 통해 func1를 호출하는 것과 클래스 이름 Foo을 통해 func1를 호출하는 것은 동일한 의미
- 인스턴스.메서드()'와 같은 방식을 주로 사용

인스턴스.메서드() VS 클래스.메서드(인스턴스)

get / set 속성값과 프로퍼티

- Python의 모든 속성과 메서드는 외부에서 접근 가능(public)
- 어떤 객체 지향 언어에서는 외부로부터 바로 접근할 수 없는 private 객체 속성을 지원함
  - 프로그래머는 private 속성의 값을 읽고 쓰기 위해 getter 메서드와 setter 메서드를 사용함.
- 파이썬에서는 getter나 setter 메서드가 필요 없음.
  - 파이썬은 모든 속성과 메서드는 public이고, 우리가 예상한대로 쉽게 동작하기 때문임.

```
class Duck:
    def __init__(self, input_name):
        self.hidden_name = input_name

    def get_name(self):
        print('inside the getter')
        return self.hidden_name

    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name


d = Duck('duck')
print(d.get_name())
```

```
inside the getter
duck
```

property

- property() 함수를 이용하여 클래스내에서 property를 정의할 수 있음
- property() 함수의 사용법

```
class Duck:
    def __init__(self, input_name):
        self.hidden_name = input_name

    def get_name(self):
        print('inside the getter')
        return self.hidden_name

    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name

    name = property(get_name, set_name)

d = Duck('duck')
print(d.name)
```

```
inside the getter
duck
```

```
class Duck:
    def __init__(self, input_name):
        self.hidden_name = input_name

    def get_name(self):
        print('inside the getter')
        return self.hidden_name

    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name

    name = property(get_name, set_name)

d = Duck('duck')
print(d.name)
d.name = 'dog'
print(d.name)
```

```
inside the getter
duck
inside the setter
inside the getter
dog
```

@property, @name.setter

- getter 메서드 앞에 @property 데커레이터를 사용
- setter 메서드 앞에 @name.setter 데커레이터를 사용

```
class Duck:
    def __init__(self, input_name):
        self.hidden_name = input_name

    @property
    def inputname(self):
        print('inside the getter')
        return self.hidden_name

    @inputname.setter
    def inputname(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name

d = Duck('duck')
print(d.inputname)
d.inputname = 'dog'
print(d.inputname)
```

```
inside the getter
duck
inside the setter
inside the getter
dog
```

```
class Duck:
    def __init__(self, input_name):
        self.hidden_name = input_name

    @property
    def inputname(self):
        print('inside the getter')
        return self.hidden_name

    @inputname.setter
    def inputname(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name

d = Duck('duck')
print(d.inputname)
d.inputname = 'dog'
print(d.inputname)
print(d.hidden_name)
```

@property의 사용 예

```python
class Number:
    def __init__(self, number):
        self.value = number

    @property
    def power(self):
        return self.value * self.value

num = Number(3)
print(num.value)
print(num.power)
```

```
3
9
```

```python
class Number:
    def __init__(self, number):
        self.value = number

    @property
    def power(self):
        return self.value * self.value

num = Number(3)
print(num.value)
print(num.power)
num.power = 3
```

```
Traceback (most recent call last):
  File "/Users/rtdatum/PycharmProjects/lecture6/code8.py", line 69, in <module>
    num.power = 3
AttributeError: can't set attribute
3
9
```

네임 맹글링(name mangling)

- 파이썬은 클래스 정의 외부에서 볼 수 없도록 하는 속성에 대한 네이밍 컨벤션이 있음.
  - 속성 이름 앞에 두 언더스코어(__)를 붙이면 됨
- 정보은닉(Information Hiding) 의도를 위해 사용

```
class Duck:
    def __init__(self, input_name):
        self.__name = input_name

    @property
    def name(self):
        print('inside the getter')
        return self.__name

    @name.setter
    def name(self, input_name):
        print('inside the setter')
        self.__name = input_name

duck = Duck('cat')
print(duck.name)
duck.name = 'dog'
```

```
inside the getter
cat
inside the setter
```

```
class Duck:
    def __init__(self, input_name):
        self.__name = input_name

    @property
    def name(self):
        print('inside the getter')
        return self.__name

    @name.setter
    def name(self, input_name):
        print('inside the setter')
        self.__name = input_name

duck = Duck('cat')
print(duck.name)
duck.name = 'dog'
print(duck.__name)
```

```
Traceback (most recent call last):
  File "/Users/rtdatum/PycharmProjects/lecture6/code8.py", line 88, in <module>
    print(duck.__name)
AttributeError: 'Duck' object has no attribute '__name'
```

네임 맹글링(name mangling)

