기초 컴퓨터 프로그래밍

- Python 기본: 클래스, 모듈, 패키지 -

충남대학교 의과대학 의공학교실

구 윤 서 교수

[Remind] A python code example

```python
# Import modules
import tensorflow as tf
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# X data
x_data = list(range(-20,21))

# W data
np.random.seed(2020)

mu = 0
sigma = 20
n = len(x_data)

noises = np.random.normal(mu, sigma, n)

W_answer = 3
b_answer = -3

y_temp = list(np.array(x_data)*W_answer + b_answer)
y_data = list(np.array(y_temp) + np.array(noises))
```

모듈 √

변수

√ 함수

값

연산자

[Remind] 함수 사용 시 장점

- 다수의 개발자가 프로그램을 기능 별로 나누어서 작성하고 통합하는 경우가 잦음.

- 프로그램을 기능 별로 나누는 방법
  - 함수 (Function)
  - 클래스 (Class)
  - 모듈 (Module)
  - 패키지 (Package)

[Remind] 메모리(Memory)

- 메모리(Memory): 값을 실제로 저장하는 물리적인 공간

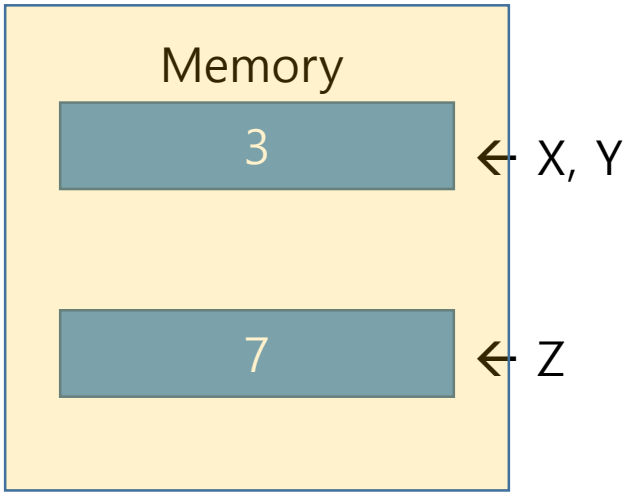
```
In [9]: X = 3
        Y = 3
        Z = 7
        print(X,Y,Z)
        print(id(X),id(Y),id(Z))
```

```
3 3 7
1927966240 1927966240 1927966368
```

