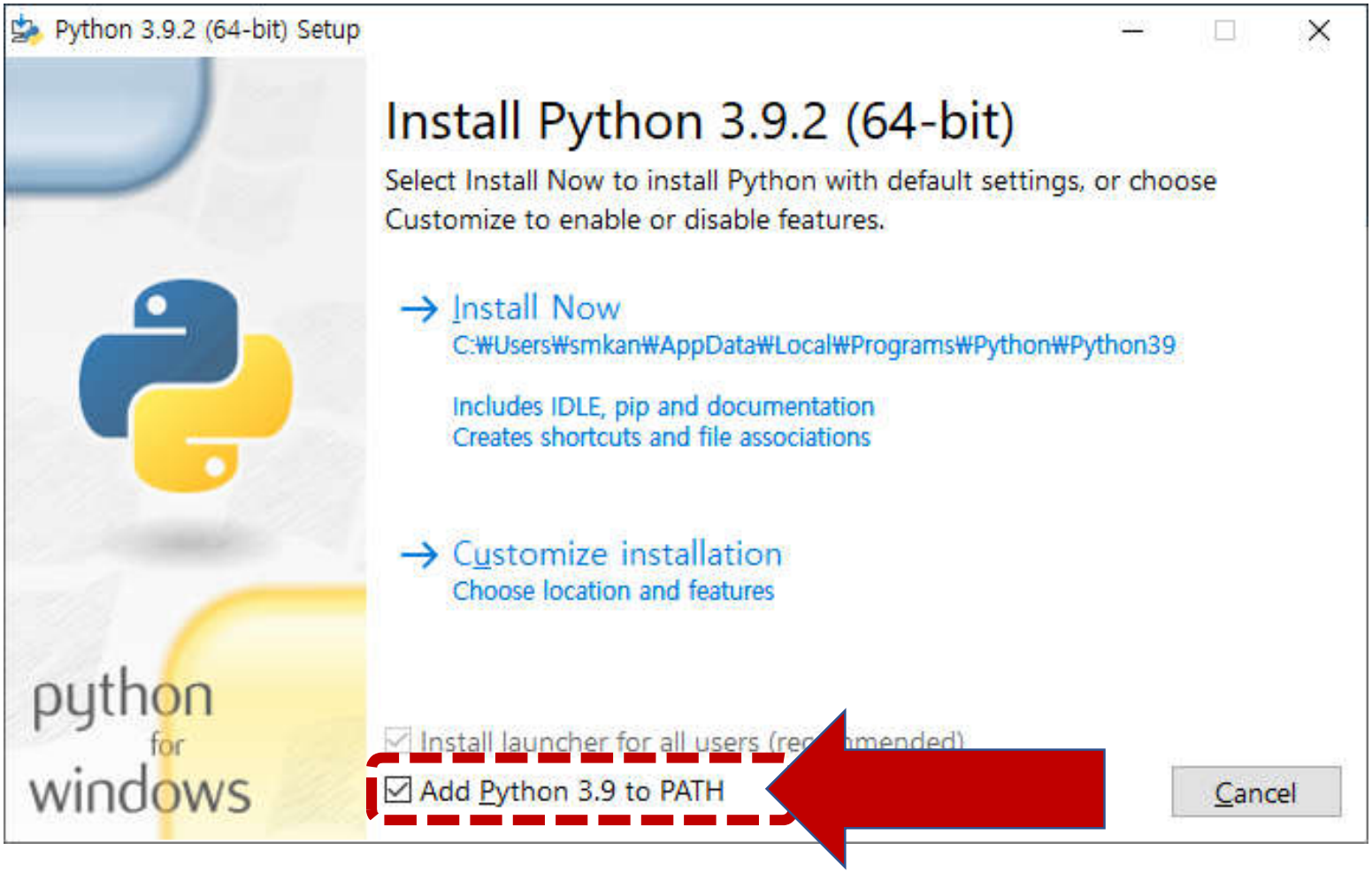
핵심 정리

- Python 설치

  ⇨ python.org 에서 "**Downloads**" 선택 후 설치

  ⇨ **환경 변수 등록을 꼭 선택**



- 소스 코드를 작성할 Text Editor

  ⇨ 강의 에서는 "visual studio code" 사용

  ⇨ **code.visualstudio.com**

Interactive mode

- 콘솔(터미널) 창에서 "**Python**" 입력
- 한 줄 단위로 코드 실행
- 종료 하려면 "**quit()**" 또는 "**Ctrl + Z**"

변수 이름만 입력하면 변수값 출력

Script File 작성

- 확장자가 "**.py**" 인 파일로 작성
- 실행하려면

  ⇨ **python hello.py**

변수값 을 출력하려면 print() 함수 사용

핵심 정리

- Python 버전
  - ⇨ "**3.9.2**" 버전 사용
  - ⇨ 콘솔 창에서 "**python --version**" 으로 버전 확인 가능
  - ⇨ 주제에 따라 update 되는 경우 버전이 다를 수 있음

- 소스 코드 편집기
  - ⇨ "**visual studio code**" 사용

- 강의에 사용된 소스
  - ⇨ "**github.com/codenuri/PYTHON**"

핵심 정리

타입과 변수, 배열, 함수, 연산자, 조건문과 반복문 .....

모든 언어의 공통의 특징

클래스, 상속, 인터페이스 .....

객체지향 언어 공통의 특징

코루틴, 제너레이터, async .....

요즘 언어의 새로운 특징

- sample.py

  ⇨ 프로그램 작성시 필요한 "**다양한 문법을 Python 은 어떻게 표현**" 하는지를 미리 살펴보는 예제

  ⇨ 각 주제 별로 "**이어지는 강좌에서 자세히 설명**"

핵심 정리

- entry-point

  ⇨ 파일의 첫번째 문장 부 터 실행

- import

  ⇨ 파이썬의 다양한 표준 라이브러리를 불러올 때 사용

  ⇨ "**모듈**" 강좌에서 자세히 설명

- 주석 (Comment)

single-line comment	#
multi-line comment	''' ''' 또는 """ """

핵심 정리

- 변수의 선언과 사용
  - ⇨ 데이터 타입을 명시할 필요 없다.
  - ⇨ 하나의 변수 다양한 타입을 가리킬 수 있다.
  - ⇨ "**변수와 타입**" 강좌에서 자세히 설명

- 시퀀스 (sequence)
  - ⇨ 배열 대신 "**리스트(list)**" 사용
  - ⇨ list, tuple, set, dictionary, bytes, ...
  - ⇨ "**시퀀스**" 강좌에서 자세히 설명

- 조건문과 반복문
  - ⇨ {} 대신 들여쓰기로 구분
  - ⇨ switch 와 do~while 문이 없다.

- 함수
  - ⇨ 함수를 편리하게 사용할 수 있는 다양한 문법 제공
  - ⇨ "**함수**" 강좌에서 자세히 설명

핵심 정리

- 클래스
  - ⇨ 생성자, 소멸자, 클래스 메소드, 클래스 변수 등.
  - ⇨ "**클래스**" 강좌에서 자세히 설명

- 최신 기술
  - ⇨ "**Generator**", "**Coroutine**", "**asyncio**"
  - ⇨ 후반부 강좌에서 자세히 설명

- 파이썬 언어의 핵심 기술
  - ⇨ "**Decorator**", "**Descriptor**"

핵심 정리

- statement
  - ⇨ 문장의 끝에 세미콜론 를 표기하지 않아도 된다.
  - ⇨ 하나의 문장을 여러줄로 표기 하려면 '₩' 를 사용.
  - ⇨ [], (), {} 안에서는 여러줄로 표기 가능

- 조건문/제어문/함수/클래스에서 { }대신에 들여쓰기 사용
  - ⇨ 한 칸 이상의 들여 쓰기
  - ⇨ "4개의 공백" 을 권장

핵심 정리

print

`print('AAA')`	출력 후 개행
`print('CCC', end='')`	출력 후 개행 안함

print 를 사용한 변수값 출력

변수값 만 출력	`print(n1, n2, n3, sep=', ')`
% 사용	`print('n1 = %d' % n1 )` `print('n1 = %d, n2 = %d' % (n1, n2) )`

format 사용	print('n1 = {0}, n2 = {1}'.format(n1, n2)) # 문자열.format()
	print('n1 = {v1}, n2 = {v2}'.format(v1 = n1, v2 = n2))

f-string	`print(f'n1 = {n1}, n2 = {n2}')` `print(f'{n1 = }, {n2 = }')`

핵심 정리

- module 을 import 하지 않아도 사용 가능한 "**python 표준 함수**"
- CPython 환경에서 "**C언어로 작성**"되어 속도가 빠르다.

핵심 정리

- “3.10(베타)” 버전에서 71개의 표준 함수 제공

abs()	dir()	isinstance()	range()
aiter()	divmod()	issubclass()	repr()
all()	enumerate()	iter()	reversed()
any()	eval()	len()	round()
anext()	exec()	list()	set()
ascii()	filter()	locals()	setattr()
bin()	float()	map()	slice()
bool()	format()	max()	sorted()
breakpoint()	frozenset()	memoryview()	staticmethod()
bytearray()	getattr()	min()	str()
bytes()	globals()	next()	sum()
callable()	hasattr()	object()	super()
chr()	hash()	oct()	tuple()
classmethod()	help()	open()	type()
compile()	hex()	ord()	vars()
complex()	id()	pow()	zip()
delattr()	input()	print()	__import__()
dict()	int()	property()	

https://docs.python.org/ko/3/library/functions.html

핵심 정리

abs()	**dir()**	isinstance()	range()
aiter()	divmod()	issubclass()	repr()
all()	enumerate()	iter()	reversed()
any()	**eval()**	**len()**	round()
anext()	exec()	list()	set()
ascii()	filter()	locals()	setattr()
bin()	float()	map()	slice()
bool()	format()	max()	sorted()
breakpoint()	frozenset()	memoryview()	staticmethod()
bytearray()	getattr()	min()	str()
bytes()	globals()	next()	sum()
callable()	hasattr()	object()	super()
chr()	hash()	**oct()**	tuple()
classmethod()	**help()**	**open()**	type()
compile()	**hex()**	ord()	vars()
complex()	id()	pow()	zip()
delattr()	input()	print()	__import__()
dict()	int()	property()	

핵심 정리

- 70여개의 표준 함수로 부족 하다면
  - ⇨ 다양한 분야의 "**200 개 이상의 표준 라이브러리**"를 사용하면 된다.
  - ⇨ "모듈" 이라는 형태로 제공된다.
  - ⇨ https://docs.python.org/ko/3/library/index.html

- module 을 import 하는 방법

---

import 모듈이름

---

from 모듈이름 import 함수(클래스)이름

핵심 정리

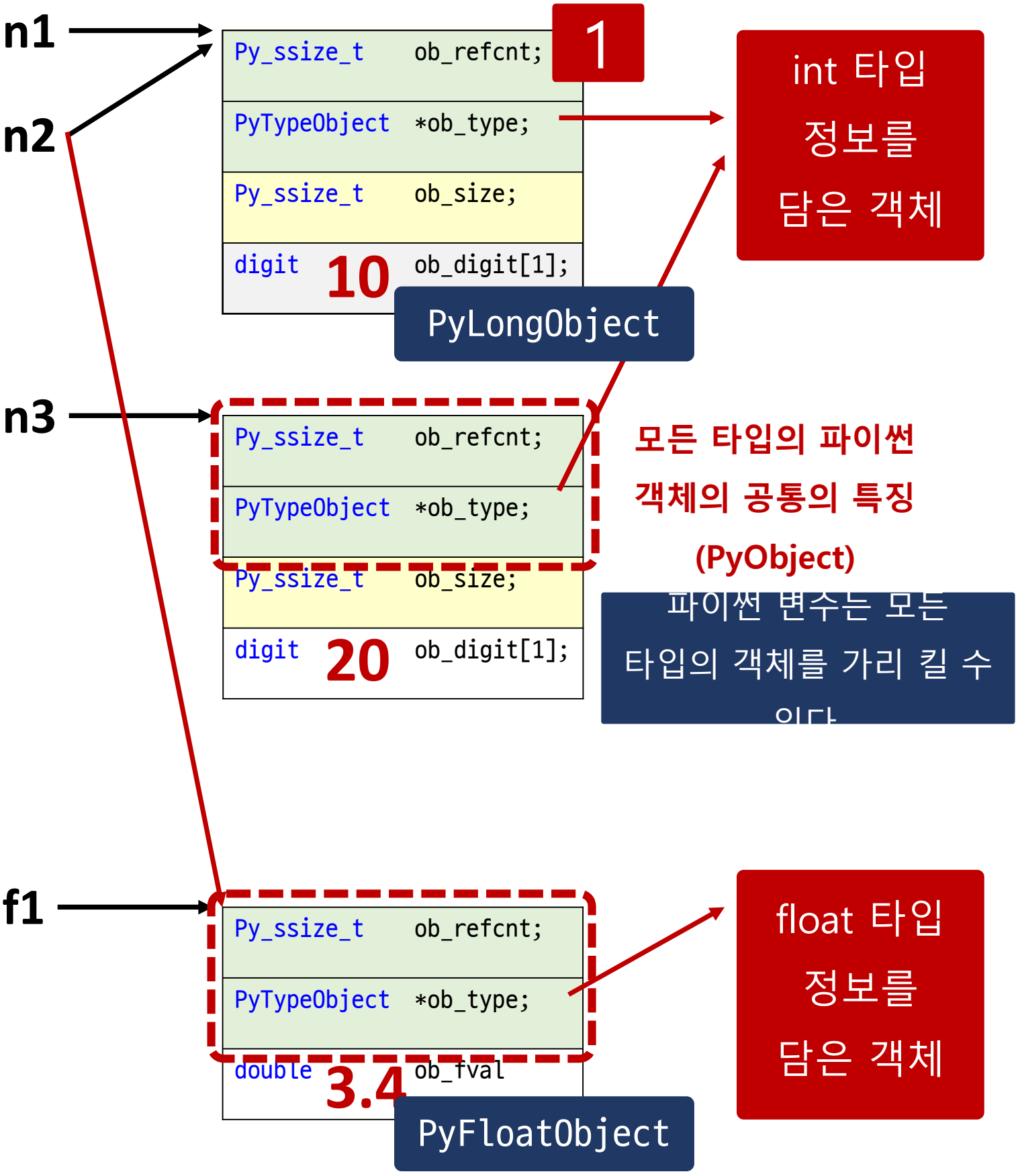
- range(first, last)
  - ⇨ "**first ~ last 사이의 숫자를 연속적으로 발생**" 시키는 generator
  - ⇨ "**iterable**" 강좌에서 자세히 설명

- 성능을 측정하려면
  - ⇨ time 모듈의 time 함수 사용

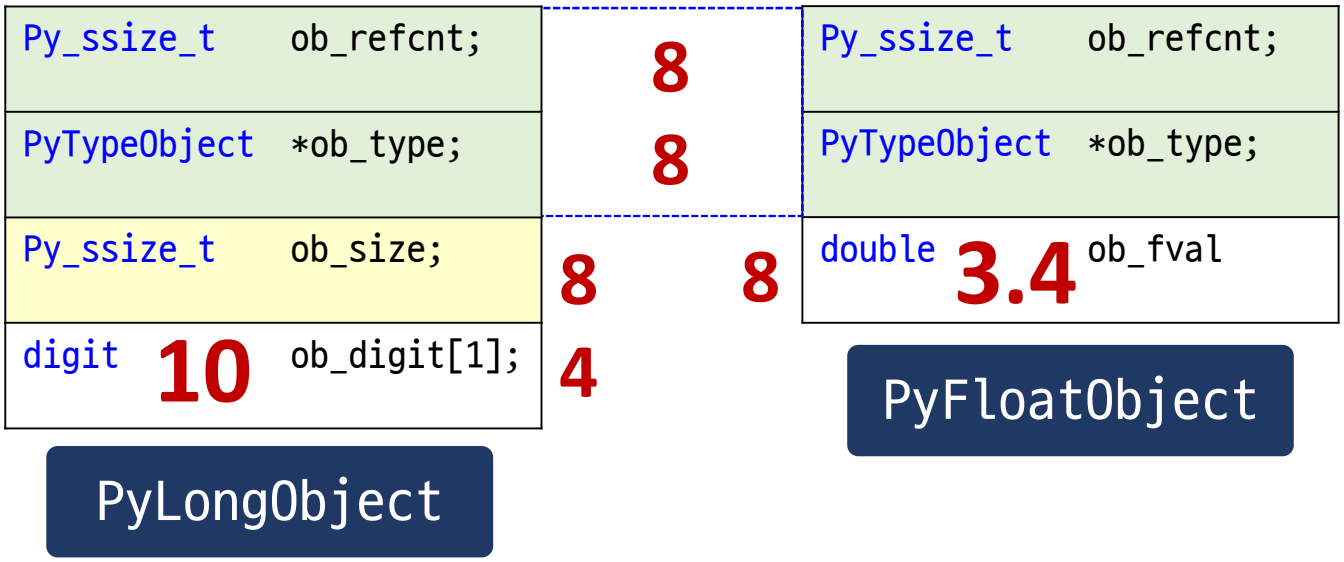
핵심 정리

- Python 에서 엑셀을 제어 하려면
  - ⇨ 외부 패키지를 설치 해야 한다.
  - ⇨ "**pip install 패키지 이름**"
  - ⇨ excel 파일을 읽고 쓰기 위한 다양한 패키지가 있다.
  - ⇨ **openpyxl**, xlrd, xlsxwrtier, xlwt, pyzlsb, pyexcel, pandas

- openpyxl 참고 사이트
  - ⇨ **https://openpyxl.readthedocs.io/en/stable/**



핵심 정리



- "**변수(객체)의 다양한 정보를 조사**" 하는 방법

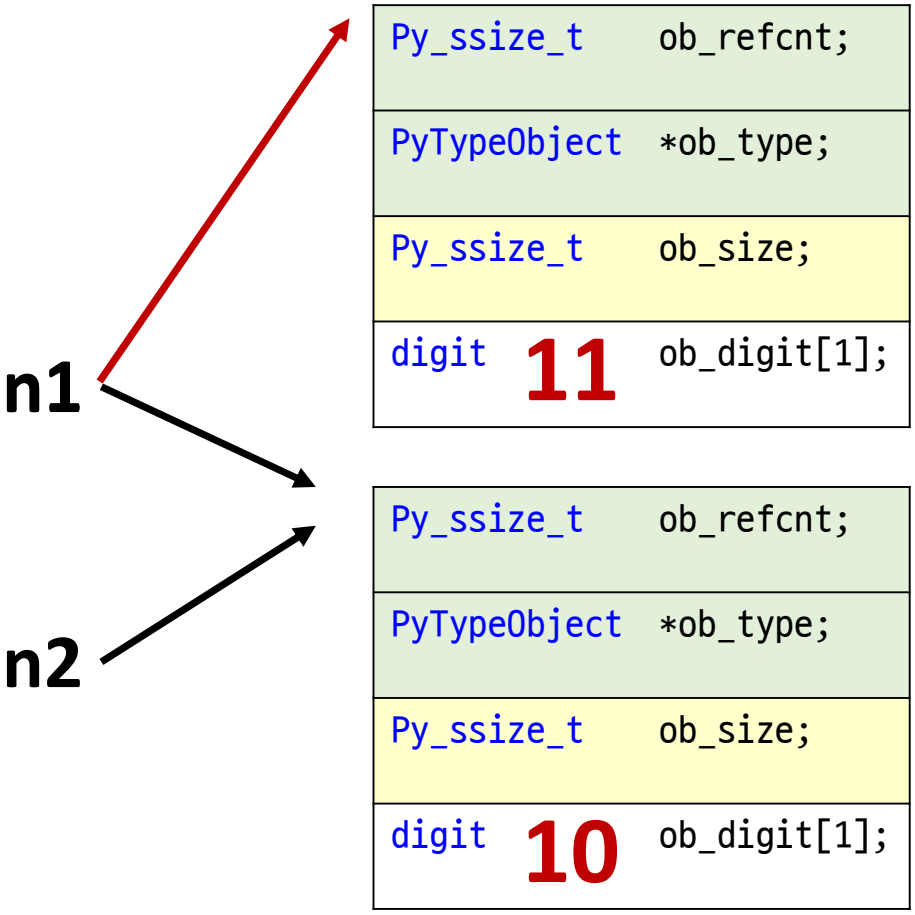
id(n)	객체의 id 값. CPython 에서 id 값은 주소
sys.getrefcount(n)	객체의 참조 계수
sys.getsizeof(n)	객체의 메모리 크기
isinstance(n, 타입)	객체가 특정 타입인지 조사
type(n)	객체의 타입 정보를 담은 "type" 객체 반환

