Chapter – 15 Decorators

Contents

Chapter – 15 Decorators

ပရိုဂရမ်တွေကို မိမိကိုယ်တိုင် ရေးကြသလို သူများရေးထားတဲ့ ပရိုဂရမ်တွေ၊ ကုဒ်တွေကိုလည်း ယူသုံးကြသည်။ သူများရေးထားသည့် ကုဒ်တွေ၊ function တွေက မိမိအလုပ်မှာ အံဝင်ခွင်ကျ အသုံးမတည့်သည့် အခါများတွင် အနည်းငယ် ပြုပြင်ခြင်းတွေ လုပ်ရန် လိုအပ်သည်။ ထိုအခြေအနေများတွင် decorator များသည် အလွန်အသုံးတည့်သည့် အရာများ ဖြစ်လာသည်။

Decorator ဆိုတာ function တစ်ခု သို့မဟုတ် method တစ်ခုရဲ့ လုပ်ဆောင်ချက်တွေကို ပိုကောင်းအောင်လုပ်ပေးတာပါ။ တစ်ခါတစ်ရံ ရှိပြီးသား function များသည် မိမိ အလိုရှိသည့် output ကို မထုတ်ပေးနိုင်သည့်အခါ ထို function များကို မိမိ စိတ်ကြိုက်ဖြစ်အောင် အနည်းငယ်ပြုပြင်ရန် လိုအပ်သည်။ တစ်ခါတစ်ရံ function များကို ခေါ်သုံးခွင့်သာ ရှိသည်။ ထို မူလ function သို့မဟုတ် မူလ method ထဲသို့ ဝင်ရောက်ပြင်ဆင် ရေးသားခွင့် မရှိပေ။ ထိုအခါမျိုးတွင် decorator များကို အသုံးပြုရန် လိုအပ်သည်။

Decorator ဆိုတာ function တစ်ခု ပိုကောင်းသွားအောင် တန်ဆာဆင်ပြီး ပိုအသုံးကျအောင် (decorate) လုပ်ပေးတာပါ။ ကွန်ပျူတာ စကားနဲ့ပြောရရင် decorator ဆိုတာ function တစ်ခုကို ယူပြီးတော့ နောက်ထပ် လုပ်ဆောင်မှုအသစ်(functionality)တွေ ထပ်ထည့်ပေး(return ပြန်လုပ်)သည့် callable object ဖြစ်သည်။

```
Standard Definition :     A decorator in Python is any callable Pyth
on object that is used to modify a function or a class.
```

ရှိပြီးသား function တစ်ခု သို့မဟုတ် class တစ်ခုကို ပိုကောင်းသွားအောင် ထပ်ပြီး ပြင်ဆင်လိုက်တဲ့ callable object သည် decorator ဖြစ်သည်။ Callable object ဆိုတာ ခေါ်ပြီးသုံးလို့ရသည့် object တွေပါ။ ဥပမာ- function တစ်ခုသည် callable object တစ်ခု ဖြစ်သည်။

Programmer များသည် function ၏ behavior များကို သော်လည်းကောင်း class ၏ behavior များကို သော်လည်းကောင်း ပြုပြင်ပြောင်းလဲ(modify) လိုသည့်အခါ decorator များကို အသုံးပြုကြသည်။

Function decorator နှင့် class decorator ဟူ၍ decorator (၂)မျိုးရှိသည်။ Decoration လုပ်သည် function ကို decorator function ဟုခေါ်သည်။ Decoration လုပ်သည့် class ကို class decorator ဖြစ်သည်။ Decorator ဟုရေးလျှင် class လည်း ဖြစ်နိုင်သည်။ Function လည်း ဖြစ်နိုင်သည်။ အခုဆက်လက်ရှင်းပြ မည့်အချက်များသည် function decorator ကို အခြေခံထားသည်။

Decorator သဘောတရားကို နားလည်ရန်အတွက်

(၁) Nested function သဘောတရား

(၂) Function တစ်ခုက တခြား function တစ်ခုကို return ပြန်ပေးတဲ့ သဘောတရား(one function can return another function)

(၃) Function names ကို reference အဖြစ် သုံးလို့ရတဲ့သဘောတရား နှင့်

(၄) Function တစ်ခုကို parameter အဖြစ်နဲ့ တခြား function ထဲကို ထည့်ပေးလို့ရတဲ့ သဘောတရား တို့ကို နားလည်သဘောပေါက်ထားရန် လိုအပ်သည်။

