



Round 3

**PRESS
START**



《 Round 3 》

- 파이썬 자료구조
- 팩킹
- 람다 함수
- 데코레이터



New
Assignment



《 Round 3 》

- 파이썬 자료구조 《
- 팩킹
- 람다 함수
- 데코레이터



Let's
Go



```
# 리스트 자료형
ex_list = [1, "이", , [3]]

# 튜플 자료형
ex_tuple = (1, "이", [3])

# 딕셔너리 자료형
ex_dictionary = {1 : 1, 2 : "이", 3 : [3]}

# 집합 자료형
ex_set = set([1, "이", [3]])
```

List

```
# 리스트 자료형  
ex_list = [1, "0", , [3]]
```

요소 수정	요소 형태	특징
0	Anything	대괄호

- array(배열)과 list(리스트)의 차이

배열은 크기가 정해져 있으며, 기능이 존재하지 않는다.
리스트는 크기가 가변적이며 여러 기능이 존재한다.

배열 인덱스는 값에 대한 유일무이한 식별자.
리스트 인덱스는 몇 번째 데이터인가 정도의 의미.

배열은 요소를 삭제해도 순서가 그대로이다.
리스트는 중간 요소를 삭제하면 순서가 바뀐다.

그렇다고 한다...

- ```
딕셔너리 자료형
ex_dictionary = {1 : 1, 2 : "0", 3 : [3]}
```

# Set

```
집합 자료형
ex_set = set([1, "이", [3]])
```

| 요소 수정 | 요소 형태            | 특징                               |
|-------|------------------|----------------------------------|
| 0     | anything but set | set()을 통해 생성<br>중복 비허용<br>순서가 없음 |

- Set은 어떤 경우에 쓴다?

자료의 중복을 제거할 때(많이 쓰지는 않음)

자료의 집합 연산을 할 때(합집합, 차집합, 교집합)  
->> 집합연산 필요 시 가장 유용하게 쓸 수 있는 자료구조



## 《 Round 3 》

- 파이썬 자료구조
- 팩킹 《
- 람다 함수
- 데코레이터



Let's  
Go



# Packing?

```
print("가나다 abc 123")
print("가나다", "abc 123")
print("가나다", "abc", "123")
```

## 출력결과

```
가나다 abc 123
가나다 abc 123
가나다 abc 123
```

- print() 함수는 몇 개의 인자를 받든지 상관하지 않고 출력해줌
- 이처럼 함수가 받을 인자의 개수를 유연하게 지정할 수 있다면 함수 작성이 유연해짐
  - >> Packing을 쓴다면 가능!!

# Position Packing

```
def func(*args):
 print(args)
 print(type(args))
```

- 매개변수 앞에 \*을 붙여준다면, **위치인자**로 보낸 모든 객체들을 하나의 Tuple로 관리

```
func(1, 2, 3, 4, 5, 6, 'a', 'b')
```

결과

```
(1, 2, 3, 4, 5, 6, 'a', 'b')
<class 'tuple'>
```

# Position Packing

- 매개변수 앞에 \*\*을 붙여준다면, **키워드 인자**로 보낸 모든 객체들을 하나의 Dictionary로 관리





# Keyword Packing + Position Packing

```
def print_family_name(*parents, **sibling):
 print("아버지 :", parents[0])
 print("어머니 :", parents[1])
 if sibling:
 print("호적 메이트..")
 for title, name in sibling.items():
 print('{} : {}'.format(title, name))

print_family_name("홍길동", '심사임당', 누나='김태희', 여동생='윤아')
```

- 위치 팩킹과 키워드 팩킹을 같이 사용해서 함수 인자를 구성할 수 있음!

