Object-Oriented Programming

OOP

- OOP – Object Oriented Programming
  - ✓ Encapsulation
    - 자료 추상화는 불필요한 정보는 숨기고 중요한 정보만을 표현함으로써 프로그램을 간단히 만드는 것
    - 자료 추상화를 통해 정의된 자료형을 추상 자료형이라고 함
    - 추상 자료형은 자료형의 자료 표현과 자료형의 연산을 캡슐화한 것으로 접근 제어를 통해서 자료형의 정보를 은닉할 수 있음
    - 객체 지향 프로그래밍에서 일반적으로 추상 자료형을 클래스, 추상 자료형의 인스턴스를 객체, 추상 자료형에서 정의된 연산을 Method(함수), 인스턴스 생성을 위해 호출하는 Method가 생성자
  - ✓ Inheritance
    - 상속은 새로운 클래스가 기존의 클래스의 자료와 연산을 이용할 수 있게 하는 기능
    - 상속을 받는 새로운 클래스를 Sub클래스, Derived 클래스 라고 하며 새로운 클래스가 상속하는 기존의 클래스를 Super 클래스, Based 클래스 라고 하며 상속을 통해서 기존의 클래스를 상속받은 하위 클래스를 이용해 프로그램의 요구에 맞추어 클래스를 수정할 수 있고 클래스 간의 종속 관계를 형성함으로써 객체를 조직화할 수 있음
  - ✓ Polymorphism
    - 동일한 메시지에 대하여 다르게 반응하는 성질로 Method Overriding 을 이용해서 구현
  - ✓ Dynamic Binding은 실행 시간 중에 일어나거나 실행 과정에서 변경될 수 있는 바인딩으로 컴파일 시간에 완료되어 변화하지 않는 정적 바인딩과 대비되는 개념

OOP

- 자주 사용되는 용어
  - ✓ Object(객체): 프로그램에서 사용되는 모든 것
  - ✓ Class: 동일한 목적을 달성하기 위해 Attribute과 Method를 하나로 묶은 것 - Encapsulation
  - ✓ Class Object: Class와 동일한 의미로 사용하는데 특정 Class를 구체적으로 지정하기 위해 사용
  - ✓ Instance: Class를 자료형으로 해서 만들어진 객체
  - ✓ Method: Class 안에 정의된 함수
  - ✓ Class는 일종의 모델 하우스나 전시품 같은 것이고 Instance는 실제 사는 집이나 제품

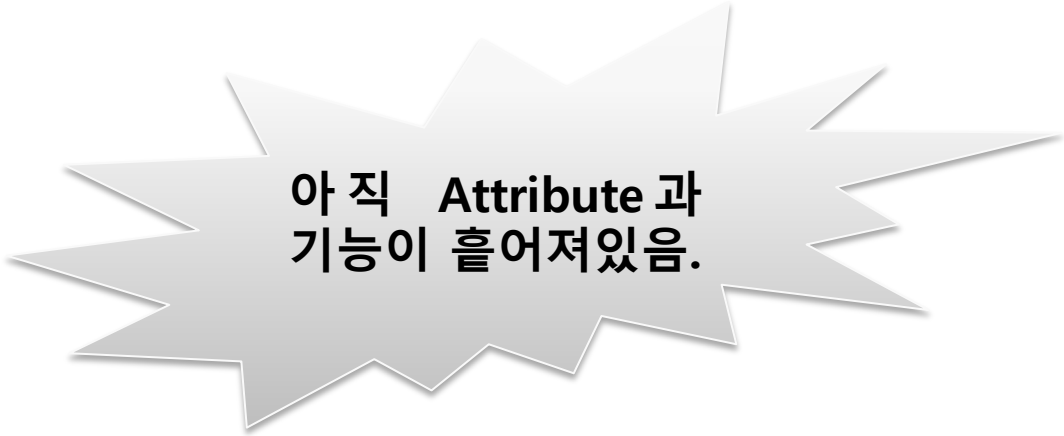
Class

- Class의 구성
  - ✓ Attribute: 사물의 특징, Field 라고도 함
    - 자동차의 Attribute : 바디의 색, 바퀴의 크기, 엔진의 배기량
  - ✓ Method: 어떤 것의 특징적인 동작
    - 자동차의 기능 : 전진, 후진, 좌회전, 우회전
  - ✓ Attribute 과 Method를 가지고 자동차를 묘사하면?
    - 18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있다.

Class

- 다음과 같이 묘사한 자동차를 코드로 표현
  - ✓ 18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있음

```
color = 0xFF0000    # 바디의 색
wheel_size = 16     # 바퀴의 크기
displacement = 2000 # 엔진 배기량

def forward(): # 전진
    pass

def backward(): # 후진
    pass

def turn_left(): # 좌회전
    pass

def turn_right(): # 우회전
    pass
```



Class

- 다음과 같이 묘사한 자동차를 코드로 표현
  - ✓ 18인치의 바퀴를 가진 2,000cc의 빨간 차는 전진, 후진, 좌회전, 우회전의 기능이 있음

```
class Car:
    def __init__(self):
        self.color = 0xFF0000     # 바디의 색
        self.wheel_size = 16      # 바퀴의 크기
        self.displacement = 2000 # 엔진 배기량

    def forward(self): # 전진
        pass

    def backward(self): # 후진
        pass

    def turn_left(self): # 좌회전
        pass

    def turn_right(self): # 우회전
        pass
```

Car Class의 정의 시작을 알림

Car Class 안에 차의 색, 바퀴 크기, 배기량을 나타내는 Attribute을 정의

Car Class 안에 전진, 후진, 좌회전, 우회전 Method를 정의

Class

- 인스턴스 생성
  - ✓ Class는 자료형
  - ✓ Car Class의 인스턴스는 다음과 같이 생성

```
num = 123       # 자료형:int, 변수: num
my_car = Car() # 자료형: Car Class, 인스턴스 이름: my_car
```

  - ✓ 인스턴스 대신 객체(Object)라는 용어를 사용하기도 함

Class

- Class 선언
  - ✓ Class는 다음과 같이 class 키워드를 이용하여 정의.

```
class Class이름:
    코드블록
```

  - ✓ Class의 코드 블록은 변수와 Method(Method)로 구성
    - Method(Method) : funtion에 대응하는 객체 지향 프로그래밍의 용어
    - 함수와 거의 동일한 의미이지만 Method는 Class나 인스턴스의 멤버라는 점이 다름
    - 함수(Function) : 일련의 코드를 하나의 이름 아래 묶은 코드의 집합
  - ✓ 인스턴스나 Class의 멤버 접근(Method와 데이터 Attribute에 접근)
    - 인스턴스의 멤버에 접근할 때는 점(.)을 이용

```
my_car = Car()
print(my_car.color )
```

my_car의 멤버에 접근

Class

- Method
  - ✓ 인스턴스가 있어야만 호출되는 Method 생성
    - Class에 인스턴스가 있어야만 호출되는 Method를 선언할 때 첫번째 매개변수는 현재 Class의 인스턴스가 되어야 하는데 관습적으로 self 라는 단어를 이용
    - 두번째 매개변수부터는 사용자가 원하는 이름으로 원하는 만큼 정의해서 사용 가능
  - ✓ Method를 호출하는 방법
    - Class 이름을 이용한 호출(언 바운드 호출)

      Class이름.Method이름(인스턴스이름, 매개변수)
    - 인스턴스 이름을 이용한 호출(바운드 호출)

      인스턴스이름.Method이름(매개변수)
    - Class 내부에서 자신의 Class에 있는 Method 호출

      self.Method이름(매개변수)

Class

❖ Student Class를 생성하고 인스턴스를 만들고 Method를 사용

```
#Class 선언
class Student:
    #인스턴스가 있어야만 호출가능한 Method 생성
    def insa(self):
        print('안녕하세요')

    #인스턴스가 있어야만 호출가능한 Method 생성
    def printName(self, name):
        #Method 내부에서 다른 Method 호출
        self.insa()
        print('이름은',name)

#인스턴스 생성
stu = Student()
#바운드 호출
stu.printName("사이버가수 아담")
#언바운드 호출
Student.printName(stu, "군계")
```

```
안녕하세요
이름은 사이버가수
아담
안녕하세요
이름은 군계
```

Class

- self

```
import time
class Student:
    # 생성자
    def __init__(self, name="사이버가수"):
        print(time.ctime(), "에 인스턴스가 생성되었습니다.")
        self.name = name

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

student = Student()
student.setName('아담')
```

Class

- Attribute
  - ✓ Class 내의 Method 외부에서 선언하면 Class Attribute: Class 내부에 1개만 만들어지며 Class 이름으로 접근할 수 있고 인스턴스 이름으로 읽는 것이 가능
  - ✓ Class Attribute는 Class 이름을 이용해서 수정하면 Class 변수가 수정되지만 인스턴스 이름으로 수정하면 인스턴스 각각에 별도로 Attribute를 생성함
  - ✓ 인스턴스가 Attribute를 읽을 때는 인스턴스에서 찾아보고 없으면 Class Attribute을 가져옴
  - ✓ Attribute가 없을 때 만들어지는 이유는 파이썬은 Attribute을 __dict__ 라고 하는 dict를 이용해서 저장하기 때문에 없으면 생성하고 있으면 읽거나 수정하기 때문
  - ✓ Method 내부에서 self 와 함께 선언되면 인스턴스 Attribute: 인스턴스 내부에 별도로 각각 사용할 수 있도록 만들어지며 인스턴스 이름으로 접근
  - ✓ Method 내부에서 self.과 함께 변수를 선언하지 않으면 그 변수는 Method의 지역변수

Class

❖ Class Attribute

```
class Student:
    schoolName = "Korea"

stu1 = Student()
stu2 = Student()

print("stu1의 참조:",id(stu1))
print("stu2의 참조:",id(stu2))
print("============================")
```

```
stu1의 참조: 4851339920
stu2의 참조: 4850827856
============================
Student의 학교: Korea
stu1의 학교: Korea
stu2의 학교: Korea
============================
Student의 학교: Seoul
stu1의 학교: Seoul
stu2의 학교: Seoul
============================
Student의 학교: Seoul
stu1의 학교: KAIST
stu2의 학교: Seoul
```