핵심 정리

- == 연산자 vs is 연산자

s1 == s3	s1 과 s2의 "**값이 동일**"한지 조사
s1 is s3	s1과 s2가 "**동일한 객체**"를 가리키는지 조사

핵심 정리



mutable vs immutable

immutable	객체의 상태를 "**변경할 수 없는 것**" **int, float, str, tuple, ...**
mutable	객체의 상태를 "**변경할 수 있는 것**" **list, byte array, set, ...**

핵심 정리

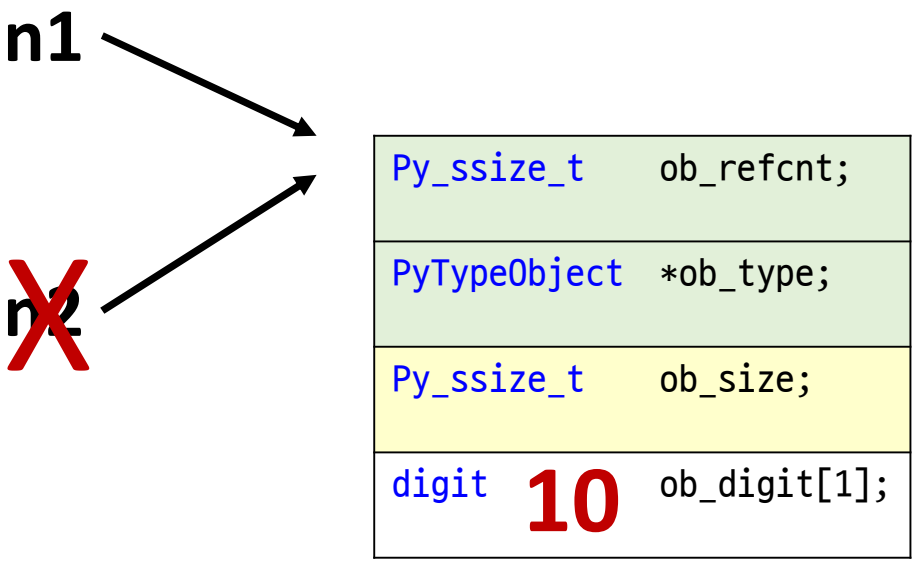
- **Object Interning**
  - ⇨ 속성이 동일한 immutable 객체를 공유 할 수 있게 하는 최적화 기술
  - ⇨ "**-5 ~ 256**" 까지의 정수 객체는 특별하게 관리된다.

-5	-4	-3	...	...	...	243	244	245	256

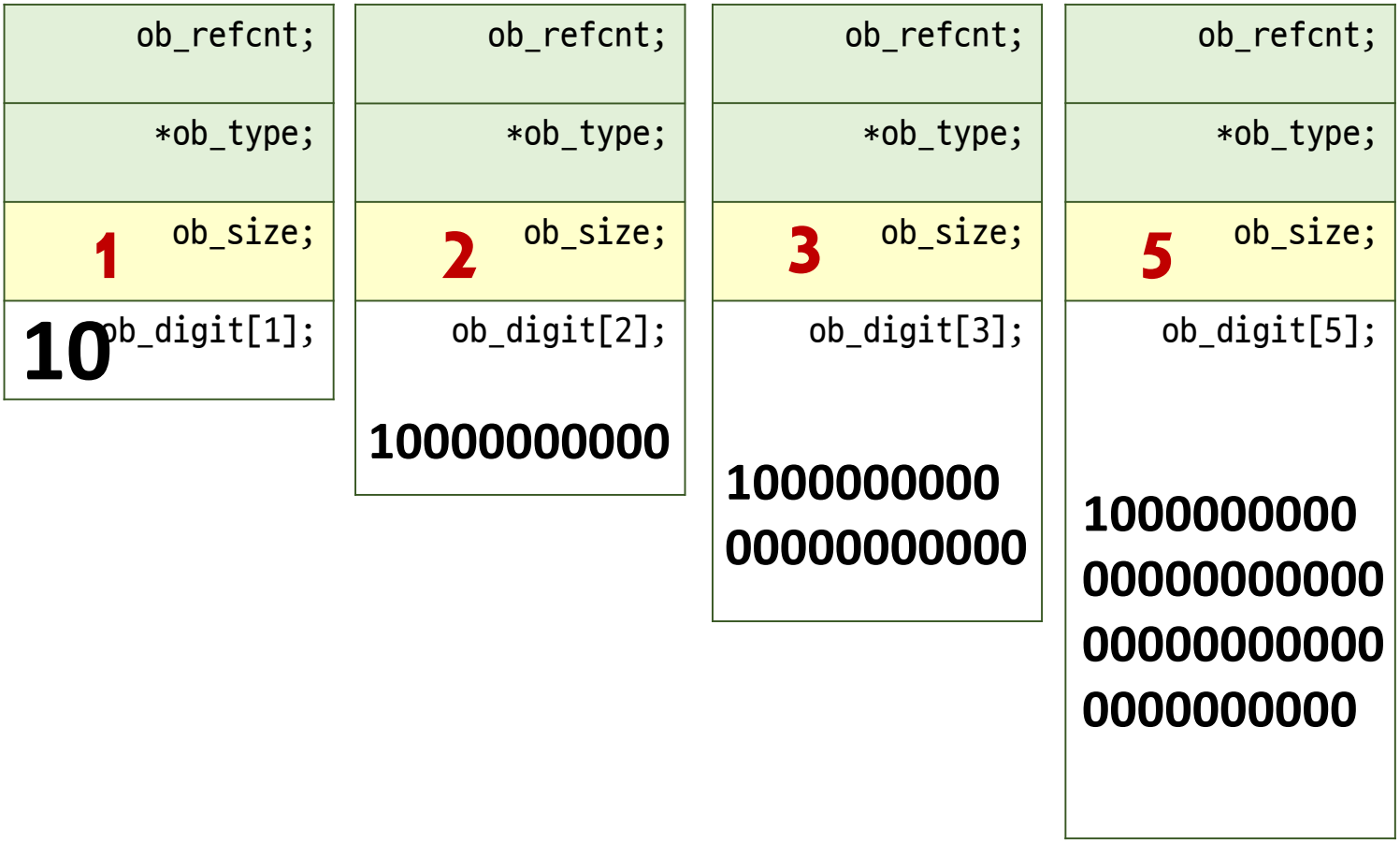
파이썬 버전에 따라 공유하는 알고리즘은 변경될 수 있다.

핵심 정리

- **del**
  - ⇨ 변수의 이름을 삭제
  - ⇨ 객체를 삭제하는 것이 아니라 "**변수의 이름만 제거, 객체의 참조계수 감소**"

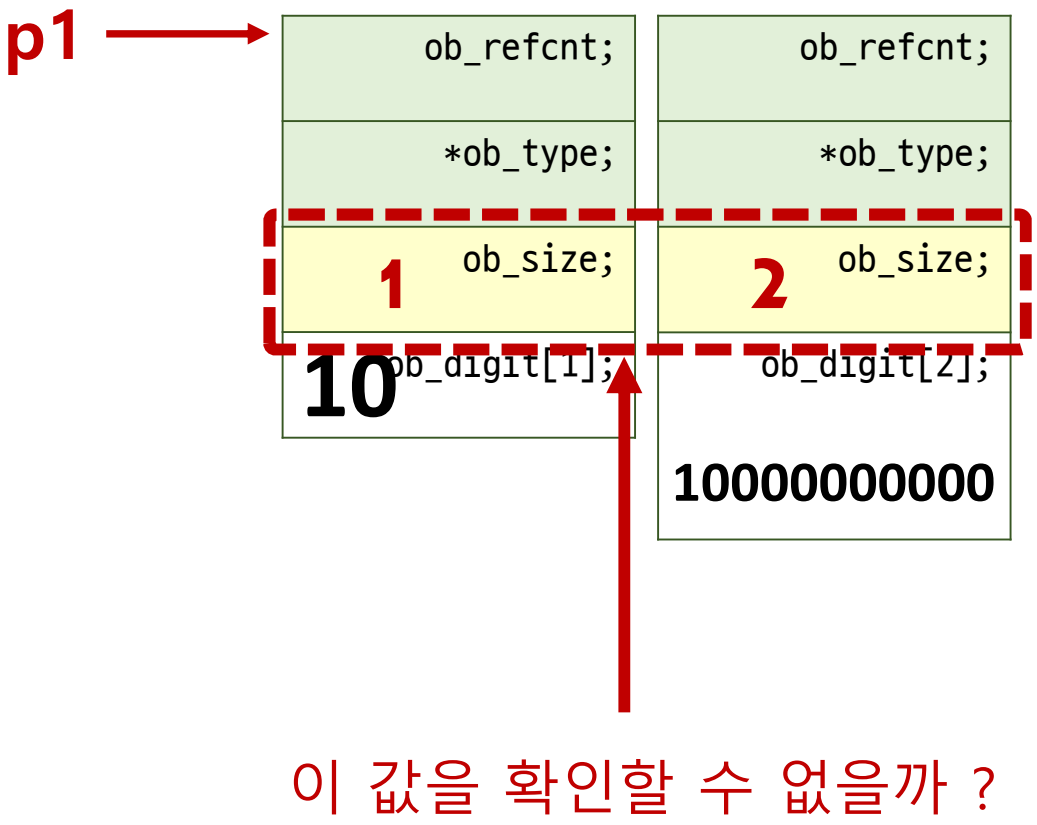


핵심 정리



- **PyLongObject**
  - ⇨ Python 의 정수 타입을 나타나는 CPython 의 구조체
  - ⇨ C 언어의 flexible array 기술 사용

핵심 정리



핵심 정리

- 임의의 타입에 대해서 동작 하는 함수를 만드는 기술
  - ⇨ "**interface**" 설계
  - ⇨ "**generic(template)**" 사용

- Python 의 함수
  - ⇨ 기본적으로 "**모든 객체를 받을 수 있다**."
  - ⇨ 함수가 사용하는 연산을 객체가 제공하지 않으면 예외 발생

핵심 정리

- 표준 타입

정수형	int
실수형	float
문자열	str
기타	bool, NoneType, range, slice, ...
Sequence	list, tuple, set, dictionary, ...

- 타입도 "**함수**"

핵심 정리

- **모든 것은 객체이다.**

  ⇨ int 타입의 객체에도 "**메소드**"가 있다.

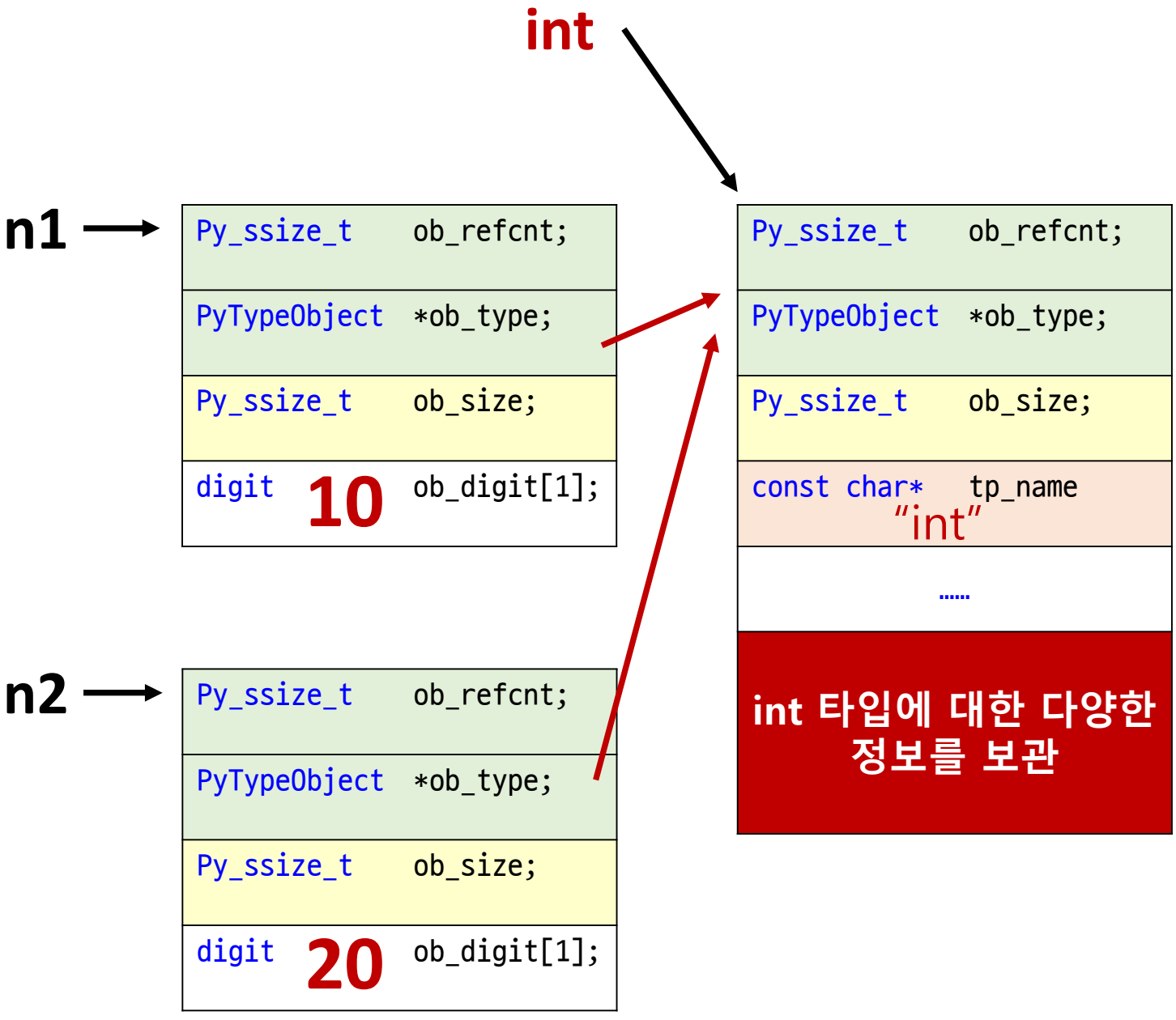
  ⇨ "**dir(타입)**" 또는 "**dir(변수)**" 으로 확인 가능

- **모든 타입은 object 로 부터 파생된다.**

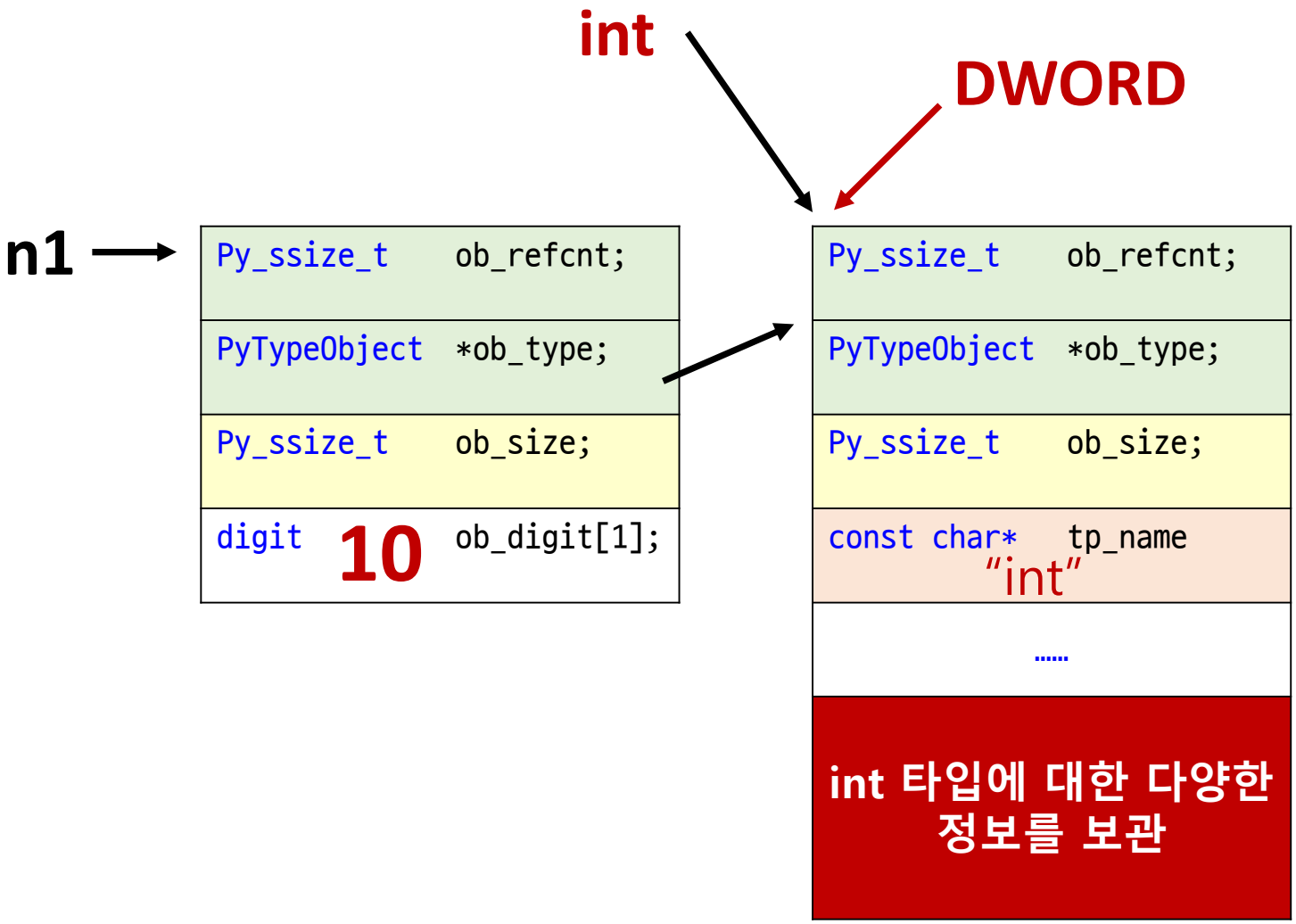
  ⇨ 상속 계층 조사.

issubclass(Derived, Base)	파생 클래스 여부 조사
타입이름.mro()	상속 계층의 타입을 리스트로 반환

핵심 정리



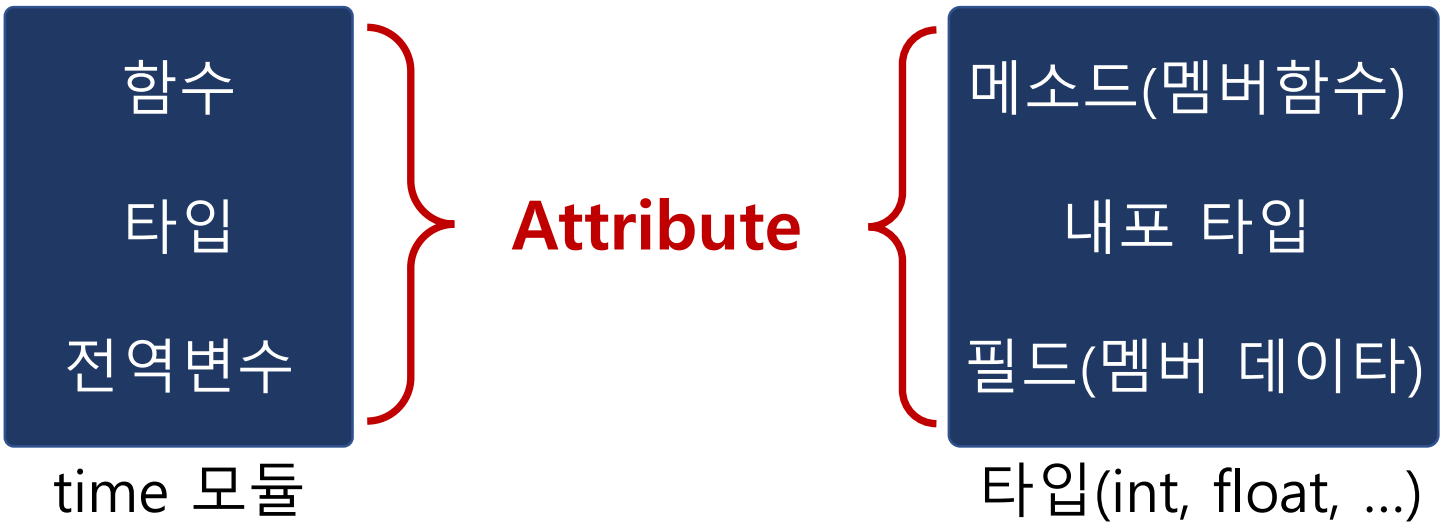
핵심 정리



print(**변수**)	**변수의 값**
print(**타입**)	**< class '타입이름' >**

- **"타입을 함수 인자 또는 반환 값으로 사용"**할 수 있다.
- type() 함수의 반환 값은 "**타입객체**" 이다.