15.1 Nested Function သဘောတရား

Function တစ်ခုထဲမှာနောက်ထပ် function တစ်ခု ထပ်ပြီးတော့ ထည့်ထားကို nested function ဟု ခေါ်သည်။

```
def outer():
    x = 3
    def inner():
        y = 3
        result = x + y
        return result
    return inner
```

inner() function ကို outer() function ၏ အတွင်း၌ ထည့်ထားသောကြောင့် nested function ဖြစ်သည်။ `return inner` တွင် function တစ်ခုသည် တခြား function တစ်ခုကို return အဖြစ် ပြန်ထုတ် ပေးသည်။

Function တစ်ခုကို လက်သည်းကွင်း() မပါဘဲထည့်ရေးလျှင် ထို function နာမည်သည် reference ဖြစ်သည်။ (Function names are reference.)

```
a = outer()
a
```

<function __main__.outer.<locals>.inner()>

```
print(a.__name__)
```

inner

Python တွင် function နာမည်သည် reference ဖြစ်သည်။ Function တစ်ခုကို နာမည် များစွာပေးနိုင်သည်။

```
a()
```

6

15.2 Function တစ်ခုကို Parameter အဖြစ်နဲ့ တခြား Function ထဲသို့ ထည့်ပေးခြင်း

Decorator Function အတွင်းရှိ function ကို wrapper function ဟုလည်း ခေါ်ဆိုလေ့ရှိသည်။ Decorator သည် တခြား function များကို argument အဖြစ်ယူပြီး decorator အတွင်းရှိ wrapper function က execute လုပ်ပေးသည်။

function1 ရဲ့ parameter အဖြစ်နဲ့ ထားပြီးတော့ function2 ကနေ လှမ်းခေါ်သည့် ဥပမာ ဖြစ်သည်။

```
def function1():
    print('hi i am function1')

def function2(func):  #func သည် function reference ဖြစ်သည်။
    print('hi i am function2, now I call function 1')
```

```
    func()

function2(function1)
```

hi i am function2, now I call function 1
hi i am function1

function1 ထဲကို function2 ထည့်လို့မရပါဘူး။ ဘာလို့လဲဆိုတော့ func ကြောင့်ပါ။ function2 ကသာ တခြား function တွေကို parameter အဖြစ်နဲ့ လက်ခံလို့ရသည်။ function2 ကို define လုပ်တုန်းက func ဆိုသည့် function reference ထည့်ရေးထားသောကြောင့် function တွေကို parameter အဖြစ်နဲ့ လက်ခံလို့ရသည်။ `def function2(func):` အောက်တွင် TypeError ပေးပုံကို ဖော်ပြထားသည်။

```
function1(function2)
```

```
TypeError: function1() takes 0 positional arguments but 1 was given
```

function1 သည် တခြားသော function ကို parameter အဖြစ်နဲ့ လက်ခံလို့ မရသည်။

15.3 Define a Decorator Function

တစ်ခါတစ်ရံ function များကို ခေါ်သုံးခွင့်သာ ရှိသည်။ ထို function ထဲသို့ ဝင်ရောက်ပြင်ဆင်ရေးသားခွင့် မရှိပေ။ Decorator တွင် decorate အလုပ်ခံရမည် function နှင့် decorate လုပ်မည့် decorator function ဟူ၍ (၂)မျိုး ကွဲပြားသည်။ Decorate လုပ်ရန်အတွက် ထို decorate အလုပ်ခံရမည့် function ကို decorator function

Decorator အကြောင်း ရှင်းပြရန် အောက်တွင် ရိုးရှင်းသည် return တစ်ခု ပြန်ပေးသည့် function တစ်ခု နမူနာအဖြစ် ဖော်ပြထားသည်။ ထို function ကို ဘာမပြင်မရေးဘဲ decorator function ကို သုံး၍ အကြီးစာလုံး ဖြစ်အောင် ပြောင်းမည်။

```
def print_str():
    return("good morning")

print_str()
```

```
'good morning'
```

