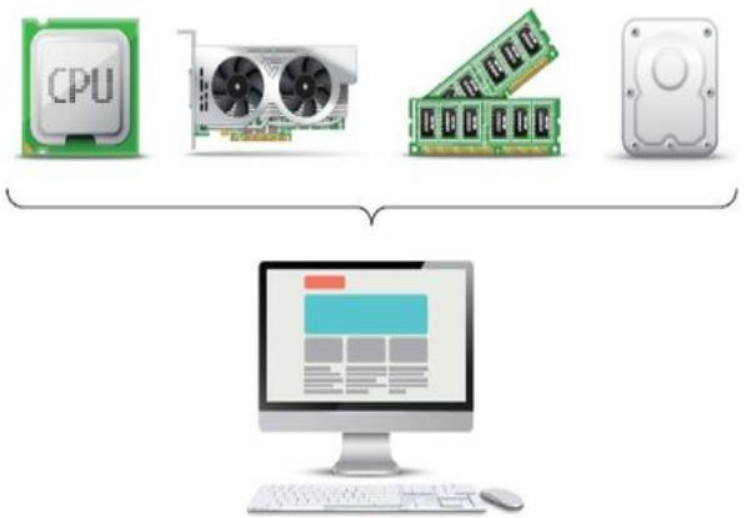
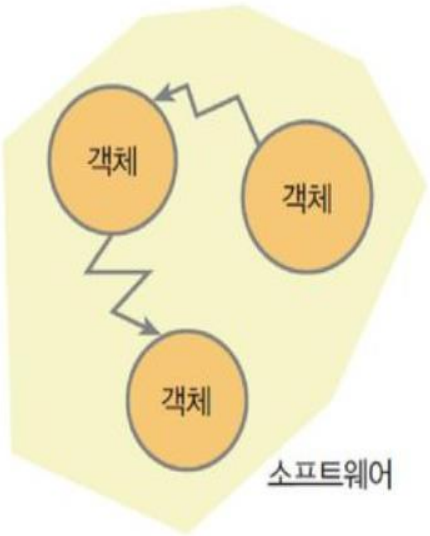
OOP(Object Oriented Programming)

❖ 객체 지향 프로그래밍이란

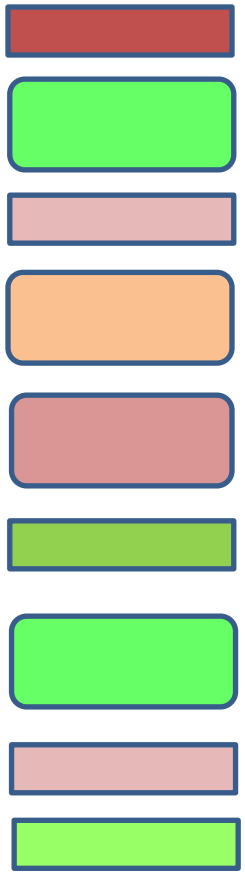
- 객체 지향 프로그래밍(Object Oriented Programming, OOP)은 프로그램에서 필요한 객체들의 역할을 정해 놓고 이 객체 안에 관련된 변수와 함수를 정의하여 객체 단위로 기능이 처리되도록 하는 프로그래밍 기법으로 재사용성이 좋은 컴퓨터 공학의 프로그래밍 기법 중 하나이다.

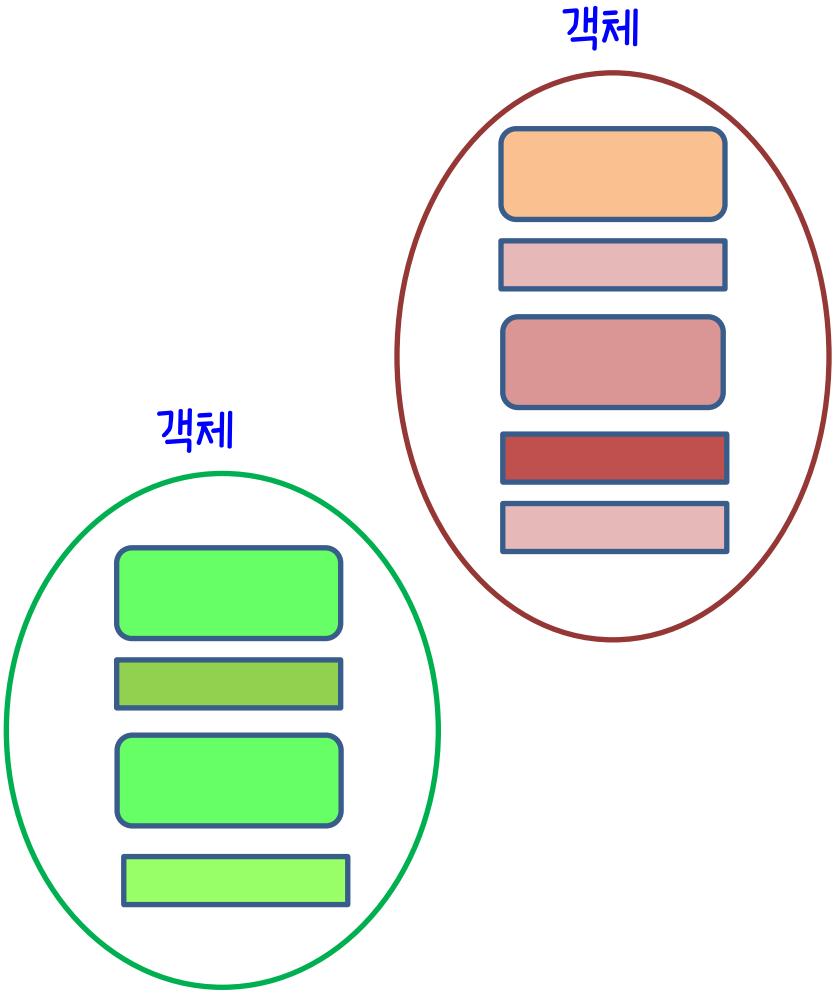




OOP(Object Oriented Programming)