Class

- Class Attribute

```
#schoolName은 Method 외부에 만들어진 것이므로 class 변수가 되고
#Class 변수는 1개만 만들어서
#Class와 Class로부터 만들어진 인스턴스 모두가 공유해서 사용
#따라서 아래 3개의 출력문은 모두 동일한 값을 출력

print("Student의 학교:",Student.schoolName)
print("stu1의 학교:",stu1.schoolName)
print("stu2의 학교:",stu2.schoolName)
print("============================")

#Class를 이용해서 Attribute을 변경하면 Class Attribute이 변경됨
Student.schoolName="Seoul"
print("Student의 학교:",Student.schoolName)
print("stu1의 학교:",stu1.schoolName)
print("stu2의 학교:",stu2.schoolName)
print("============================")

#인스턴스를 이용해서 Attribute의 값을 수정하면 Class Attribute을 수정하는 것이 아니고
#자신에게 없는 Attribute이면 생성하고 없는 Attribute이면 수정
stu1.schoolName = "KAIST"
print("Student의 학교:",Student.schoolName)
print("stu1의 학교:",stu1.schoolName)
print("stu2의 학교:",stu2.schoolName)
```

Class

- is
  - ==
    - id를 비교하는 것이 아니고 데이터를 비교하기 위한 연산자
    - a == b 는 a.__eq__(b)의 편리 구문
    - object 객체에서 상속받은 __eq__( ) 메서드는 객체의 ID를 비교
    - 내장 자료형은 __eq__( ) 메서드를 오버라이딩해서 객체의 값을 비교
    - 대형 컬렉션이나 깊이 내포된 구조체를 비교하는 경우 동치 비교는 상당한 처리를 요구
  - is
    - id 값의 일치 여부를 확인하는 연산자
    - 연산자 오버로딩이 안됨

Class

- Accessor
  - ✓ 객체 지향 언어에서는 Attribute에 직접 접근하는 것을 권장하지 않음
  - ✓ Attribute는 인스턴스를 이용해서 직접 접근이 안되도록 하고 Attribute의 값을 리턴해주는 Getter Method 와 Attribute의 값을 설정해주는 Setter Method를 만들어서 Attribute를 사용하는 것을 권장
  - ✓ Getter
    - Attribute의 값을 리턴해주는 Method
    - 이름은 get필드명 으로 하고 필드명의 첫 글자는 대문자로 하며 매개변수는 없고 Method의 내용은 Attribute의 값을 리턴하기만 함
    - 필드의 타입이 bool 인 경우 get 대신에 is를 사용
  - ✓ Setter
    - Attribute의 값을 설정해주는 Method
    - 이름은 set필드명 으로 하고 필드명의 첫 글자는 대문자로 하고 매개변수는 필드의 타입과 동일한 타입 1개로 하며 내용은 매개변수의 값을 Attribute에 대입
  - ✓ 파이썬은 대문자 대신에 앞에 _를 붙이는 경우가 많음

Class

❖ Accessor 생성과 호출

```
class Student:
    #Method를 만들 때 첫번째 매개변수는 인스턴스의 참조를 기억하는 self
    def getName(self):
        #인스턴스가 각각 소유하는 name의 값을 리턴
        return self.name

    #매개변수가 2개인 함수
    def setName(self, name):
        #매개변수로 받은 내용을 인스턴스가 각각 소유하는 name에 대입
        self.name = name

#객체 자신의 name에 값을 설정
student1 = Student()
student1.setName('아담')

student2 = Student()
student2.setName('류시아')

#객체 자신의 값을 가져오기 때문에 내용이 다름
print(student1.getName())
print(student2.getName())
```

```
아담
류시아
```

Class

- ❖ 특수 Attribute
  - ✓ __로 시작하는 Attribute으로 특별한 목적으로 사용할 수 있도록 파이썬에서 제공
    - __doc__: 함수를 설명하는 문자열로 설정하지 않는 경우 None이며 서브 Class로 상속되지 않으며 쓰기 가능
    - __name__: 함수의 이름으로 쓰기 가능
    - __qualname__: 함수의 정규화된 이름으로 쓰기 가능
    - __module__: 함수가 정의된 Module의 이름으로 설정하지 않으면 None이 되고 쓰기 가능
    - __defaults__: 인자의 기본값 이며 설정하지 않으면 None 으로 만들어지는 튜플로 쓰기 가능
    - __code__: 컴파일된 함수의 바디(body) 를 나타내는 코드 객체로 쓰기 가능
    - __globals__: 함수의 전역 변수들을 가진 딕셔너리에 대한 참조 - 함수가 정의된 Module의 전역 이름 공간(namespace)으로 읽기 전용
    - __dict__: 함수나 객체에 만들어진 Attribute 이름으로 쓰기 가능
    - __closure__: None 또는 함수의 자유 변수(free variable)들에 대한 연결을 가진 셀(cell)들의 튜플로 읽기 전용
    - __annotations__: 파라미터의 어노테이션을 가진 dict로 dict 의 키는 파라미터의 이름인데 반환 값 어노테이션이 있다면 return 을 키로 사용하며 쓰기 가능
    - __kwdefaults__: 키워드 형태로만 전달 가능한 파라미터들의 기본값을 가진 dict로 쓰기 가능

Class

- 생성자 와 소멸자
  - ✓ 생성자
    - __init__
    - 인스턴스를 생성하기 위해 호출하는 메서드
    - 초기화하다는 뜻의 initialize를 줄여서 붙여진 이름
    - 만들지 않으면 매개변수가 없는 초기화 Method가 자동으로 생성됨
    - 직접 작성하는 경우 첫번째 매개변수는 self 이며 이후에 매개변수 추가 가능
    - self 이외의 매개변수가 있는 생성자를 만든 후 그 생성자를 이용해서 인스턴스를 생성할 때는 생성자에 매개변수를 전달해야 함
    - 초기화 Method를 직접 생성하면 기본적으로 제공되는 초기화 Method를 만들어지지 않음
    - 초기화 Method를 만드는 이유는 인스턴스 변수를 생성해서 초기화하거나 인스턴스를 생성할 때 기본적으로 수행할 작업을 하도록 하기 위해서
  - ✓ 소멸자:__del__()
    - 인스턴스가 소멸될 때 호출되는 Method
    - 외부 자원을 사용하는 경우 해제하는 코드를 주로 작성
    - self 이외의 매개변수를 받지 않음

Class

- 생성자 와 소멸자
  - ✓ Garbage Collection
    - 파이썬에서는 특별히 사용자가 메모리 관리를 할 필요는 없음
    - 메모리는 자동으로 할당되고 자동으로 해제
    - 파이썬의 모든 데이터는 객체로 관리되며 참조 횟수(Reference Count – Retain Count)를 가지고 있고 이 값이 0가 되면 파이썬이 자동으로 해제
    - 참조 횟수는 데이터의 참조를 변수에 대입하게 되면 1씩 증가하며 가리키는 변수가 소멸되거나 변수에 None을 대입하면 1 감소
    - 참조 횟수를 확인하고자 하면 sys Module의 getrefcount 함수에 객체를 대입하면 되는데 이 때 참조 횟수는 1이 증가해서 나오게 되는데 이유는 getrefcount 함수 자체가 데이터를 참조해서 참조 횟수를 1증가 시켰기 때문

Class

- 생성자 와 소멸자
  - ✓ 매개변수가 없는 초기화 Method를 이용해서 인스턴스를 생성하고 Method 호출

```
class Student:
    def getName(self):
        return self.name

    def setName(self, name):
        self.name = name

student1 = Student()
student1.setName('아담')
print(student1.getName())

student2 = Student()
print(student2.getName())
```

Class

- 생성자 와 소멸자
  - ✓ 매개변수가 없는 초기화 Method를 이용해서 인스턴스를 생성하고 Method 호출

```
박문석
Traceback (most recent call last):
  File "//Mac/Home/Documents/source/python/six.py", line 46, in <module>
    print(student2.getName())
  File "//Mac/Home/Documents/source/python/six.py", line 36, in getName
    return self.name
AttributeError: 'Student' object has no attribute 'name'
```

Class

❖ 초기화 method 와 소멸자를 추가

```
import time
class Student:
    # 생성자
    def __init__(self, name="사이버가수"):
        print(time.ctime(), "에 인스턴스가 생성되었습니다.")
        self.name = name

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    # 소멸자
    def __del__(self):
        print(time.ctime(), "에 인스턴스가 소멸되었습니다.")
```

```
Sat Aug 14 11:21:27 2021 에 인스턴스가 생성되었습니다.
아담
Sat Aug 14 11:21:27 2021 에 인스턴스가 생성되었습니다.
사이버가수
Sat Aug 14 11:21:27 2021 에 인스턴스가 소멸되었습니다.
Sat Aug 14 11:21:27 2021 에 인스턴스가 소멸되었습니다.
```

Class

- 초기화 method 와 소멸자를 추가

```
student1 = Student()
student1.setName('아담')
print(student1.getName())

student2 = Student()
print(student2.getName())

#객체의 소멸은 None을 대입해서 retain count를 1감소 시켜서 0을 만들어서 삭제
student1 = None
student2 = None
```

Class

❖ 참조 횟수 확인

```
import sys

student1 = Student()
print(sys.getrefcount(student1))
student1 = None
#아래 문장은 에러
#print(student1.getName())

student2 = Student()
print(sys.getrefcount(student2))
student3 = student2
print(sys.getrefcount(student2))
student2 = None
print(student3.getName())
```

```
Sat Aug 14 11:26:22 2021 에 인스턴스가 생성되었습니다.
2
Sat Aug 14 11:26:22 2021 에 인스턴스가 소멸되었습니다.
Sat Aug 14 11:26:22 2021 에 인스턴스가 생성되었습니다.
2
Sat Aug 14 11:26:22 2021 에 인스턴스가 소멸되었습니다.
3
사이버가수
```

Class

- Static Method
  - ✓ 인스턴스를 생성하지 않고 Class를 이용해서 직접 호출할 수 있는 Method
  - ✓ Method 내에서 인스턴스 변수를 생성하거나 호출할 수 없음
  - ✓ Class 변수는 호출할 수 있음
  - ✓ @staticmethod 데코레이터로 장식
  - ✓ 매개변수는 self 없이 바로 정의
  - ✓ 인스턴스를 이용해서 호출 가능

```
class Class이름:
    @staticmethod
    def Method이름( 매개변수 ):
        pass
```

@staticmethod 데코레이터로 수식

self 매개변수는 사용하지 않음

Class

- Class Method
  - ✓ @classmethod 데코레이터로 수식
  - ✓ 정적 Method와 유사하지만 첫번째 매개변수로 Class 인스턴스가 전달됨
  - ✓ Class 인스턴스 매개변수 이름은 cls 라는 매개변수 이름을 사용하는 것이 관례
  - ✓ 인스턴스를 이용해서 호출 가능

```
class Class이름:
    # ...

    @classmethod
    def Method이름(cls):
        pass
```