Decorate လုပ်ပေးမည့် function သည် တခြား function ကို လက်ခံယူရမည်။ ထို့ကြောင့် တခြား function ကို လက်ခံယူရန် func ဆိုသည့် function reference ထည့်ရေးထားသည်။ ထိုအပြင် decorator function ၏ အတွင်း၌ function တစ်ခု ထပ်ရှိရမည်။ တစ်နည်းအားဖြင့် decorator function သည် nested function ဖြစ်သည်။

```
def str_upper(func):              # func ဖြင့် မူလ function ကို လက်ခံယူရသည်။
    def inner():                  # decorate လုပ်မည့် function တစ်ခု define လုပ်သည်။
        str1 = func()             # လက်ခံထားသည့် func ကို str1 ဖြင့် assign လုပ်သည်။
        return str1.upper()       # အကြီးစာလုံး ပြောင်းသည်။
```

```
    return inner                        # return လုပ်သည်။

def print_str():
    return("Good Morning")

d = str_upper(print_str)        # str_upper()သည် decorator function ဖြစ်သည်။
print(d)                         # function reference
print(d())                       # function return
```

```
<function str_upper.<locals>.inner at 0x00000129847180D8>
GOOD MORNING
```

အပေါ်တွင်ပြခဲ့သည့်နည်းသည် decorator function ကို မှန်ကန်သည့်အသုံးပြုသည့်နည်း မဟုတ်ပါ။ @ သင်္ကေတ၏ နောက်တွင် decorator function နာမည်ကို ရေးသည်။

```
@decorator_function_name
```

အောက်က decorator ဥပမာတွင် `print_str()`သည် decorate အလုပ်ခံရမည့် function ဖြစ်သည်။ `str_upper(func)`သည် decorate လုပ်ပေးမည့် function ဖြစ်သည်။

```
def str_upper(func):
    def inner():
        str1 =  func()
        return str1.upper()
    return inner

@str_upper                                  @decorator_function_name
def print_str():
    return("good morning")

print(print_str())   #မူလ print_str() function ကို ခေါ်ရင် အကြီးစာလုံးပြပါပြီ။
```

```
GOOD MORNING
```

@ သင်္ကေတ၏ နောက်တွင် decorator function နာမည်ကို ထည့်ရေးသည့် နည်းသည် မှန်ကန်သည့်နည်း ဖြစ်သည်။ အထက်တွင် ရှင်းပြခဲ့သည့် ဥပမာတွင် parameter တစ်ခုမျှ မပါဝင်ပါ။ Parameter များပါဝင်သည့် function ဖြစ်လျှင် မည်ကဲ့သို့ ပြုလုပ်မည်နည်း?

15.4 Decorator Function With Parameter

အောက်တွင် ကိန်းတစ်လုံးကို အခြားကိန်း တစ်လုံးဖြင့် စားသည့် function တစ်ခုကို ဖော်ပြထားသည်။ သုညဖြင့် စားလျှင် ZeroDivisionError ပေးသောကြောင့် ထို Error မဖြစ်ပေါ်စေဘဲ သတိပေးချက် ထုတ်ပြန်ပေးမည့် decorator function တစ်ခုကို နမူနာ အဖြစ် ရှင်းပြထားသည်။

```
def div(a,b):           # a ကို b နှင့် စားမည့် function ဖြစ်သည်။
```

```
    return(a/b)

print(div(4,0))
```

```
ZeroDivisionError: division by zero
```

ဤ decorator function သည် parameter များကို လက်ခံယူရမည် ဖြစ်သည်။ စားမည့်ကိန်း (denominator)သည် သုည(zero)ဖြစ်လျှင် “give proper input” ဟူသည့် message ထုတ်ပေးမည့် decorator function ဖြစ်သည်။

Decorator function ကို define လုပ်သည်။ func ဖြင့် မူလ function ကို လက်ခံယူရသည်။ ထိုနောက် parameter a နှင့် b ကို x နှင့် y အဖြစ် လက်ခံသည်။

y သည် စားမည့်ကိန်းဖြစ်သောကြောင့် `y == 0` ဖြစ်လျှင် message ထုတ်ပေးသည်။

```
def div_decorator(func):          # func ဖြင့် မူလ function ကို လက်ခံယူရသည်။
    def inner(x,y):               # decorate လုပ်မည့် function တစ်ခု define လုပ်သည်။
        if y == 0:                # လက်ခံထားသည့် func ကို b ကို 0 နှင့် ညီမည် စစ်သည်။
            return("give proper input") # သုညဖြစ်လျှင် message ထုတ်ပေးသည်။
        return func(x,y)    # y သည် 0 မဟုတ်လျှင် x နှင့် y ကို ပြန်ပေးမည်။
    return inner

@div_decorator                        @decorator_function_name
def div(a,b):
    return (a/b)

div(5,0)
```

```
'give proper input'
```