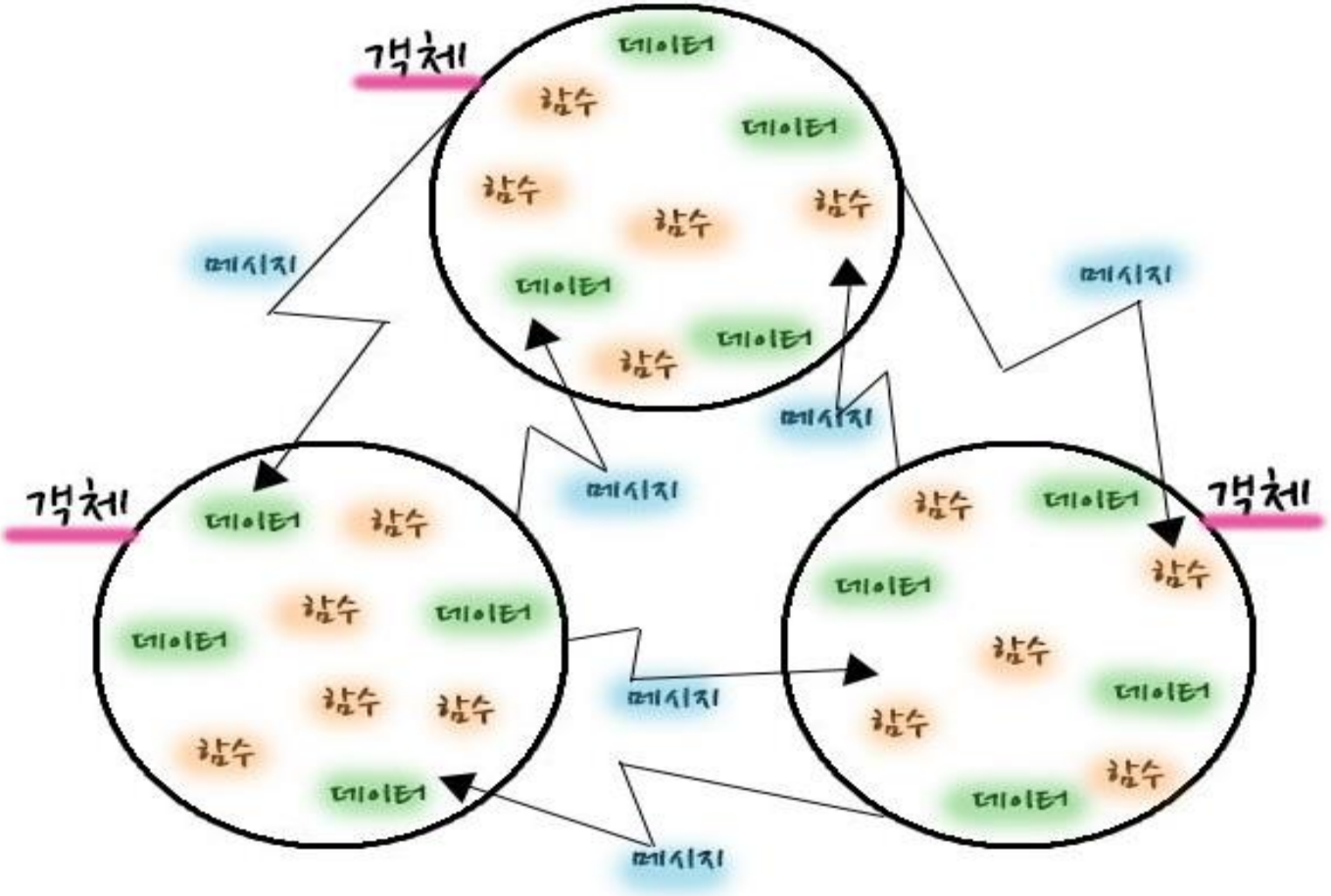
OOP의 장점





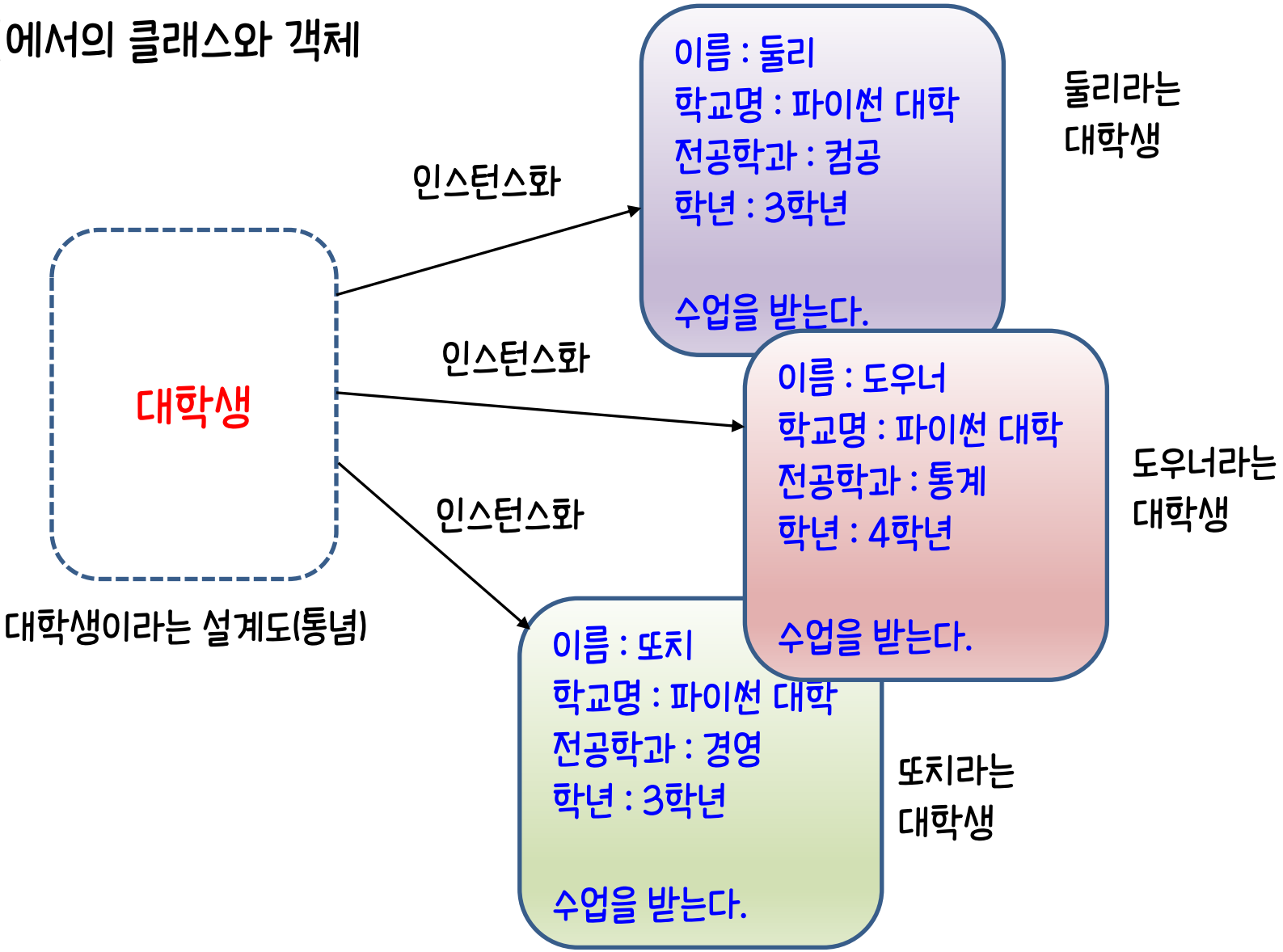
OOP(Object Oriented Programming)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어



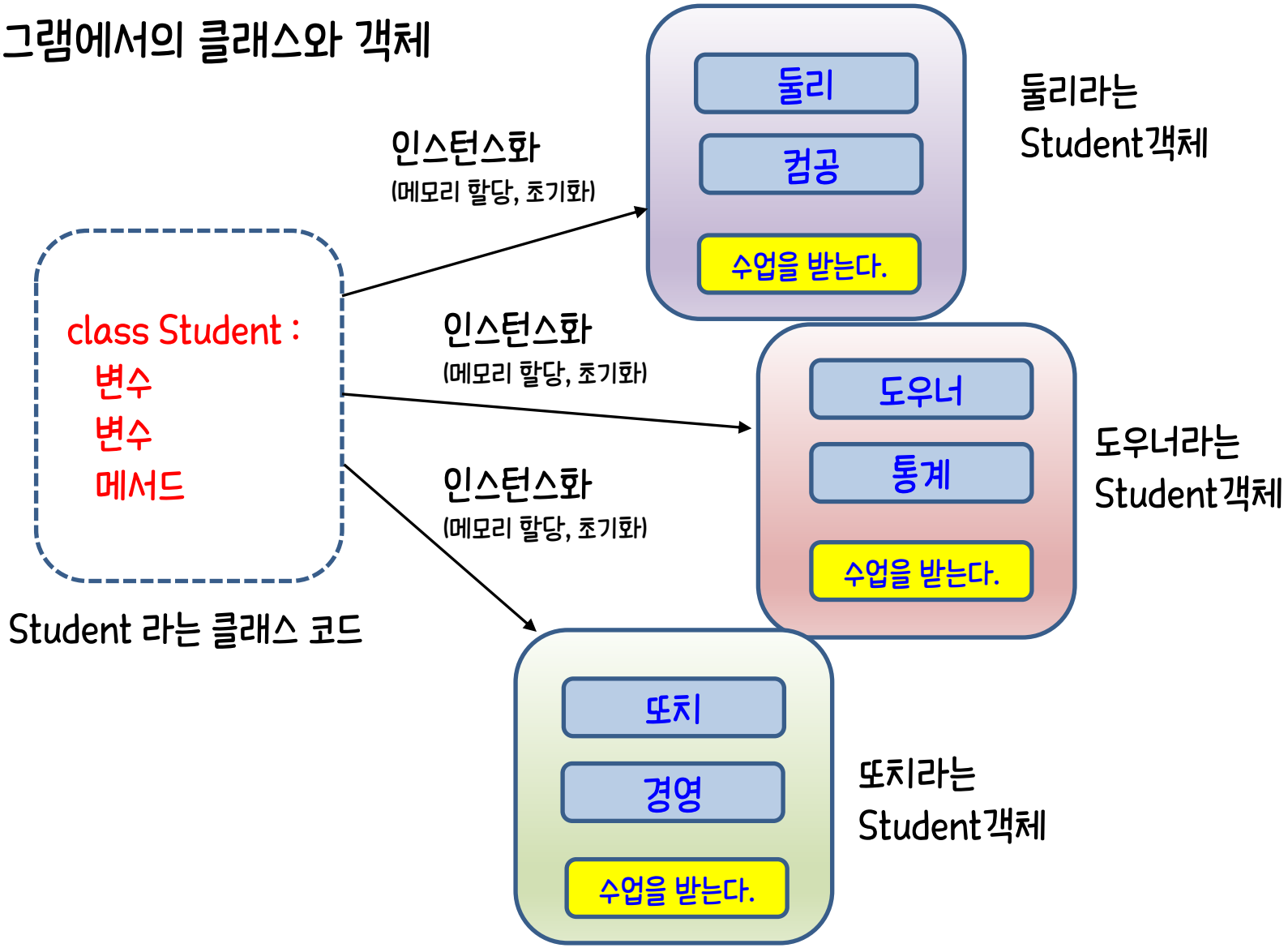
OOP(Object Oriented Programming)