메모리의 물리적 주소

√

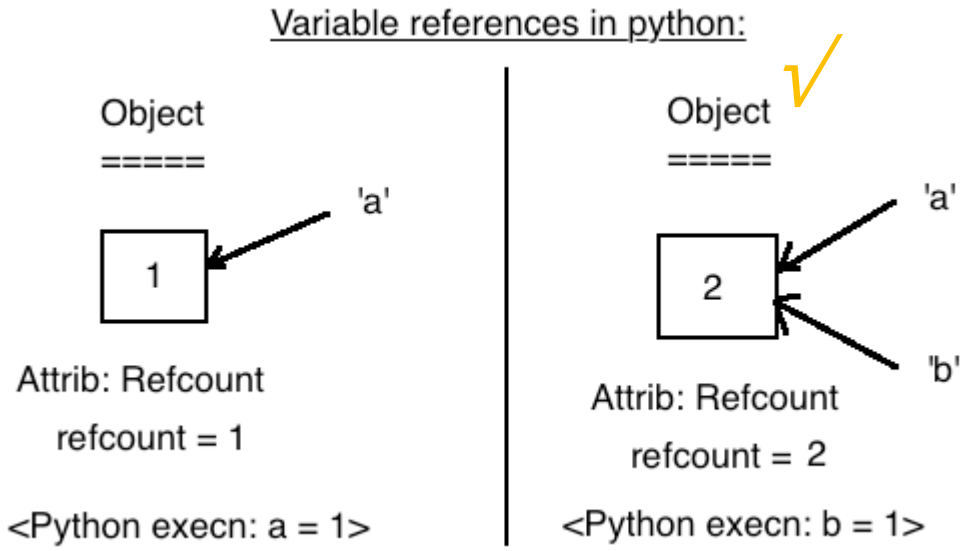
Python의 모든 자료형은 객체



[Remind] Python memory management process

- Python memory management process

```
a=1
b=1
c=2
```



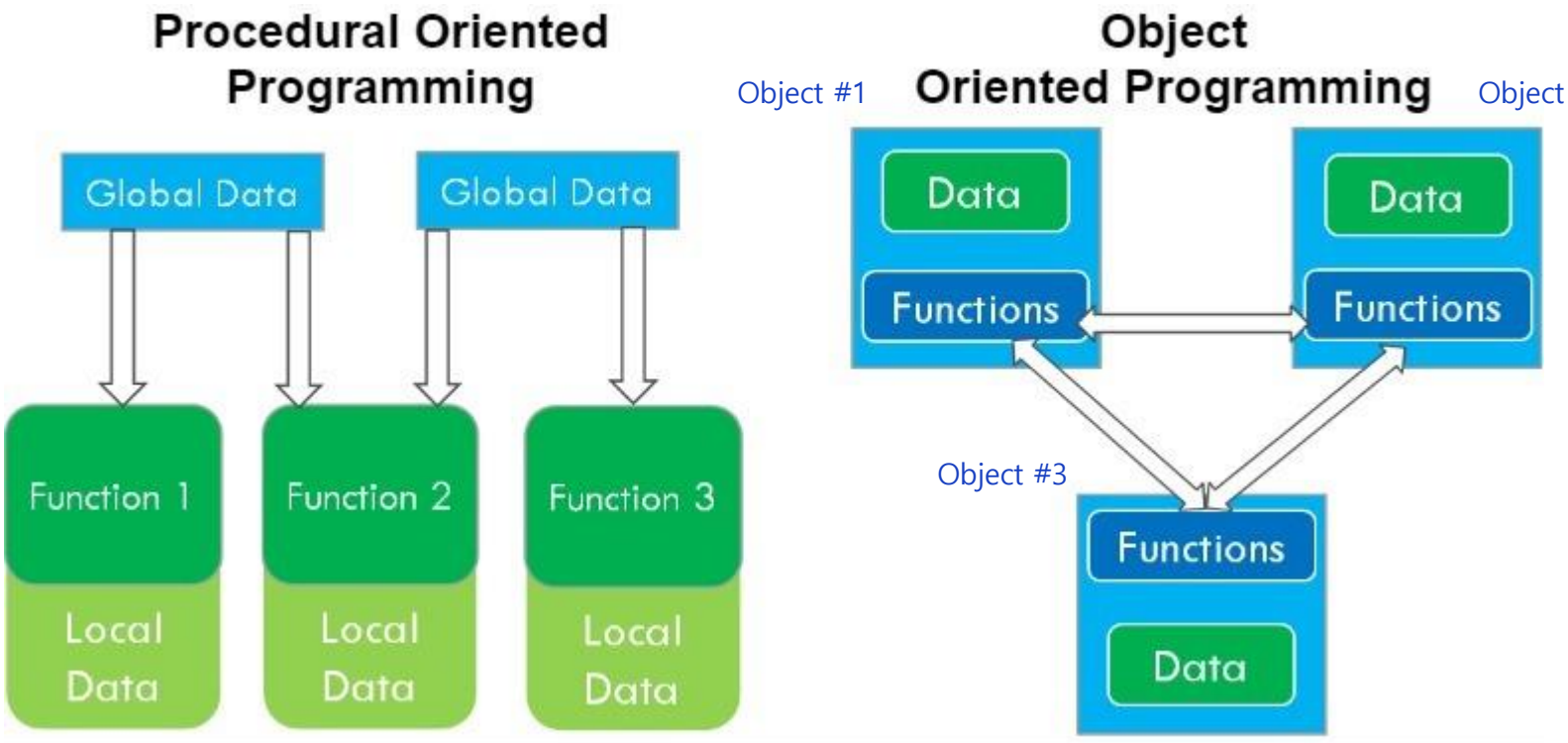
https://www.codementor.io/arch/variable-references-in-python-u9z8j2gk0

[Remind] 숫자형

```
x = 10
print(type(x))
<class 'int'>

y = 10.
print(type(y))
<class 'float'>

print(10/2) # int/int → float
5.0
```

절차 지향적 프로그래밍 (Procedure oriented programming)