`return func(x,y)` သည် True if statement ၏ return ဖြစ်သည်။ `return func(x,y)` သည် False if statement ၏ return ဖြစ်သည်။ `return inner` သည် inner function ၏ return ဖြစ်သည်။ inner function ၏ return သည် True if statement ၏ return သို့မဟုတ် False if statement ၏ return ဖြစ်လိမ့်မည်။

15.5 Multiple decorator in Single Function

Function တစ်ခုကို တစ်ခုထက်ပိုများသည့် decorator function များဖြင့် decorate လုပ်နိုင်သည်။ အပေါ်တွင် တွေ့ခဲ့ပြီးသည့် good morning ဟု print ထုတ်ပေးသည့် `print_str():` function ကို decorator (၂)ခု တစ်ပြိုင်နက် တွဲ၍ decorate လုပ်သည့်နည်းကို ဖော်ပြမည်။ good morning ဆိုသည့် string ကို ပထမဦးစွာ အကြီးစာလုံး ပြောင်းရန် decorator function တစ်ခုတည်ဆောက်မည်။ ထိုနောက် good နှင့် morning ဆိုသည့် စာလုံး(၂)လုံး ဖြစ်အောင် ခွဲသည့်(split လုပ်သည့်) decorator function တည်ဆောက်မည်။ decorator function (၂)ခုကို တွဲသုံးမည်။

```
def print_str():                    # decorate အလုပ်ခံရမည့် function
    return("good morning")

print(print_str())
```

```
good morning
```

```
def upper_d(func):                  # အကြီးစာလုံး ပြောင်းပေးမည့် decorator function
    def inner():
        str1 = func()
        return str1.upper()         # အကြီးစာလုံး ပြောင်းပေးသည်။
    return inner

def split_d(func):     # စာလုံး(၂) ဖြစ်အောင် ခွဲသည့် (split လုပ်သည့်) decorator function
    def wrapper():     # စာလုံး(၂) ဖြစ်အောင် ခွဲသည့် (split လုပ်သည့်) inner function ဖြစ်သည်။
        str2= func()
        return str2.split()
    return wrapper

@split_d                    # စာလုံး(၂) ဖြစ်အောင် ခွဲသည့် (split လုပ်သည့်) decorator function
@upper_d                    # အကြီးစာလုံး ပြောင်းပေးမည့် decorator function
def print_str():
    return("good morning")

print(print_str())
```

```
['GOOD', 'MORNING']
```

Function တစ်ခုကို တစ်ခုထက်ပိုများသည့် decorator function များဖြင့် decorate လုပ်နိုင်သည့် အခါ မည်သည့် decorator function က အရင် execute လုပ်ရမည်ကို ဂရုစိုက် ရေးသားရန်လိုသည်။ မူလ function နှင့် အနီးကပ်ဆုံး ရှိသည့် decorator function က အရင် execute လုပ်သည်။

`@split_d` → နောက်မှ အလုပ်လုပ်မည့် decorator function

`@upper_d` → အရင် အလုပ်လုပ်မည့် decorator function

`def print_str():` decorate အလုပ်ခံရမည့် မူလ function

သတိပြုရန်အချက်မှာ `@split_d` နှင့် `@upper_d` ကို အပေါ်နှင့်အောက် ဖလှယ်လိုက်ပါက AttributeError ပေးလိမ့်မည်။ AttributeError: 'list' object has no attribute 'upper' အပေါ်က ဥပမာသည် function တစ်ခုထဲတွင် တစ်ခုထက်ပိုများသည့် decorator များဖြင့် အသုံးပြုနည်း ဖြစ်သည်။

```
def first(func):
    def inner():
        return ("first " + func() + " first")
```

```
    return inner

def second(func):
    def wrapper():
        return ("second " + func() + " second")
    return wrapper

@first
@second
def print_str():
    return("good morning")

print(print_str())
```

```
first second good morning second first
```