Class Method를 정의하기 위해서는…
1.@classmethod 데코레이터를 앞에 붙여줌

2.Method의 매개변수를 하나 이상 정의

Class

- Static Method와 Class Method

```
class Student:
    @classmethod
    def cmethod(cls):
        print("Class Method")
        print(cls)

    @staticmethod
    def smethod():
        print("정적 Method")

#클래스를 이용한 method
Student.cmethod()
Student.smethod()

#인스턴스를 이용한 method
student = Student()
student.cmethod()
student.smethod()
```

```
Class Method
<class '__main__.Student'>
정적 Method
Class Method
<class '__main__.Student'>
정적 Method
```

Class

- Class Method
  - ✓ 클래스 멤버 와 인스턴스 멤버 구분
    - 클래스 멤버는 클래스의 이름 공간에 생성
    - 인스턴스 멤버는 인스턴스 객체의 이름 공간에 생성
    - 클래스 멤버는 모든 인스턴스 객체에 의해서 공유
    - 인스턴스 멤버는 각각의 인스턴스 객체 내에서만 참조

Class

- __slots__ Attribute
  - ✓ Class 정의 안에 __slots__ Attribute을 이용하면 인스턴스에 새로운 Attribute을 정의하는 것을 제한할 수 있음
  - ✓ __slots__ Attribute에 list로 Attribute을 나열하면 나열한 Attribute만 사용이 가능하고 새로운 Attribute은 생성할 수 없음
  - ✓ __slots__ Attribute을 정의하면 __dict__ Attribute을 사용할 수 없음

```
class Student:
    name = ""
    score = 0


student = Student()
student.name = "제시카"
student.score = 90
student.tel = "01031391997"
print(student.tel)
```

01031391997

Class

- __slots__ Attribute

```
class Student:
    __slots__ = ["name", "score"]

student = Student()
student.name = "제시카"
student.score = 90
student.tel = "01031391997"
print(student.tel)
```

```
Traceback (most recent call last):

  File "<ipython-input-21-63c0ad90247d>", line 8, in <module>
    student.tel = "01031391997"

AttributeError: 'Student' object has no attribute 'tel'
```

Class

- private 멤버 명명 규칙
  - ✓ private 멤버는 Class 내부에서는 접근이 가능하지만 Class 외부(인스턴스)에서 접근이 되지 않는 멤버
  - ✓ public 멤버는 Class 외부(인스턴스)에서 접근이 가능한 멤버
  - ✓ python의 모든 멤버는 public
  - ✓ 속성을 생성할 때 두 개의 밑줄(__)을 추가하고 이름을 설정하면 Class 외부에서는 접근이 안 됨

Class

- property
  - ✓ 변수를 호출하는 것처럼 사용하지만 실제로는 getter 와 setter Method를 호출하는 것으로 처리되는 Attribute

    변수명 = property(fget=None, fset=None, fdel=None, doc=None)

    - fget은 getter
    - fset은 setter
    - fdel은 삭제할 때 사용할 Method

Class

- property

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

maria = Student('마리아', 20)
maria.name = "Maria"
maria.__age -= 1  # Class 외부에서 비공개 Attribute에 접근하면 에러가 발생함
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/var/folders/42/fxl3_n31121f16yj69fbrb_m0000gn/T/ipykernel_29861/1322497898.py in
      <module>
       6 maria = Student('마리아', 20)
       7 maria.name = "Maria"
----> 8 maria.__age -= 1  # Class 바깥에서 비공개 Attribute에 접근하면 에러가 발생함

AttributeError: 'Student' object has no attribute '__age'
```

Class

- property

```
class Student:
    def __init__(self, name="noname"):
        self.__name = name

    def setName(self, name):
        print("setter 호출")
        self.__name = name

    def getName(self):
        print("getter 호출")
        return self.__name

    name = property(getName, setName)


stu = Student()
# 아래 문장은 변수를 호출한 것이 아니고 Accessor를 호출한 것
stu.name = "adam"
print(stu.name)
```

Class

❖ Decorator를 이용한 Property

```
class Student:

    def __init__(self, name="noname"):
        self.__name = name

    @property
    def name(self):
        print("getter 호출")
        return self.__name

    @name.setter
    def name(self, name):
        print("setter 호출")
        self.__name = name

stu = Student()
stu.name = "아담"
print(stu.name)
```

Class

- Operator Overloading
  - ✓ 연산자의 기능을 수정해서 사용하는 것
  - ✓ 연산자 오버로딩(중복정의)이란 인스턴스끼리 서로 연산을 할 수 있게끔 기존에 있는 연산자의 기능을 바꾸어 중복으로 정의하는 것
  - ✓ + 는 수치 데이터를 더하는 기능을 가진 연산자인데 문자열 데이터에 +를 하면 더하는 기능을 수행하는 것이 아니고 문자열 데이터를 연결하는 기능을 수행
  - ✓ 파이썬에서는 특정 이름의 Method(__연산자이름__)를 재정의하면 Operator Overloading가 구현

Class

❖ Operator Overloading

메서드(Method)	연산자(Operator)	사용 예
__add__(self, other)	+ (이항)	A + B, A += B
__pos__(self)	+ (단항)	+A
__sub__(self, other)	- (이항)	A - B, A -= B
__neg__(self)	- (단항)	-A
__mul__(self, other)	*	A * B, A *= B
__truediv__(self, other)	/	A / B, A /= B
__floordiv__(self, other)	//	A // B, A //= B
__mod__(self, other)	%	A % B, A %= B
__pow__(self, other)	pow(), **	pow(A, B), A ** B
__lshift__(self, other)	<<	A << B, A <<= B
__rshift__(self, other)	>>	A >> B, A >>= B
__and__(self, other)	&	A & B, A &= B
__xor__(self, other)	^	A ^ B, A ^= B
__or__(self, other)	|	A | B, A |= B
__invert__(self)	~	~A
__abs__(self)	abs()	abs(A)

Class

- Student Class에 + Operator Overloading

```
class Student:
    def __init__(self, name="noname"):
        self.__name = name

    def setName(self, name):
        print("setter 호출")
        self.__name = name

    def getName(self):
        print("getter 호출")
        return self.__name

    def __add__(self, other):
        return self.name + "," +  other.name


stu1 = Student()
stu1.name = "아담"

stu2 = Student()
stu2.name = "류시아"

print(stu1 + stu2)
```

Class

- Student Class에 __str__ Method 중복 정의:
  - ✓ print Method에 객체를 대입할 때 호출되는 Method

```
class Student:
    def __init__(self, name="noname"):
        self.__name = name

    def setName(self, name):
        print("setter 호출")
        self.__name = name

    def getName(self):
        print("getter 호출")
        return self.__name

    def __str__(self):
        return self.name

stu = Student()
stu.name = "아담"
print(stu)
```

Class

- Operator Overloading
  - ✓ 비교 연산 메서드

연산자	메서드
<	object.__lt__(self, other)
<=	object.__le__(self, other)
>	object.__gt__(self, other)
>=	object.__ge__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)

Class

- Student Class에 __str__ Method 중복 정의:
  - ✓ 비교 연산자 오버로딩

```
class Student:
    def __init__(self, name="noname"):
        self.__name = name

    def setName(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    def __eq__(self, other):
        return self.name == other.name
```

Class

- Student Class에 __str__ Method 중복 정의:
  - ✓ 비교 연산자 오버로딩

```
stu1 = Student()
stu1.name = "아담"

stu2 = Student()
stu2.name = "아담"

print("is:", stu1 is stu2)
print("==:", stu1 == stu2)
```

```
is: False
==: True
```

Class

- Operator Overloading
  - ✓ 해시 값 접근 메서드 - __hash__
  - ✓ 속성 값 접근 메서드

메서드	설명
`__getattr__(self, name)`	정의되어 있지 않은 속성을 참조할 때, 이 메서드가 호출된다. 속성 이름 name은 문자열이다.
`__getattribute__(self, name)`	__getattr__( ) 메서드와 같으나 속성이 정의되어 있어도 호출된다.
`__setattr__(self, name, value)`	self.name = value와 같이 속성에 치환(대입)이 일어날 때 호출된다.
`__delattr__(self, name)`	del self.name에 의해서 호출된다.

Class

- Operator Overloading
  - ✓ 형 변환 메서드

메서드	연산자
__complex__(self)	complex( )
__int__(self)	int( )
__float__(self)	float( )
__round__(self)	round( )
__index__(self)	operator.index( )

- 문자열 변환 메서드: __str__(), __repr__()
- 바이트 변환 메서드: __bytes__()
- bool 변환 메서드: __bool__()

Class

- Operator Overloading
  - ✓ 시퀀스 자료형의 메서드

메서드	연산자
object.__len__(self)	len(object)
object.__contains__(self, item)	item in object
object.__getitem__(self, key)	object[key]
object.__setitem__(self, key, value)	object[key] = value
object.__delitem__(self, key)	del object[key]

Class

- Operator Overloading – 인덱싱(__getitem__)

```
class Square:
    def __init__(self, end):
        self.end = end

    def __len__(self):
        return self.end

    def __getitem__(self, k):
        if type(k) != int:
            raise TypeError('...')
        if k < 0 or self.end <= k:
            raise IndexError('index {} out of range'.format(k))
        return k * k

s1 = Square(10)
print(len(s1))
print(s1.__getitem__(4))
print(s1[4])
print(s1[20])
```

Class

- Operator Overloading – 인덱싱(__getitem__)

```
10
16
16
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/var/folders/98/j21dd2qs46b69mwc94dgyxnr0000gn/T/ipykernel_22784/1846584318.py in
      <module>
      17 print(s1.__getitem__(4))
      18 print(s1[4])
---> 19 print(s1[20])

/var/folders/98/j21dd2qs46b69mwc94dgyxnr0000gn/T/ipykernel_22784/1846584318.py in
      __getitem__(self, k)
      10             raise TypeError('...')
      11         if k < 0 or self.end <= k:
---> 12             raise IndexError('index {} out of range'.format(k))
      13         return k * k
      14

IndexError: index 20 out of range
```

Class

- Operator Overloading – 매핑 자료형(__getitem__, __setitem__)

