

# Python Programming

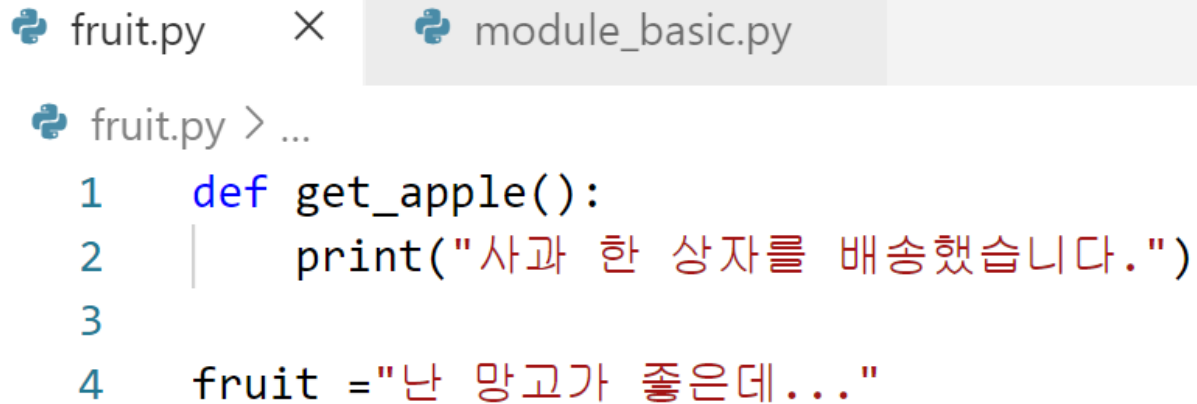
For Beginner

# Module, Class, Object

For Python 3

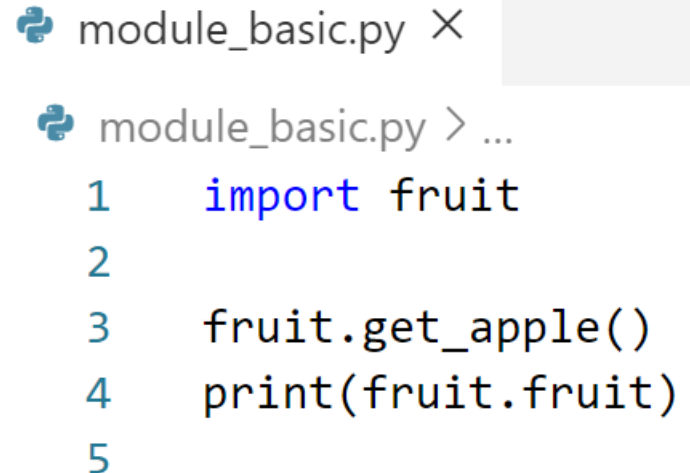
# 모듈

1. 함수나 변수, 클래스 등의 python 코드로 이뤄진 파일
2. Import를 통해 사용
  - 파일 명만 사용
3. 점(.) 연산자로 멤버 접근



```
fruit.py  ×  module_basic.py

fruit.py > ...
1  def get_apple():
2      print("사과 한 상자를 배송했습니다.")
3
4  fruit = "난 망고가 좋은데..."
```



```
module_basic.py  ×

module_basic.py > ...
1  import fruit
2
3  fruit.get_apple()
4  print(fruit.fruit)
5
```

# 클래스

1. 메서드와 데이터를 함께 다루기 위한 자료형
2. 소형 모듈을 만드는 청사진이나 정의
3. 클래스 내의 멤버는 점(.) 연산자로 접근
4. 모듈과 클래스 비교
  - 모듈은 한 프로그램에서 딱 한 번 존재
  - 클래스는 한 프로그램에서 여러 모양으로 존재 가능
5. 형식
  - Self – 관례, 호출된 객체 자신 전달

```
Class ClassName:  
    def _init_(self, param):  
        클래스 본문
```

```
class Fruit:  
    def __init__(self):  
        self.fruit = "난 망고가 좋은데..."  
  
    def get_apple(self):  
        print("사과 한 상자를 배송했습니다.")
```

# 객체 (Object)

## 1. Instantiate (인스턴스화)

- 모듈의 임포트에 해당
- 결과는 객체

## 2. 인스턴스화 형식

- 변수 = ClassName(object)

## 3. 생성자

- 객체 생성에 사용
- def \_\_new\_\_(cls)
- 클래스 자체를 받으며 할당
- 자동으로 실행되므로 생략

## 4. 초기자

- def \_\_init\_\_(self)
- 객체 내부에서 사용할 속성 초기화
- 객체 생성시 전달되는 값 검증 수행

song\_class.py ✕

class\_call\_song.py

song\_class.py > ...

```
1 class Song(object):
2
3     def __init__(self, lyrics):
4         self.lyrics = lyrics
5
```

song\_class.py

class\_call\_song.py ✕

class\_call\_song.py > ...

```
1 from song_class import Song
2
3 happy_bday = Song(["생일 축하 합니다.",
4                   "사랑하는 귀도 반 로쌌",
5                   "생일 억수로 축하 합니다."])
6
7 bulls_on_parade = Song(["조개껍질 묶어",
8                         "그녀의 목에 걸고"])
9
10 happy_bday.sing_a_song()
11
12 bulls_on_parade.sing_a_song()
```