핵심 정리



dir(object)	object 가 가진 attribute 의 목록을 리스트로 반환 object 는 **모듈/타입/변수** 모두 가능
getattr(object, '이름')	object 에서 해당 attribute 를 찾아서 반환

- **builtins** 모듈
  - ⇨ python 표준 함수와 표준 타입을 제공하는 모듈
  - ⇨ 명시적으로 import 하지 않아도 사용가능 하다.

핵심 정리

- "**모든 것은 객체이다.**"

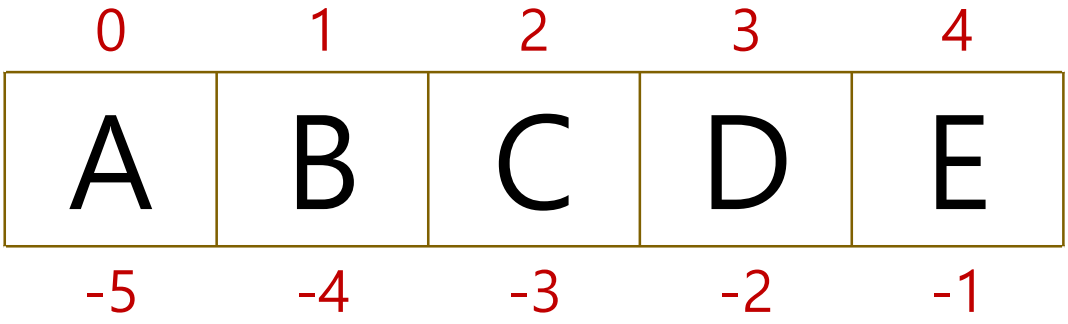
  ⇨ int, float, str 타입도 메소드가 있다.

  ⇨ **dir(str)**

- str 의 주요 메소드

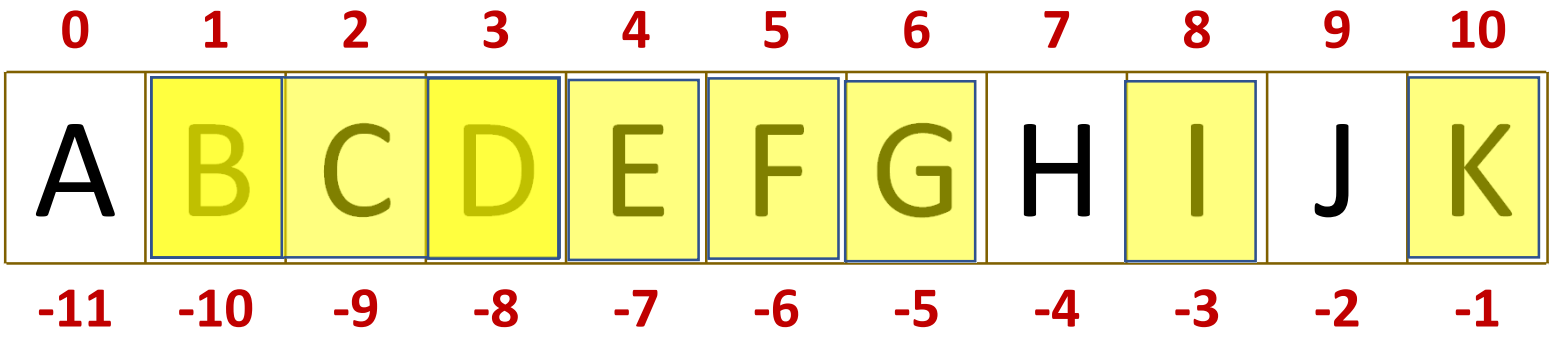
upper, lower	대문자(소문자)로 변경한 새로운 문자열 반환
count	문자열 안에 특정 문자열이 몇 번 나오는가 ?
find, index	문자열이 처음 나오는 곳의 위치 없으면 -1(find) 또는 ValueError예외(index)
strip, lstrip, rstrip	공백 제거
replace	치환
join	기존 문자열 사이에 새로운 문자열 넣기
split	문자열을 각 단어로 분리해서 list 에 담아서 반환

핵심 정리



핵심 정리

indexing	[] 연산자를 통해서 한 글자만 반환
slicing	[] 연산자를 통해서 여러 글자 반환

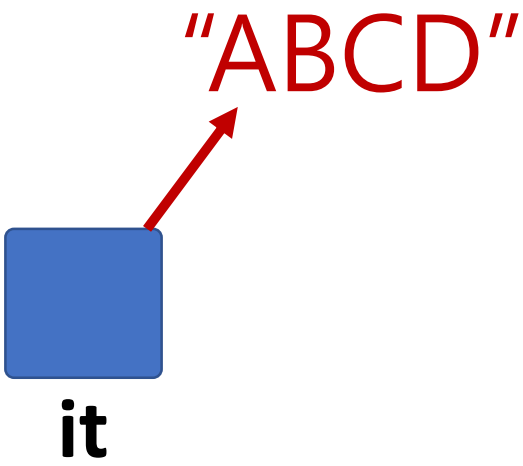



```
slice(first, last, step)
slice(count)
```

- slicing 을 하려면
  - ⇨ slice 객체를 생성해서 [slice객체] 를 하거나
  - ⇨ **[: :] 표기법 사용**

핵심 정리

- iterator
  - ⇨ 객체가 가진 각 요소에 순차적으로 접근하는 도구
  - ⇨ Python 에서는 "**iter()**" 표준 함수를 통해서 얻을 수 있다.



- 반복자 관련 기본 함수

iter()	반복자를 꺼내는 함수
next()	반복자가 가리키는 값을 반환하고, 반복자는 다음으로 이동
reversed()	역 반복자 반환

핵심 정리

- "**iterable**" type
  - ⇨ iter() 함수로 반복자를 얻을 수 있는 타입
  - ⇨ str, list, tuple, dictionary, set ...

- "**range**" type
  - ⇨ 반복자를 통해서 주어진 구간의 값을 순차적으로 얻을 때 사용하는 타입

`range(10)`	`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
`range(1, 10)`	`1, 2, 3, 4, 5, 6, 7, 8, 9`
`range(1, 10, 2)`	`1, 3, 5, 7, 9`
`range(9, 1, -3)`	`9, 6, 3`

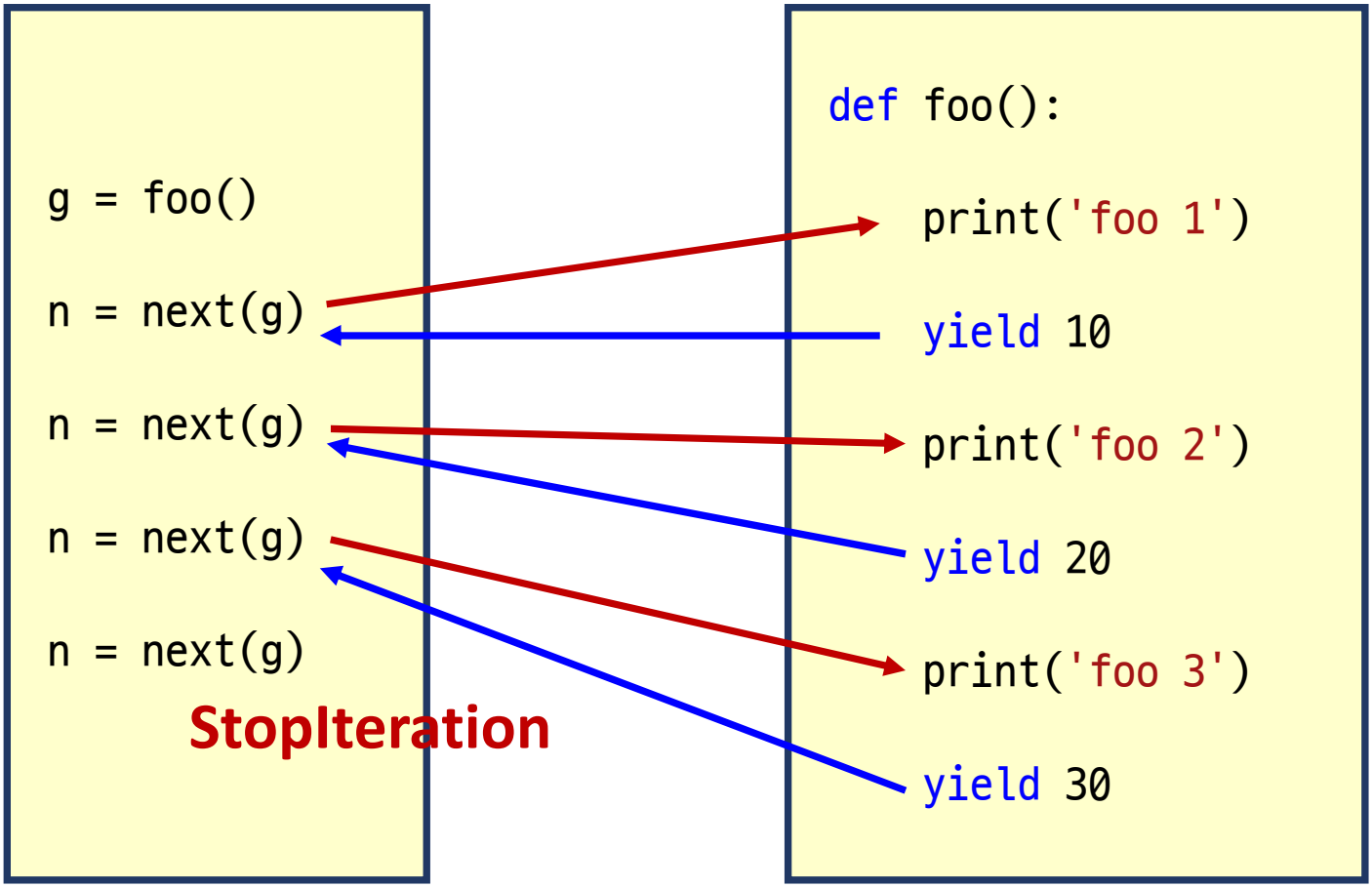
핵심 정리

- for 문

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

핵심 정리

- generator
  - ⇨ **yield 키워드를 사용해서 반환하는 함수**
  - ⇨ 반환 값이 "**generator**" 이다.
  - ⇨ generator 는 "**반복자와 유사하게 동작**"한다.



핵심 정리

- generate infinite sequence

- 지연된 실행

  ⇨ **미리 값을 구하지 않고, 필요할 때 연산을 수행**

핵심 정리

- **Generator**

  ⇨ yield 반환을 사용하는 함수

- **Generator Expression (제너레이터 표현식)**

  ⇨ Generator를 반환하는 표현식

  ⇨ 함수를 사용하지 않고, "**표현식으로 generator를 만드는 문법**".

  ⇨ 괄호( )가 필요

  ⇨ list, tuple 등의 시퀀스를 만들 때 많이 사용

"Coroutine" 강좌 참고

핵심 정리

- 시퀀스 (sequence)

  ⇨ 한 개의 이름으로 여러 개의 객체를 관리 할 때 사용

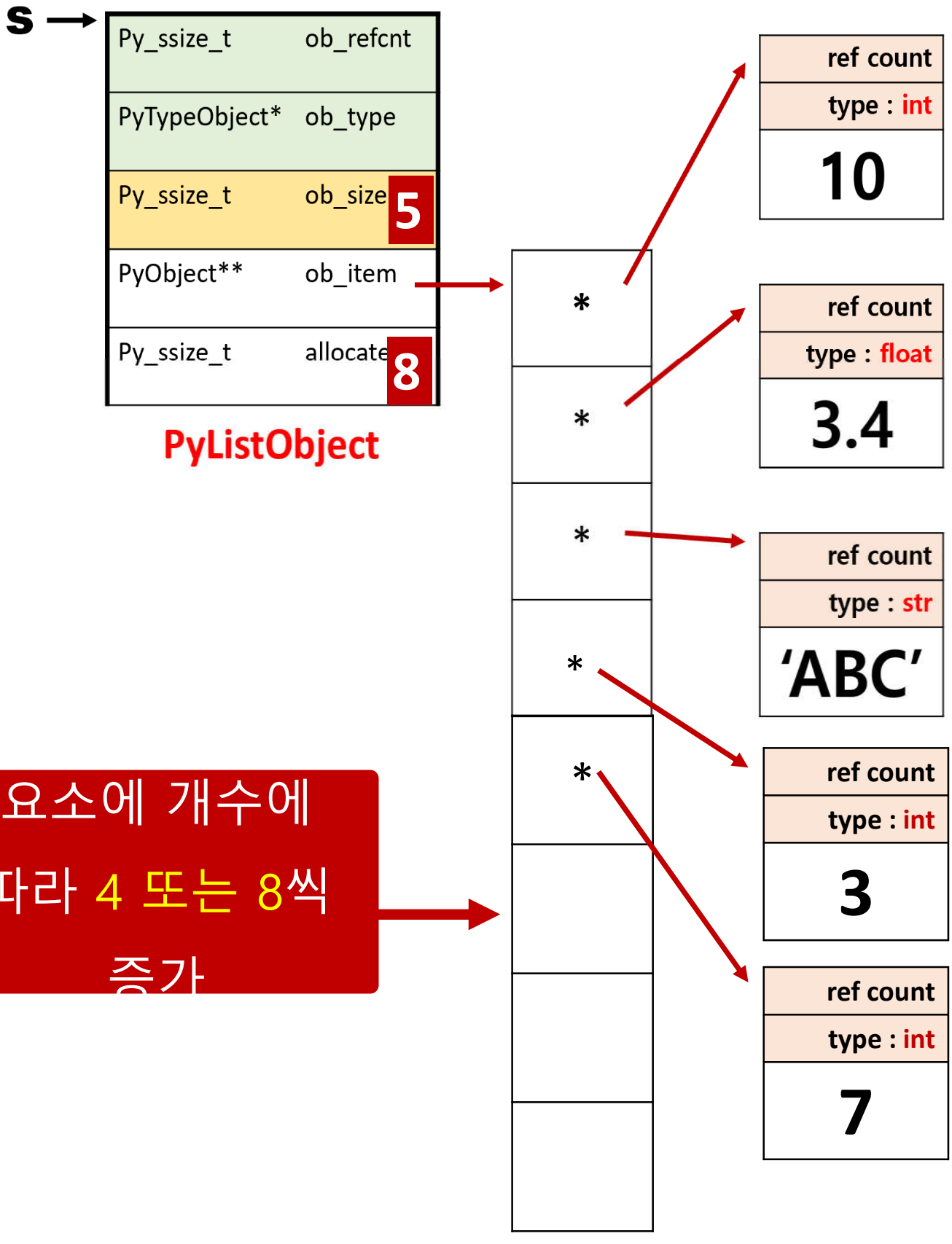
- Sequence 의 종류

mutable	list, set, dictionary, bytearray collection.deque, array.array
immutable	tuple, str, byte

- Sequence 표기법

list	[10, 3.4, 'ABC']
tuple	(10, 3.4, 'ABC')
set	{ 10, 3.4, 'ABC' }
dictionary	{ 'x':10, 'y' : 20 }

핵심 정리



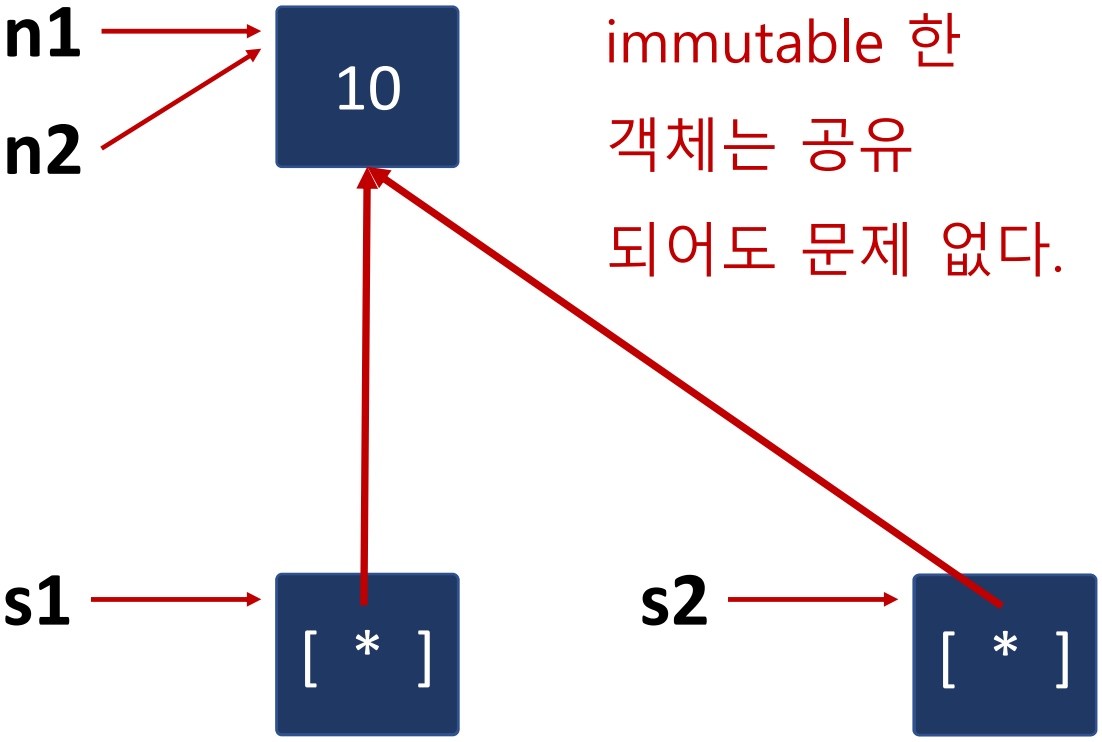
핵심 정리

- list 활용
  - ⇨ "**함수가 여러 개의 값을 반환**"하거나
  - ⇨ "**여러 개의 값을 인자로 받을 때**"
  - ⇨ list를 사용하는 경우가 많이 있다