```
class MyDict:
    def __init__(self):
        self.d = {}

    def __getitem__(self, k): # key
        return self.d[k]
    def __setitem__(self, k, v):
        self.d[k] = v

    def __len__(self):
        return len(self.d)

m = MyDict( )
m.__setitem__('night','darkness')
m['day'] = 'light'

print(m['day'])
print(m['night'])
print(len(m))
```

```
light
darkness
2
```

Class

- Operator Overloading
  - ✓ __call__(): 인스턴스이름(매개변수) 를 호출하면 호출되는 메서드
  - ✓ 인스턴스 객체 생성
    - __new__()
    - 메모리 할당을 위한 함수로 init 보다 먼저 호출됨
  - ✓ with 사용
    - __enter__() 와 __exit__() 메서드 구현

Class

❖ Singleton Pattern 구현

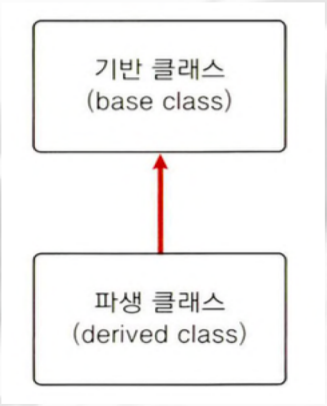
```
class Singleton:
    __instance = None # 유일한 객체를 저장하기 위한 클래스 변수

    def __new__(cls, *args, **kwargs):
        if cls.__instance is None:
            # 새로운 객체를 생성한다.
            cls.__instance = object.__new__(cls, *args, **kwargs)
        return cls.__instance

class Sub(Singleton): # 싱글톤으로부터 상속받는다.
    pass

s1 = Sub( )
s2 = Sub( )
print(s1 is s2)
```

True

Inheritance

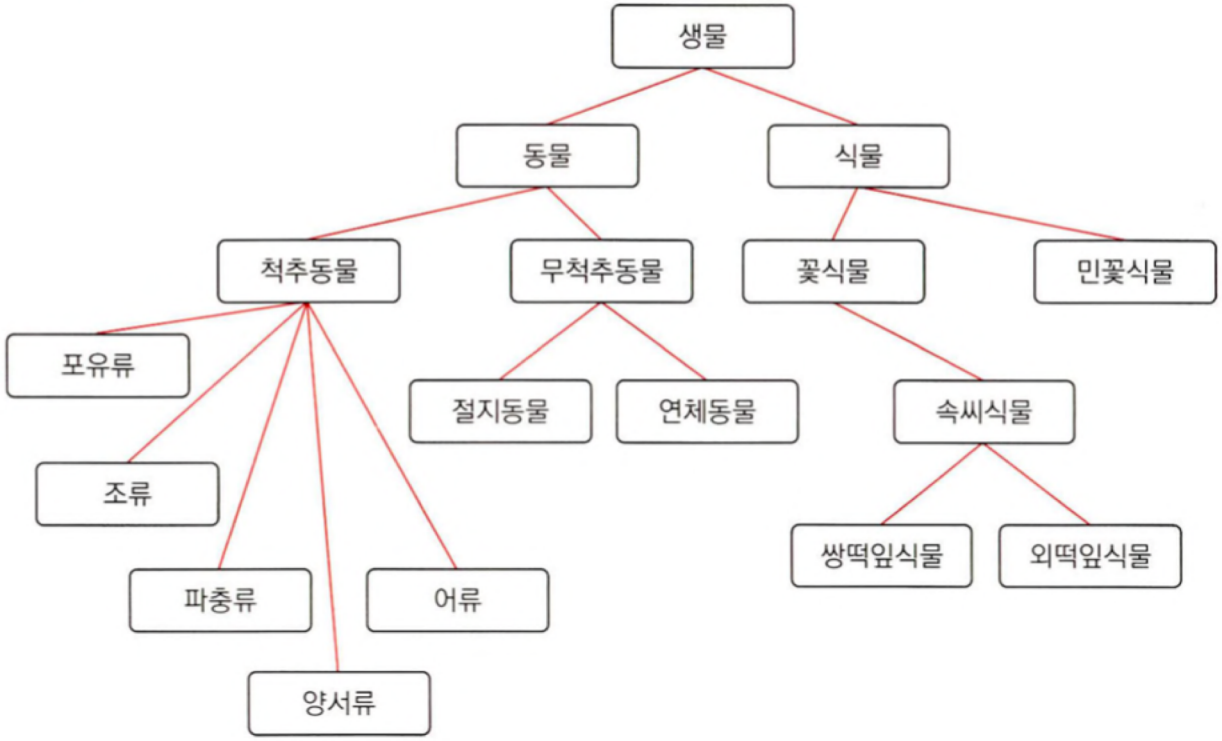
- Inheritance(상속)
  - ✓ 하위 Class가 상위 Class의 모든 것을 물려 받는 것
  - ✓ 용어
    - 상속하는 Class를 기반 Class(base class), 상속을 받아 새롭게 만들어지는 Class를 파생 Class(derived class)라고 하는데 기반 Class는 슈퍼 Class(superclass)라고 부르기도 하고 파생 Class는 서브 Class(subclass)라고도 함
    - 하나의 Class로부터 상속받는 것을 단일 상속이라 하고 여러 개의 Class로부터 상속받는 것을 다중 상속이라고 함
  - ✓ 상속의 목적
    - 여러 Class에 공통으로 존재하는 코드를 상위 Class로 묶어서 중복을 제거하기 위해서
    - 제공되는 Class의 기능이 부족해서 기능 확장을 위해서
  - ✓ Class 상속은 다음과 같이 Class를 만들 때 ( )(괄호)를 붙이고 안에 기반 Class 이름을 작성

```
class 기반Class이름:
    코드
class 파생Class이름(기반Class이름):
    코드
```



Inheritance

❖ Inheritance(상속)



Inheritance

❖ 단일 상속

```
class Person:
    def greeting(self):
        print('안녕하세요.')

class Student(Person):
    def study(self):
        print('공부하기')

student = Student()
student.greeting()    # 기반 Class Person의 Method 호출
student.study()       # 파생 Class Student에 추가한 study Method
```

안녕하세요.
공부하기

Inheritance

- Inheritance
  - ✓ 현재 Class가 특정 Class의 파생 Class인지 확인할 때는 issubclass 함수를 사용하는데 파생 Class가 맞으면 True, 아니면 False를 반환

    issubclass(Student, Person)

    True
  - ✓ __bases__ Attribute을 이용하면 상위 Class 목록을 가져옴
  - ✓ 하위 Class의 init() Method에서 상위 Class의 초기화 Method를 super().init()을 이용해서 호출해야 상위 Class의 Attribute이나 함수 사용 가능하며 명시적으로 특정 상위 클래스의 method를 호출하고자 하는 경우는 super(클래스이름, self). 으로 호출
  - ✓ 상위 Class의 init을 호출하지 않고 상위 Class의 Attribute에 바로 접근하면 에러
  - ✓ Method Overriding: 상위 Class에 존재하는 Method를 하위 Class에서 재정의해서 사용하는 것으로 상위 Class의 Method를 호출하지 않으면 상위 Class의 Method를 대치하게 되고 상위 Class의 Method를 호출한 후 내용을 작성하면 기능을 확장
  - ✓ 인스턴스가 특정 클래스의 인스턴스 인지 확인은 isinstance(인스턴스, 클래스)를 이용

Inheritance

❖ 상위 Class의 Method 호출하기

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

class Student(Person):
    def __init__(self):
        print('Student __init__')
        self.school = '파이썬 학교'

student = Student()
print(student.school)
print(student.hello)    # 기반 Class의 Attribute을 출력하려고 하면 에러가 발생함
```

```
Student __init__
파이썬 학교
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-18ad596c4386> in <module>()
     11 james = Student()
     12 print(james.school)
---> 13 print(james.hello)    # 기반 Class의 Attribute을 출력하려고 하면 에러가 발생함
AttributeError: 'Student' object has no attribute 'hello'
```

Inheritance

❖ 상위 클래스의 프로퍼티 호출

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

class Student(Person):
    def __init__(self):
        print('Student __init__')
        super().__init__()              # super()로 기반 Class의 Method 호출
        self.school = '파이썬 학교'

student = Student()
print(student.school)
print(student.hello)
```

Student __init__
Person __init__
파이썬 학교
안녕하세요.

Inheritance

- Method 오버라이딩

```
class Person:
    def greeting(self):
        print('안녕하세요.')

class Student(Person):
    def greeting(self):
        super().greeting()
        print('반갑습니다.')

student = Student()
student.greeting()
```

안녕하세요.
반갑습니다.

Inheritance

- 다중 상속
  - ✓ 여러 개의 Class로부터 상속받는 경우
  - ✓ 파이썬은 다중 상속을 지원하고 부모 Class에 동일한 Method나 Attribute이 있을 때는 왼쪽에서부터 우선권을 부여

```
class 기반Class이름1:
    코드


class 기반Class이름2:
    코드


class 파생Class이름(기반Class이름1, 기반Class이름2):
    코드
```

  - ✓ method 탐색 순서(Method Resolution Order, MRO)는 클래스 이름.mro() 를 이용해서 확인 가능

Inheritance

❖ 다중 상속

```
class Base1:
    def __init__(self):
        print('첫번째 상위 클래스의 초기화 method')

    def method(self):
        print('첫번째 상위 클래스의 method')

class Base2:
    def __init__(self):
        print('두번째 상위 클래스의 초기화 method')

    def method(self):
        print('두번째 상위 클래스의 method')
```

```
첫번째 상위 클래스의 초기화 method
두번째 상위 클래스의 초기화 method
하위 클래스의 초기화 method

첫번째 상위 클래스의 method
두번째 상위 클래스의 method
하위 클래스의 method
```

Inheritance

- 다중 상속

```
class Derived(Base1, Base2):
    def __init__(self):
        #첫번째 상위 클래스의 초기화 method
        super().__init__()
        #두번째 상위 클래스의 초기화 method
        super(Base1, self).__init__()
        print('하위 클래스의 초기화 method')

    def method(self):
        #첫번째 상위 클래스의 method() 호출
        super().method()
        #두번째 상위 클래스의 method() 호출
        super(Base1, self).method()
        print('하위 클래스의 method')

drived = Derived()
print()
drived.method()
```

Inheritance

- Abstract(추상)
  - ✓ Abstract Method: 내용이 없는 Method로 하위 Class에서 재정의해서 사용해야 하는 Method로 Abstract Class에 존재해야 함
  - ✓ Abstract Class
    - 자신의 인스턴스를 생성할 수 없는 Class로 Abstract Method를 소유하고 있는 Class
    - 생성
      - abc Module을 가져 온 후 Class의 ( )(괄호) 안에 metaclass=ABCMeta (abc는 abstract base class의 약자) 를 지정
      - 추상 Method를 만들 때 위에 @abstractmethod를 붙여서 추상 Method로 지정
      - 추상 클래스를 상속받은 클래스는 반드시 추상 method를 재정의 해야 함
  - ✓ 목적
    - Polymorphism 구현
    - 템플릿 Method 패턴 구현

Inheritance

- 추상 클래스