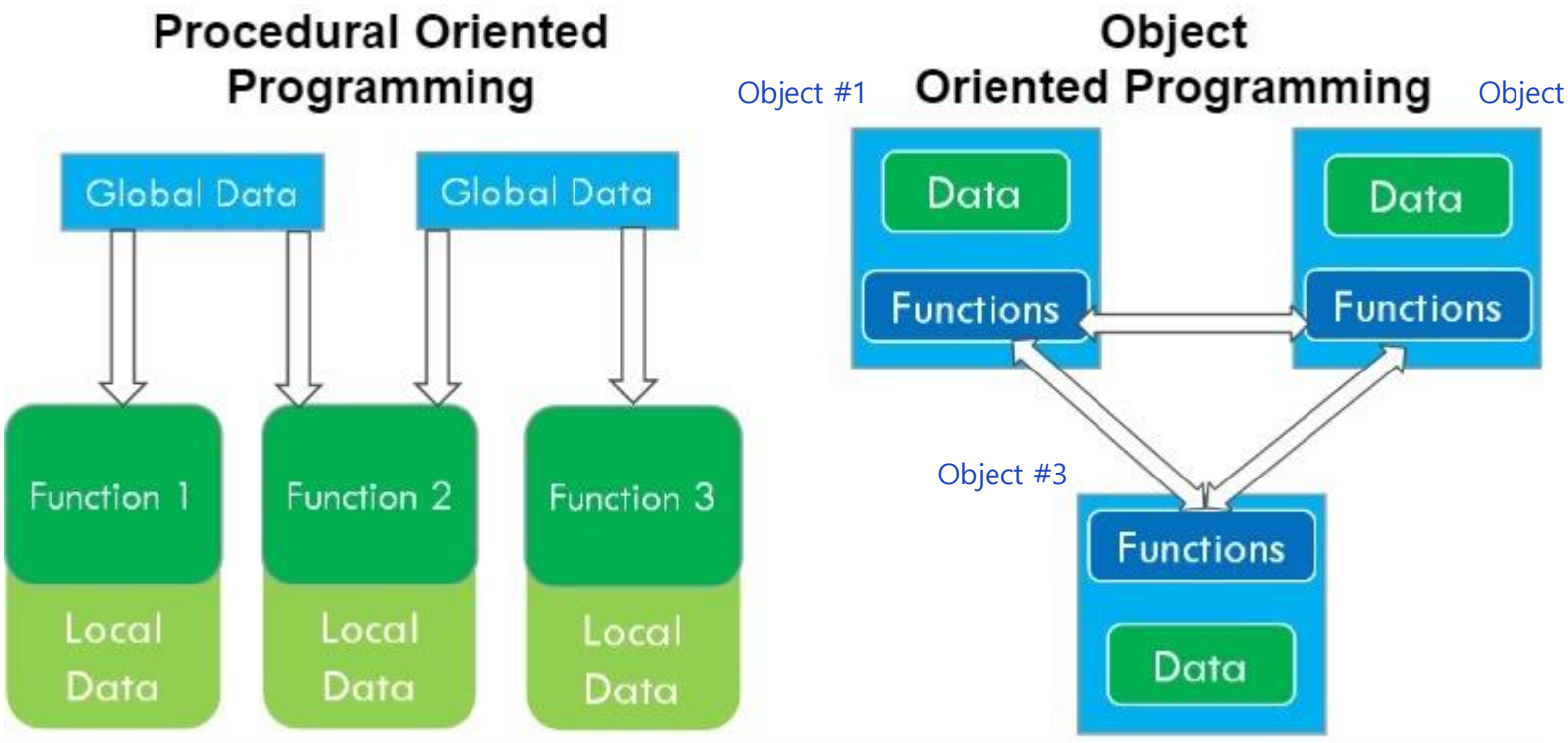
- 전체 프로그램을 이해하기 쉬운 함수(function) 단위로 나누고 함수를 호출하는 방식의 프로그래밍
- 큰 문제를 해결하기 위하여 작은 문제 단위로 나누어서 해결
- C, Pascal



Comparing procedural vs object-oriented programming. Source: Sakpal 2018.

객체 지향적 프로그래밍 (Object oriented programming)

- 사물을 객체(Object)로 추상화하여, 객체 간의 상호 작용을 기반으로 프로그래밍 하는 방법
- 객체를 생성한 후 그 객체들을 조합하여 문제 해결
- Python, JAVA, C++



Comparing procedural vs object-oriented programming. Source: Sakpal 2018.

POP vs. OOP

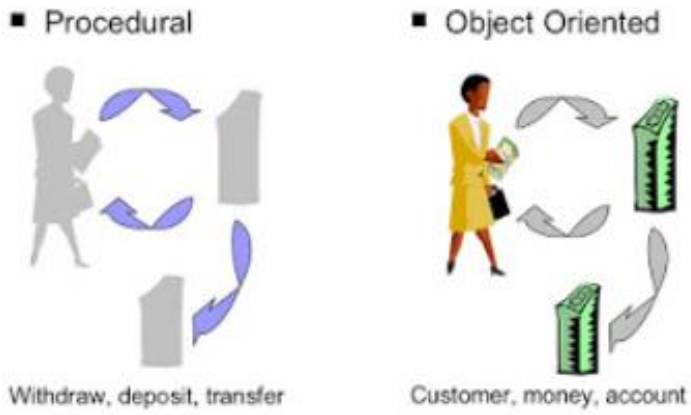
- 절차 지향 모델링 (POP)
  - 프로그램을 기능 중심으로 바라보는 방식 → "무엇을 어떤 절차로 할 것인가?"가 핵심
  - 어떤 기능을 어떤 순서로 처리하는가에 초점

- 객체 지향 모델링 (OOP)
  - 기능이 아닌 객체가 중심 → "누가 어떤 일을 할 것인가?"가 핵심
  - 객체를 도출하고 각각의 역할을 정의해 나가는 것에 초점

왜 객체지향 방식을 사용하는가?

- 모든 프로그램 구현에 필요한 것은 아님 (ex. C는 클래스 X)
- 기능으로 구분하는 경우 프로그램 개발 시 필요한 기능 (메뉴, 도구, 파일 열기/저장 등)이 많아 질수록 나누기가 어려움
- 객체로 구분하는 경우 실생활에 가까워 구분이 자연스럽고 보다 직관적임
- 객체지향 방식은 특히 사용자 이벤트 (ex. 마우스 클릭) 기반의 GUI 동작 구현에 용이