핵심 정리

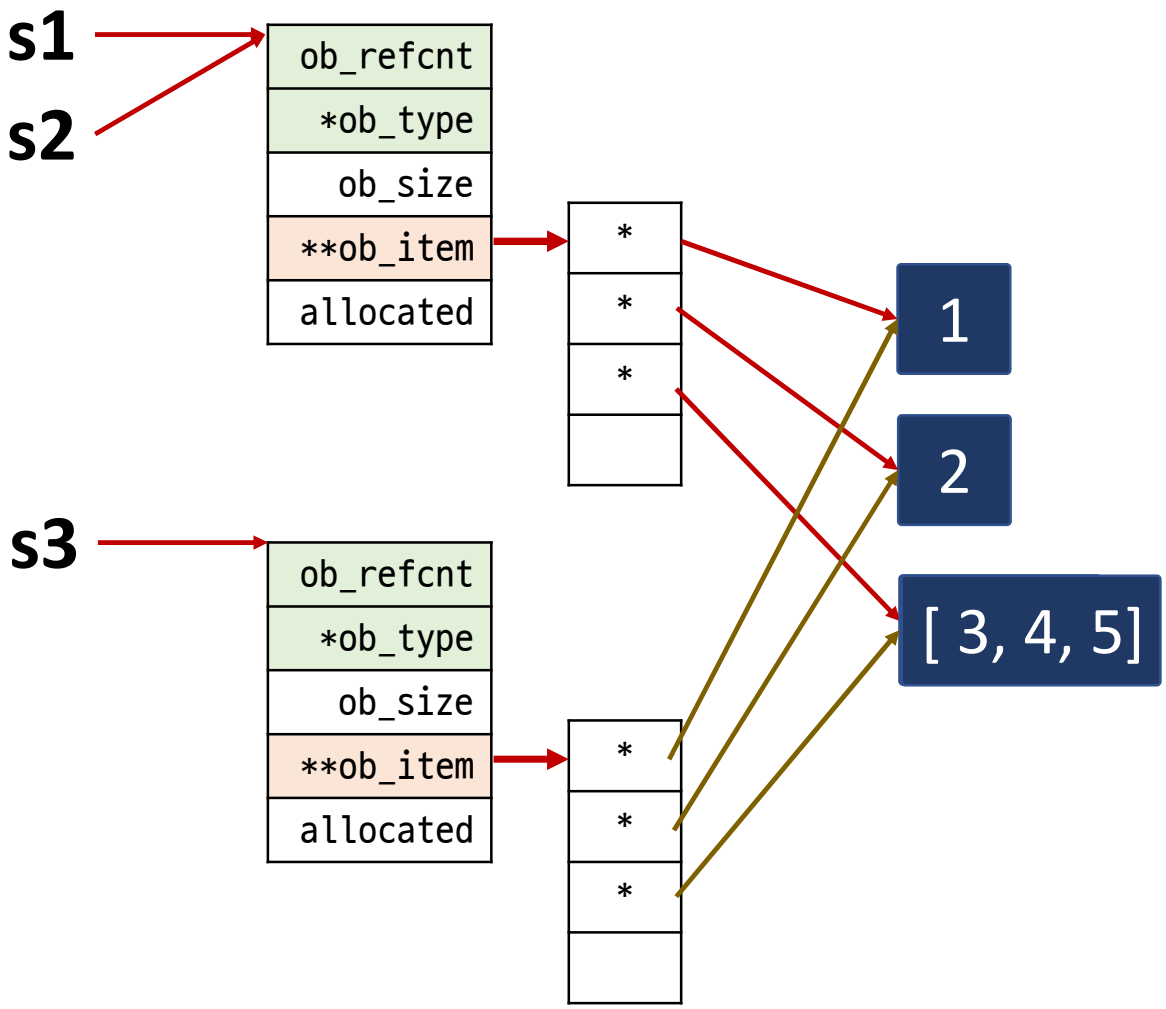
- list 타입 사용
- [ ] 연산자 사용
- 지능형 리스트(List Comprehension)

핵심 정리

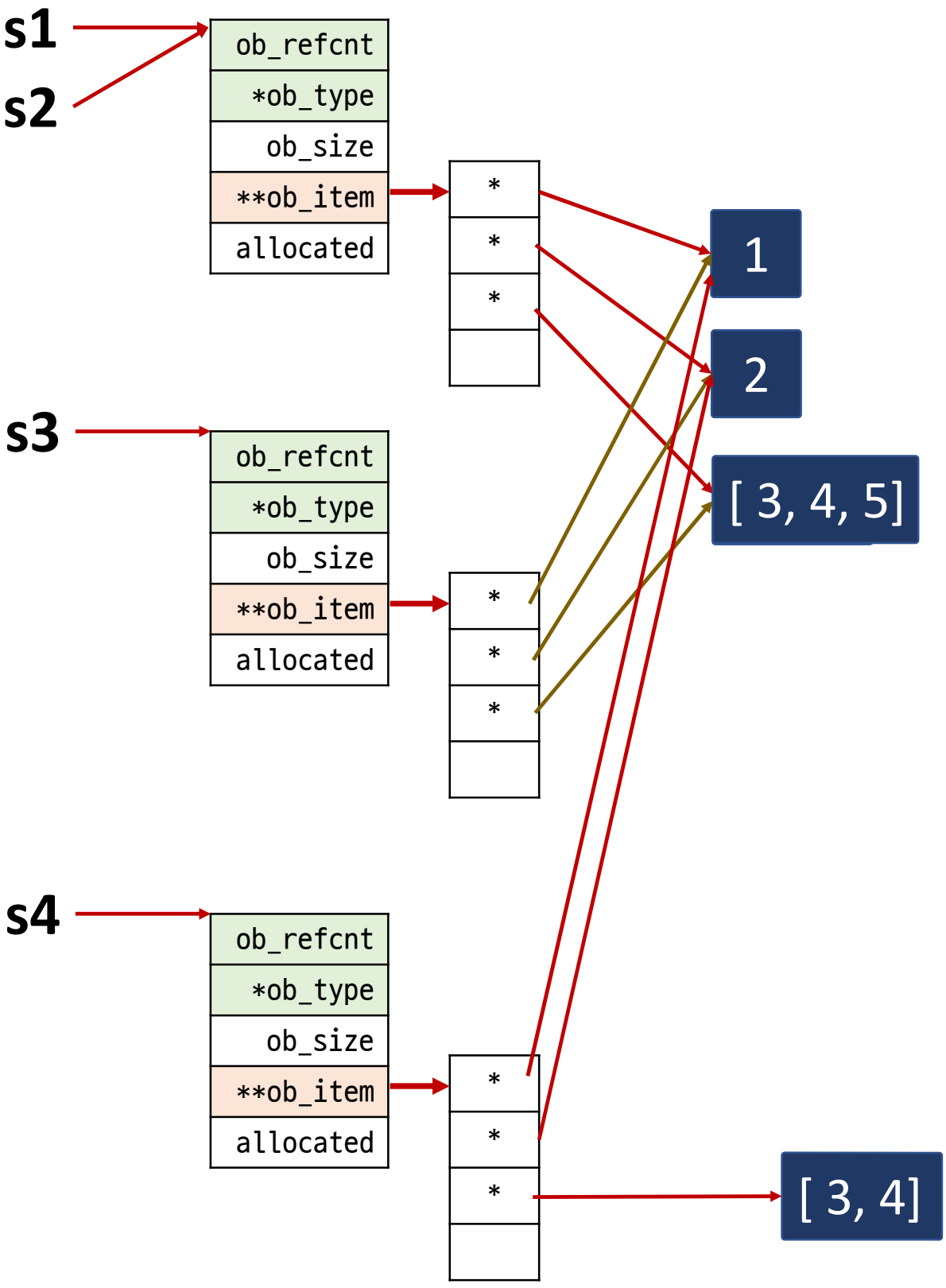


immutable 한
객체는 변할 수
있으므로
공유되지 않는다.

핵심 정리



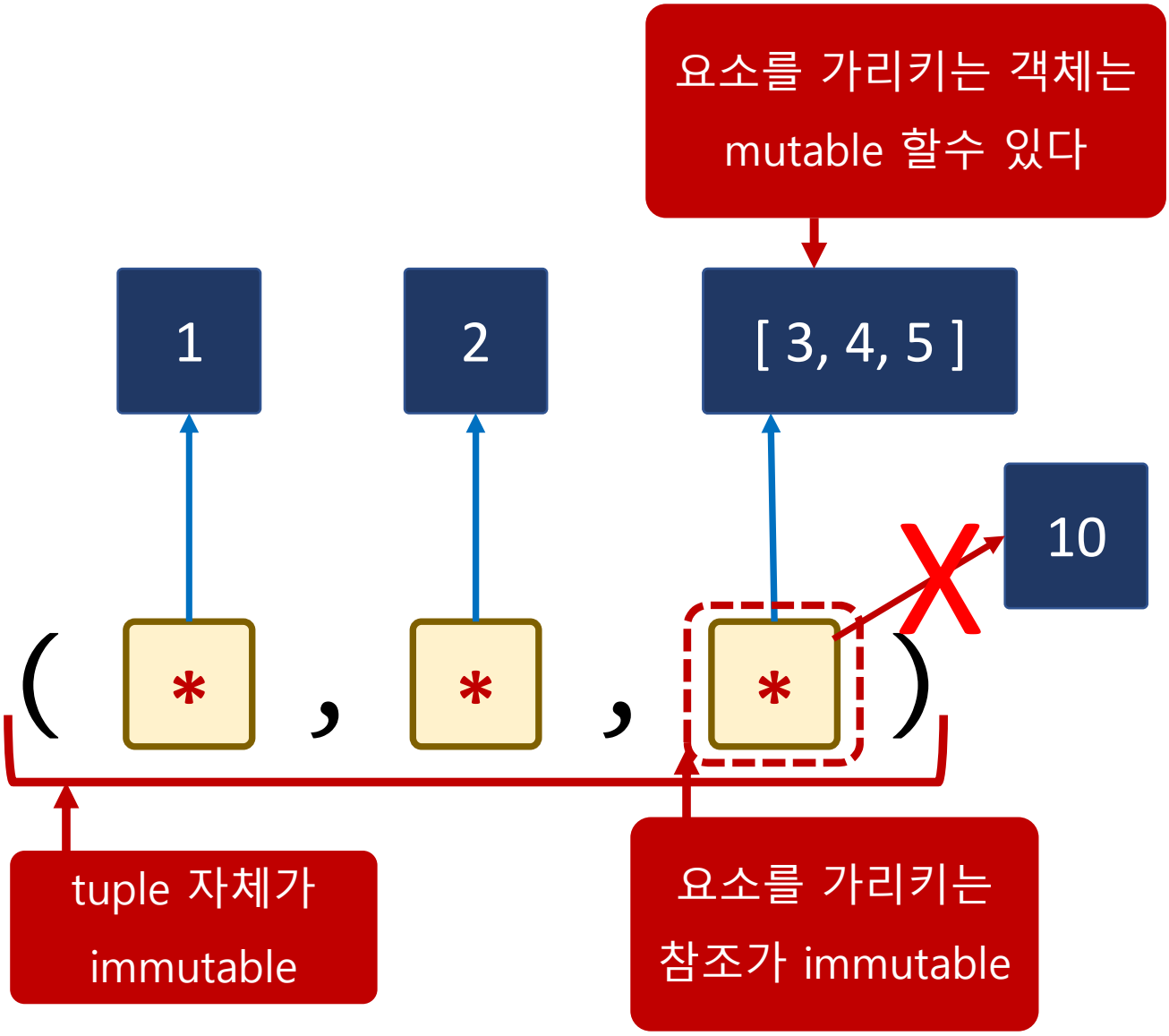
핵심 정리



핵심 정리

- list vs tuple

list	[1, 2, 3]	mutable
tuple	(1, 2, 3)	immutable



핵심 정리

- tuple 객체를 생성하는 방법
  - ⇨ tuple 이름을 사용
  - ⇨ ( ) 를 사용하는 방법 (정확히는 **콤마(,)** 를 사용)

핵심 정리

- enumerate
  - ⇨ 반복자 처 럼 동작 하는 객체
  - ⇨ iterable 한 객체를 인자로 받아서 생성
  - ⇨ 순회시 "**(index, 요소)**" 형태로 반환

```
s = ['apple', 'banana']
        0, 'apple'
        1, 'banana'
for   idx, t    in enumerate(s):
    print(idx, t)
```

핵심 정리

- set

  ⇨ "**집합 연산(교집합, 합집합, 차집합 등) 을 제공**"하는 sequence

핵심 정리

- 키 값을 가지고 데이터를 저장하는 자료구조

'name'	'kim'
'age'	20
'addr'	'seoul'

핵심 정리

- **함수를 인자로 전달 받는 add_emoticon 함수**

  ⇨ 인자로 전달 받은 함수를 다시 호출한다.

  ⇨ 함수 호출 "**전/후 에 작업을 추가**"할 수 있다.

```
def add_emoticon(func):
    print(':-) ', end='')
    func()
    ……
```

- add_emoticon 함수는 기존 함수에 "**기능(이모티콘 출력)을 추가**"한다.

핵심 정리

- 생각해 볼 문제

  ⇨ 기존코드에 “say_hello()” 를 사용하고 있었는데

  ⇨ “**이모티콘이 추가된 hello 메시지로 변경**”하려면...

```
say_hello() ➔ add_emoticon(say_hello)
```

“**함수의 이름도 변경**” 되고,

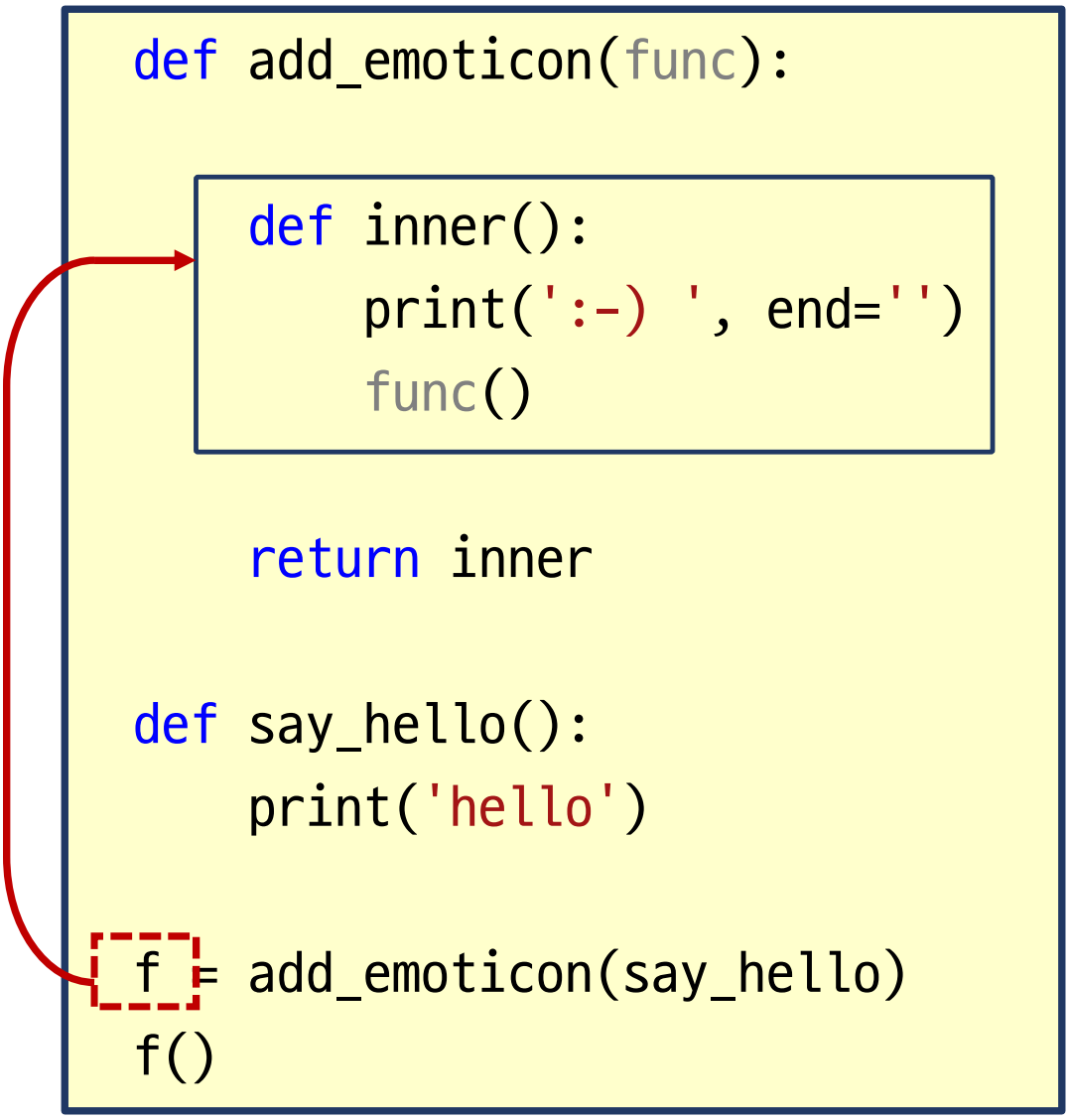
“**함수의 사용법(인자 모양)도 변경**”되어야 한다.

함수의 “**이름과 사용법을 동일하게 유지**” 하면서

“**기능을 추가**”할 수 없을까 ?

핵심 정리

- inner 함수를 사용하는 add_emoticon



핵심 정리

- say_hello 함수에 "**add_emoticon 의 기능을 추가**"하고 싶으면

```
say_hello = add_emoticon(say_hello)
```

핵심 정리

- Decorator 안의 inner 함수는
  - ⇨ 다양한 형태의 인자를 모두 받을 수 있어야 한다.
  - ⇨ 자신이 받은 모든 인자를 원본 함수(say_hello) 에 그대로 전달해야 한다. ("**perfect forwarding**"-C++용어)
  - ⇨ 원본함수의 반환 값도 그대로 반환해야 한다.

핵심 정리

- **Decorator 의 기본 모양**

```python
from functools import wraps

def add_emoticon(func):
    # 초기화가 필요한 경우

    @wraps(func)
    def inner( *args, **kwargs ):
        # 사전 작업
        result = func( *args, **kwargs )
        # 사후 작업
        return result

    return inner
```

핵심 정리

```
say_hello = add_emoticon(say_hello)
```

inner 함수

```
say_hello = add_emoticon('^-^;')(say_hello)
```

```
decorator(say_hello)
```

inner 함수

- add_emoticon 은 "**이모티콘을 인자로 받고 함수를 반환**" 해야 한다.
- add_emoticon 이 반환한 함수는 "**함수를 인자로 받고 inner 함수 반환**" 해야 한다.