```
import abc
#추상 클래스
class ChineseRestaurant(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def jajangmyeon(self):
        pass

class Magnoliaceae(ChineseRestaurant):
    pass

magnoliaceae = Magnoliaceae()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/42/fxl3_n31121f16yj69fbrb_m0000gn/T/ipykernel_29861/3438241028.py in
    <module>
      7         pass
      8
----> 9 magnoliaceae = Magnoliaceae()

TypeError: Can't instantiate abstract class Magnoliaceae with abstract methods jajangmyeon
```

Inheritance

❖ 추상 클래스

```
import abc
class ChineseRestaurant(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def jajangmyeon(self):
        pass


class Magnoliaceae(ChineseRestaurant):
    def jajangmyeon(self):
        print('이연복이 만든 짜장면')


magnoliaceae = Magnoliaceae()
magnoliaceae.jajangmyeon()
```

이연복이 만든 짜장면

특별한 기능

- Delegation
  - ✓ 존재하지 않는 Method를 호출한 경우 에러 발생
  - ✓ 존재하지 않는 Method를 호출했을 때 에러를 발생시키지 않고자 하는 경우에는 __getattr__ Method를 구현해서 처리를 위임할 수 있음
  - ✓ 이 Method는 self 이외에 문자열로 된 매개변수를 1개 소유하는데 이 매개변수가 호출된 Method 이름

```
class Delegation:
    def __init__(self, data):
        self.data = data

    def __getattr__(self, name):
        print(name + "호출")
        #self.data 의 count Method 호출
        return getattr(self.data, name)

instance = Delegation([100,200,300,200,200])
print(instance.count(200))
```

```
count 호출
3
```

특별한 기능

- Iterator
  - ✓ 이터레이터(iterator)는 값을 차례대로 꺼낼 수 있는 인스턴스(object)
  - ✓ 100번을 반복한다면 for i in range(100):처럼 만들어 사용
  - ✓ 위의 경우 0부터 99까지 값을 차례대로 꺼낼 수 있는 이터레이터를 만들어 낸 후 반복할 때마다 이터레이터에서 숫자를 하나씩 꺼내서 사용
  - ✓ 연속된 숫자를 미리 만들면 숫자가 적을 때는 상관없지만 숫자가 아주 많을 때는 메모리를 많이 사용하므로 성능에 불리
  - ✓ 파이썬에서는 이터레이터만 생성하고 값이 필요한 시점이 되었을 때 값을 만드는 방식을 사용
  - ✓ 데이터 생성을 뒤로 미루는 것인데 이런 방식이 지연 평가(lazy evaluation)
  - ✓ 이터레이터는 반복자라고도 함
  - ✓ 반복 가능한 인스턴스는 문자열, 리스트, 딕셔너리, 세트 등
  - ✓ 요소가 여러 개 들어있고 한 번에 하나씩 꺼낼 수 있는 인스턴스
  - ✓ 인스턴스가 반복 가능한 인스턴스인지 알아보는 방법은 인스턴스에 __iter__ Method가 들어있는지 확인

특별한 기능

- Iterator
  - ✓ 리스트 [1, 2, 3]을 dir로 살펴보면 __iter__ Method가 존재
  - ✓ 리스트에서 __iter__를 호출하면 이터레이터가 리턴
  - ✓ 이터레이터를 가지고 __next__를 호출할 때마다 리스트에 들어있는 1, 2, 3이 리턴
  - ✓ 3 다음에 __next__를 호출하면 StopIteration 예외가 발생하는데 [1, 2, 3]이므로 1, 2, 3 세 번만 반복가능
  - ✓ 이터레이터는 __next__로 요소를 계속 꺼내다가 꺼낼 요소가 없으면 StopIteration 예외를 발생시켜서 반복을 종료
  - ✓ 직접 구현 가능

```
class 이터레이터이름:
    def __iter__(self):
        코드

    def __next__(self):
        코드
```

  - ✓ enumerate 함수는 순서가 있는 데이터의 모임을 index 와 data 의 tuple 로 리턴

특별한 기능

- Iterator

```
li = [1, 2, 3]
it = li.__iter__()
#iterator 출력하기
print(it)
print(it.__next__())
print(it.__next__())
print(it.__next__())
print(it.__next__())
```

```
<list_iterator object at 0x7fd93a4833a0>
1
2
3
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-1-1ef04a89f7f8> in <module>
      6 print(it.__next__())
      7 print(it.__next__())
----> 8 print(it.__next__())

StopIteration:
```

특별한 기능

- Iterator

```
class IteratorImpl:
    def __init__(self, stop):
        self.current = 0     # 현재 숫자 유지, 0부터 지정된 숫자 직전까지 반복
        self.stop = stop     # 반복을 끝낼 숫자

    def __iter__(self):
        return self          # 현재 인스턴스를 반환

    def __next__(self):
        if self.current < self.stop:     # 현재 숫자가 반복을 끝낼 숫자보다 작을 때
            r = self.current             # 반환할 숫자를 변수에 저장
            self.current += 1            # 현재 숫자를 1 증가시킴
            return r                     # 숫자를 반환
        else:                            # 현재 숫자가 반복을 끝낼 숫자보다 크거나 같을 때
            raise StopIteration          # 예외 발생

for i in IteratorImpl(3):
    print(i, end=' ')
```

0 1 2

특별한 기능

- enumerate

```
study = ['파이썬', '자료구조', '알고리즘', '웹 프로그래밍', '데이터 분석']
for idx, subject in enumerate(study):
    print(idx, ':', subject)
```

0 : 파이썬
1 : 자료구조
2 : 알고리즘
3 : 웹 프로그래밍
4 : 데이터 분석

특별한 기능

- Generator
  - ✓ Generator는 Iterator의 특수한 한 형태
  - ✓ Generator 함수(Generator function)는 함수 안에 yield를 사용하여 데이터를 하나씩 리턴
  - ✓ Generator 함수가 처음 호출되면 그 함수 실행 중 처음으로 만나는 yield 에서 값을 리턴
  - ✓ Generator 함수가 다시 호출되면 직전에 실행되었던 yield 문 다음부터 다음 yield 문을 만날 때까지 문장들을 실행
  - ✓ Generator 함수를 변수에 할당하면 그 변수는 generator Class 인스턴스가 되는데 __next__ 함수가 구현되어 있음

특별한 기능

- Generator

```
def gen():
    yield 1
    yield 2
    yield 3

# Generator 인스턴스
g = gen()
print(type(g))  # <class 'generator'>

# for 루프 사용 가능
for x in g:
    print(x)
```

```
<class 'generator'>
1
2
3
```

특별한 기능

- Generator

```
def reverse(data):
    for idx in range(len(data)-1, -1, -1):
        yield data[idx]

for ch in reverse('Teacher'):
    print(ch)
```

```
r
e
h
c
a
e
T
```

특별한 기능

- Generator

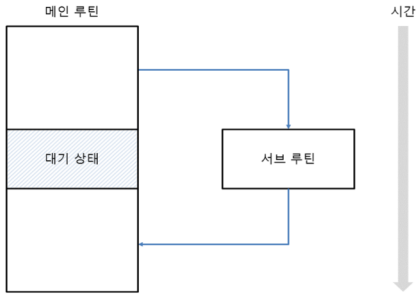
```
def upper_generator(x):
    for i in x:
        yield i.upper() # 함수의 반환값을 바깥으로 전달

study = ['Python', 'DataStructure', 'Algorithm', 'Web_Programming', 'Data_Analysis']

for i in upper_generator(study):
    print(i)
```

PYTHON
DATASTRUCTURE
ALGORITHM
WEB_PROGRAMMING
DATA_ANALYSIS

특별한 기능

❖ 일반적인 함수 호출

```
def add(a, b):
    c = a + b     # add 함수가 끝나면 변수와 계산식은 사라짐
    print(c)
    print('add 함수')

def calc():
    add(1, 2)     # add 함수가 끝나면 다시 calc 함수로 돌아옴
    print('calc 함수')

calc()
```

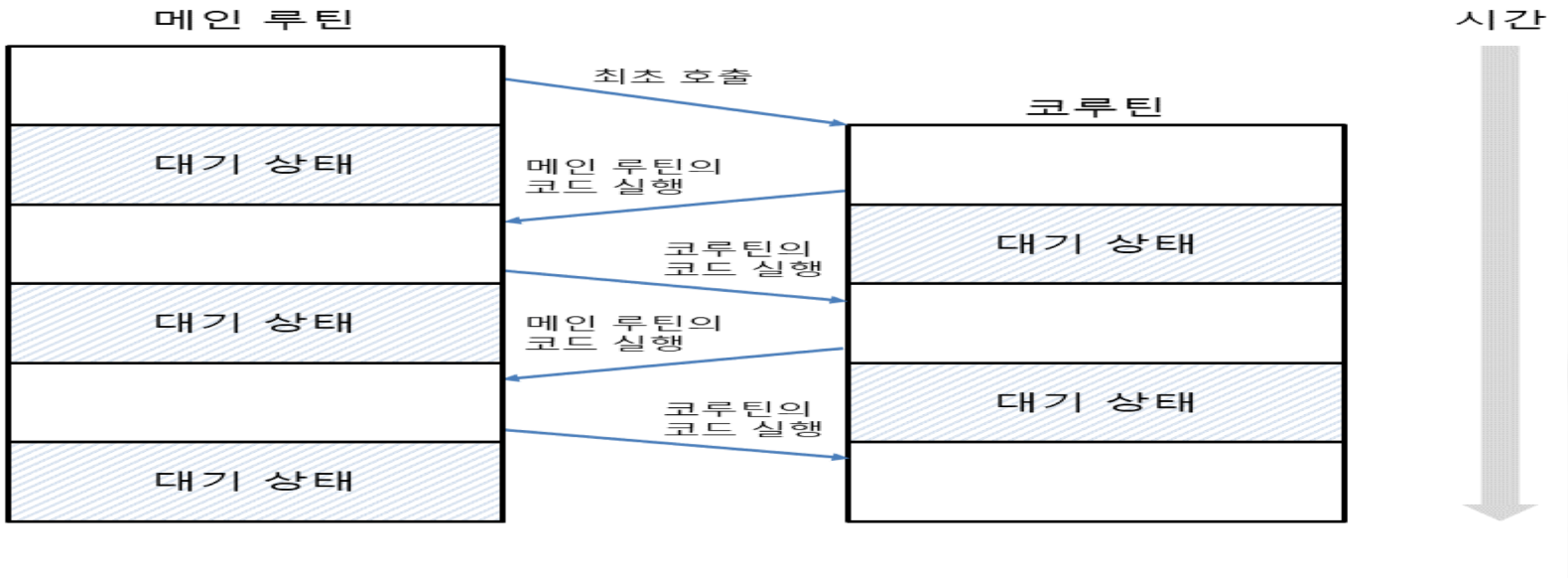


✓ 위의 코드와 같은 경우 calc()가 메인 루틴이 되고 add()가 서브 루틴

✓ 메인 루틴이 먼저 시작하고 서브 루틴이 수행을 완료하고 종료되면 메인 루틴의 나머지 부분이 수행

특별한 기능

- coroutine(coroutine)
  - ✓ cooperative routine를 의미하는데 서로 협력하는 루틴이라는 의미로 메인 루틴과 서브 루틴처럼 종속된 관계가 아니라 서로 대등한 관계이며 특정 시점에 상대방의 코드를 실행
  - ✓ coroutine은 함수가 종료되지 않은 상태에서 메인 루틴의 코드를 실행한 뒤 다시 돌아와서 coroutine의 코드를 실행
  - ✓ 일반 함수는 호출을 하면 코드를 한 번만 실행할 수 있지만 coroutine은 코드를 여러 번 실행 가능



특별한 기능

- coroutine
  - ✓ coroutine은 제네레이터의 특별한 형태
  - ✓ 제네레이터는 yield로 값을 발생시켰지만 coroutine은 yield로 값을 받아올 수 있음
  - ✓ coroutine에 값을 보내면서 코드를 실행할 때는 send Method를 사용
  - ✓ send Method가 보낸 값을 받아오려면 yield를 괄호로 묶어준 뒤 변수에 저장