**GUI: Graphic User Interface*



POP (C, Pascal)	OOP (Python, Java, C++)
계산기 기능에 따라 구분 - **입력 기능** - **계산 기능** - **출력 기능**	계산기를 구성하는 객체로 구분 - **키보드** - **디스플레이** - **CPU**
각각의 기능을 함수로 구현	각각의 객체를 클래스로 구현

https://dzone.com/articles/top-5-object-oriented-programming-and-design-cours

POP vs. OOP

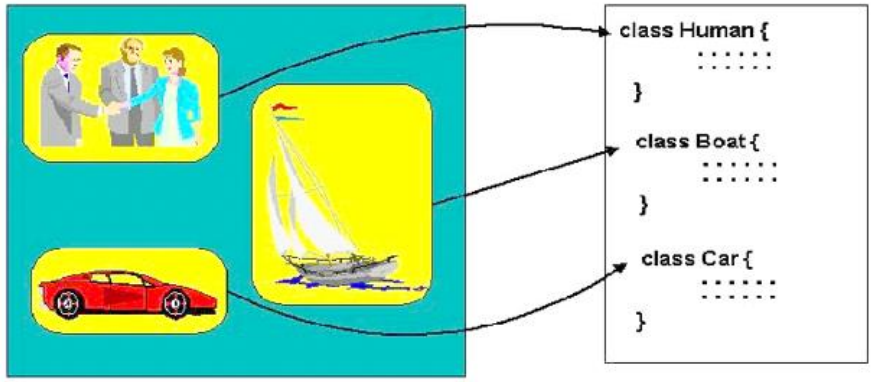
- 복잡한 프로그래밍의 경우 많은 기능을 수반하기 때문에 절차 지향보다는 객체 지향이 적합 → 각 객체가 하는 역할이 많아도, 많은 역할을 객체로 묶어서 프로그래밍이 용이해짐.

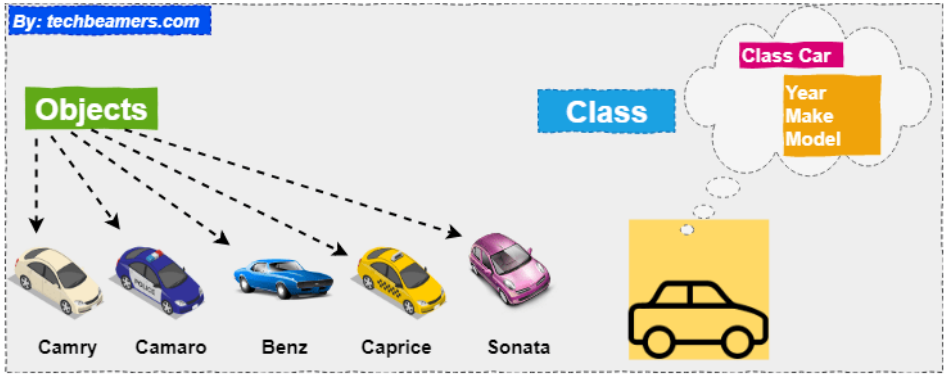
- 간단한 프로그래밍의 경우 비교적 많지 않은 기능을 수반하기 때문에 객체 지향보다는 절차 지향이 적합 → 작은 기능을 객체별로 나눌 경우, 오히려 복잡해져 필요한 메모리 공간이 많아지고 실행 속도가 느려짐.

객체 (Object)

- 객체는 클래스에 의해서 생성하고, 1개의 클래스는 무수히 많은 객체 생성 가능

- 객체는 데이터와 실행 가능한 기능으로 구성
  - 내부 데이터 변경을 막기 위하여 캡슐화 가능
  - 객체는 상속으로 기능을 확장 가능
  - 객체의 기능은 재정의(override) 가능



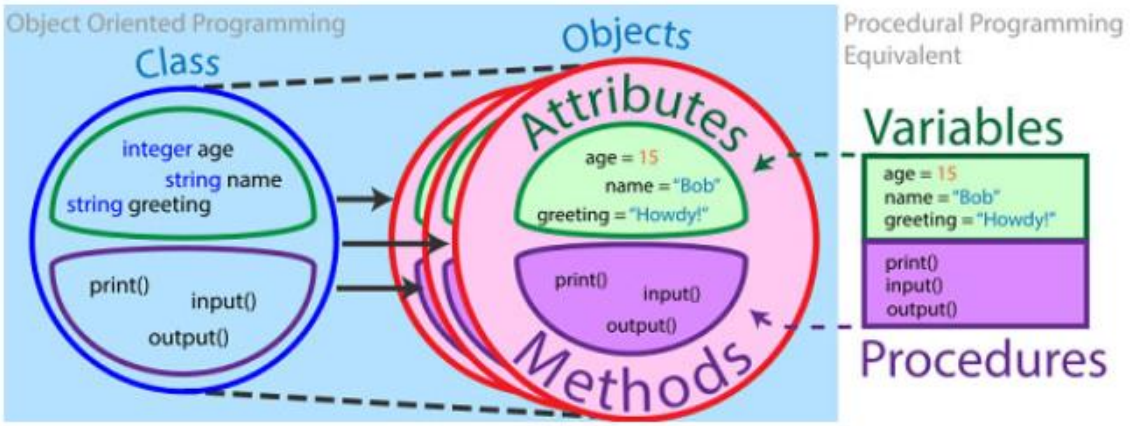


클래스 (Class)