핵심 정리

- **커링(Currying)**
  - ⇨ "**인자가 2개인 함수를 인자가 한 개인 함수의 연속적인 호출**"로 사용 (또는, 인자가 5개인 함수를 3개, 2개의 인자로 나누어 호출)
  - ⇨ 함수형 언어에서 널리 사용되는 기술

- **@register(obj) decorator**
  - ⇨ 함수에 기능을 추가하기 위한 목표가 아니라 "**함수를 객체에 등록**"하기 위한 decorator
  - ⇨ register 함수 자체는 "**최초에 한번만 호출**"된다는 특징을 활용한 기술

핵심 정리

- 함수가 한 이상의 값을 반환 하려면

list	반환 타입의 개수가 일정 하지 않을 때
tuple	반환 타입의 개수가 일정 할 때

- 반환 값이 없는 함수의 반환 값을 받는 경우

  ⇨ None

핵심 정리

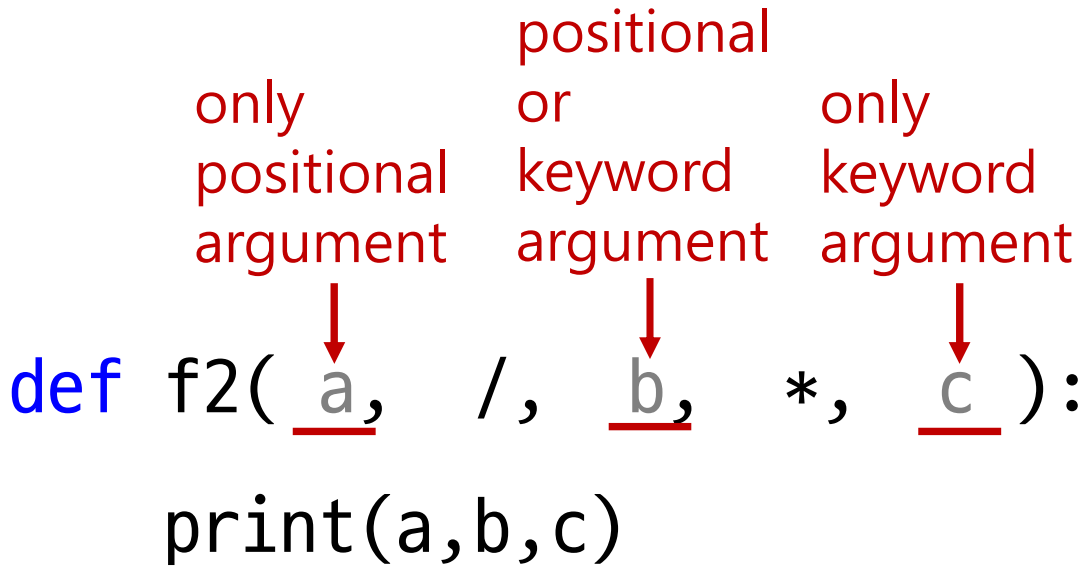
- 함수 인자를 전달하는 방법

positional argument	위치에 따른 인자 전달
keyword argument	매개 변수의 이름으로 인자 전달

- 주의!

  ⇨ 모든 positional argument 는 반드시 keyword argument 앞에 있어야 한다.

- **/, *** 를 사용한 매개 변수 표기

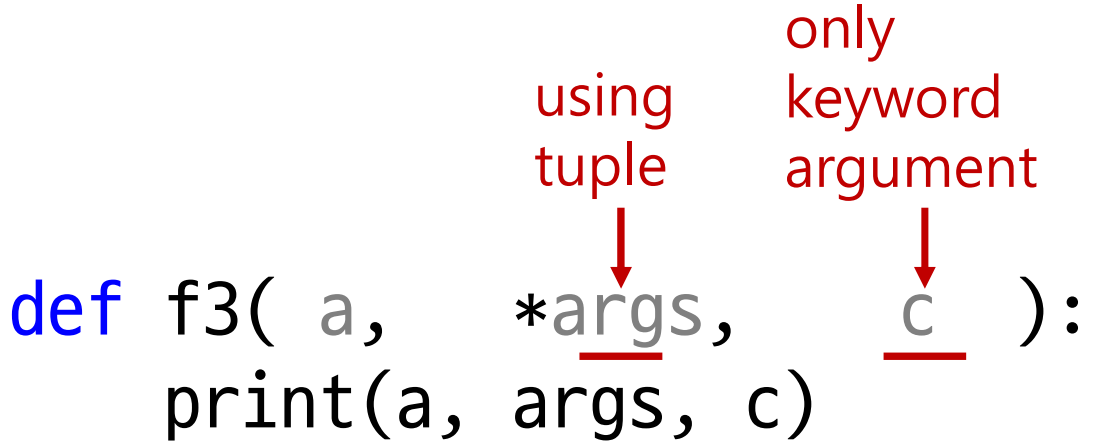


핵심 정리

- default parameter
  - ⇨ 인자를 전달하지 않으면 디폴트 값이 적용
  - ⇨ "**마지막 인자부터 차례대로 지정**"해야 한다. (단, keyword only argument 는 가능)

- 주의!
  - ⇨ mutable 한 타입의 경우는 버그의 원인이 될 수 있다.
  - ⇨ "**함수도 객체이다.**" 강좌 참고

핵심 정리




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

핵심 정리

- parameter pack

f(***args**)	모든 positional argument 를 tuple로 받음
f(****kwargs**)	모든 keyword argument 를 dictionary 로 받음

- "**모든 positional argument 는 반드시 keyword argument 앞에 있어야**" 한다.

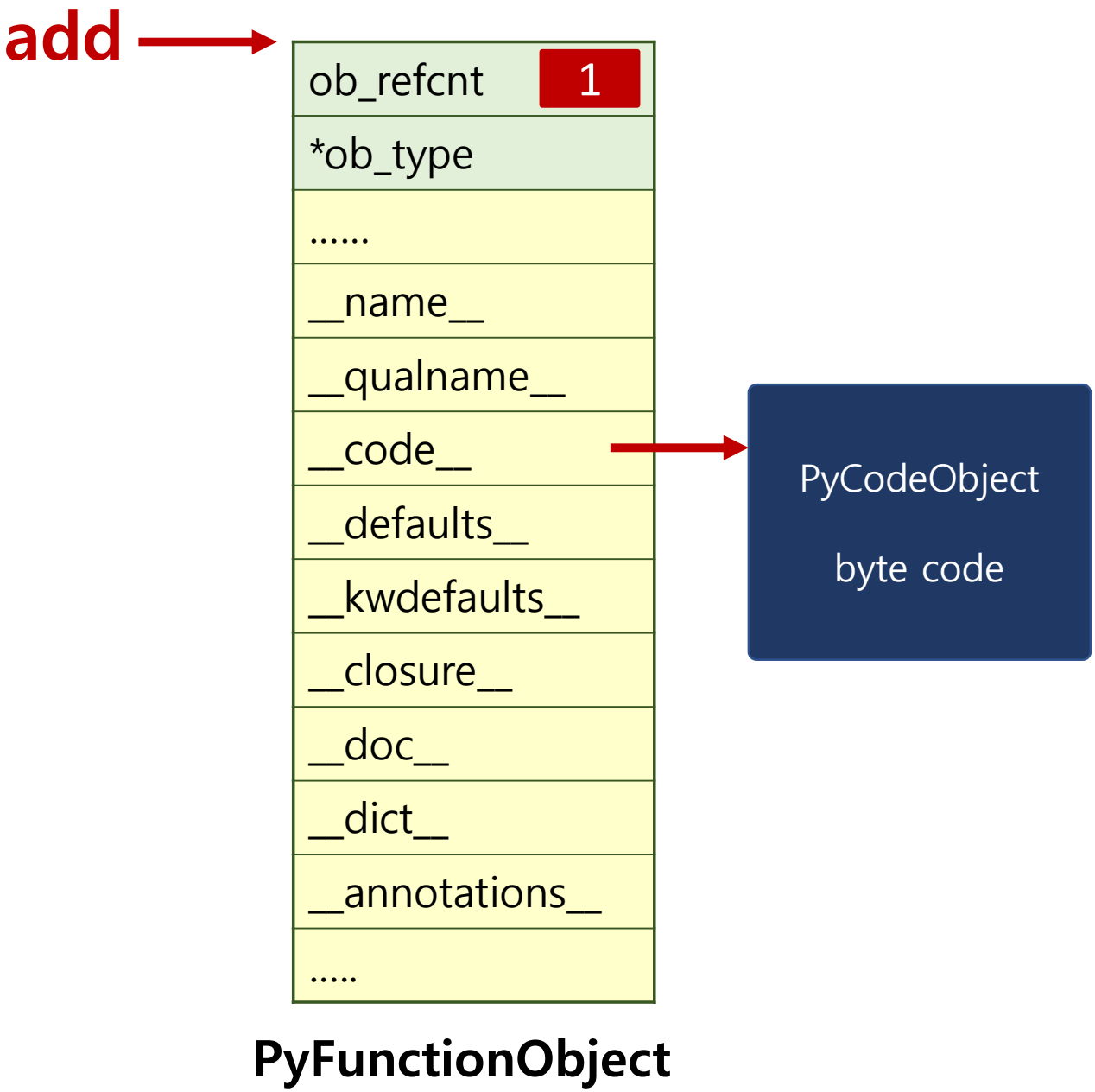
- perfect forwarding
  - ⇨ 전달 받은 모든 인자를 다른 함수 에게 전달하는 기술(C++ 용어)
  - ⇨ Python 에서는 "**Decorator**" 만들 때 널리 사용

```
def chronometry(f, *args, **kwargs):
    f(*args, **kwargs)
```

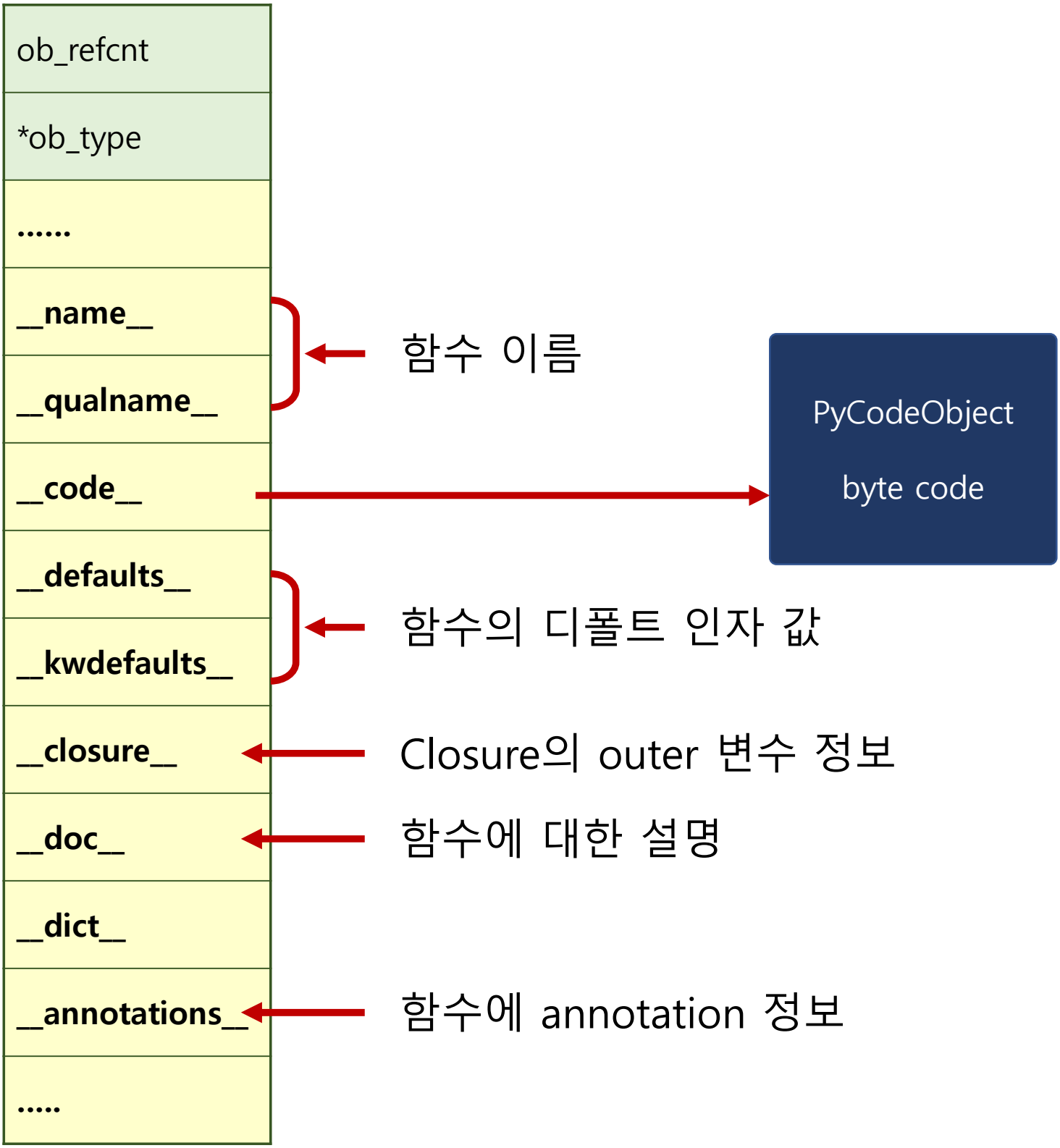
핵심 정리

- inner function, nested function
  - ⇨ 함수 안에 만드는 함수
  - ⇨ “**decorator**” 만들 때 널리 사용

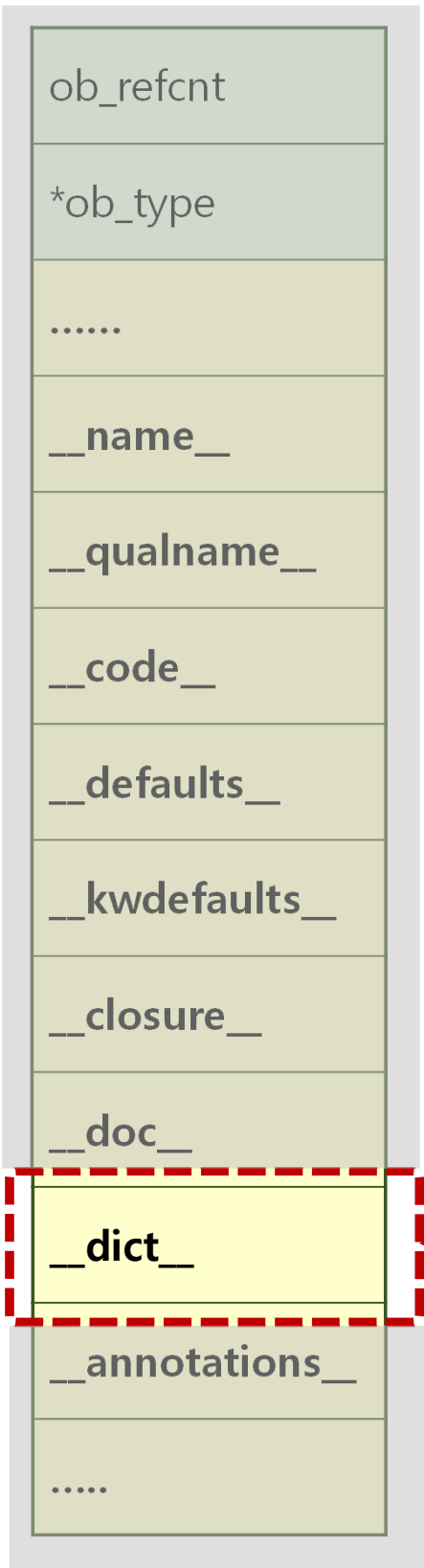
핵심 정리



핵심 정리



핵심 정리



함수와 관련된 다양한 정보를 보관.
"Decorator" 등 다양한 고급 기법에서 많이 활용

dictionary

핵심 정리

- **사용자 정의 함수** vs **built-in-functions**

  ⇨ built-in-functions 안에는 **“__dict__ 멤버”**가 없다.

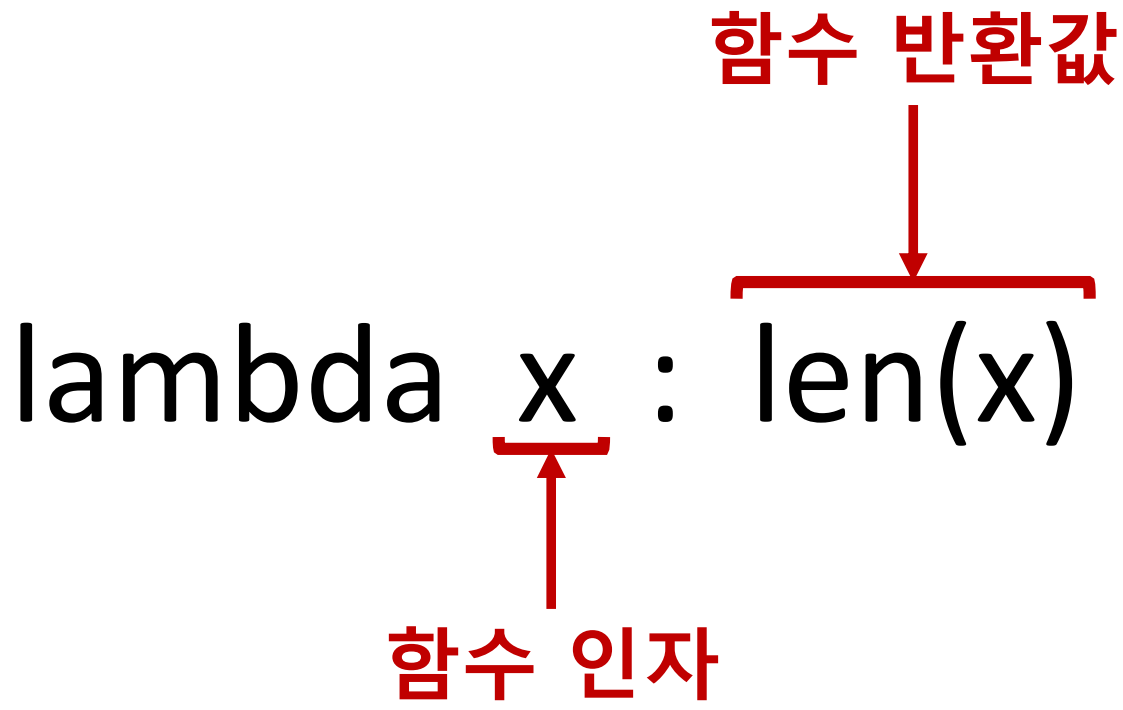
  ⇨ **dir(add) vs dir(sum)**

핵심 정리

- **mutable 타입을 디폴트 값으로 사용하지 말라.**

  ⇨ None 을 사용하는 것이 좋다.

핵심 정리

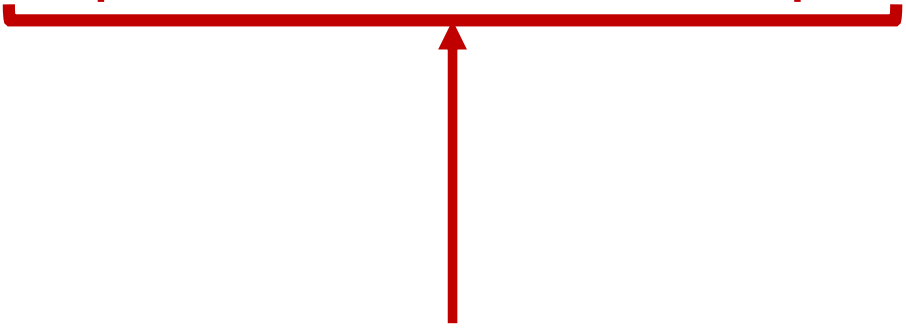


핵심 정리

- **람다표현식에서 if 사용하기**

  ⇨ : 없이 표현

```
lambda x : exp1 if 조건 else exp2
```



조건을 만족하면 exp1 반환

만족하지 못하면 exp2 반환

핵심 정리

- **first class object (일급 객체) 란 ?**
  - 모든 요소는 "**함수의 실제 매개변수**"가 될 수 있다.
  - 모든 요소는 "**함수의 반환 값**"이 될 수 있다.
  - 모든 요소는 "**할당 명령문의 대상**"이 될 수 있다.
  - 모든 요소는 "**동일 비교의 대상**"이 될 수 있다.