❖ 현실에서의 클래스와 객체



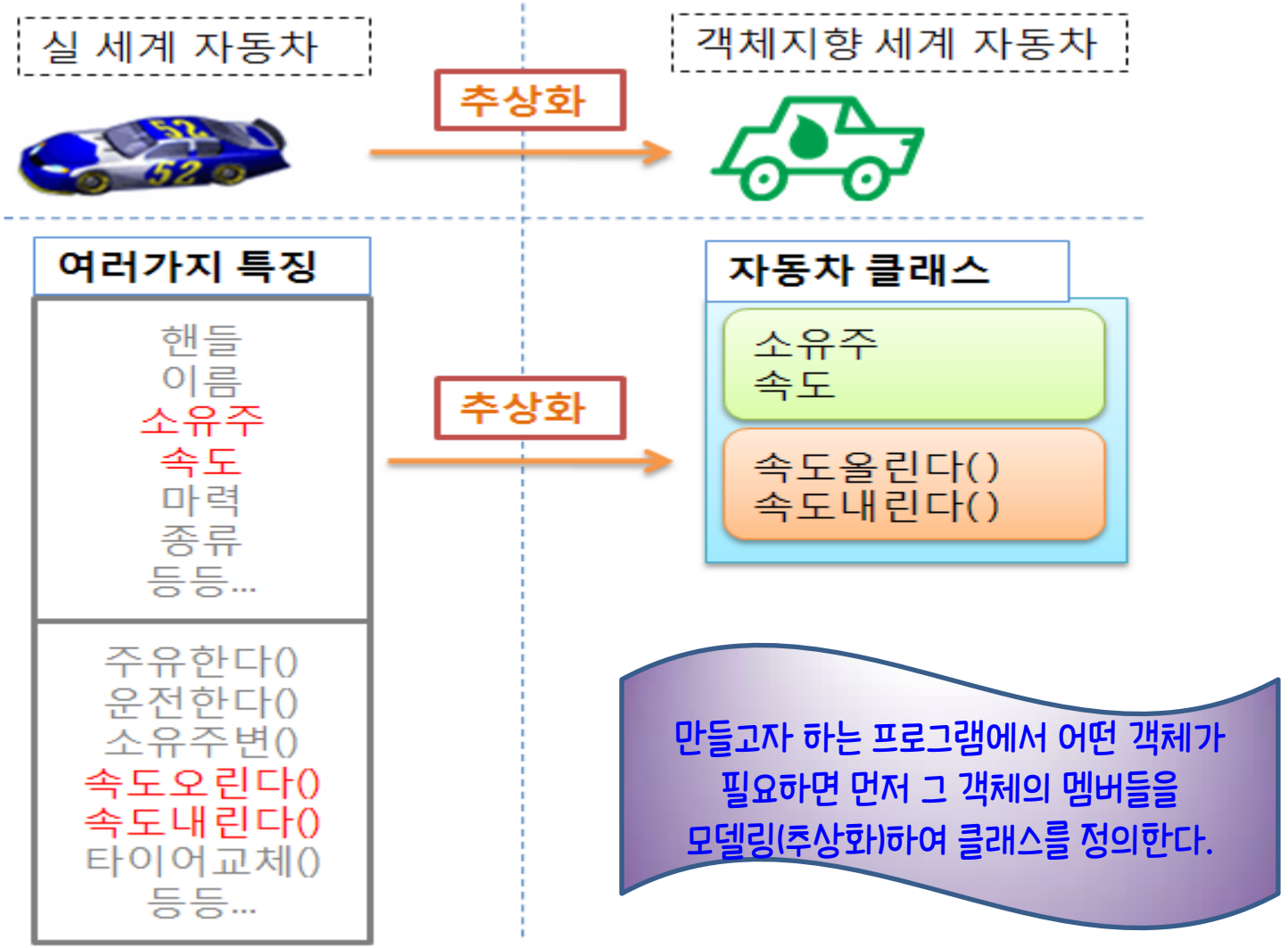
OOP(Object Oriented Programming)

❖ 프로그램에서의 클래스와 객체



OOP(Object Oriented Programming)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어



OOP(Object Oriented Programming)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어

"객체 지향 프로그래밍은 프로그램을 유연하고 변경이 용이하게 만들기 때문에 대규모 소프트웨어 개발에 많이 사용된다. 또한 소프트웨어 개발과 유지보수를 간편하게 하며, 보다 직관적인 코드 분석을 가능하게 하는 장점을 갖고 있다."

"같은 목적과 기능을 위해 객체로 묶인 코드 요소(변수, 함수)들은 객체 내부에서만 강한 응집력을 발휘하고 객체 외부에 주는 영향은 줄이게 된다. 코드가 객체 위주로 (또는 순수하게 객체로만) 이루어질 수 있도록 지향하는 프로그래머의 노력은 코드간의 결합도를 낮추는 결과가 된다."

OOP(Object Oriented Programming)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어

개발하려는 시스템 또는 프로그램에서 어떤 역할이 필요하다.

그 역할을 제공하는 객체를 생성한다.

객체를 만들려면 객체의 구조를 정의한 클래스가 필요하다.

API 로 제공되는 경우라면... 또는 이전에 만들어져 있는 상황이면
앗싸!! ~~ 바로 객체 생성해서 사용한다. 룰루랄라~~~

그렇지 않다면...ㅜ 객체를 모델링하고 클래스로 구현한 후 객체를 생성한다.

OOP(Object Oriented Programming)

❖ 파이썬은 객체지향 프로그래밍(OOP) 언어

클래스 정의 방법

```
class UserInfo :
    변수정의
    변수정의
    함수정의
    함수정의
    함수정의
```

객체 생성 방법

```
변수 = 클래스명()
```

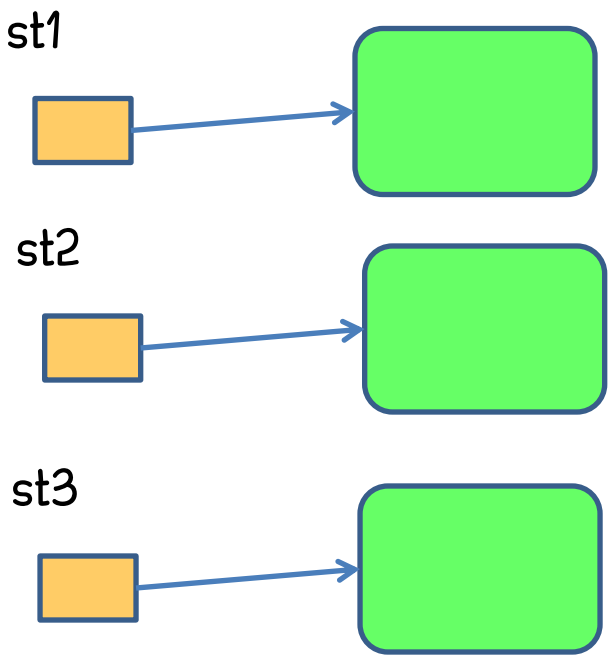
객체 사용 방법

```
변수.변수명
변수.메서드명()
```

OOP(Object Oriented Programming)

❖ 클래스 정의와 객체 생성

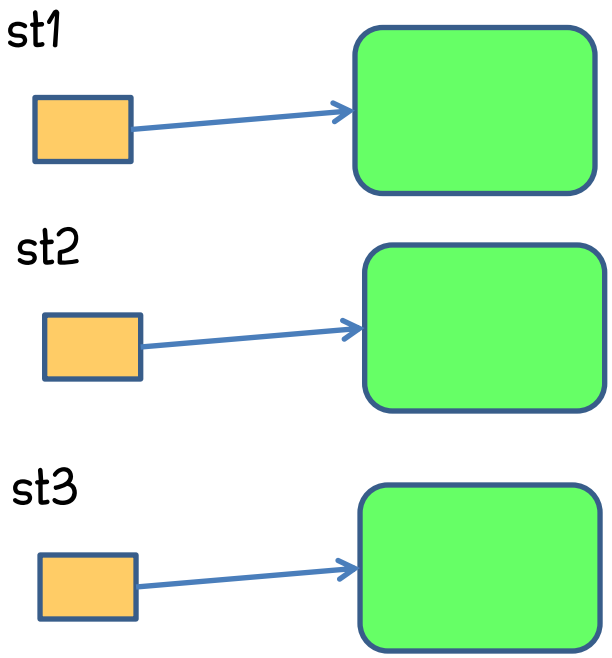
```
class Student :
        pass

st1 = Student()
st2 = Student()
st3 = Student()
```