```
def tot_coroutine():
    tot = 0
    while True:         # coroutine을 계속 유지하기 위해 무한 루프 사용
        x = (yield)     # coroutine 바깥에서 값을 받아옴
        tot = tot + x
        print('현재까지의 합:', tot)

co = tot_coroutine()
next(co)        # coroutine 안의 yield까지 코드 실행(최초 실행)

co.send(1)     # coroutine에 숫자 1을 보냄
co.send(2)     # coroutine에 숫자 2을 보냄
co.send(3)     # coroutine에 숫자 3을 보냄
```

```
현재까지의 합: 1
현재까지의 합: 3
현재까지의 합: 6
```

특별한 기능

- coroutine
  - ✓ yield 뒤에 값(변수)을 지정한 뒤 괄호로 묶어주면 값을 받아오면서 바깥으로 값을 전달
  - ✓ yield를 사용하여 바깥으로 전달한 값은 next 함수(__next__ Method)와 send Method의 반환 값으로 리턴

    변수 = (yield 변수)

    변수 = next(coroutine인스턴스)

    변수 = coroutine인스턴스.send(값)

특별한 기능

- coroutine

```
def tot_coroutine():
    total = 0
    while True:
        x = (yield total)    # coroutine 바깥에서 값을 받아오면서 바깥으로 값을 전달
        total += x

co = tot_coroutine()
print(next(co))      # 0: coroutine 안의 yield까지 코드를 실행하고 coroutine에서 나온 값 출력

print(co.send(1))    # 1: coroutine에 숫자 1을 보내고 coroutine에서 나온 값 출력
print(co.send(2))    # 3: coroutine에 숫자 2를 보내고 coroutine에서 나온 값 출력
print(co.send(3))    # 6: coroutine에 숫자 3을 보내고 coroutine에서 나온 값 출력
```

```
0
1
3
6
```

특별한 기능

- coroutine
  - ✓ coroutine은 실행 상태를 유지하기 위해 while True:를 사용해서 끝나지 않는 무한 루프로 동작
  - ✓ coroutine을 강제로 종료하고 싶다면 close Method를 사용
  - ✓ coroutine 인스턴스에서 close Method를 호출하면 coroutine이 종료될 때 GeneratorExit 예외가 발생
  - ✓ 이 예외를 처리하면 coroutine의 종료 시점을 알 수 있음

특별한 기능

- coroutine

```
def number_coroutine():
    try:
        while True:
            x = (yield)
            print(x, end=' ')
    except GeneratorExit:     # coroutine이 종료 될 때 GeneratorExit 예외 발생
        print()
        print('coroutine 종료')

co = number_coroutine()
next(co)

for i in range(20):
    co.send(i)

co.close()
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
coroutine 종료
```

특별한 기능

- Decorator
  - ✓ Class를 만들면서 Method에 데코레이터를 지정할 때는 self를 주의
  - ✓ 인스턴스 Method는 항상 self를 받으므로 데코레이터를 만들 때도 wrapper 함수의 첫 번째 매개변수는 self로 지정(Class Method는 cls).
  - ✓ func을 호출할 때도 self와 매개변수를 그대로 대입
  - ✓ 매개변수가 있는 데코레이터: 데코레이터를 호출할 때 매개변수를 대입할 수 있도록 하는 방법으로 Flask에서 URL 매핑을 할 때 이런 방식을 사용하는데 데코레이터의 매개변수를 처리하는 함수로 감싸고 함수를 매개변수로 처리하는 함수를 만든 후 실제 수행되는 함수를 생성해야 함

특별한 기능

❖ Decorator

```
def is_multiple(x):                 # 데코레이터가 사용할 매개변수를 지정
    def real_decorator(func):       # 호출할 함수를 매개변수로 받음
        def wrapper(a, b):          # 호출할 함수의 매개변수와 똑같이 지정
            r = func(a, b)          # func을 호출하고 반환값을 변수에 저장
            if r % x == 0:          # func의 반환값이 x의 배수인지 확인
                print('{0}의 반환값은 {1}의 배수입니다.'.format(func.__name__, x))
            else:
                print('{0}의 반환값은 {1}의 배수가 아닙니다.'.format(func.__name__, x))
            return r                # func의 반환값을 반환
        return wrapper              # wrapper 함수 반환
    return real_decorator           # real_decorator 함수 반환

@is_multiple(3)     # @데코레이터(인수)
def add(a, b):
    return a + b

print(add(10, 20))
print(add(2, 5))
```

```
add의 반환값은 3의 배수입니다.
30
add의 반환값은 3의 배수가 아닙니다.
7
```

Module

- Module
  - ✓ Module
    - 독자적인 기능을 갖는 구성 요소를 의미
    - 서로 연관된 작업을 하는 코드들의 모임으로 작성 중인 Module의 크기가 어느 정도 커지게 되면 관리 가능한 작은 단위로 다시 분할
    - 파이썬에서는 개별 소스 파일을 일컫는 말
    - 파이썬 프로그램으로 작성된 파일도 가능하고 C 나 Fortran 등으로 만든 파이썬 확장 파일도 Module이 될 수 있음
    - __name__: 모듈의 이름을 나타내는 내장 변수
  - ✓ Module의 종류
    - 표준 Module: 파이썬에 포함된 Module
    - 사용자 정의 Module: 프로그래머가 직접 작성한 Module
    - 서드 파티(3rd Party) Module: 파이썬 재단도 프로그래머도 아닌 다른 프로그래머 또는 업체에서 제공한 Module

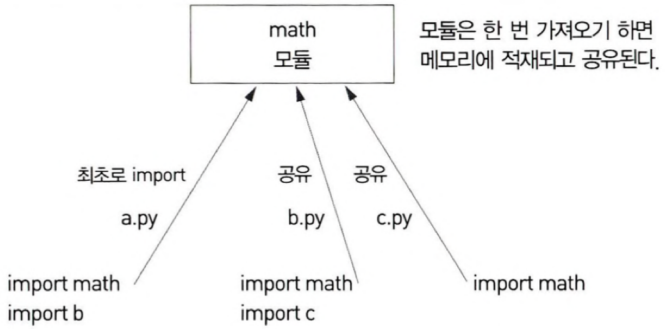
Module

- Module
  - ✓ Module 사용
    - Module을 사용할 수 있도록 추가하는 방법은 import 모듈이름
    - 확장자는 생략
    - 파일에 있는 변수나 Method는 파일명.변수 또는 파일명.함수()로 호출
    - import 방법
      - import math: Module의 이름을 이용해서 가져온 경우로 math.해서 구성 요소를 사용
      - from math import sin, cos, pi: math Module에서 sin, cos, pi 만 가져온 경우로 math Module에서 sin, cos, pi 만 현재 모듈에 포함시키는 것으로 math.을 생략하고 사용
      - from math import * : math Module의 모든 구성 요소를 현재 모듈에 포함시킴
      - import math as ma: math라는 Module 이름 대신에 ma라는 이름 사용
      - from math import pi as py: pi 대신에 py 사용
    - Module 이름이 문자열로 되어 있는 경우 __import__(Module이름) 으로 가져오는 것이 가능

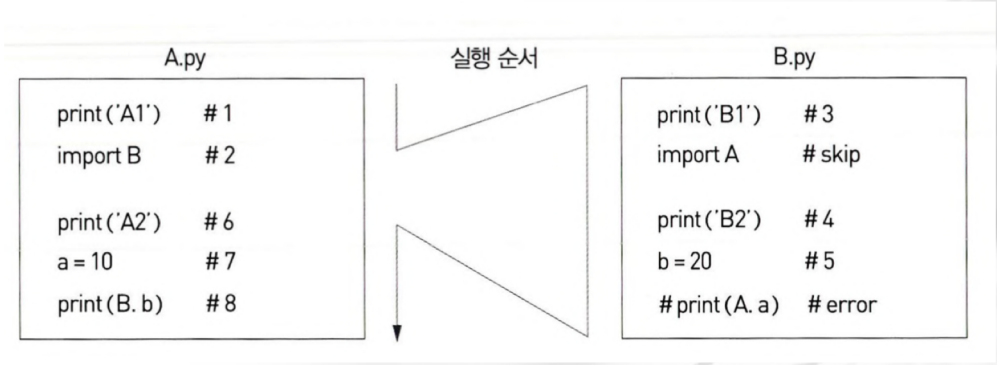
Module

❖ 모듈의 공유

- ✓ Module은 가져오면 메모리에 적재된 상태가 되기 때문에 다시 가져 올 필요가 없음
- ✓ 모듈 가져오기를 다시 수행해도 이미 가져온 모듈을 공유



- ✓ 외부에서 A를 가져오려 할 경우 B.py 파일에서의 import A 문은 실행되지 않고 넘어가지 때문에 재귀적인 무한 반복에 빠지지 않는다.



Module

- 모듈 확인
  - ✓ 모듈을 강제로 재적재 하고자 하는 경우는 imp 모듈의 reload 함수에 모듈 이름을 대입하면 됨
  - ✓ 현재 가져온 모듈 확인 – sys.modules 에 dict로 저장되어 있음
  - ✓ 클래스나 함수는 그들이 속한 모듈 이름을 갖는 __module__ 속성을 가지고 있음