- int, float, str 타입의 모든 객체는 일급 객체 이다.
- Python 에서는 "**함수도 일급 객체**" 이다.
- 함수형 언어에서 많이 사용되는 기술 (map, filter, zip, reduce)

핵심 정리

- 함수형 언어는 “map”, “filter” 등의 함수를 널리 활용

핵심 정리

- class 키워드
- 객체를 생성하는 방법
- 메소드 호출의 원리

```
c.stop() ➔ stop(c)
```

- 메소드(멤버 함수) 종류

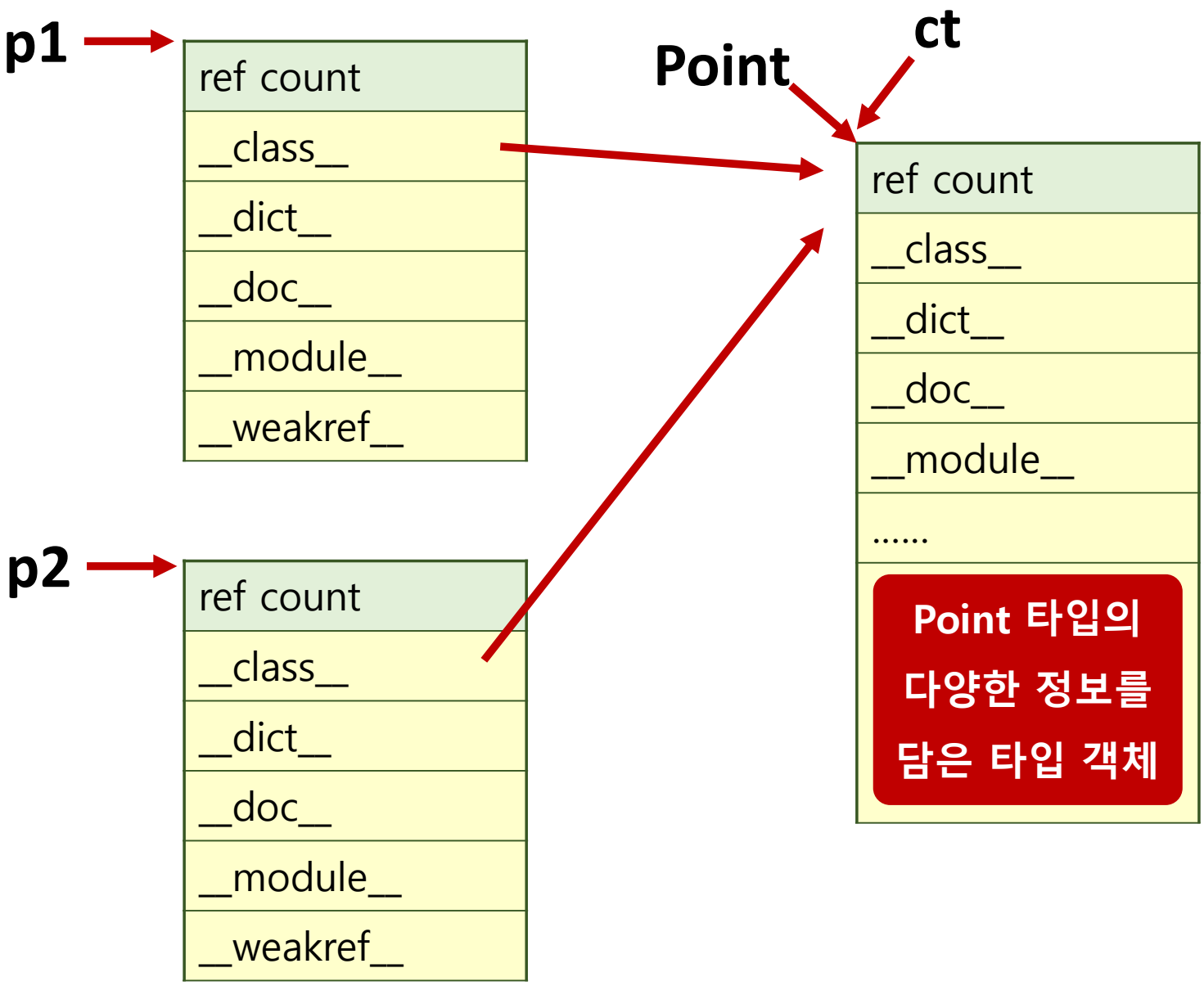
인스턴스 메소드	1번째 인자로 self 를 가진다.
정적 메소드	인자로 self 를 사용하지 않는다.

핵심 정리

- constructor 와 desctructor

constructor	**__init__** 이라는 이름의 메소드
destructor	**__del__** 이라는 이름의 메소드

핵심 정리

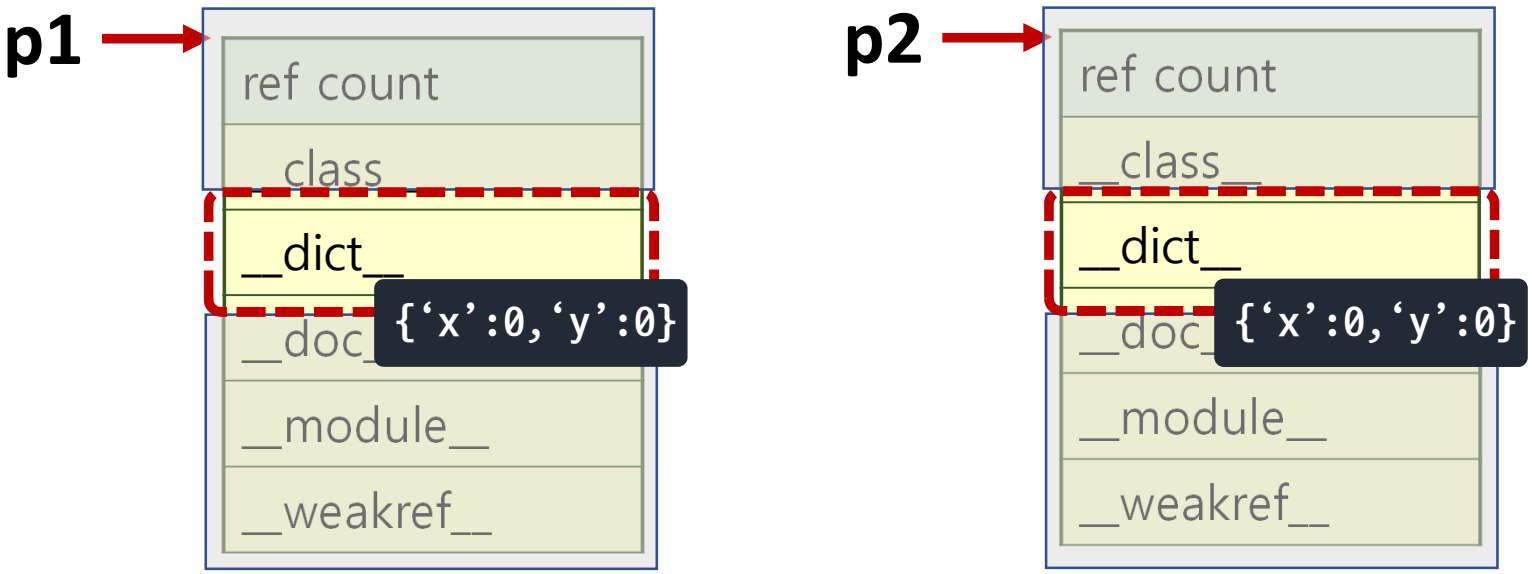


- 객체.__class__

  ⇨ 객체의 타입

핵심 정리

- **인스턴스 필드 (instance field)**
  - ⇨ 멤버 데이터
  - ⇨ 객체당 따로 보관 되는 데이터



- **인스턴스 필드를 추가 하려면**
  - ⇨ __init__ 메소드에서 필드를 만들고 초기화.
  - ⇨ __dict__ 에 추가된다.

핵심 정리

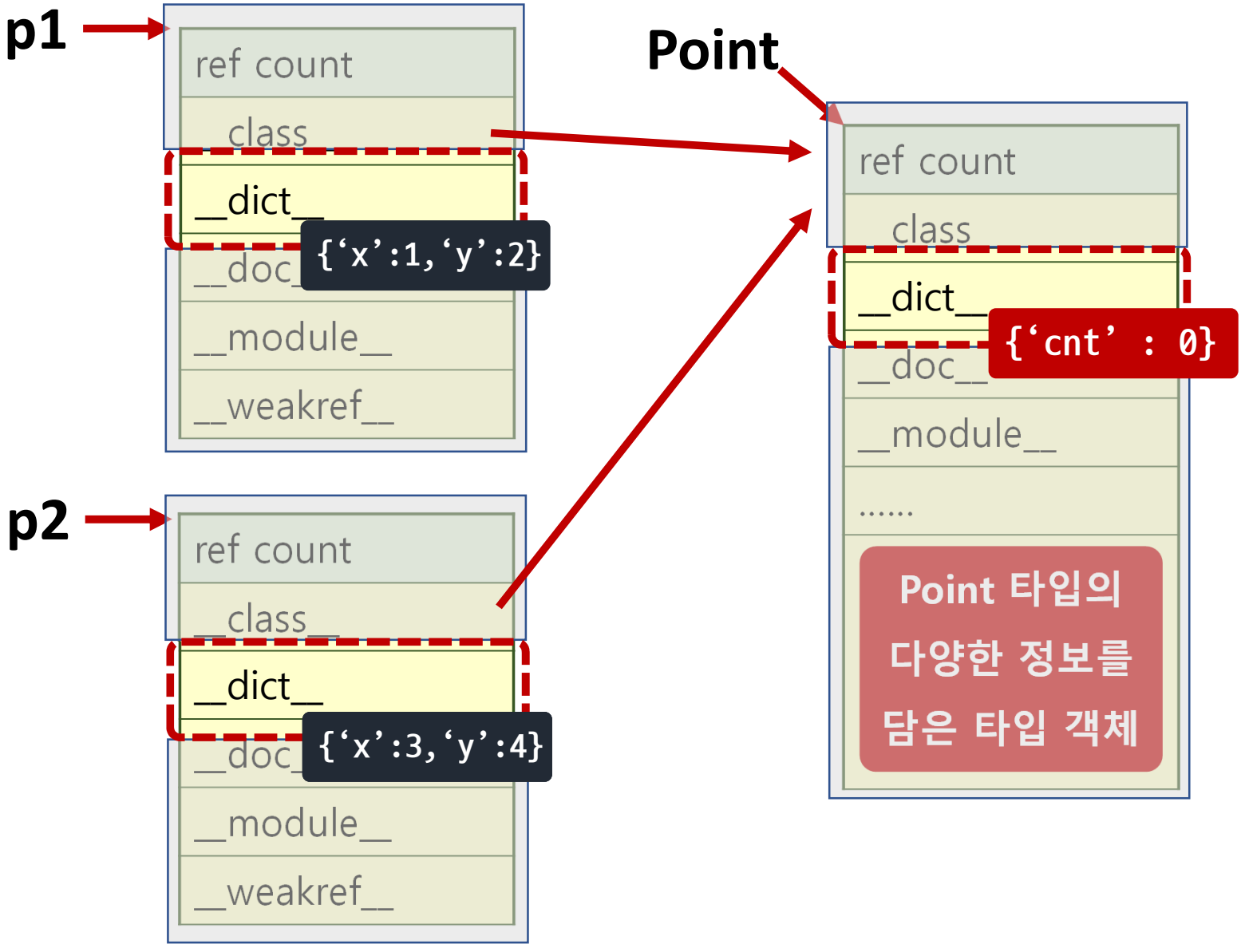
- **private 멤버**
  - ⇨ 접근 지정자를 위한 문법을 제공되지 않는다.
  - ⇨ 관례상 private 멤버는 "_" 또는 "__"를 붙여서 표현.
  - ⇨ "__" 가 붙은 멤버는 "**name mangling**" 발생

__이름 ➔ _클래스이름__이름

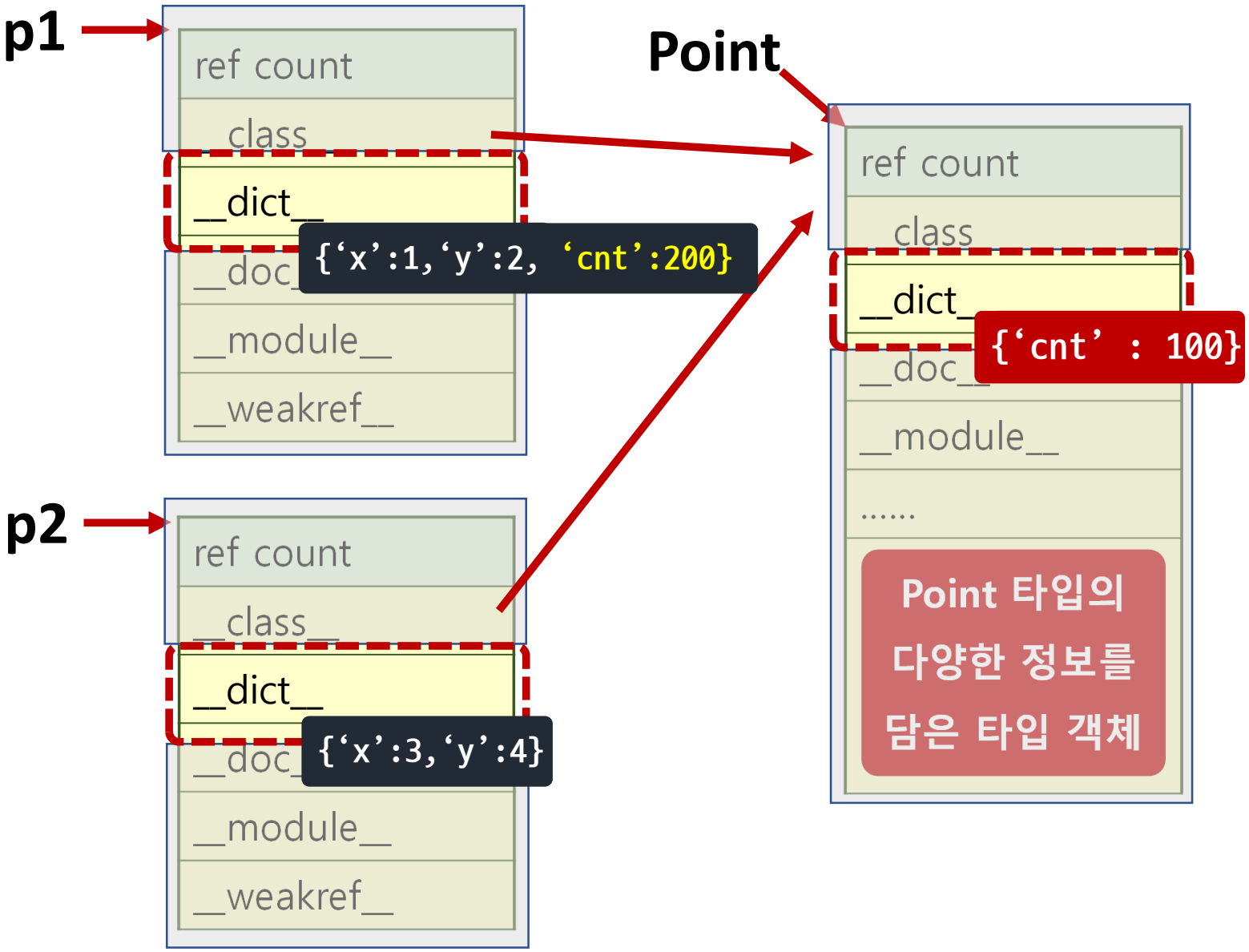
핵심 정리

instance field vs static field

instance field	객체별로 "**따로 보관**"되는 데이터
static field	모든 객체가 "**공유**" 하는 데이타



핵심 정리



객체 이름으로 static field 접근하는 방법

`n = p1.cnt`	p1.__dict__ 검색 후 없으면 Point.__dict__ 검색
`p1.cnt = 200`	p1.__dict__ 에 "**cnt**" 항목을 추가

핵심 정리

instance method vs static method

instance method	객체가 있어야만 호출 가능.
static method	객체가 없어도 호출 가능.

@staticmethod

`def method():` `  pass`	클래스 이름으로만 호출 가능

`@staticmethod`
`def method():`
`  pass`
	클래스 이름과 객체이름으로 모두 호출 가능

핵심 정리

- **메소드 안에서 자신의 타입을 알고 싶다.**
  - ⇨ instance method 안에서는 "**self.__class__**"
  - ⇨ static method 에서는 알 수 없다.

- **@staticmethod vs @classmethod**
  - ⇨ 메소드를 "**호출하는 방법은 동일**"
  - ⇨ "**메소드의 모양이 다르다**"

`@staticmethod`
`def method():`
`  pass`
	필요한 인자만 만들면 된다.

`@classmethod`
`def method(cls):`
`  pass`
	1번째 인자로 반드시 클래스 객체를 받아야 한다.

핵심 정리

- **사용자 정의 타입이 len() 을 지원하려면**

  ⇨ 약속된 메소드인 "**__len__()**" 을 제공해야 한다.

- **special method**

  ⇨ "**__xxx__()**" 형태의 메소드

  ⇨ 직접 호출할 수도 있지만, 특정 상황에서 자동으로 호출되는 메소드 (이름이 약속되어 메소드)

  ⇨ 100여개의 special method.

  ⇨ "**파이썬 표준 타입 과 유사하게 동작**"하는 클래스를 설계하기 위해서 반드시 필요.

핵심 정리

- 객체가 문자열로 변경될 수 있으려면

  ⇨ **__str__()**, 또는 **__repr__()** 메소드 제공

- __str__ vs __repr__

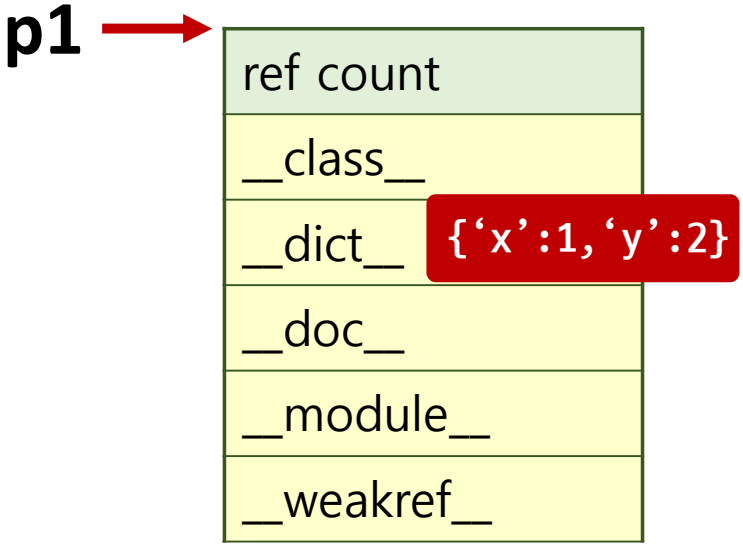
`__str__`	객체의 상태를 문자열로 반환 사용자에게 보여 주기 위한 용도
`__repr__`	디버깅 등 내부적인 용도

`s = repr(p1)`	p1.__repr__() 호출
`s = str(p1)`	① p1.__str__() 호출, 없으면 ② p1.__repr__() 호출
`print(p1)`	① p1.__str__() 호출, 없으면 ② p1.__repr__() 호출

핵심 정리

- instance field

  ⇨ 객체의 "__dict__" 에 보관 된다.

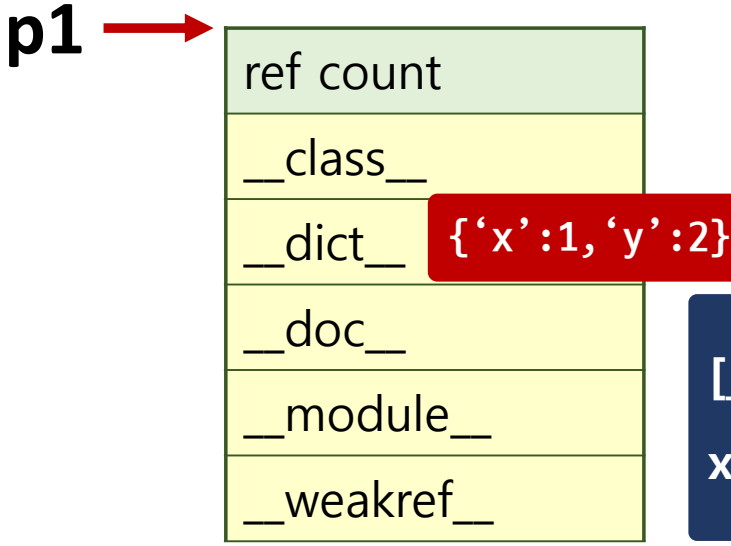


- __dict__ 사용시

  ⇨ 외부에서 "**새로운 멤버를 자유롭게 추가**"할 수 있다.