OOP(Object Oriented Programming)

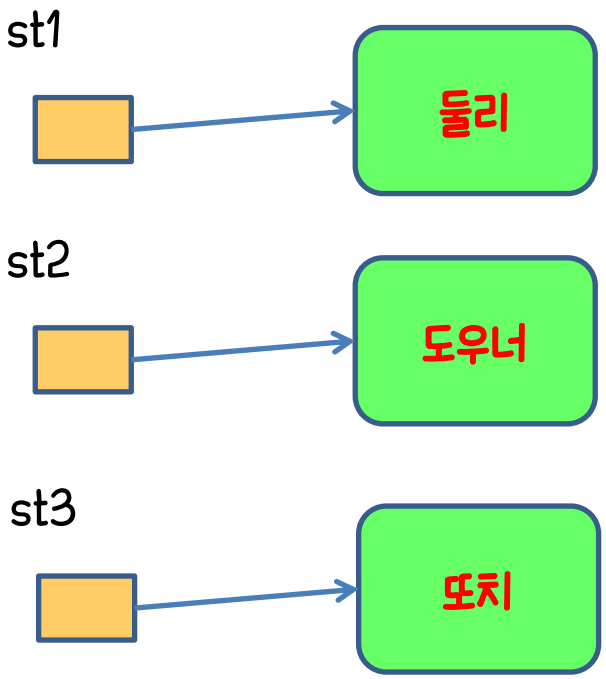
❖ 클래스 정의와 객체 생성

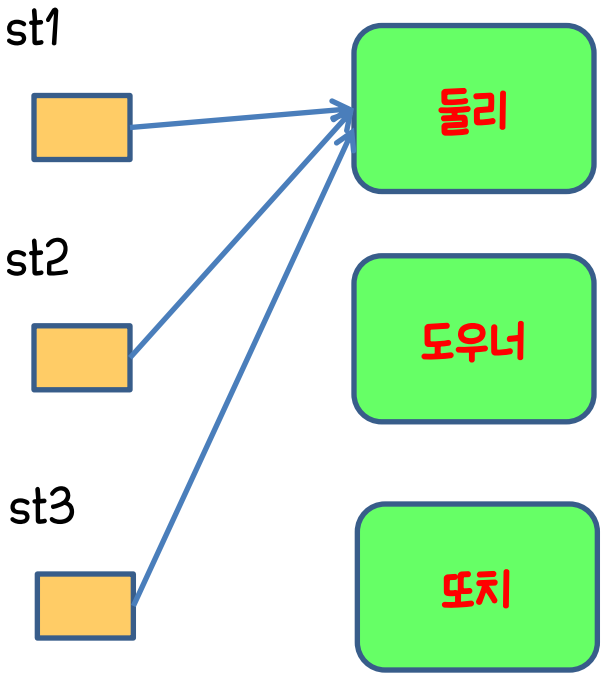
```
class Student :
    def __init__(self):
        print("인스턴스 생성!!")

    def __del__(self):
        print("인스턴스 삭제!!")

st1 = Student()
st2 = Student()
st3 = Student()

print("[ st1이 참조하고 있는 Student 인스턴스 삭제]")
del st1

print("종료")
```



OOP(Object Oriented Programming)

st2 = st1
st3 = st1





OOP(Object Oriented Programming)

❖ 파이썬 객체 지향 구문의 특성

- 스크립트 언어는 가볍게 구현하는 것이 목적
- 파이썬은 스크립트 언어임에도 불구하고 C++과 모듈러 3의 문법을 계승하여 OOP 구문지원
- 클래스 정의, 다중상속, 연산자 오버로딩 지원
- 파이썬은 객체 지향을 지원만 할 뿐 완성도는 떨어짐
- 객체지향의 가장 기본적인 개념 – 클래스
- 클래스 – 속성과 동작을 하나의 범주로 묶어서 실세계의 사물을 프로그램 코드로 흉내 냄
- 모델링 – 사물을 분석하여 필요한 속성과 동작을 추출하는 것
- 캡슐화 – 모델링한 결과를 클래스로 포장하는 것

Contents

❖ 목차

1. 클래스

❖ 클래스

- 객체지향의 가장 기본적 개념
- 관련된 속성과 동작을 하나의 범주로 묶어 실세계의 사물을 흉내 냄
- 모델링
  - 사물 분석하여 필요한 속성과 동작 추출
- 캡슐화
  - 모델링 결과를 클래스로 포장

capsule

```
balance = 8000

def deposit(money):
    global balance
    balance += money

def inquire():
    print("잔액은 %d원입니다." % balance)

deposit(1000)
inquire()
```

실행결과

```
잔액은 9000원입니다.
```

OOP(Object Oriented Programming)

```
def deposit(name, money):
    if name == "둘리" :
        global balancedooly
        balancedooly += money
    elif name == "또치" :
        global balanceddochi
        balanceddochi += money
    elif name == "도우너" :
        global balancedouner
        balancedouner += money

def inquire(name):
    if name == "둘리":
        print("%s의 잔액은 %s원입니다." % (name, format(balancedooly, ',')))
    elif name == "또치":
        print("%s의 잔액은 %s원입니다." % (name, format(balanceddochi, ',')))
    elif name == "도우너":
        print("%s의 잔액은 %s원입니다." % (name, format(balancedouner, ',')))

dooly = "둘리"
ddochi = "또치"
douner = "도우너"

balancedooly = 8000
balanceddochi = 8000
balancedouner = 8000
```

객체 개념을 적용하지 않은 프로그램

서로 연관된 속성과 기능을 클래스라는 하나의 범주로 묶는다. -> 클래스로 만든다.

캡슐화

OOP(Object Oriented Programming)

```
class Account:
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    def deposit(self, money):
        self.balance += money

    def inquire(self):
        print("%s의 잔액은 %s원입니다." % (self.name, format(self.balance, ',')))
```

객체 개념 적용한 프로그램

```
obj1 = Account("둘리", 8000)
obj2 = Account("또치", 8000)
obj3 = Account("도우너", 8000)
```

1. 클래스

❖ 객체의 속성은 변수로, 동작은 함수로 표현

- 멤버
  - 클래스 구성하는 변수와 함수
- 메서드
  - 클래스에 소속된 함수

❖ 객체와 인스턴스

```
a = Cookie
```

a 는 객체이다. (O)
a 객체는 Cookie 클래스의 인스턴스이다. (O)

인스턴스라는 말은 특정 객체(a)가 어떤 클래스(Cookie)의 객체인지를 관계 위주로 설명할 때 사용한다.

1. 클래스

❖ 생성자

- 클래스 선언 형식

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

- __init__ 생성자
  - 통상 객체 초기화

human

```
class Human:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def intro(self):
        print(str(self.age) + "살 " + self.name + "입니다.")
kim = Human(29, "김상형")
kim.intro()
lee = Human(45, "이승우")
lee.intro()
```

실행결과

```
29살 김상형입니다.
45살 이승우입니다.
```

1. 클래스

- 객체 생성 구문
  - 생성된 객체의 참조값을 __init__의 첫 번째 인수 self로 전달
  - 객체생성함수 호출 시 전달한 아규먼트를 두 번째 이후의 인수로 전달
  - 새로 생성되는 객체의 멤버변수를 정의하고 초기화

```
객체 = 클래스명(인수)
```

```
obj1 = Account(10, "둘리")
```

```
def __init__(self, name, balance):
    self.name = name
    self.balance = balance
```

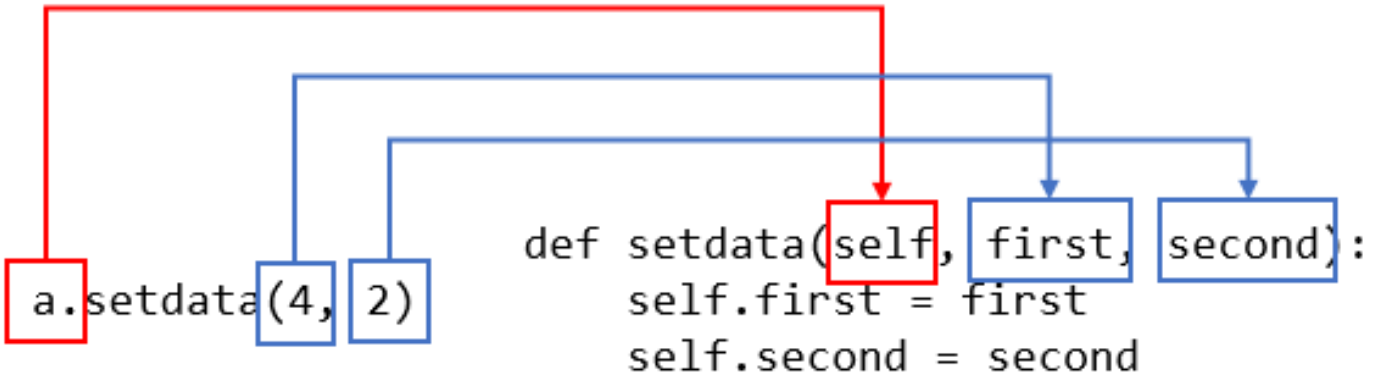
① Account 클래스 메모리 할당(Account 클래스 객체 생성)
② 할당된 주소는 생성자의 첫 번째 매개변수인 self 에게 전달되고 나머지 아규먼트들은 그 다음 매개변수들에게 전달된다.
③ 초기화 메서드에서는 Account 클래스가 할당된 메모리 영역에 name과 balance 변수의 방을 만들고 각각 매개변수의 값을 대입하여 초기화하면 객체 생성 과정은 완료된다.
④ 생성 완료된 객체의 참조값을 obj1에 대입한다. 이제부턴 이 객체를 obj1 객체라 부른다.