```
# 적재된 모듈 이름 확인
import sys
from math import sin

print(type(sys.modules))
print()
#print('₩n'.join(sys.modules.keys()))

print()
print(sin.__module__)

print()
a = 1
current_module = sys.modules[__name__] # 현재 모듈 얻어내기
print(getattr (current_module, 'a')) # 현재 모듈에서 a의 값 얻어내기
```

Module

- 모듈을 가져오는 순서
  - ✓ 파이썬은 Module 가져오기를 수행하면 지정한 디렉토리에서 찾아가기 시작
  - ✓ sys.path 변수에 그 순서가 기재

```
import sys
for path in sys.path:
    print(path)
```

```
/Users/munseokpark
/opt/anaconda3/lib/python37.zip
/opt/anaconda3/lib/python3.7
/opt/anaconda3/lib/python3.7/lib-dynload

/opt/anaconda3/lib/python3.7/site-packages
/opt/anaconda3/lib/python3.7/site-packages/aeosa
/opt/anaconda3/lib/python3.7/site-packages/IPython/extensions
/Users/munseokpark/.ipython
```

Module

- 모듈을 호출하는 경로를 추가
  - ✓ 소스 코드에 설정

    sys.path.append("검색할 경로")
  - ✓ 환경 변수에 추가 – 터미널에서 수행
    - 윈도우

      set PYTHONPATH = 검색경로;
    - Mac

      export PYTHONPATH = 검색경로
    - 리눅스의 환경 변수에 추가

      C Shell 인 경우: setenv PYTHONPATH 검색경로

      bash Shell 인 경우: export PYTHONPATH = 검색경로

Module

- 직접 작성한 모듈 사용
  - ✓ mymath.py – 소스파일을 작성하는 경로에 작성

```
mypi = 3.14

def area(r):
    return mypi * r * r
```

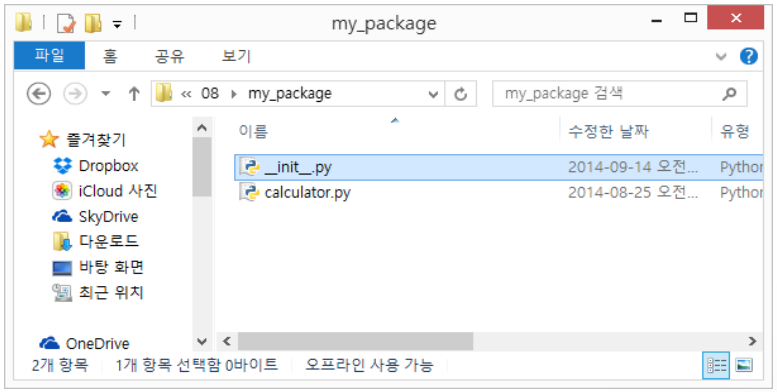
  - ✓ test.py

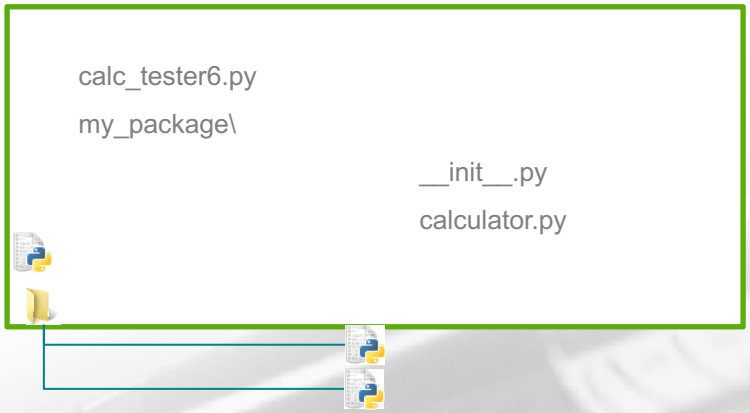
```
import sys
import mymath

#앞에서 만든 파일 경로 추가
sys.path.append("./")
print(mymath.mypi)
print(mymath.area(5))
```

Module

- package
  - ✓ Module을 모아놓는 디렉토리
  - ✓ Module 꾸러미로 해석하면 이해하기 편함
  - ✓ 디렉토리가 파이썬의 package로 인정받으려면 __init__.py 파일을 그 경로에 갖고 있어야 함
  - ✓ __init__.py 파일에는 package를 초기화하는 어떠한 파이썬 코드도 포함할 수 있음
  - ✓ package에서 특정한 Module을 가져올 때는 아래 그림과 같은 경우

    from my_package import calculator





Module

- site-packages
  - ✓ 파이썬의 기본 라이브러리 package 외에 추가적인 package를 설치하는 디렉토리
  - ✓ 각종 서드 파티 Module을 이 곳에 설치함

Module

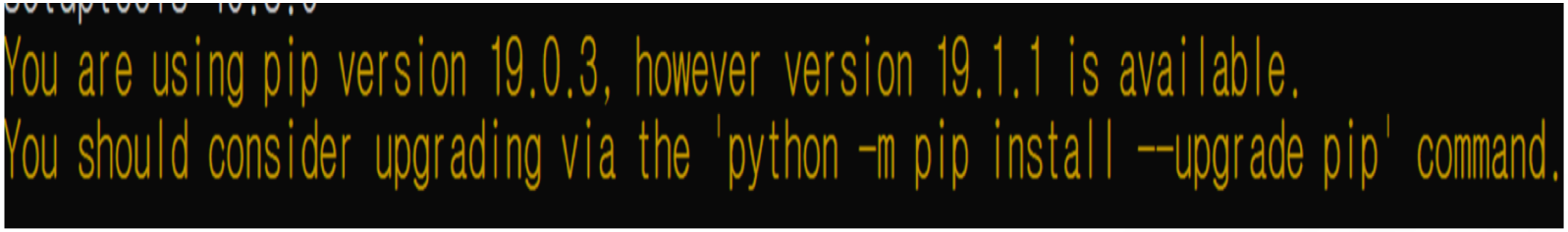
❖ 외부 Package 설치 – command 창에서 가능

pip install 외부Module이름

pip install 외부Module이름 –upgrade

pip install 외부Module이름=버전

pip install 외부Module이름>=버전 처럼 부등호 사용 가능



✓ 위와 같은 경고가 발생하는 경우에는

python -m pip install --upgrade pip 를 입력하셔서 pip를 업그레이드하면 설치가 가능

✓ pip 의 경우는 관리자 권한이 아니면 뒤에 --user를 붙여야 하는 경우가 있음

Module

- 기타 설치
  - ✓ anaconda
    - anaconda의 경우는 pip 대신에 conda를 사용
    - anaconda update
      - conda update --force conda
      - conda update anaconda
      - conda update conda
  - ✓ Pycharm을 사용하는 경우는 설정 메뉴에서 설치하는 것도 가능
  - ✓ Colab 의 경우는 !pip install 패키지이름 을 import 구문 위에 추가

Module

- package 관련 명령
  - ✓ 설치된 package 검색
    - pip list --format=columns : package 목록 출력
    - pip list --outdated --format=columns: package 목록과 마지막 업데이트 버전 출력
  - ✓ package 검색
    - pip search package이름
  - ✓ package 삭제
    - pip uninstall package이름 -y
  - ✓ 가상 환경의 package 설정 저장
    - pip freeze > 파일명
  - ✓ 설정 복원
    - pip install –r 파일명

Module

- 직접 설치
  - ✓ package 다운로드: 인터넷이 안 되는 환경에서 설치하기 위해서 다운로드

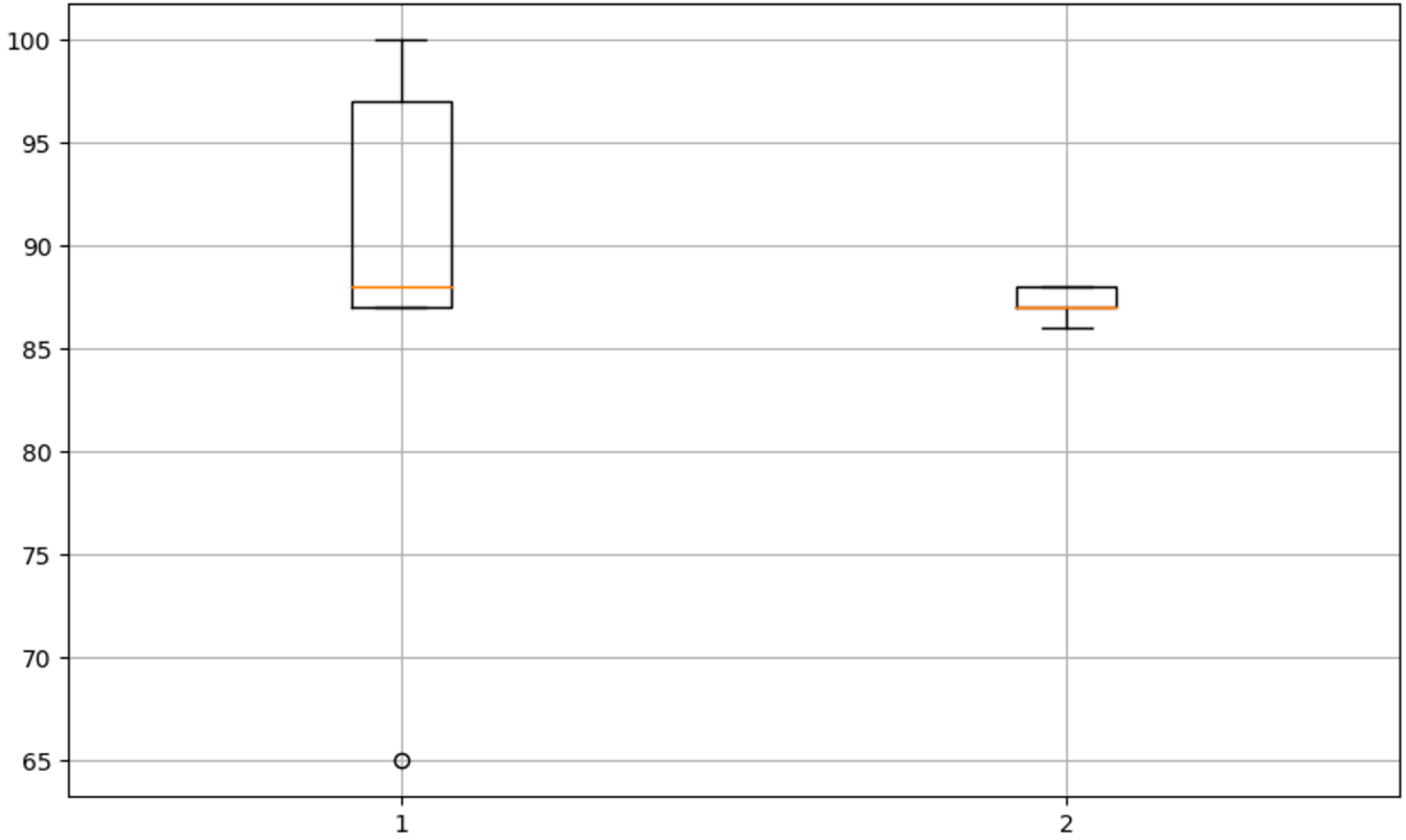
    pip download package이름
  - ✓ 직접 다운로드 받아서 설치하는 경우 의존되는 라이브러리를 자동으로 설치해 주지 않기 때문에 **pip install 파일명** 명령으로 순서대로 설치

Module

- matplotlib 패키지 사용
  - ✓ matplotlib package의 pyplot 이라는 Class의 정적 Method인 boxplot 이라는 Method를 이용하면 상자 그래프를 그릴 수 있음
  - ✓ matplotlib package는 python을 설치한 경우에는 제공되지 않기 때문에 설치해야 하고 anaconda를 설치한 경우에는 이미 설치되어 있음
  - ✓ 그래프를 그리는 순서
    - pyplot 의 정적 Method인 figure(figsize=(가로크기, 세로크기))를 이용해서 그래프 그릴 영역을 생성하고 변수에 저장
    - 그래프를 그리는 함수 호출
    - 옵션 설정
    - pyplot의 show 라는 정적 Method를 호출해서 화면에 그래프 출력
    - 첫번째 생성된 변수의 savefig를 이용해서 그림 파일로 저장

Module

- matplotlib 패키지 사용



Module

- matplotlib 패키지 사용
  - ✓ matplotlib 설치 확인