- 클래스는 **객체를 만들기 위한 도구**
- 클래스의 구성

속성 (Attribute)	객체를 구성하는 데이터
메소드 (Method)	속성에 대해 어떤 기능을 수행하는 함수
생성자, 소멸자	객체 생성과 소멸 시 자동 호출되는 특별한 메소드
연산자 중복	연산자(+, -등) 기호를 이용하여 표현할 수 있도록 함

생성자는 **def__init__(self,...):** 으로 정의함
소멸자는 **def__del__(self,...):** 으로 정의함



https://yeasir007.blogspot.com/2019/01/introducing-object-oriented.html

클래스 예시 #1

```
class Dog :
    """ 강아지를 이름과 나이로 표현하는 클래스 """
    """ 속성은 강아지 객체를 구성하는 데이터임. """

    def __init__(self, name, age):
        """ 강아지 객체를 생성하는 생성자 메소드 """
        self.name = name
        self.age = age

    def bark(self):
        print(self.name, 'is barking')
```

속성

생성자

메소드 (method)

self: 메소드의 첫번째 인자 (관례적으로 사용(i.e. 다른 이름 사용가능))이며, 인스턴스의 매개변수를 전달할 때는 self 매개변수는 생략하고 전달

__init__ : 생성자(Constructor)라고 하며 객체가 생성될 때 자동으로 호출되는 메서드를 의미 (초기화 함수)

클래스 예시 #1

```
class Dog :
    # 강아지를 이름과 나이로 표현하는 클래스
    # 속성은 강아지 객체를 구성하는 데이터
    def __init__(self, name, age): # 생성자 (Constructor)
        self.name = name # 속성
        self.age= age
    def bark(self): # 메소드 (Method)
        print(self.name, 'is barking')

x = Dog('Jack', 3)
y = Dog('Daisy', 2)

x.bark()
y.bark()

print(x.name, 'is', x.age, 'years old.')
print(y.name, 'is', y.age, 'years old.')
```

```
Jack is barking
Daisy is barking
Jack is 3 years old.
Daisy is 2 years old.
```

클래스 예시 #2

- Windows 계산기 App 예제
  - 계산기에 숫자 3을 클릭하고 + 기호를 클릭한 후 4를 클릭하면 7을 출력
  - 다시 한 번 + 기호를 클릭하고 3을 클릭하면 10을 출력
  - 즉, 계산기는 이전에 계산한 결과를 항상 메모리에 저장해야 함

클래스 예시 #2

- 그런데 만일 한 프로그램에서 2대의 계산기가 필요한 경우? → 각 계산기는 각각의 결과 저장 및 유지 필요

- 해결 방법: 전역변수 사용 및 별도의 함수 생성

- 문제점: 계산기가 3개, 5개, 10개로 점점 더 많이 필요해진다면? → 전역 변수와 같은 기능을 하는 함수를 계속 추가 필요

```python
result1 = 0
result2 = 0

def add1(num):
    global result1
    result1 += num
    return result1

def add2(num):
    global result2
    result2 += num
    return result2

print(add1(3))
print(add1(4))
print(add2(3))
print(add2(7))
```

```
3
7
3
10
```

클래스 예시 #2

- 해결 방법: **클래스 사용**
  - **Calculator 클래스의 계산기 cal1, cal2 인스턴스가 각각의 역할을 수행**
  - 다른 계산기의 결과와 상관없이 독립적인 값 유지
  - **클래스를 사용하면 계산기 대수가 늘어나더라도 객체 생성만 하면 되기 때문에 함수를 사용하는 경우와 달리 매우 간단하게 구현 가능**

```python
class Calculator:
    def __init__(self):
        self.result = 0

    def add(self, num):
        self.result += num
        return self.result

cal1 = Calculator()
cal2 = Calculator()

print(cal1.add(3))
print(cal1.add(4))
print(cal2.add(3))
print(cal2.add(7))
```

```
3
7
3
10
```

클래스 실습

```python
result1 = 0
result2 = 0

def add1(num):
    global result1
    result1 += num
    return result1

def add2(num):
    global result2
    result2 += num
    return result2

print(add1(3))
print(add1(4))
print(add2(3))
print(add2(7))
```

```
3
7
3
10
```

[함수 사용]

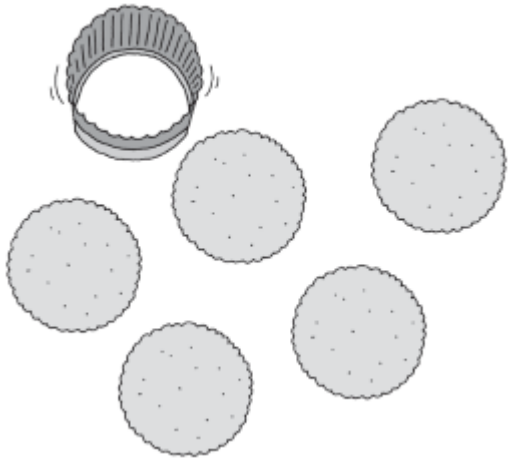
```python
class Calculator:
    def __init__(self):
        self.result = 0

    def add(self, num):
        self.result += num
        return self.result

cal1 = Calculator()
cal2 = Calculator()

print(cal1.add(3))
print(cal1.add(4))
print(cal2.add(3))
print(cal2.add(7))
```

```
3
7
3
10
```

