# Fastcampus Data Science Extension SCHOOL

## Python

# Index

- Review
- Programming Paradigms
  - Sequential Programming
  - Procedural Programming
  - Object Oriented Programming
- Lambda
- map
- filter
- reduce

# Decorator

# Back to the Fibonacci..

```
start_time = time.time()
fib_rec(10)
end_time = time.time()
print(end_time-start_time)
```

```
start_time = time.time()
fib_binet(10)
end_time = time.time()
print(end_time-start_time)
```

# Let's wrap with decorator

```
import time


def time_checker(function):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = function(*args, **kwargs)
        end_time = time.time()
        print("execution time: {time} sec".format(
                time=end_time-start_time))
        return result
    return wrapper
```

## and just add `@`

```
@time_checker
def fib_rec(num):
        ...
```

## Decorator는

- function의 앞, 뒤로 해야 할 일이나 로깅, 벤치마킹 등 다양한 용도로 쓰일 수 있습니다.
- 데이터 전처리 과정 또한 미리 정의해둔 뒤, 붙여 사용할 수 있습니다.

## Do it yourself!

"Hi, {name}. You might be loved with {lang}" 이라는 문자열이 존재할 때, 이 문자열의 앞 뒤로 `<h1>, <em>` 태그가 붙도록 하는 데코레이터를 생성하세요

ex output)

```
<h1><em> {{text}} </em></h1>
```

`Advanced problem: Decorator 하나로 html 태그 이름을 지정할 수 있도록 수정`

# Programming Paradigms

# Sequencial Programming

```
print("wake up")
print("go to work")
print("have lunch")
print("hard work")
print("back to home")
print("have dinner")
print("get some sleep")
if tomorrow == "weekend":
        goto 12
else :
        goto 1
print("zzz")
print("have dinner")
print("get some sleep")
if tomorrow == "weekend":
        goto 12
else :
        goto 1
```

# Procedural Programming

```
def wake():
    return "wake up"
def eat():
    return "eat something"
def work():
    return "work"
def sleep():
    return "sleep"

while True:
    if today=="weekday":
        wake()
        work()
        eat()
        work()
        sleep()
    else:
        sleep()
        wake()
        eat()
        sleep()
```

# Object-Oriented Programming

```
Class Person:
    def __init__(self):
        self.health=100
        self.hunger=100
        self.damage=1

Class Hero:
    def __init__(self, a, b):
        self.health=a
        self.hunger=100
        self.damage=b

me = Person()
iron_man = Hero(1000,1000)
hulk = Hero(10000,800)
hawk_eye = Hero(100,300)
```

# lambda

## lambda

- 익명함수(이름이 없는 함수)
- 간단한 수식을 함수로 지정해 한 두번 쓸 용도로 사용할 때
- 두 줄 이상 실행될 함수는 그냥 함수로 정의하는게 나음!

## lambda

- python은 모든 것이 객체로 존재
- 간단한 연산 함수 조차 객체로 존재하여 리소스를 점유

## 그러나 남발하면..

- Code 자체의 Readibility를 헤칠 뿐 아니라, 재사용에 대한 고민없이 사용하다 lambda를 반복사용하여 Heap을 괴롭힐 수 있습니다.
-

## lambda - traditional function

```
def get_next_integer(a):
        return a + 1
```

## lambda - lambda function

```
lambda a: a+1
```

## lambda - usual example

```
>>> (lambda a: a+1)(12)
13
```

## lambda

- only expression, not statement

```
lamb =
```

# map, filter, reduce

## map

- map(function, iter)
- list의 각 element에 대해 특정한 함수를 적용

# map - example

```
def get_squared(num_list):
    squared = []
    for num in num_list:
        squared.append(num**2)
    return squared
```

# map - example

```
def squared_lambda(x):
    return x ** 2

list(map(squared_lambda, [1,2,3,4]))
```

# map with lambda - example

```
list(map(lambda x: x**2, [1,2,3,4]))
```

# map is rather than for

```
def print_with_sleep(x):
    time.sleep(1)
    return x ** 2
```

```
m = map(print_with_sleep, [1,2,3])
next(m)
next(m)
next(m)
..
```

```
for i in range(1,3+1):
    print_with_sleep(i)
```

# timing is perfect!

```
m = map(print_with_sleep, [1,2,3])
for i in m:
    print(i)
```

```
m = map(print_with_sleep, [1,2,3])
list(m)
```

## map is rather than for

- map은 제너레이터를 생성해 함수와 인자를 바인딩만 하고, 필요할 때 순회하며 값을 처리
- for는 한번에 처리하므로 for 수행 중 다른 일을 할 수 없음

## filter

- filter(function, iter)
- 특정함수를 만족하는 요소만 남기는 필터

# filter - example

```
def even_selector(x):
    if x % 2 == 0:
        return True
    else:
        return False

filter(even_selector, range(1,10+1))
```

# filter with lambda - example

```
filter(lambda x: x%2==0, range(1,10+1))
```

# reduce

- reduce(function, iter[, initializer])
- iterable object의 모든 element에 대하여 연산결과를 출력
- python3 기본 내장함수에서 제외되어 functools에서 import함
    - why? Rossum은 map, filter, reduce에 대해 readibility가 떨어진다는 이유로 제외하고자 하였음

```
from functools import reduce
```

## before reduce

3min
1부터 100까지 모든 숫자의 합을 반복문을 활용하여 연산하시오.

# reduce example

```
result = 0
for i in range(1,100+1):
    result += element
```

# reduce example

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

reduce(adder, range(1,100+1))
```

## reduce with lambda

```
reduce((lambda x,y:x+y), range(1,100+1)
```

# reduce with initializer

```
default = 10
for i in range(1,10+1):
    dafault += i
```

```
reduce(lambda x,y:x+y, range(1,10+1), 10)
```

# Deep dive into reduce

```
reduce(lambda x,y:y+x, range(1,10+1)
```

```
reduce(lambda x,y:y+x, 'fastcampus')
```

# Do it yourself

```
recycle_bin = [
1, 2, "Fastcampus", ['dog', 'cat', 'pig'], 5, 4, 5.6, False
"패스트캠퍼스", 100, 3.14, 2.71828, {'name':'Kim'}, True,
]
```

1. 위 리스트의 요소 중 정수와 실수인 요소만 리스트로 구성하기
2. 위 리스트의 요소 중 정수만 각각 제곱하여 리스트로 구성하기
3. 위 리스트의 요소 중 정수만 각각 제곱한 수들의 합계 출력하기

> Hint: isinstance(1, int)

# Do it yourself

| Order ID | Quantity | Unit Price |
|---|---|---|
| 181121001 | 2 | 2400 |
| 181121002 | 12 | 9800 |
| 181121003 | 5 | 124800 |
| 181121004 | 10 | 76000 |
| 181121005 | 24 | 2810 |

- 앞서 배운 개념들을 활용해 아래 문제를 해결하세요.

1. 튜플과 리스트를 활용해 위 테이블을 변환하세요
2. 각 주문 별 총 주문가격을 산출하세요
3. 11월 21일에 발생한 총 매출, 평균 구매금액을 산출하세요