- 초기화 메서드 외에 다른 일반 메서드는 객체를 통해서 호출 가능함
  - 객체.메서드()

1. 클래스

```
class FourCal:
    def setdata(self, first, second):
        self.first = first
        self.second = second
```

```
a = FourCal()
a.setdata(4, 2)
```



1. 클래스

❖ 상속

- 기존 클래스 확장하여 멤버 추가하거나 동작 변경
  - 클래스 이름 다음의 괄호 안에 부모 클래스 이름 지정

```
class 이름(부모):
    ....
```

student

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

    def intro(self):
        print(str(self.age) + "살 " + self.name + "입니다")


class Student(Human):
    def __init__(self, age, name, stunum):
        super().__init__(age, name)
```

1. 클래스

```
        self.stunum = stunum

    def intro(self):
        super().intro()
        print("학번 : " + str(self.stunum))

    def study(self):
        print("하늘천 따지 검을현 누를황")


kim = Human(29, "김상형")
kim.intro()
lee = Student(34, "이승우", 930011)
lee.intro()
lee.study()
```

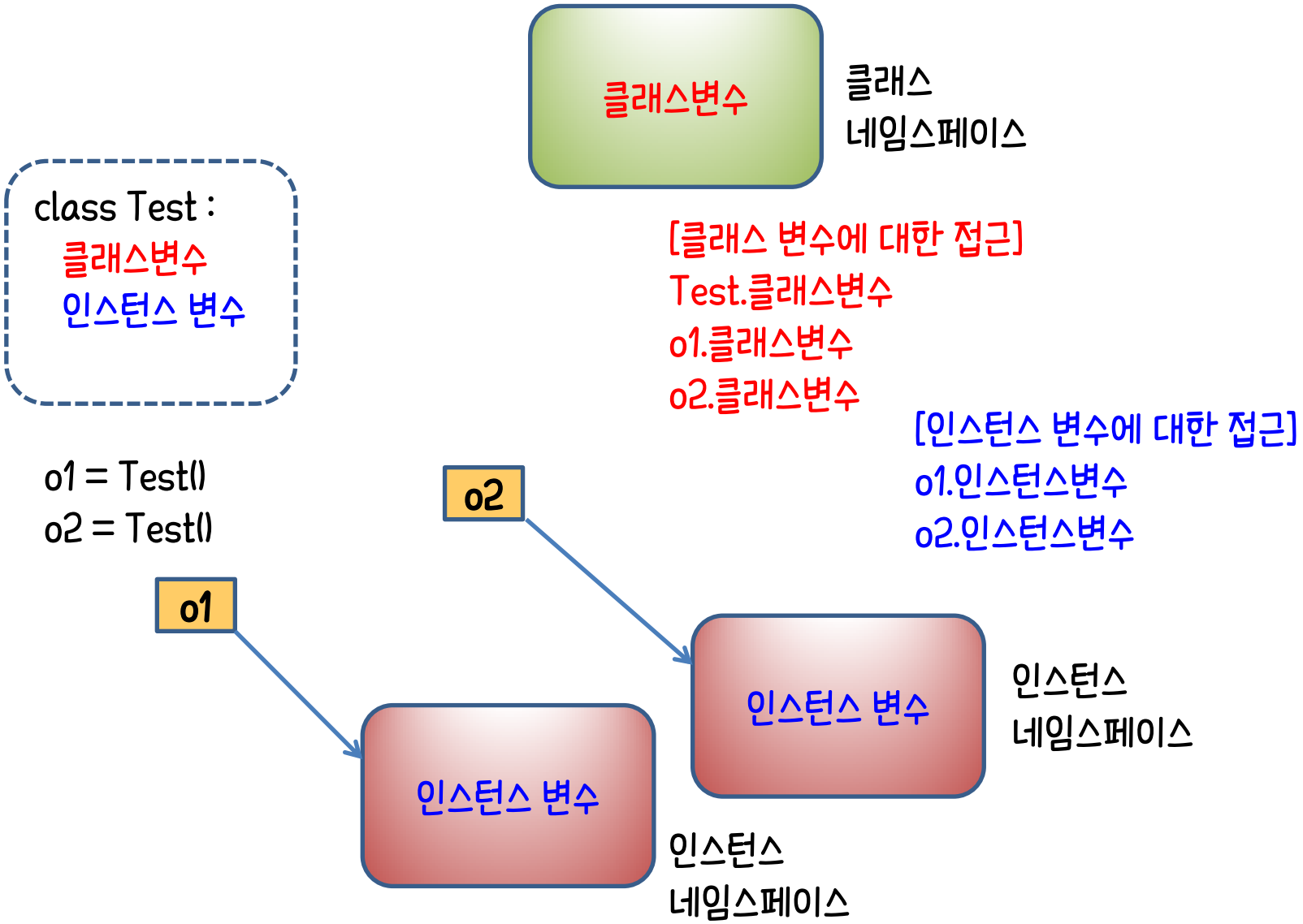
실행결과

```
29살 김상형입니다
34살 이승우입니다
학번 : 930011
하늘천 따지 검을현 누를황
```

- super() 메서드
  - 자식 클래스에서 부모의 초기화 메서드를 호출할 때 사용

OOP(Object Oriented Programming)

❖ 클래스 변수와 인스턴스 변수



1. 클래스

❖ 액세서

- 파이썬 클래스의 멤버는 모두 공개되어 누구나 외부에서 액세스 가능
- 일정한 규칙 마련하고 안전한 액세스 보장
  - 게터(Getter) 메서드
    - 멤버 값 대신 읽음
  - 세터(Setter) 메서드
    - 멤버 값 변경

getset

```
class Date:
    def __init__(self, month):
        self.month = month
    def getmonth(self):
        return self.month
    def setmonth(self, month):
        if 1 <= month <= 12:
            self.month = month

today = Date(8)
today.setmonth(15)
print(today.getmonth())
```

실행결과

```
8
```

1. 클래스

❖ 액세서

- 객체의 멤버 변수의 경우에는 외부로 부터의 잘못된 사용을 막기 위해서 은닉(hidden)하는 구문을 사용하지만 파이썬은 이와 관련한 구문을 지원하지 않는다.

외부로 부터의 직접 접근이 불가하도록 해당 멤버변수 앞에 __ 를 붙인다.

__를 붙인 멤버 변수의 값을 설정 및 추출하는 기능을 지원하는 setter 와 getter 메서드를 준비하고 property() 함수를 사용해서 property 객체를 만들어 변수에 담는다.

객체를 통해서 이 변수에 값을 대입하면 setter 가 호출되고 추출하면 getter 가 호출된다.

__를 붙인 멤버 변수의 값을 설정 및 추출하는 기능을 지원하는 setter 와 getter 메서드를 준비하고 getter 에는 @property 를 붙이고 setter 에는 @변수명.setter 를 붙인다.

객체를 통해서 이 변수에 값을 대입하면 setter 가 호출되고 추출하면 getter 가 호출된다.

2. 여러 가지 메서드

❖ 클래스 메서드

- 특정 객체에 대한 작업을 처리하는 것이 아니라 클래스 전체에 공유
  - @classmethod 데커레이터를 사용해서 정의
  - 첫 번째 인수로 클래스에 해당하는 cls 인수를 정의한다.

classmethod

```
class Car:
    count = 0
    def __init__(self, name):
        self.name = name
        Car.count += 1
    @classmethod
    def outcount(cls):
        print(cls.count)

pride = Car("프라이드")
korando = Car("코란도")
Car.outcount()
```

실행결과

```
2
```

2. 여러 가지 메서드

❖ 정적 메서드

- 클래스에 포함되는 단순 유틸리티 메서드
- 특정 객체에 소속되거나 클래스 관련 동작 하지 않음
- @staticmethod 데커레이터를 사용해서 정의

staticmethod