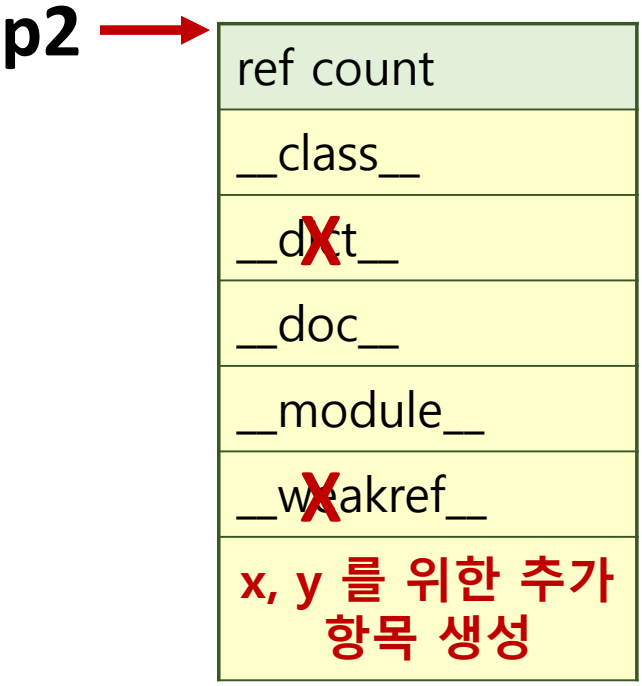
  ⇨ dictionary 는 "**메모리 사용량이 많다.**"

핵심 정리



[__slots__ 을 사용하지 않은 경우]

x, y 가 "__dict__" 보관



[__slots__ 을 사용하는 경우]

x, y 가 별도의 항목으로 보관

핵심 정리

- __slots__ 사용시
  - ⇨ 객체의 "__dict__", "__**weakref**__" 가 없다.
  - ⇨ slots 에 지정하지 않은 새로운 필드를 추가할 수 없다.
  - ⇨ 메모리 사용량이 약간 줄어든다.

핵심 정리

- field 를 외부에서 직접 접근하면
  - ⇨ "**객체가 잘못된 상태**"를 가질 수 있다.
  - ⇨ getter/setter 사용하는 것이 좋다.

- property
  - ⇨ getter/setter 를 필드처럼 접근하는 기술
  - ⇨ 원리 "**descriptor**" 강좌 참고

핵심 정리

- 연산자 재정의

  ⇨ 약속되어 있는 "**special method**" 를 제공하면 된다.

__add__(self, other)	**__radd__(self, other)**	**__iadd__(self, other)**
__sub__(self, other)	__rsub__(self, other)	__isub__(self, other)
__mul__(self, other)	__rmul__(self, other)	__imul__(self, other)¶
__matmul__(self, other)	__rmatmul__(self, other)	__imatmul__(self, other)
__truediv__(self, other)	__rtruediv__(self, other)	__itruediv__(self, other)
__floordiv__(self, other)¶	__rfloordiv__(self, other)	__ifloordiv__(self, other)
__mod__(self, other)	__rmod__(self, other)	__imod__(self, other)
__divmod__(self, other)	__rdivmod__(self, other)	
__pow__(self, other[, modulo])	__rpow__(self, other[, modulo])	__ipow__(self, other[, modulo])
__lshift__(self, other)	__rlshift__(self, other)	__ilshift__(self, other)
__rshift__(self, other)	__rrshift__(self, other)	__irshift__(self, other)
__and__(self, other)	__rand__(self, other)	__iand__(self, other)
__xor__(self, other)	__rxor__(self, other)	__ixor__(self, other)
__or__(self, other)	__ror__(self, other)	__ior__(self, other)
__neg__(self)	__pos__(self)	__abs__(self)

핵심 정리

```
'  '.join(['To', 'Be', 'Or', 'Not', 'To', 'Be'])
```

'To Be Or Not To Be'

핵심 정리

- "__call__" special method
  - ⇨ 객체를 함수 처럼 ( ) 연산자를 사용해서 호출 가능하게 한다
  - ⇨ 상태를 가지는 함수를 만들 수 있다.

핵심 정리

- decorator

  ⇨ 함수로 만들 수도 있고

  ⇨ "**클래스로 만들 수도 있다.**"

```
@add_emoticon
def say_hello(name):
    print(f'hello, {name} !')
```

```
say_hello = add_emoticon(say_hello)
```

객체의 생성
__init__(say_hello)

```
say_hello('kim')
```

```
say_hello.__call__('kim')
```

핵심 정리

- callable object
  - ⇨ ( ) 를 사용해서 호출 가능한 객체
  - ⇨ 내장 함수 "**callable**" 을 사용해서 확인 가능

내장함수	len(), print() 등
사용자 정의 함수	def 또는 lambda 로 만드는 함수
제너레이터 함수	yield 를 사용하는 함수 반환값은 "generator" 이다.
내장 메소드	list, dict 등의 타입이 가진 메소드 s.append(10)
사용자 정의 메소드	사용자 정의 클래스의 메소드
타입, 클래스	int(10), Point(1,2)
사용자 정의 타입의 객체	"**__call__**" 메소드를 재정의한 타입의 객체

핵심 정리

- 상속 표기법

```
class Derived(Base):
    ......
```

- 기반 클래스의 메소드를 호출하는 방법

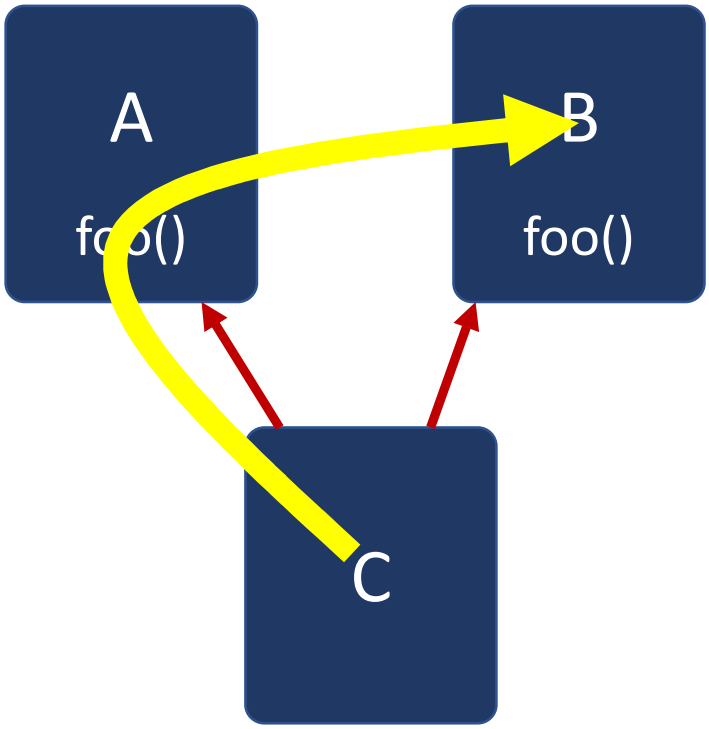
기반 클래스 이름 사용	**Base.goo(self)** 해당 클래스 의 메소드 호출
super() 사용	**super().goo()** **MRO(Method Resolution Order)** 의 순서에 따라 클래스 선택

핵심 정리

- 상속과 __init__ 메소드
  - ⇨ 파생 클래스의 __init__ 메소드 에서 "**기반 클래스의 __init__ 메소드를 명시적으로 호출**" 해야 한다.
  - ⇨ "**super().__init__()**" 을 권장

핵심 정리

- 파이썬은 "**다중 상속을 지원**"



- **MRO (Method Resolution Order)**

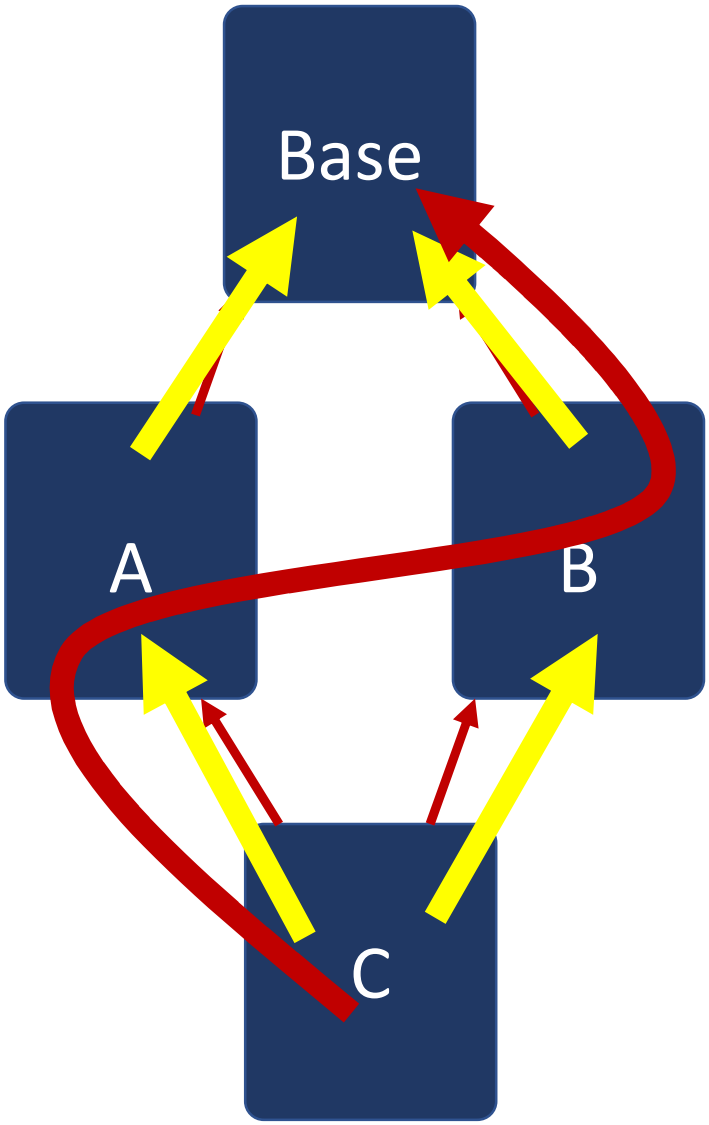
  ⇨ 메소드를 찾는 순서

  ⇨ "**클래스이름.mro()**" 로 확인 가능

- **super(타입, 객체).메소드()**

  ⇨ MRO 순서에서 "타입" 다음 클래스의 메소드 검색

핵심 정리

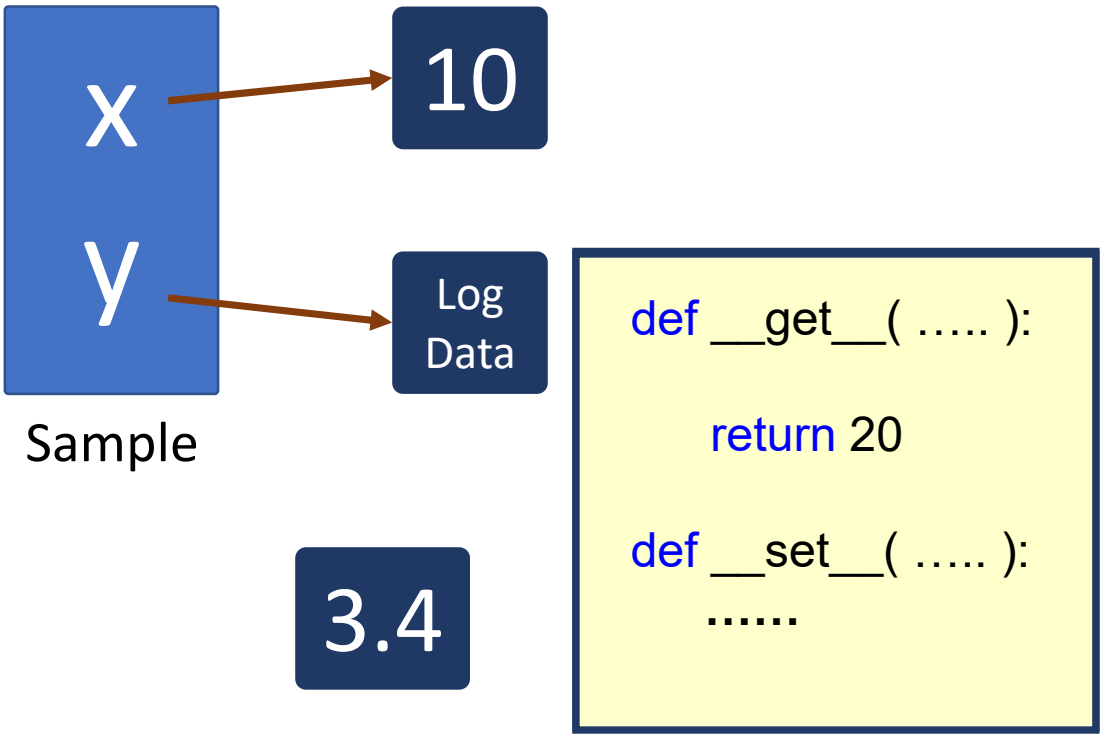


- **기반 클래스의 __init__ 메소드 호출시**

  ⇨ 클래스 이름을 직접 사용하면 최상위 기반 클래스의 "**__init__ 메소드는 2번 호출**"된다.

  ⇨ "**super() 를 사용해서 해결**"

핵심 정리



- **Descriptor 란 ?**

  ⇨ "__get__, __set__, __delete__" 중 한 개 이상의 메소드를 제공하는 클래스

- **Descriptor 특징**

  ⇨ Descriptor 타입의 static field 접근시 __get__, __set__ 메소드가 자동으로 호출

핵심 정리

- **Descriptor 란 ?**

  ⇨ "**__get__, __set__, __delete__**" 중 한 개 이상의 메소드를 제공하는 클래스

- **Descriptor 특징**

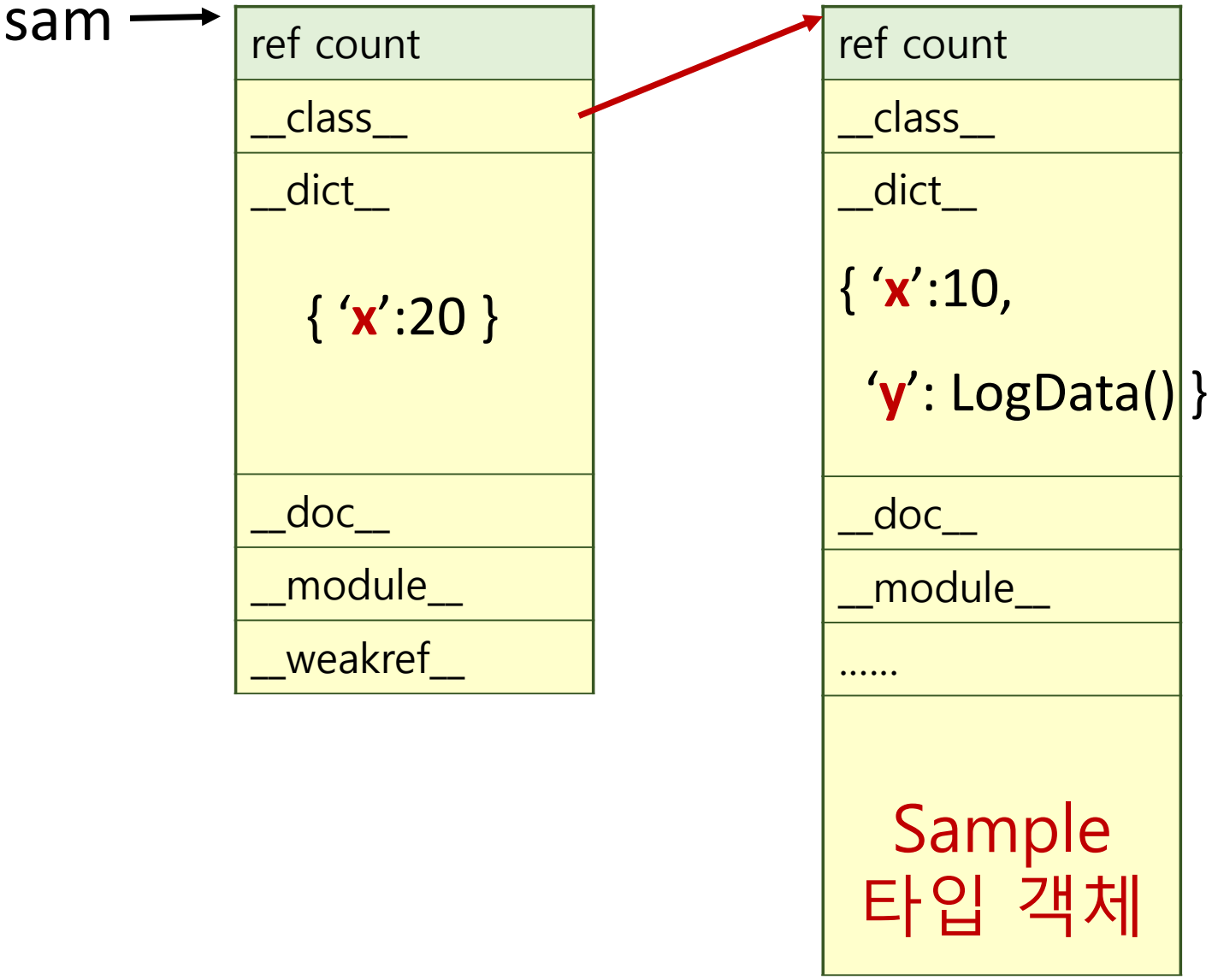
  ⇨ Descriptor 타입의 static field 접근시 __get__, __set__ 메소드가 자동으로 호출

- **클래스이름, 객체 이름으로 모두 사용가능**

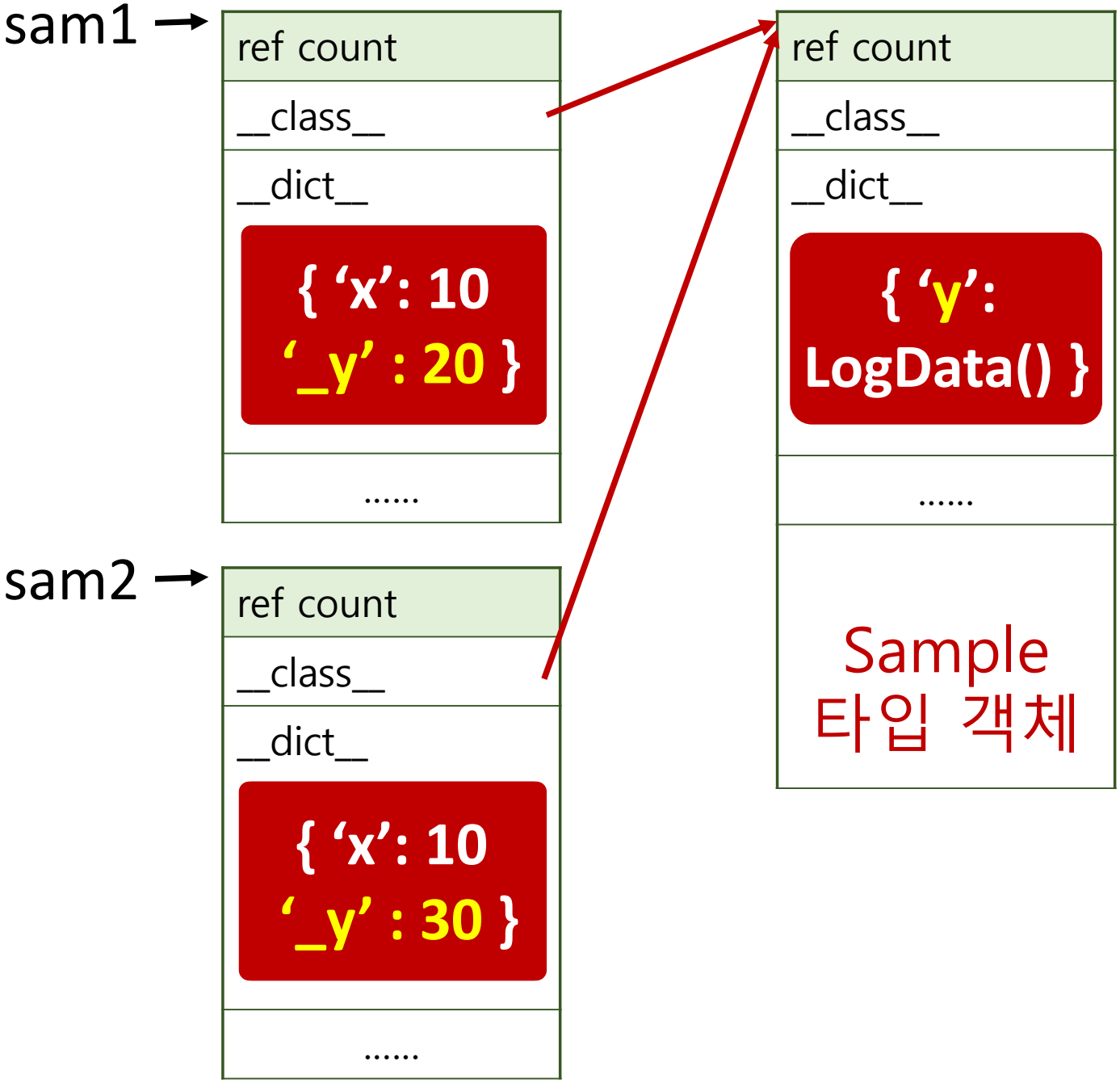
- **주의 사항**

  ⇨ "**static field 로 만든 경우만 동작**"한다.