## 《 Round 3 》

- 파이썬 자료구조
- 팩킹
- 람다 함수 《
- 데코레이터



Let's  
Go





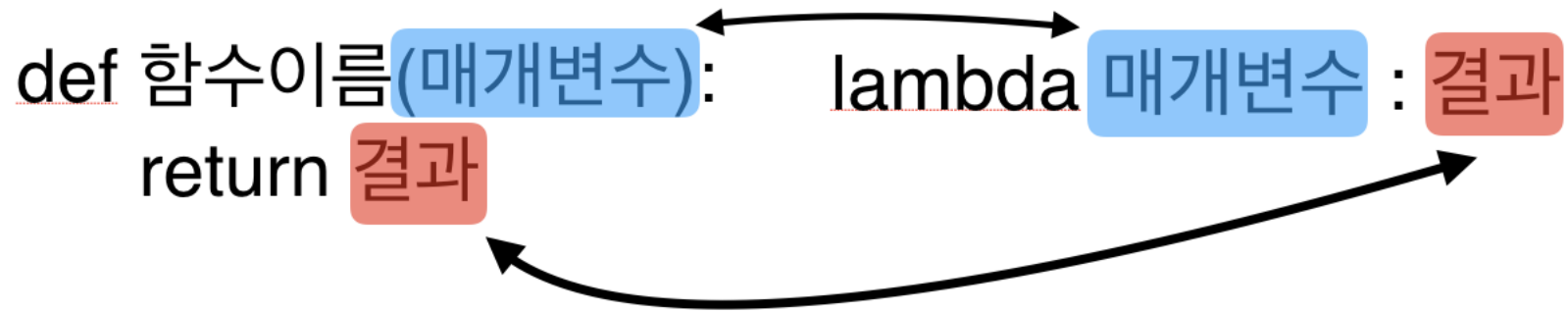
# Lambda function?



- 북적북적한 식당에서 8인용 테이블을 혼자 차지하고 1인분만 시킨다면...?
- 마찬가지로 단 한번만 사용할 함수를 인스턴스화 한다면 메모리를 비효율적으로 이용하게 된다.  
(파이썬에서는 모든 것이 객체이기 때문에 함수 또한 클래스를 통해 생성된 객체 인스턴스임)

# Lambda function

`def 함수이름(매개변수):`      `lambda 매개변수 :` **결과**  
`return` **결과**



- 람다 함수(aka 익명함수)는 사용 이후 힙(heap) 영역에서 바로 제거되는 함수. 메모리 효율성이 높아짐!
  - 간결하게 함수를 표현할 수 있음
- 경우에 따라서 개발자가 간단한 함수를 짜는 것에 있어서 시간낭비를 줄여줄 수 있음

# Lambda function

```
target = [' rabbit', 'tiger', 'dog ', 'elephant ']
```

- 위 list를 앞뒤 불필요한 공백을 제외한 문자의 길이로 정렬한다면...

일반적인 방법

```
def my_key(string):
 return len(string.strip())
```

```
target = [' cat ', ' tiger ', ' dog', 'snake ']
print(sorted(target, key=my_key))
```

람다함수 사용

```
target = ['cat', 'tiger', 'dog', 'snake']
print(sorted(target, key=lambda x : len(x.strip())))
```

메모리 비효율성 발생

## 《 Round 3 》

- 파이썬 자료구조
- 팩킹
- 람다 함수
- 데코레이터 《



Let's  
Go



# Decorator?



- 기존 함수를 그대로 유지하면서 간단한 기능들을 추가하고 싶다면...?  
ex) 실행시간 측정, 기록 기능 추가...
- 데코레이터는 간단한 선언으로 기존 코드의 기능을 간편하게 개선시킬 수 있음!

# Decorator

- 만약 기존 방법으로 big\_number라는 함수에 시간 측정 기능을 도입한다면...

일반적인 방법

데코레이터

```
import time

def big_number(n):
 return n ** n ** n

def make_time_checker(func):
 def new_func(*args, **kwargs):
 start_time = time.perf_counter()
 result = func(*args, **kwargs)
 end_time = time.perf_counter()
 print('실행시간:', end_time - start_time)
 return result
 return new_func

new_func = make_time_checker(big_number)
new_func(7)
```

```
def make_time_checker(func):
 def new_func(*args, **kwargs):
 start_time = time.perf_counter()
 result = func(*args, **kwargs)
 end_time = time.perf_counter()
 print('실행시간:', end_time - start_time)
 return result
 return new_func

@make_time_checker
def big_number(n):
 return n ** n ** n

@make_time_checker
def big_number2(n):
 return (n+1) ** (n+1) ** (n+1)
```

# NEXT STAGE