[클래스 사용]

[용어 정리] 객체 (Object) vs. 인스턴스 (Instance)

- 인스턴스 (Instance): 클래스로 만든 객체
- a = Cookie() 이렇게 만든 a는 객체
- a 객체는 Cookie의 인스턴스
- 즉, 인스턴스라는 말은 특정 객체(a)가 어떤 클래스(Cookie)의 객체인지를 관계 위주로 설명할 때 사용
- "a는 인스턴스"보다는 "a는 객체"라는 표현이 어울리며 "a는 Cookie의 객체"보다는 "a는 Cookie의 인스턴스"라는 표현이 어울림



•과자 틀 → 클래스 (class)
•과자 틀에 의해서 만들어진 과자 → 객체 (object)

```
>>> class Cookie:
>>>     pass
```

```
>>> a = Cookie()
>>> b = Cookie()
```

Python Class 활용 객체 지향 프로그래밍

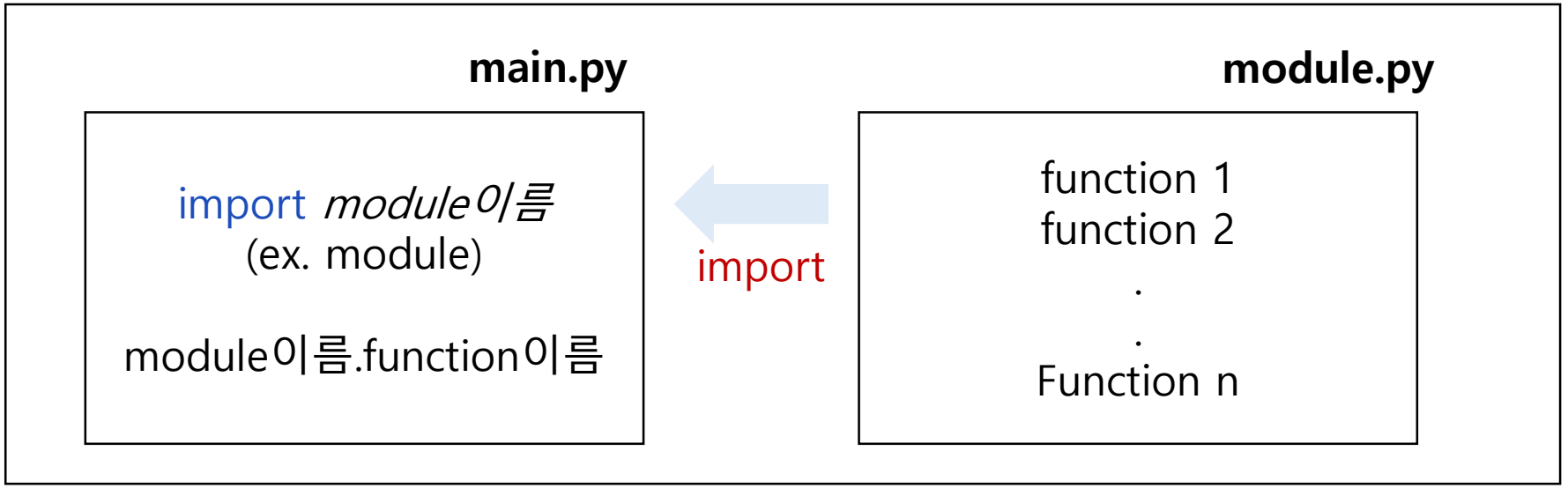
- Encapsulation (캡슐화), Inheritance (상속), Overriding (재정의)

https://www.codefires.com/implementation-object-oriented-programming-oop-python/

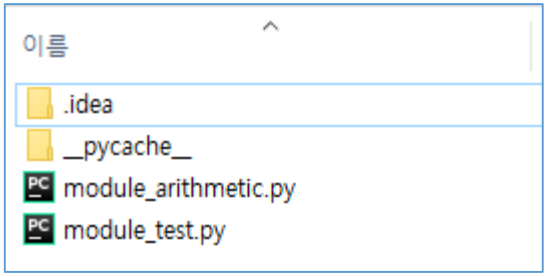
Module

- 재사용을 위해 작성한 Python 파일(.py)
- Module로 작성한 .py파일과 module을 사용하는.py을 동일 폴더에 저장
- import 문을 사용해서 Module을 호출 (또는 from)

[동일 폴더]



Module 실습 (@PyCharm)



