

Day-7 특이한 함수들 (Closure / Decorator / Lambda)

7.1. 클로저 (Closure)

7.1.1. 클로저란?

클로저란 무엇일까?

- 함수 내부에서 정의된 함수
- 해당 스코프 외부에서 호출되는 경우에도 **자유 변수**에 액세스 할 수 있는 함수 객체

아래의 조건을 충족해야 한다

- 어떤 함수의 내부 함수일 것
- 그 내부 함수가 외부 함수의 변수를 참조할 것
- 외부 함수가 내부 함수를 리턴할 것

예시를 확인해보자. `inner_func()` 은 클로저라고 이야기 할 수 있다.

```
def outer_func():  
    message = 'Hi' # inner_func 외부에 선언되어있다  
  
    def inner_func():  
        print(message) # message 호출  
  
    return inner_func # inner_func을 반환한다  
  
my_func = outer_func() # outer_func은 inner_func을 반환하고 종료된다  
my_func() # Hi
```

위 동작을 좀 더 이해하기위해, 인터프리터 동작 방식을 알아보자.

7.1.1.1. 파이썬 인터프리터가 변수를 검색하는 순서

파이썬은 다음 세 범위(Scope)를 우선 순위로 해 변수를 검색한다.

1. **Local Scope** : 먼저 함수 내 해당 변수가 선언됐는지 확인한다. 이 변수를 Local variable이라고 한다
2. **Enclosing Scope (Free Scope)**: Local에서 찾지 못한 경우, 해당 함수가 선언되는 외부 함수에서 변수가 선언됐는지를 확인한다. 이렇게 선언된 변수를 **자유 변수(free variable)**라고 한다.
3. **Global Scope**: 위 두 단계에서 찾지 못한 경우, 함수의 외부에서 변수를 찾는다. 이를 **전역 변수(Global Variable)**이라고 한다

7.1.2. 클로저는 왜 쓸까?

여러가지 이유로 사용된다. 클로저는 유연한 메커니즘을 제공한다.

- 관리와 책임을 명확히 할 수 있다
- 사용 환경에 맞게 임의대로 내부 구조를 조정할 수 있다

다만 다음을 주의해야 한다

- 내부 함수가 외부 함수의 변수를 참조하기 때문에, 메모리 누수의 위험이 있다

7.1.2.1. 상태 저장 기능

```
# 데이터 캡슐화
def counter():
    count = 0 # int형, immutable object임에 주의
    def increment_counter():
        nonlocal count # 이런 경우 nonlocal을 붙여 스코프를 명시한다. 붙이지 않으면 0
        count += 1
        return count
    return increment_counter
```

```
my_counter = couneter()
print(my_counter()) # 1
print(my_counter()) # 2
print(my_counter()) # 3
```

7.1.2.1. 팩토리 함수

7.1.2.1.3. 콜백 함수

tkinter는 GUI 프로그래밍을 위한 라이브러리다. closure가 callback 함수로 사용되는 좋은 예시기도 하다.

```
import tkinter as tk

app = tk.Tk()
app.title("GUI App")
app.geometry("320×240")

label = tk.Label(
    app,
    font=("Helvetica", 16, "bold"),
)
label.pack()

def callback(text):
    def closure():
        label.config(text=text)

    return closure

button = tk.Button(
    app,
    text="Greet",
    command=callback("Hello, World!"),
```

```
)  
button.pack()  
  
app.mainloop()
```

위 코드에서 `command`는 인수를 취하지 않는 callable 객체를 받는다. `callback`을 구현함으로써 이를 충족시킬 수 있다.

7.2. 데코레이터 (Decorator)

클로저의 개념을 더 자주 사용하는 곳이 있는데, 바로 데코레이터다. 데코레이터는 함수를 수정하지 않고 기능을 추가할 수 있는 기능이다.

7.2.1. 기본 사용법

다른 함수를 인자로 받는 callable function이라고 할 수도 있다.

```
def decorator_function(original_function):  
    def wrapper_function():  
        print(f'wrapper executed this before {original_function.__name__}')  
        return original_function()  
  
    return wrapper_function  
  
def display():  
    print('display function ran')  
  
decorated_display = decorator_function(display)  
decorated_display()
```

위 코드는 `display` 함수를 실행하기 전에 `wrapper_function` 함수를 실행하는 데코레이터 함수이다.

데코레이터를 사용하면 아래와 같이 코드를 작성할 수 있다. `@`(어노테이션)을 이용해 사용할 수 있다.

```

def decorator_function(original_function):
    def wrapper_function():
        print(f'wrapper executed this before {original_function.__name__}')
        return original_function()

    return wrapper_function

@decorator_function
def display():
    print('display function ran')

display()

```

데코레이터는 함수를 수정하지 않고 기능을 추가할 수 있다.

데코레이터가 인수를 받게 하고 싶으면 어떻게 할까?

함수를 한번 더 감싸주면 된다.

```

def is_multiple(x):
    def my_decorator(func):
        def my_func(*args, **kwargs):
            result = func(*args, **kwargs)
            print(f'function name : {my_func.__name__}')
            if result % x == 0:
                print(f'{x}'s multiple number!")
            else:
                print(f'{x}'s non-multiple number!")
            return result
        return my_func
    return my_decorator

@is_multiple(2)
def add(a, b):
    sum = a+b
    print(f"sum : {sum}")
    return sum

```

```
print(add(2, 1))
print(add(10, 22))
```

```
sum : 3
function name : my_func
2's non-multiple number!
3
sum : 32
function name : my_func
2's multiple number!
32
```

아래와 같은 실행결과가 나온다.