```
C:₩Users₩munseokpark>pip list --format=columns
Package           Version
----------------- -------
cycler            0.10.0
kiwisolver        1.1.0
matplotlib        3.1.1
numpy             1.16.4
pip               19.1.1
pyparsing         2.4.0
python-dateutil   2.8.0
setuptools        40.8.0
six               1.12.0
```

Module

- matplotlib 패키지 사용

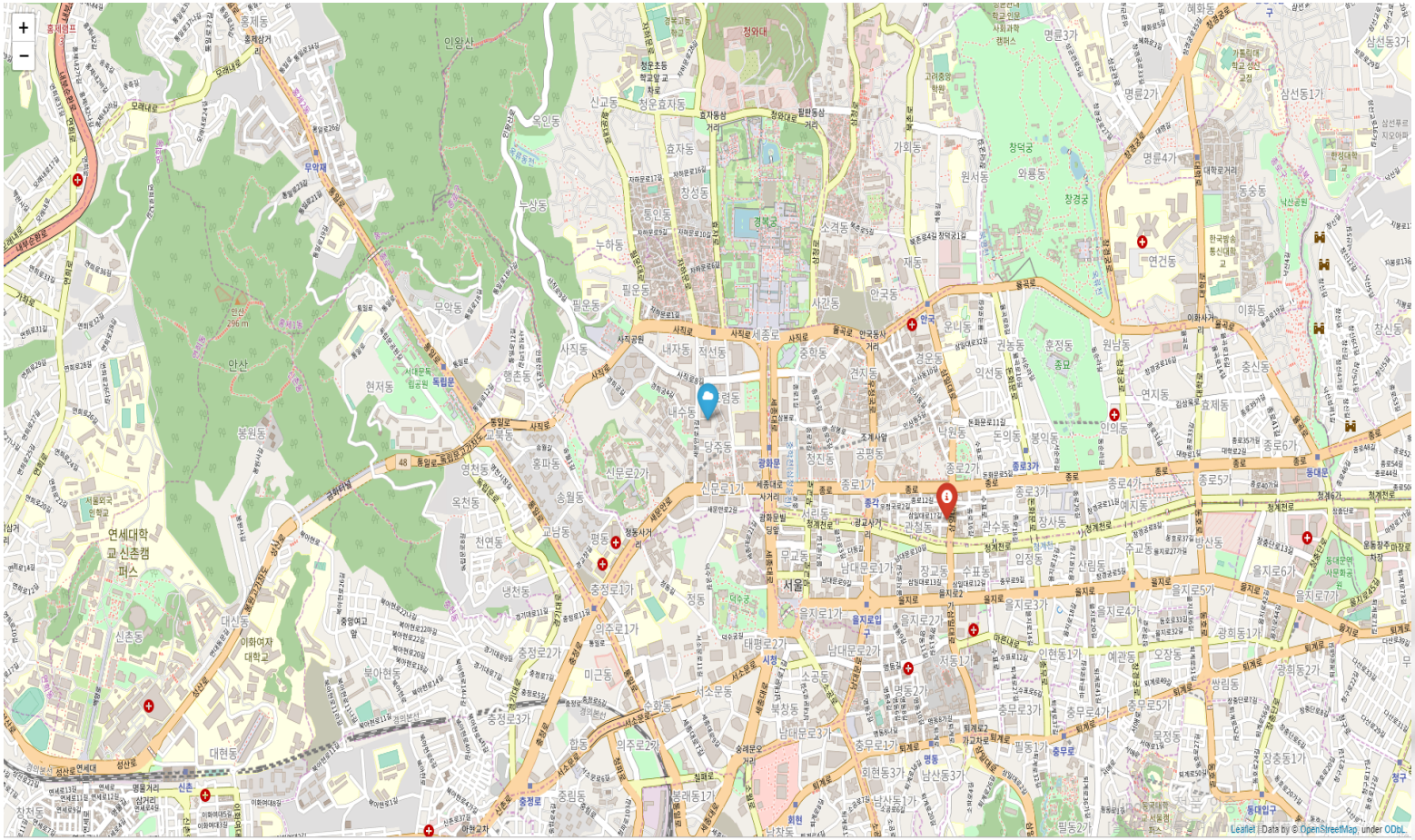
```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(10,6))
plt.boxplot(([100, 87, 97, 65, 88], [87,88,86,87,88]))
plt.grid()
plt.show()
fig.savefig("graph.png")
```

Module

- folium
  - ✓ folium 이라는 package는 지도를 출력하고 그 위에 마커나 도형을 그려서 지역별로 데이터를 표시하는 것이 가능
  - ✓ anaconda를 설치해도 기본적으로 설치되어 있지 않으므로 직접 설치해서 사용
  - ✓ 설치: pip install folium
  - ✓ 다운로드가 안되는 경우 https://github.com/itggangpae/python 에서 다운로드

Module

- folium



Module

❖ folium 사용 – 현재 작업 디렉토리에서 map.html 파일을 크롬에서 열어서 확인

```
import folium
m = folium.Map(location=[37.572656, 126.973304], zoom_start=15)
folium.Marker(location=[37.572656, 126.973304], popup="KB 국민카드",
              icon=folium.Icon(icon='cloud')).add_to(m)
folium.Marker(location=[37.569027, 126.987279], popup="메가IT",
              icon=folium.Icon(color='red')).add_to(m)
m
m.save("map.html")
```

파이썬 배포

- __main__.py
  - ✓ 디렉토리에 __main__.py 파일이 존재하는 경우 터미널에서 아래 명령을 수행하면 __main__.py 파일이 실행

    python 디렉토리이름
  - ✓ zip 파일로 압축되어 있는 경우에도 가능

    python zip파일이름

파이썬 배포

- 설치 파일 생성
  - ✓ distutils(Python Distribution Utility) 모듈은 파이썬 프로그램을 배포하고 설치할 때 사용하는 파이썬 표준 도구
  - ✓ 소스와 문서, 데이터, 스크립트 등을 한 번에 묶어서 배포할 수 있음
  - ✓ 최상위 디렉토리에 setup.py 파일을 생성

```
# setup.py
from distutils.core import setup

setup(name = "spam", version = "1.0",
description = " setup test",
author = "Park MoonSeok",
author_email = "ggangpae1@gma11.com",
url = "http://pythonworld.net/",

py_modules = ['A1, 'B', 'mymath02'],
packages = ['Speech',
'Speech/Recognition', 'Speech/S1gnalProcessi ng1, 1Speech/Synthesis ' ] ,
# package_dir = {'Speech': 'Speech?'},
package_data = {'Speech': ['images/*.jpg']},
data_files = [ ('data' , ['Speech/images/al.jpg1])],
)
```

파이썬 배포

- 설치 파일 생성
  - ✓ setup 함수의 옵션
    - 개별적인 모듈을 지정하고자 하는 경우 py_modules를 사용하는데 ['A', 'B']와 같이 확장자를 뺀 모듈 이름만 추가하면 되고 파일의 위치가 패키지 안에 있을 경우는 package.module과 같이 지정할 수 있음
    - 패키지를 포함하려면 인수 packages를 사용하는데 하위 패키지는 루트 패키지를 지정하면 자동으로 포함되지 않으므로 각각 지정해야 하는데 만일 패키지 이름과 패키지가 저장된 디렉토리가 다를 경우에는 옵션 package_dir을 사용 – Speech 는 Speech2에 저장되어 있음

      package_dir = {'Speech' : 'Speech2 '}
    - 패키지에 포함되지 않은 데이터 파일을 지정하려면 인수 data_files를 사용
    - '설치경로,:[파일목록]

파이썬 배포

- 배포한 만들기 와 설치
  - ✓ 소스 배포판
    - 생성 – dist 디렉토리에 생성됨

      python setup.py sdist
    - 소스 배포 설치

      python setup.py install
  - ✓ 바이너리 배포판
    - 생성 – 소스가 아닌 실제 설치할 결과물 생성

      python setup.py bdist
    - 윈도우용 바이너리 배포함
    - python setup.py bdist_wininst

파이썬 배포

- 파이썬이 없어도 실행 가능한 배포판 생성
  - ✓ Py2exe
    - 파이썬 프로그램을 윈도우용 실행 파일로 변환
    - 파이썬이 시스템에 설치되어 있지 않아도 실행할 수 있음(http://www.py2exe.org).
  - ✓ py2app
    - 맥 OS X에 독립 실행형(Standalone) 응용 프로그램을 생성
    - https://pypi.python,org/pypi/py2app/
  - ✓ cx_Freeze
    - 독립 실행형 실행 파일을 만들어 주는데 플랫폼에 독립적으로 실행
    - http://cx-freeze. sourceforge, net/