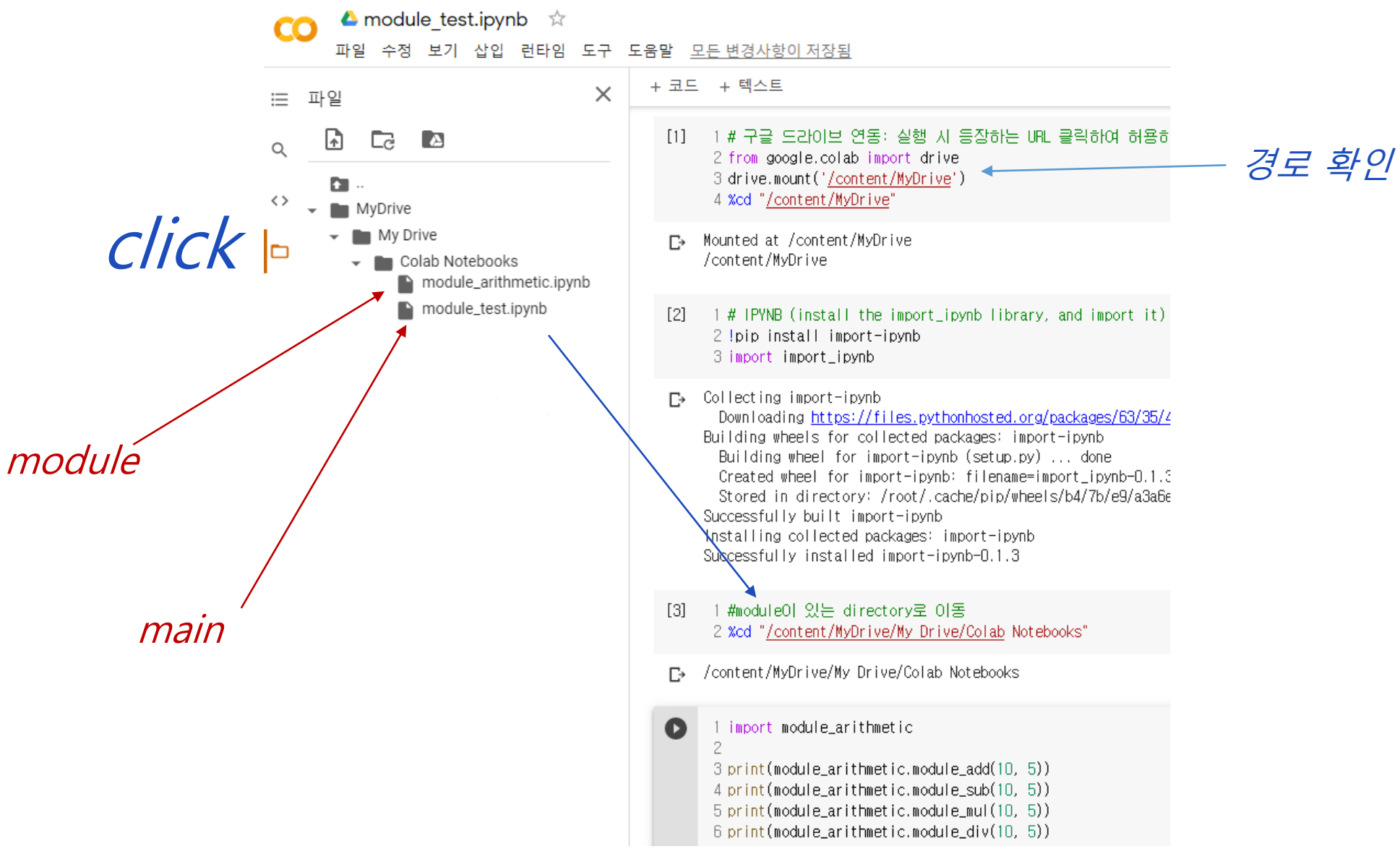
module_test.py

```
import module_arithmetic

print(module_arithmetic.module_add(10, 5))
print(module_arithmetic.module_sub(10, 5))
print(module_arithmetic.module_mul(10, 5))
print(module_arithmetic.module_div(10, 5))
```

module_arithmetic.py

```
def module_add(a, b):
    return a + b

def module_sub(a, b):
    return a - b

def module_mul(a, b):
    return a * b

def module_div(a, b):
    return a / b
```

Module 실습 (@Google Colab)



Module 실습 (@Google Colab)

[1]
```
# 구글 드라이브 연동: 실행 시 등장하는 URL 클릭하여 허용하면 인증KEY가 나타나고, 복사하여 URL아래 빈칸에 붙여넣으면 마운트에 성공
from google.colab import drive
drive.mount('/content/MyDrive')
%cd "/content/MyDrive"
```

```
Mounted at /content/MyDrive
/content/MyDrive
```

[2]
```
# IPYNB (install the import_ipynb library, and import it)
!pip install import-ipynb
import import_ipynb
```

```
Collecting import-ipynb
  Downloading https://files.pythonhosted.org/packages/63/35/495e0021bfdcc924c7cdec4e9fbb87c88dd03b9b9b22419444dc370c8a45/import-ipynb-0.1.3.tar.gz
Building wheels for collected packages: import-ipynb
  Building wheel for import-ipynb (setup.py) ... done
  Created wheel for import-ipynb: filename=import_ipynb-0.1.3-cp36-none-any.whl size=2976 sha256=441cd637cf7cb84f88eb1fb214d469cc2aff2439d0c77916acf7361f7e186961
  Stored in directory: /root/.cache/pip/wheels/b4/7b/e9/a3a6e496115dffdb4e3085d0ae39ffe8a814eacc44bbf494b5
Successfully built import-ipynb
Installing collected packages: import-ipynb
Successfully installed import-ipynb-0.1.3
```