```
class Car:
    @staticmethod
    def hello():
        print("오늘도 안전 운행 합시다.")
    count = 0
    def __init__(self, name):
        self.name = name
        Car.count += 1
    @classmethod
    def outcount(cls):
        print(cls.count)

Car.hello()
```

실행결과

```
오늘도 안전 운행 합시다.
```

2. 여러 가지 메서드

❖ 연산자 메서드

- 연산자 사용하여 객체끼리 연산
- 연산자 오버로딩
  - 클래스별로 연산자 동작을 고유하게 정의

연산자	메서드	우변일 때의 메서드
==	__eq__	
!=	__ne__	
<	__lt__	
>	__gt__	
<=	__le__	
>=	__ge__	
+	__add__	__radd__
-	__sub__	__rsub__
*	__mul__	__rmul__
/	__div__	__rdiv__
/(division 임포트)	__truediv__	__rtruediv__

2. 여러 가지 메서드

//	__floordiv__	__rfloordiv__
%	__mod__	__rmod__
**	__pow__	__rpow__
《	__lshift__	__rlshift__
》	__rshift__	__lshift__

eqop

```
class Human:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def __eq__(self, other):
        return self.age == other.age and self.name == other.name

kim = Human(29, "김상형")
sang = Human(29, "김상형")
moon = Human(44, "문종민")
print(kim == sang)
print(kim == moon)
```

실행결과

```
True
False
```

2. 여러 가지 메서드

❖ 특수 메서드

- 특정한 구문에 객체 사용될 경우 미리 약속된 작업 수행

메서드	설명
__str__	str(객체) 형식으로 객체를 문자열화한다.
__repr__	repr(객체) 형식으로 객체의 표현식을 만든다.
__len__	len(객체) 형식으로 객체의 길이를 조사한다.

clsstr

```
class Human:
    def __init__(self, age, name):
        self.age = age
        self.name = name
    def __str__(self):
        return "이름 %s, 나이 %d" % (self.name, self.age)

kim = Human(29, "김상형")
print(kim)
```

실행결과

```
이름 김상형, 나이 29
```

Contents

❖ 목차

1. 모듈

❖ 모듈의 작성 및 사용

■ 모듈

- 파이썬 코드를 저장하는 기본 단위
- 편의상 스크립트를 여러 개의 파일로 나눈 하나
- .py 빼고 파일명으로 불림
- 파이썬에서 자주 사용하는 기능은 표준 모듈로 제공됨
- 직접 제작 가능

util

```
INCH = 2.54

def calcsum(n):
    sum = 0
    for num in range(n + 1):
        sum += num
    return sum
```

1. 모듈

utiltest

```
import util

print("1inch =", util.INCH)
print("~10 =", util.calcsum(10))
```

실행결과

```
1inch = 2.54
~10 = 55
```

1. 모듈

❖ 테스트 코드

- 모듈에 간단한 테스트 코드를 작성할 수 있음

util

```
INCH = 2.54

def calcsum(n):
    sum = 0
    for num in range(n + 1):
        sum += num
    return sum

print("인치 =", INCH)
print("합계 =", calcsum(10))
```

실행결과

```
인치 = 2.54
합계 = 55
```

1. 모듈

- 타 모듈에 함수 제공하는 모듈에서는 테스트 코드를 조건문으로 감쌈

util2

```
INCH = 2.54

def calcsum(n):
    sum = 0
    for num in range(n + 1):
        sum += num
    return sum

if __name__ == "__main__":
    print("인치 =", INCH)
    print("합계 =", calcsum(10))
```

utiltest2

```
import util2

print("1inch =", util2.INCH)
print("~10 =", util2.calcsum(10))
```

1. 모듈

❖ 모듈 경로

- 모듈은 임포트하는 파일과 같은 디렉토리에 있어야 함

```
Traceback (most recent call last):
  File "C:\PyStudy\utiltest2.py", line 1, in <module>
    import util2
ModuleNotFoundError: No module named 'util2'
```

- 모듈을 특정 폴더에 두려면 임포트 패스에 추가

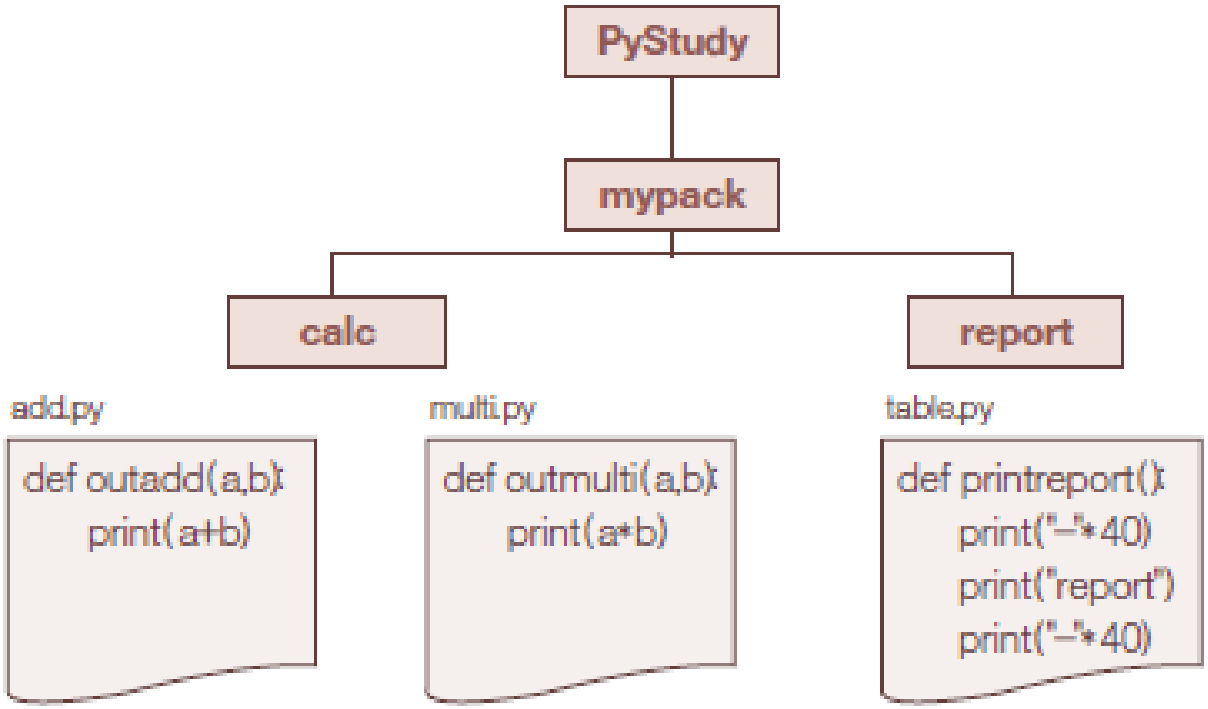
```
>>> import sys
>>> sys.path
['C:\\PyStudy', 'C:\\Python\\Lib\\idlelib', 'C:\\Python\\python36.zip', 'C:\\
Python\\DLLs', 'C:\\Python\\lib', 'C:\\Python', 'C:\\Python\\lib\\site-packages']
```

```
>>> sys.path.append("C:\\Temp")
>>> import util2
>>> util2.calcsum(10)
55
```

2. 패키지

❖ 패키지

- 모듈을 담는 디렉토리
- 디렉토리로 계층 구성하면 모듈을 기능 등에 따라 분류 가능



2. 패키지

usepackage

```
import sys
sys.path.append("C:/PyStudy")

import mypack.calc.add
mypack.calc.add.outadd(1,2)

import mypack.report.table
mypack.report.table.outreport()
```

실행결과

```
3
---------------------------------------
report
---------------------------------------
```

- 함수명이나 모듈명에 충돌 발생하지 않음
- 단일 모듈에 비해 호출문 길어지는 불편함 있음
  - from 패키지 import 모듈

```
from mypack.calc import add
add.outadd(1,2)
```

2. 패키지

❖ __init__.py

- 모든 모듈 한꺼번에 불러올 때는 어떤 모듈 대상인지 밝혀두어야
- 임포트할 대상 모듈 리스트 명시
- 패키지 로드될 때의 초기화 코드 작성해 둠

importall

```
import sys
sys.path.append("C:/PyStudy")

from mypack.calc import *
add.outadd(1,2)
multi.outmulti(1,2)
```

실행결과

```
Traceback (most recent call last):
  File "C:\PyStudy\Test.py", line 5, in <module>
    add.outadd(1,2)
NameError: name 'add' is not defined
```

2. 패키지

__init__