여기서 my_func이 아니라, add를 출력하게 하고 싶으면 어떻게 할까?

`functools.wraps` 를 사용하면 된다. 인자로 받은 의 속성을 그대로 사용해, wrapper 함수를 wrapped 함수처럼 보이게 갱신한다.

```
import functools

def is_multiple(x):
    def my_decorator(func):
        @functools.wraps(func) # 추가 됐다
        def my_func(*args, **kwargs):
            result = func(*args, **kwargs)
            print(f'function name : {my_func.__name__}')
            if result % x == 0:
                print(f'{x}'s multiple number!")
            else:
                print(f'{x}'s non-multiple number!")
            return result
        return my_func
    return my_decorator

@is_multiple(2)
def add(a, b):
    sum = a+b
```

```

    print(f"sum : {sum}")
    return sum

print(add(2, 1))
print(add(10, 22))

```

다음과 같은 실행 결과가 나온다.

```

sum : 3
function name : add
2's non-multiple number!
3
sum : 32
function name : add
2's multiple number!
32

```

7.2.2. 파이썬에서의 데코레이터

파이썬 표준 라이브러리에서는 메소드를 데커레이트 하기 위한 `property()` , `classmethod()` , `staticmethod()` 를 제공하나 여기선 살펴보지 않는다.

데커레이터를 활용처를 알아보기 위해 몇 가지 예시를 살펴보자.

7.2.2.1 functools.lru_cache()를 이용한 메모이제이션

메모이제이션은 이전에 실행한 값비싼 함수의 결과를 저장해, 이전에 사용된 인수에 대해 다시 계산할 필요가 없게 해준다. 앞에 붙은 LRU는 사용한지 가장 오래된(Least Recently Used)의 약자로, 오래 사용되지 않은 값을 캐시에서 제거해 메모리 사이즈를 관리한다는 의미다.

다음 코드의 출력을 비교해보자.

```
import time
```

```
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__ == '__main__':
    # check elapsed time
    start = time.time()
    print(fibonacci(40))
    print(f'Elapsed time: {time.time()-start}')
```

실행에 15초가 걸린다

```
102334155
Elapsed time: 15.613154888153076
```

이제 캐시를 사용해보자

```
import functools
import time

@functools.lru_cache() # 추가되었다
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__ == '__main__':
    # check elapsed time
    start = time.time()
    print(fibonacci(40))
    print(f'Elapsed time: {time.time()-start}')
```

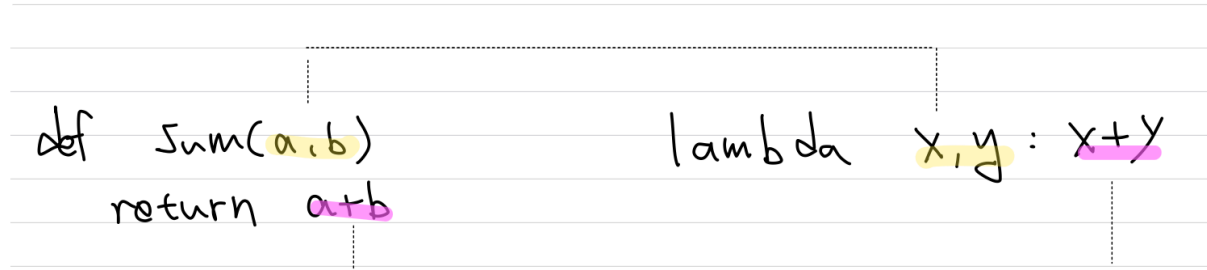
매우 적은 시간이 걸린다

```
102334155
```


Elapsed time: 4.172325134277344e-05

7.3. 람다(Lambda)

람다는 한 줄 코딩이 가능한, 이름 없는 함수다. `lambda` 키워드를 사용해 만들 수 있다.



The image shows a handwritten comparison between a regular function and a lambda function. On the left, a regular function is defined as `def sum(a, b):` followed by an indented `return a + b`. On the right, a lambda function is written as `lambda x, y: x + y`. Dotted lines connect the parameters `a, b` to `x, y` and the return value `a + b` to `x + y`, illustrating that a lambda function is essentially a single-line function definition.

예제 코드를 확인해보자.

```
add = lambda a, b: a + b
print(add(1, 2)) # 3
```

함수를 인자로 넘겨야 할 때 주로 사용하며, 이외의 부분에서 남발할 시 코드의 가독성을 해칠 수 있다.

아래와 같은 특징을 가진다

- 보통 `map`, `filter`, `sort` 등등에 사용
- 익명함수
- 힙영역에서 사용

아래 코드를 확인해보자

```
my_list = ['apple', 'bananaaaaaaaaaaaaaaaaaaaaa', 'cherry']
# 위 리스트를 길이 순으로 정렬하자
my_list2 = sorted(my_list, key=lambda x: len(x)) # 길이를 반환하는 함수를 인자로
print(my_list2)
```