- 속성 이름 앞에 두 언더스코어(__)를 붙이면 속성을 private로 만들지 않지만, 이 속성이 외부에서 발견할 수 없도록 이름을 맹글링(mangling)함

```python
from pprint import pprint

class Duck:
    def __init__(self, input_name):
        self.__name = input_name
        self.temp = ""

    @property
    def name(self):
        print('inside the getter')
        return self.__name

    @name.setter
    def name(self, input_name):
        print('inside the setter')
        self.__name = input_name


duck = Duck('cat')
pprint(dir(duck))
```

```
['_Duck__name',
 'temp',
 '__module__',
 '__init__',
 'name',
 '__dict__',
 '__weakref__',
 '__doc__',
```

dir 함수 - 클래스와 인스턴스 내부에서 사용할 수 있는 정보를 확인할 때 사용

다양한 메서드 타입

- 인스턴스 메서드(Instance Method)
  - 메서드 정의시 첫 번째 인자가 self인 메서드
  - 파이썬은 이 메서드를 호출할 때 객체를 전달
- 클래스 메서드(Class Method)
  - 특정 객체에 대한 작업 처리하는 것이 아니라 클래스 전체에 공유
    - @classmethod 데커레이터
    - 첫 번째 인수로 클래스에 해당하는 cls 인수
- 정적 메서드
  - 클래스에 포함되는 단순 유틸리티 메서드
  - 특정 객체에 소속되거나 클래스 관련 동작 하지 않음
  - @staticmethod 데커레이터

클래스 메서드 예제

```python
class A:
    count = 0

    def __init__(self):
        A.count += 1

    def echo(self):
        print('ya ho')

    @classmethod
    def kids(cls):
        print('A has ', cls.count, 'litte objects')
        print('A has ', A.count, 'litte objects')


var_1 = A()
var_2 = A()
var_3 = A()

A.kids()
```

```
A has  3 litte objects
A has  3 litte objects
```

Duck typing

- Duck Typing
  - 파이썬과 같은 동적타입의 언어에서 본질적으로 다른클래스라도 객체의 적합성은 객체의 실제 유형이 아니라 특정 메소드와 속성의 존재에 의해 결정
  - 'If it walks like a duck and it quacks like a duck, then it must be a duck'
  - '오리처럼 걷고, 오리처럼 꽥꽥거리면, 그것은 틀림없이 오리다.'

```
class Duck:
    def fly(self):
        print("Duck flying")

class Sparrow:
    def fly(self):
        print("Sparrow flying")

class Whale:
    def fly(self):
        print("Whale flying?")

for animal in Duck(), Sparrow(), Whale():
    animal.fly()
```

```
Duck flying
Sparrow flying
Whale flying?
```

특수 메서드

- 파이썬의 특수 메서드를 사용하면 연산자를 새롭게 정의해서 사용해서 사용할 수 있음

```
class Word:
    def __init__(self, txt):
        self.txt = txt;

    def eq(self, word2):
        return self.txt.lower() == word2.txt.lower()


first = Word('A')
second = Word('a')
third = Word('C')

print(first.eq(second))
print(first == second)
print(first.eq(third))
print(first == third)
```

```
True
False
False
False
```

```
class Word:
    def __init__(self, txt):
        self.txt = txt;

    def __eq__(self, other):
        return self.txt.lower() == other.txt.lower()

first = Word('A')
second = Word('a')
third = Word('C')

print(first == second)
print(first == third)
```

```
True
False
```

특수 메서드

__eq__(self, other)	self == other
__ne__(self, other)	self != other
__lt__(self, other)	self < other
__gt__(self, other)	self > other
__le__(self, other)	self <= other
__ge__(self, other)	self >= other

__add__(self, other)	self + other
__sub__(self, other)	self - other
__mul__(self, other)	self * other
__floordiv__(self, other)	self // other
__truediv__(self, other)	self / other
__mod__(self, other)	self % other
__pow__(self, other)	self ** other

__str__(self)	str(self)
__repr__(self)	repr(self)
__len__(self)	len(self)

```
class Word:
    def __init__(self, txt):
        self.txt = txt;

    def __eq__(self, other):
        return self.txt.lower() == other.txt.lower()

    def __mul__(self, other):
        return self.txt * other


first = Word('A')
print(first * 4)
```

```
AAAA
```

특수 메서드

- print 함수는 인자값을 모두 string으로 변환(str())하여 화면(sys.stdout)에 출력

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

특수 메서드

- __str__ 의 사용예

```
class Word:
    def __init__(self, txt):
        self.txt = txt;

    def __eq__(self, other):
        return self.txt.lower() == other.txt.lower()

    def __mul__(self, other):
        return self.txt * other

    def __str__(self):
        return 'value : ' + str(self.txt)


first = Word('A')
print(first)
```

```
<__main__.Word object at 0x7ff7ef992df0>
```

```
class Word:
    def __init__(self, txt):
        self.txt = txt;

    def __eq__(self, other):
        return self.txt.lower() == other.txt.lower()

    def __mul__(self, other):
        return self.txt * other

    def __str__(self):
        return 'value : ' + str(self.txt)


first = Word('A')
print(first)
```

```
value : A
```