```
__all__ = ["add", "multi"]

print("add module imported")
```

- import * 로 읽을 때 add 및 multi 모듈 모두 읽어 옴
- __init__.py의 초기화 코드 실행

- __init__.py 목록에 어떤 모듈 작성할 것인지는 패키지 개발자 재량

- 한 번 임포트한 모듈은 컴파일 상태로 캐시에 저장
  - 확장자 pyc
  - 한 모듈을 각각 다른 파이썬 버전끼리 공유 가능

3. 서드 파티 모듈

❖ 모듈의 내부

- 각 모듈은 기능별로 나누어져 다수 함수 포함
- dir 내장 함수 사용
  - 모듈에 있는 함수나 변수 목록 조사

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

3. 서드 파티 모듈

❖ 외부 모듈의 목록

- 서드 파티 모듈 (Third Party Module)
  - 파이썬 외 회사 및 단체가 제작하여 배포하는 모듈

모듈	설명
Django, Flask	웹 프레임워크
BeauifulSoup	HTML, XML 파서
wxPython, PyGtk	그래픽 툴킷
pyGame	게임 제작 프레임워크
PIL	이미지 처리 라이브러리
pyLibrary	유틸리티 라이브러리

3. 서드 파티 모듈

❖ pip (Python Package Index)

- 외부 모듈 관리 용이
- pip 명령 패키지명

명령	설명
install	패키지를 설치한다.
uninstall	설치한 패키지를 삭제한다.
freeze	설치한 패키지의 목록을 보여 준다.
show	패키지의 정보를 보여 준다.
search	pyPI에서 패키지를 검색한다.

wxtest

```
import wx

app = wx.App()
frame = wx.Frame(None, 0, "파이썬 만세")

frame.Show(True)
app.MainLoop()
```

Contents

❖ 목차

1. 반복자

❖ 열거 가능 객체

■ for 반복문

- 객체 요소를 순서대로 읽는 제어문
- 컨테이너 순회
  - __iter__ 메서드 호출하여 반복자 구하고,
  - 이것이 __next__ 메서드 호출하여 컨테이너 요소 읽으며 이동

```
for num in [11, 22, 33]:
    print(n)
```

■ 반복가능(Iterable) 객체

- 반복자로 요소를 순서대로 읽을 수 있는 것
- for문과 자주 함께 사용

1. 반복자

foriter

```
nums = [11, 22, 33]
it = iter(nums)
while True:
    try:
        num = next(it)
    except StopIteration:
        break
    print(num)
```

실행결과

```
11
22
33
```

- for문 내부동작 이해하고 필요한 조건 맞추어 반복가능 객체 만들 수 있음
  - 요소 순회할 반복자 제공

- Seq 클래스
  - 일련의 이름 목록 저장해두고 순서대로 하나씩 읽어 줌

1. 반복자

❖ 제너레이터 (Generator)

▪ return 대신 yield 로 값을 리턴하는 함수

- yield 문은 return 문과 유사하며 마지막 값과 상태를 저장함
- 함수가 값을 리턴할 때 yield를 사용하고 있으면 제너레이터로 간주한다.

generator

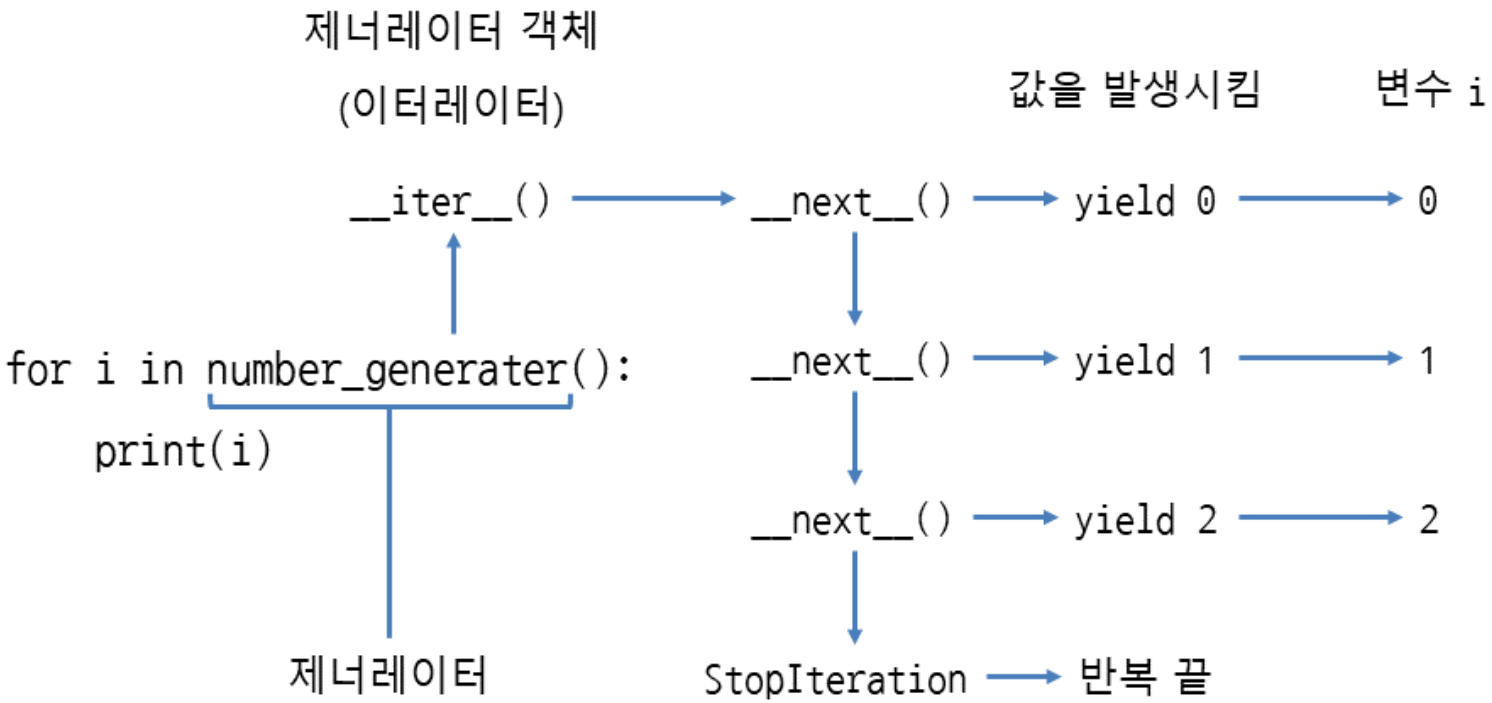
```
def seqgen(data):
    for index in range(0, len(data), 2):
        yield data[index:index+2]

solarterm = seqgen("입춘우수경칩춘분청명곡우입하소만망종하지소서대서")
for k in solarterm:
    print(k, end = ',')
```

- seqgen 제너레이터
  - __iter__, __next__ 메서드 내부 자동생성
  - 인수로 전달받은 문자열 데이터 분리하여 yield 명령으로 리턴

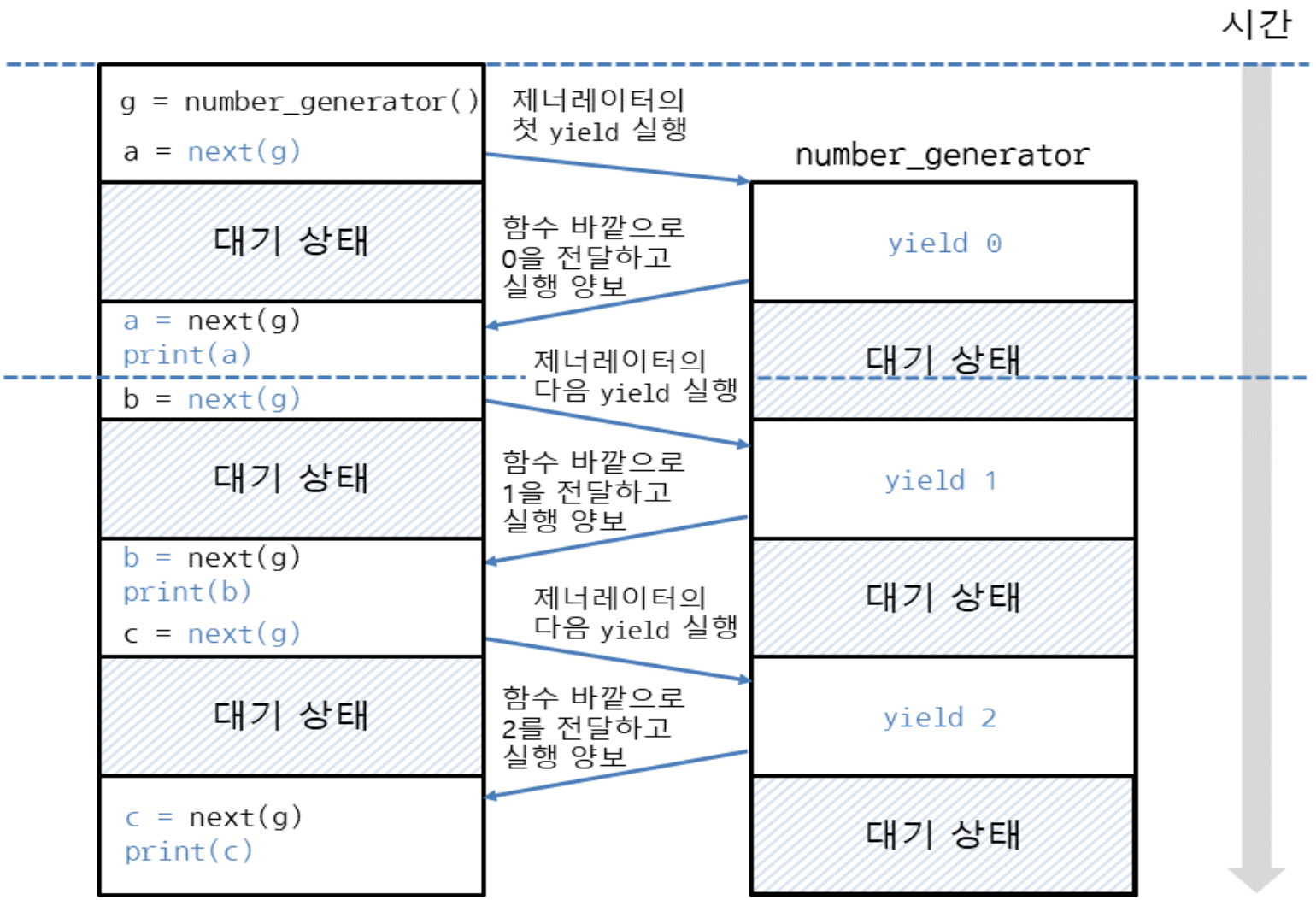
1. 반복자