[3]
```
#module이 있는 directory로 이동
%cd "/content/MyDrive/My Drive/Colab Notebooks"
```

```
/content/MyDrive/My Drive/Colab Notebooks
```

```
import module_arithmetic

print(module_arithmetic.module_add(10, 5))
print(module_arithmetic.module_sub(10, 5))
print(module_arithmetic.module_mul(10, 5))
print(module_arithmetic.module_div(10, 5))
```

```
importing Jupyter notebook from module_arithmetic.ipynb
15
5
50
2.0
```

Namespace

- 모듈 호출 시 범위 지정하는 방법 (from, import)
- 모듈 전체 혹은 모듈 내 함수와 클래스 등 필요한 부분만 선택해서 호출 가능

```
# Alias 설정하기 - 모듈명을 별칭으로 써서
import module_arithmetic as ma
print(ma.module_add(10, 5))

# 모듈에서 특정 함수 또는 클래스만 호출하기
from  module_arithmetic import module_div
print(module_div(10, 5))

# 모듈에서 모든 함수 또는 클래스를 호출하기
from module_arithmetic  import *
print(module_mul(10, 5)) # 전체 호출
```

Built-in Modules

- Python 기본 제공 라이브러리로 문자 처리, 웹, 수학 등 다양한 모듈이 제공
- 바로 import 문으로 활용 가능

```
#난수
import random
print (random.randint (0,100)) # 0~100사이의 정수 난수를 생성
print (random.random()) # 일반적인 난수 생성

#시간
import time
print(time.localtime()) # 현재 시간 출력

#웹
import urllib.request
response = urllib.request.urlopen("https://www.google.com")
print(response.read())
```

Built-in Modules

[1]
```
#난수
import random
print (random.randint (0,100)) # 0~100사이의 정수 난수를 생성
print (random.random()) # 일반적인 난수 생성
```

```
88
0.3214363078543103
```

[2]
```
#시간
import time
print(time.localtime()) # 현재 시간 출력
```

```
time.struct_time(tm_year=2020, tm_mon=10, tm_mday=11, tm_hour=14, tm_min=41, tm_sec=23, tm_wday=6, tm_yday=285, tm_isdst=0)
```

[3]
```
#웹
import urllib.request
response = urllib.request.urlopen("https://www.google.com")
print(response.read())
```

```
:#'0,202162,1151585,5662,730,224,756,2900,1448,207,2415,789,10,1226,364,1499,611,206,383,246,5,1354,648,3451,315,3,66,453,532,
```

Built-in Modules

- OS module: Directory (folder)생성 (현재 위치)

```
#OS module을 이용해서 folder 생성

import os
os.mkdir("log")

if not os.path.isdir("log"): # "log"라는 folder가 없는 경우에만 folder 생성
    os.mkdir("log")
```

Built-in Modules (@Local 환경)

- OS module: 외부 프로그램 실행 (ex. 메모장)

```
#OS module을 이용해서 외부 프로그램 실행 (ex. 메모장)

os.system('notepad')
```

Packages

- 여러 Module들의 모음
- Directory (folder)와 Module (.py 파일)로 구성

가상의 game 패키지 예

```
game/
    __init__.py
    sound/
        __init__.py
        echo.py
        wav.py
    graphic/
        __init__.py
        screen.py
        render.py
    play/
        __init__.py
        run.py
        test.py
```

- game, sound, graphic, play는 폴더 이름, 확장자가 .py인 파일은 파이썬 모듈
- game 폴더가 이 패키지의 root directory
- sound, graphic, play는 sub-directory

https://wikidocs.net/1418

Packages

- __init__.py 파일: 해당 directory가 package의 일부임을 표시
  - 만약 game, sound, graphic 등 directory에 __init__.py이 없다면 package로 인식되지 않음.

- Python 3.3 버전부터는 __init__.py 파일이 없어도 package로 인식하지만(PEP 420), 하위 버전 호환을 위해 __init__.py 파일을 생성하는 것이 안전

가상의 game 패키지 예

```
game/
    __init__.py
    sound/
        __init__.py
        echo.py
        wav.py
    graphic/
        __init__.py
        screen.py
        render.py
    play/
        __init__.py
        run.py
        test.py
```

Packages- Namespace

- dot 연산자(.) 사용 시 import a.b.c에서 가장 마지막 항목 c는 반드시 module 또는 package이어야 함.

첫 번째는 echo 모듈을 import하여 실행하는 방법으로, 다음과 같이 실행한다.

```
>>> import game.sound.echo
>>> game.sound.echo.echo_test()
echo
```

두 번째는 echo 모듈이 있는 디렉터리까지를 from ... import하여 실행하는 방법이다.

```
>>> from game.sound import echo
>>> echo.echo_test()
echo
```

세 번째는 echo 모듈의 echo_test 함수를 직접 import하여 실행하는 방법이다.

```
>>> from game.sound.echo import echo_test
>>> echo_test()
echo
```