핵심 정리



핵심 정리



- **핵심**

  ⇨ "**Descriptor 자체는 static field**" 이지만 __set__을 통해 저장되는 "**데이터는 각 객체에 저장(instance field)**"

핵심 정리

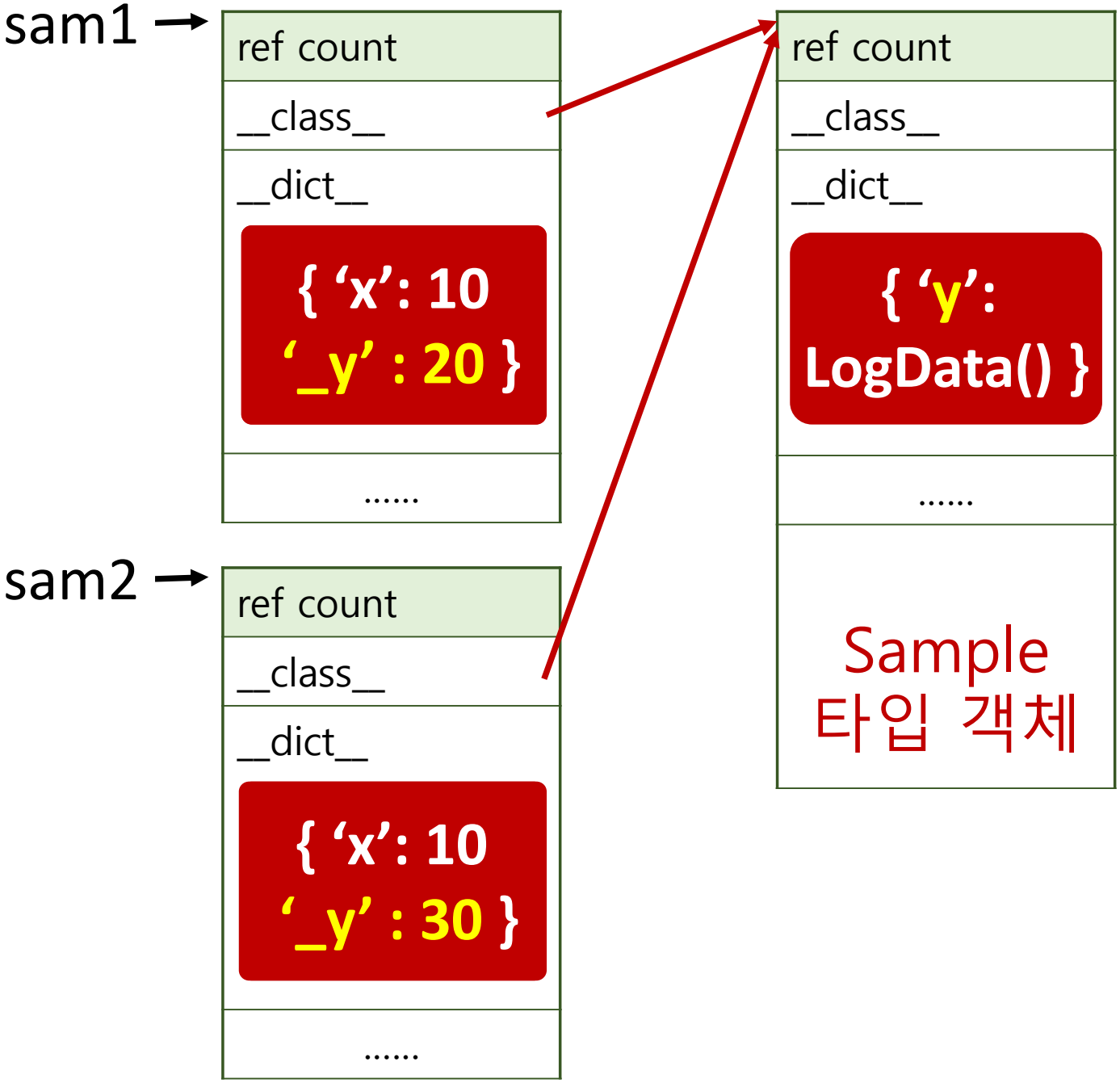
Descriptor 의 종류

Data Descriptor	__set__ 또는 __delete__ 메소드를 제공하는 Descriptor.
Non Data Descriptor	__get__ 메소드만 제공하는 Descriptor.

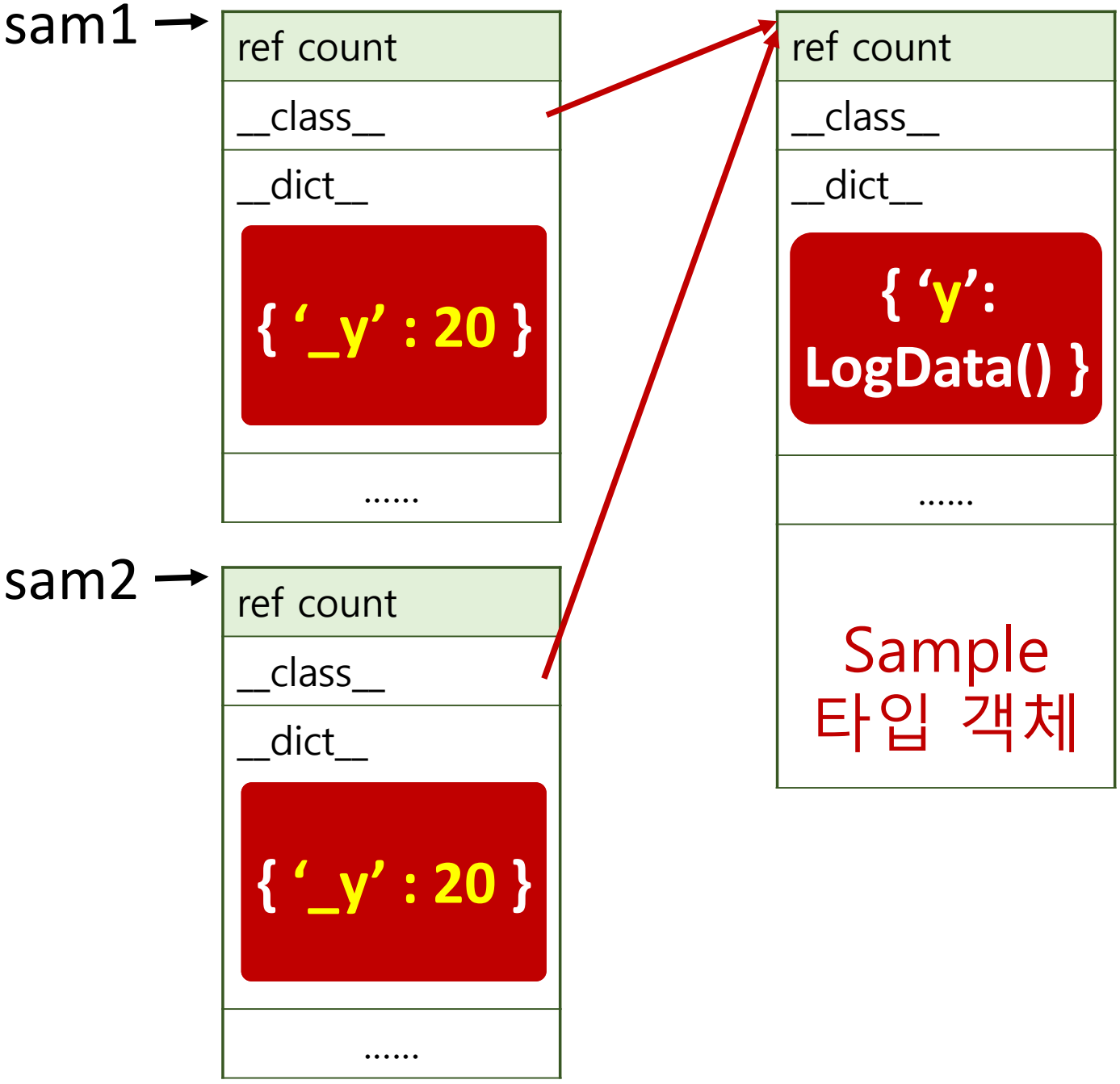
Sample 클래스 에서

sam1.x	x 필드(인스턴스 필드)에 직접 접근하는 것
sam1.y	"**getter/setter 를 통해서 _y 인스턴스 필드 에 접근**"하는 것

핵심 정리

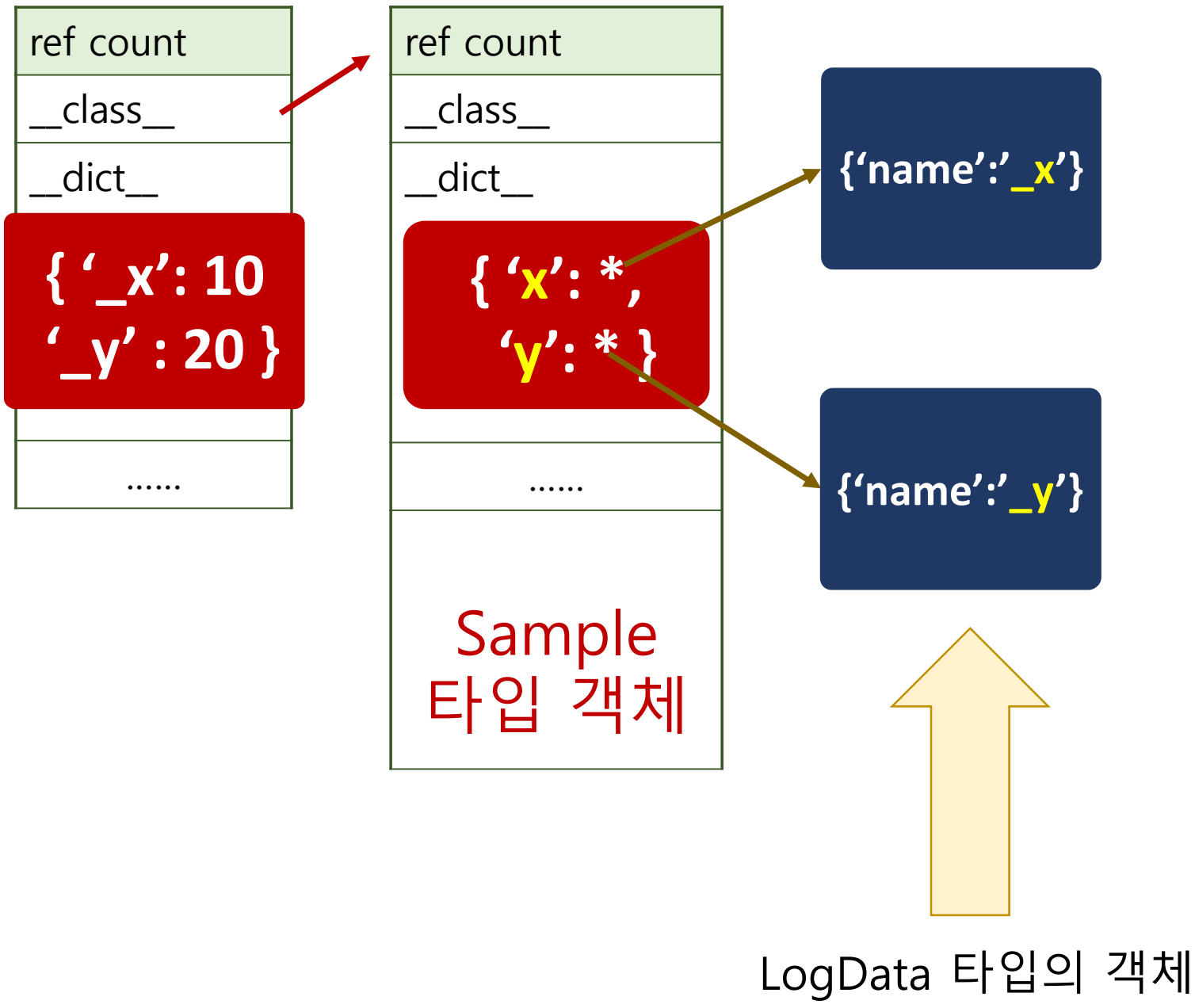

sam1
ref count
__class__
__dict__
{ 'x': 10
'_y' : 20 }
.....
sam2
ref count
__class__
__dict__
{ 'x': 10
'_y' : 30 }
.....
ref count
__class__
__dict__
{ 'y':
LogData() }
.....
Sample
타입 객체

핵심 정리



- 동일한 Descriptor 를 가지고 2개이상의 static field 를 만들 경우, "**저장하는 데이터의 이름을 다르게**" 할 수 있어야 한다.

핵심 정리



핵심 정리

- **방법 1. 모듈의 경로를 사용 환경에서 임시로 추가 하는 방법**

```
import sys

sys.path.append("G:\\MODULE_LIB")
```

- **방법 2. PYTHONPATH 환경변수에 등록**

linux .bash_profile	**PYTHONPATH**=**$PYTHONPATH**:/usr/lib/MODULE_LIB
windows	**PYTHONPATH**=**$PYTHONPATH**;G:₩₩MODULE_LIB

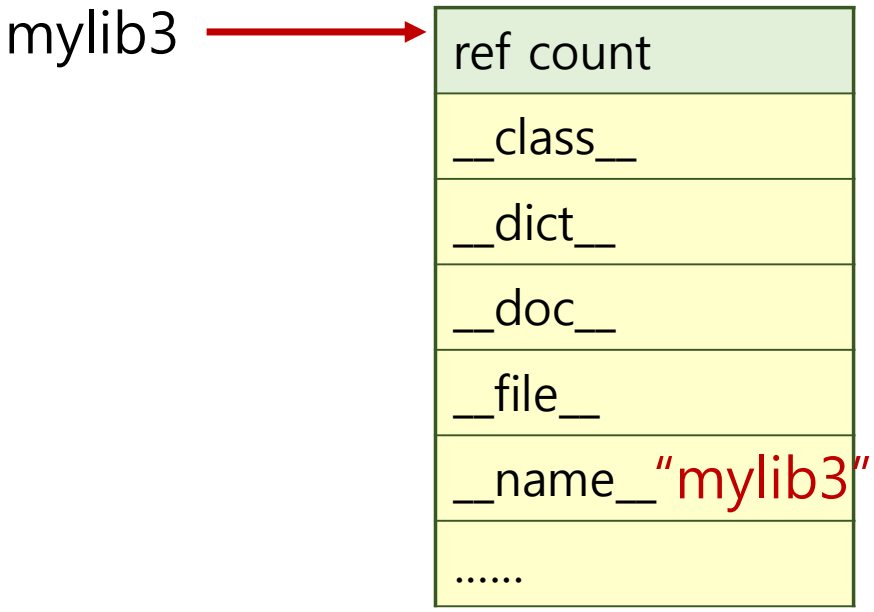
핵심 정리

Interactive Mode 에서 모듈 다시 로드

```
import importlib

importlib.reload( module name )
```

핵심 정리



핵심 정리

- **문자열로 모듈의 다양한 속성(타입, 함수등) 얻기**

```
f = getattr(mylib3, '함수 또는 클래스이름')
```

핵심 정리

- **builtins 모듈**
  - ⇨ 표준 타입과 표준 함수를 가지고 있는 모듈
  - ⇨ import 하지 않아도 사용가능
  - ⇨ "**__builtins__**" 이름으로 접근할 수 있다.

- **모든 모듈의 정보를 얻으려면**
  - ⇨ sys 모듈의 "**sys.modules**" 사용 - dictionary

- **현재 모듈의 참조를 얻으려면**
  - ⇨ **sys.modules[__name__]**

핵심 정리

함수가 실패 했음을 알리는 방법

None 반환	호출자가 반드시 실패를 처리할 필요는 없다.
NotImplemented 반환	연산자 재정의 함수에서 널리 사용.
예외 발생	호출자가 발생된 예외를 처리 하지 않으면 프로그램 종료.

핵심 정리

```
async def getpage(url):
        await asyncio.sleep(1);
        print(f'complete {url}')
```

await 를 사용하는 함수 앞에는 반드시 async 를 붙여야 한다.

```
main()
getpage(1)
getpage(2)
getpage(3)
getpage(4)
getpage(5)
```

event loop

핵심 정리

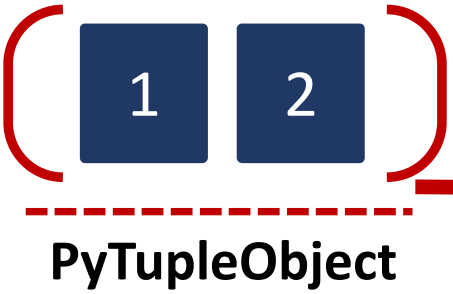
sample.c

`int add(int a, int b)`	정수 반환
`void sub(double a, double b, double* ret)`	포인터를 인자로 사용
`int sum(int* arr, int sz)`	배열을 인자로 사용
`int getarea(Rect* r)`	구조체를 인자로 사용

빌드 하는 방법

.dll	`cl sample.c /LD`
.so	`gcc -shared -fPIC -o libsample.so sample.c`

star.add(1, 2)



PyTupleObject

```c
int add(int a, int b)
{
    return a + b;
}
```

```c
PyObject* py_add(PyObject* self,
                 PyObject* args)
{
}
```

```c
PyMethodDef Methods[] = {

    // 모듈에 있는 메소드의 정보
};
```

```c
struct PyModuleDef star_module = {

    // 모듈 자체의 정보
};
```

```c
PyMODINIT_FUNC PyInit_star(void)
{
    // 모듈 초기화 함수
}
```

star.c

핵심 정리

빌드를 위한 준비물

star.c	add 함수가 있는 소스
setup.py	빌드 정보를 담은 파일
C 컴파일러	windows : cl 컴파일러(visual studio) linux : gcc 컴파일러

빌드 명령

빌드 + 설치	**python setup.py install**
빌드 설치 분리	python setup.py build python setup.py install
컴파일러 변경	python setup.py build **--compiler=mingw32** python setup.py install

핵심 정리