# Object Oriented Programming

For Python 3

# OOP 용어 정리

1. class 선언
2. object
3. def
4. self
5. inheritance
6. composition
7. attribute
8. is a – 연어 is a 물고기
9. Has a – 연어 has a 지느러미

# 객체와 클래스의 관계

1. 객체 = 특성 + 행위 (동작) + 정체성
  - 자동차의 특성 : 색상, 타이어 수, 에어백의 수
  - 자동차의 동작: 전진, 후진, 멈추기 등
  - 자동차의 정체성: GV80
2. 클래스 = 데이터 구조 + 메서드
  - 객체의 설계도, 틀, 청사진.
  - 클래스로 구현한 객체 → 인스턴스
  - 객체를 만드는 과정 → 인스턴스 화



# 상속

1. 부모 클래스의 기능을 자식 클래스가 모두 물려 받음.

2. 기본 형식

- 자식 클래스 선언 시 괄호로 부모 클래스 포함
- 부모 클래스 기능을 기술할 필요 없음

```
class ParentClass:
```

```
    ... 내용 ...
```

```
class ChildClass(ParentClass):
```

```
    ...내용...
```

```
class Country:
```

```
    name = '국가명'
```

```
    population = '인구'
```

```
    capital = '수도'
```

```
    def show(self):
```

```
        print('국가 클래스 메소드입니다.')
```

```
class Korea(Country):
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def show_name(self):
```

```
        print('국가 이름은 : ', self.name)
```

# 상속 – 부모와 자식이 상호작용하는 3가지 방법

1. 자식이 암시적으로 부모의 동작을 수행 한다.
  - 기본 클래스의 함수를 모든 서브 클래스에서 실행 가능
2. 자식이 부모의 동작을 재정의(override) 한다.
  - 자식에서 명시적으로 같은 이름의 함수 정의
3. 자식이 부모의 동작을 대체 한다.
  - 재정의의 특수한 케이스
  - 부모 클래스의 함수 실행 전후에 다른 동작 실행이 필요한 경우
  - 내장함수 super 활용 → 다른 클래스의 동작이 필요할 때 사용

# 합성

1. 클래스와 모듈을 이용해 상속의 효과 내기.
  - 합성 대상이 꼭 클래스 일 필요는 없다.
2. 기능을 교체나 바꾸는 작업에 적당
  - 소유자 클래스가 합성 대상 갖는 관계
3. 소유자 클래스에서 명시적으로 함수 정의

# 상속 vs. 합성

## 1. 코드 재사용 문제 해결 장치.

- 상속 – 기본 클래스의 기능을 묵시적으로 쓸 수 있는 체계 제공.
- 합성 – 모듈과 다른 클래스의 함수 호출 기능

## 2. 선택을 위한 3가지 기준

- 다중 상속 사용 금지
- 코드가 서로 관련 없는 위치나 상황에서 쓰이는 경우, **합성**
- 공통 개념이 들어 맞고 명확하게 연관된 재사용할 수 있는 코드가 있다면, **상속**

## 3. OOP에서의 규칙은 법이 아니고 관습(Convention)

# is-a와 has-a

## 1. Is-a 관계

- 클래스 관계로 엮인 객체와 클래스
- 상속 관계
- 물고기와 연어

## 2. Has-a 관계

- 클래스와 객체가 서로 참조로 엮인 경우
- 합성 관계
- 연어와 아가미

#합성에 사용할 클래스

```
class Dron:
    def __init__(self, kind):
        self.kind = kind

    def fly(self):
        print('드론이 날아 옵니다')
```

# 객체합성

```
class Tesla():
    def __init__(self, dron_kind=''):
        if dron_kind:
            # Dron 객체를 Tesla의 인스턴스 멤버로 할당
            self.dron = Dron(dron_kind)
        else:
            #dron 인스턴스 멤버는 있지만 값은 없는 상태
            self.dron = None

    def get_dron(self, dron_kind):
        self.dron = Dron(dron_kind)

    def startflying(self):
        if self.dron:
            self.dron.fly()
        else:
            print('차에 드론이 없습니다.')
```

조!

# 추상 클래스(abstract class)

1. 추상 메서드를 하나 이상 가진 클래스
2. 자식 클래스에서 추상 메서드 구현 강제
3. abc 모듈 import
4. 형식

```
class AbstractClass(metaclass=ABCMeta):  
    @abstractmethod  
    def abstract_method(self):  
        pass
```

```
from abc import *
```

```
class AbstractCountry(metaclass=ABCMeta):  
    name = '국가명'  
    population = '인구'  
    capital = '수도'  
  
    def show(self):  
        print('국가 클래스의 메소드입니다.')  
    @abstractmethod #추상 메서드 추가  
    def show_capital(self):  
        print('국가의 수도는?')
```

수고했습니다!