```
def number_generator():
    yield 0
    yield 1
    yield 2
```

```
for i in number_generator():
    print(i)
```



1. 반복자


시간
g = number_generator()
a = next(g)
제너레이터의
첫 yield 실행
number_generator
대기 상태
함수 바깥으로
0을 전달하고
실행 양보
yield 0
a = next(g)
print(a)
대기 상태
제너레이터의
다음 yield 실행
b = next(g)
대기 상태
함수 바깥으로
1을 전달하고
실행 양보
yield 1
b = next(g)
print(b)
c = next(g)
대기 상태
제너레이터의
다음 yield 실행
대기 상태
함수 바깥으로
2를 전달하고
실행 양보
yield 2
c = next(g)
print(c)
대기 상태

1. 반복자

제너레이터 표현식

리스트 컴프리핸션에서 대괄호를 괄호로 사용하면 제너레이터 표현식으로 간주한다.

```
for n in [i for i in range(100)] :
    print(n, end=",")

for n in (i for i in range(100)) :
    print(n, end = ',')
```

2. 데코레이터

❖ 일급 시민 (First Class Citizen) → 일급객체

- 파이썬 등 함수형 언어에서의 함수
  - 이름 가짐
  - 다른 변수에 대입할 수 있음
  - 데이터를 인수로 전달할 수 있음
  - 결과값을 리턴할 수 있음
  - 컬렉션에 저장할 수 있음

funcvalue

```
def add(a, b):
    print(a + b)

plus = add
plus(1, 2)
```

실행결과 3

2. 데코레이터

❖ 지역 함수

- 다른 함수 안에 정의되는 도우미 함수
- 함수 내부의 반복되는 코드를 통합하여 관리를 용이하게 함

localfunc

```
def calcsum(n):
    def add(a, b):
        return a+b

    sum = 0
    for i in range(n+1):
        sum = add(sum, i)
    return sum

print("~10 =", calcsum(10))
```

실행결과

```
~10 = 55
```

2. 데코레이터

- 상호 평등한 관계로 작성할 경우
  - 동작에는 문제 없음
  - calcsum이 add에 종속되며 독립성 떨어지고 재사용 번거로움

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

def calcsum(n):
    sum = 0
    for i in range(n + 1):
        sum = add(sum, i)
    return sum
```

2. 데코레이터

❖ 함수 데코레이터 (Decorator)

- 이미 정의된 함수에 원하는 코드 추가하는 기법 -> 함수에 대한 재사용성을 높임

- 함수 래핑 (Wrapping)
  - 원하는 코드를 추가한 후, 함수를 호출하여 기능을 확장

wrapper

```
def inner():
    print("결과를 출력합니다.")

def outer(func):
    print("-" * 20)
    func()
    print("-" * 20)

outer(inner)
```

함수 래핑

실행결과

```
--------------------
결과를 출력합니다.
--------------------
```

2. 데코레이터

- 호출 구문의 비직관성을 해결하려면?

wrapper2

```
def inner():
    print("결과를 출력합니다.")

def outer(func):
    def wrapper():
        print("-" * 20)
        func()
        print("-" * 20)
    return wrapper

inner = outer(inner)
inner()
```

2. 데코레이터

- 내부함수 정의 시 데코레이터 붙임
- @outer 데코레이터
  - inner = outer(inner) 구문으로 함수 포장하여 재정의

decorator

```
def outer(func):
    def wrapper():
        print("-" * 20)
        func()
        print("-" * 20)
    return wrapper

@outer
def inner():
    print("결과를 출력합니다.")

inner()
```

@outer 는 inner = outer(inner) 와
동일한 기능을 처리한다.
inner() 함수는 outer() 함수에 의해서
래핑된다는 것을 나타낸다.

2. 데코레이터

- 예시

tagdeco

```
def para(func):
    def wrapper():
        return "<p>" + str(func()) + "</p>"
    return wrapper

@para
def outname():
    return "김상형"

@para
def outage():
    return "29"

print(outname())
print(outage())
```

실행결과

```
<p>김상형</p>
<p>29</p>
```

2. 데코레이터

- 래핑되는 함수가 인수를 가질 경우 대리호출 시에도 인수 그대로 전달

decoarg

```
def para(func):
    def wrapper():
        return "<p>" + str(func()) + "</p>"
    return wrapper

@para
def outname(name):
    return "이름:" + name + "님"

@para
def outage(age):
    return "나이:" + str(age)

print(outname("김상형"))
print(outage(29))
```

실행결과

```
Traceback (most recent call last):
  File "C:/PyStudy/CharmTest/CharmTest.py", line 14, in <module>
    print(outname("김상형"))
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

2. 데코레이터

- wrapper 의 매개변수를 *args, **kwargs 로 정의하여 어떠한 형태의 아규먼트라도 전달받아 래핑되는 함수에게 전달하도록 구현해야 한다.

decoarg2

```
def para(func):
    def wrapper(*args, **kwargs):
        return "<p>" + str(func(*args, **kwargs)) + "</p>"
    return wrapper

@para
def outname(name):
    return "이름:" + name + "님"

@para
def outage(age):
    return "나이:" + str(age)

print(outname("김상형"))
print(outage(29))
print(outname.__name__)
```

실행결과

```
<p>이름:김상형님</p>
<p>나이:29</p>
wrapper
```

3. 동적 코드 실행

❖ eval

- 문자열 형태로 된 파이썬 표현식을 평가하고 실행하여 결과 반환
- 문자열로 구성된 내용을 파이썬 코드로 인식하여 실행할 수 있음

eval

```
result = eval("2 + 3 * 4")
print(result)

a = 2
print(eval("a + 3"))

city = eval("['seoul', 'osan', 'suwon']")
for c in city:
    print(c, end = ', ')
```

실행결과

```
14
5
seoul, osan, suwon,
```

```
>>> data = '10,15,20,13,16'
>>> r1 = [data]
>>> r1
['10,15,20,13,16']
>>> len(r1)
1
>>> r2 = eval("[" + data + "]")
>>> r2
[10, 15, 20, 13, 16]
>>> len(r2)
5
```

3. 동적 코드 실행

❖ exec

- 문자열 형태로 된 파이썬 문장(명령)을 평가하고 실행함
- eval() 함수는 표현식을 평가하고 실행하여 리턴하는 기능 exec() 함수는 문장을 실행하는 기능

exec

```
exec("value = 3")
print(value)
exec("for i in range(5):print(i, end = ', ')")
```

실행결과

```
3
0, 1, 2, 3, 4,
```

```
>>> eval('r = 10+20+30')
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    eval('r = 10+20+30')
  File "<string>", line 1
    r = 10+20+30
      ^
SyntaxError: invalid syntax
>>> exec('r = 10+20+30')
>>> r
60
>>> r = eval('10+20+30')
>>> r
60
```