`@first` → နောက်မှ အလုပ်လုပ်မည့် decorator function

`@second` → အရင် အလုပ်လုပ်မည့် decorator function

`def print_str():` → decorate အလုပ်ခံရမည့် မူလ function

@second သည် အရင် အလုပ် လုပ်သောကြောင့် good morning နှင့် နီးကပ်စွာ တည်ရှိသည်။

15.6 Decorator With Parameter

Decorator မှာ parameter တွေပါလာရင် ဘယ်လိုသုံးမလဲ? မူလ ရှိပြီးသား good morning ဟု print ထုတ်ပေးသည့် print_str(): function ကို parameter ပါသည့် decorator function ဖြင့် decorate လုပ်ပါမည်။ decorator function ထဲတွင် "Python" စာလုံးကို ထည့်ပါမည်။

```
def outer(expr):       # expr သည် parameter ကို လက်ခံမည် variable ဖြစ်သည်။
    def upper_d(func):
        def inner():
            return func() + expr
        return inner
    return upper_d

@outer("Python")              # parameter ပါသည့် decorator function
def print_str():
    return("good morning ")

print(print_str())
```

```
good morning Python
```

`@outer("Python")` မှ "Python" သည် `def outer(expr):`ဆီသို့ရောက်ပြီးနောက် expr variable ဖြင့် parameter ကို လက်ခံသည်။ ထိုနောက် func() နှင့် ပေါင်းပြီး return ပြန်ပေးသည်။

15.7 Single Decorator on Multiple Function

Decorator တစ်ခုကို function များစွာအတွက် အသုံးပြုခြင်း။

15.7.1 Single Decorator on Multiple Function Example-1

ကိန်း(၁)လုံးကို တည်ပြီး ကိန်း(၁)လုံးဖြင့် စားသည့် function တစ်ခုနှင့် ကိန်း(၃)လုံးကို အဆင့်ဆင့် စားသည့် function တစ်ခု တည်ဆောက်သည်။ ထိုကိန်းများတွင် denominator သည် သုည(zero) ဖြစ်လျှင် ZeroDivisionError ပေးမည်ဖြစ်သောကြောင့် decorator ကို အသုံးပြု၍ သတိပေးချက် ထုတ်ပြန် ပေးမည်။ အလုပ်လုပ်ပုံတူညီသောကြောင့် decorator တစ်ခုသာ ရေးပြီး function နှစ်ခုစလုံးအတွက် အသုံးပြုမည်။

```
def div1(x,y):       # 2 parameters
    return x/y

def div2(a,b,c):     # 3 parameters
    return a/b/c

print(div1(10,5))
print(div2(10,2,5))  # 10/2 = 5 / 5 = 1
```

```
2.0
1.0
```

`div_decorator (func)` ကို define လုပ်သည်။ Decorate အလုပ်ခံရမည့် function တစ်ခုတွင် parameter နှစ်ခုပါပြီး ကျန် function တစ်ခုတွင် parameter သုံးခုပါသည်။ ထို့ကြောင့် ***args** ကို အသုံးပြုသည်။ ***args** သည် parameter အရေအတွက် ကန့်သတ်ချက်မရှိ လက်ခံနိုင်သည်။

```
def div_decorator(func):
    def inner(*args):           # *args
        list1 = []              # *args များ ထည့်ရန် list1 တည်ဆောက်သည်။
        list1 = args[1:]   # args ထဲမှ first item ကိုဖယ်ပြီး ကျန်ကိန်းများကို asssing လုပ်
        for i in list1:
            if i==0:            #  list1 ထဲမှ ကျန်ကိန်းများ zero ဖြင့် ညီ မညီ စစ်သည်။
                return "Give proper input!"  # ညီလျှင် message ထုတ်ပေးသည်။
        return func(*args)      # မညီလျှင် parameter အားလုံးကို ပြန်ပေးသည်။
    return inner

@div_decorator
def div1(x,y):       # 2 parameters
    return x/y

@div_decorator
def div2(a,b,c):     # 3 parameters
```

```
    return a/b/c

print(div1(10,5))
print(div2(10,0,5))
```

```
2.0
Give proper input!!!
```

ဤနည်းဖြင့် decorator တစ်ခုကို function များစွာအတွက် အသုံးပြုနိုင်သည်။

15.7.2 Single Decorator on Multiple Function Example-2

Decorator တစ်ခု ရေးထားပြီး နေရာတော်တော်များများတွင် အသုံးပြုခွင့်ရလျှင် အလွန်အကျိုးရှိသည်။ အောက်တွင် function တော်တော်များများအတွက် သုံးနိုင်သည့် decorator တစ်ခုကို ဥပမာ အဖြစ် ဖော်ပြထားသည်။ hello() function နှင့် bye()function ကို decorate လုပ်မည့် decorator ဖြစ်သည်။

```
def hello():
    print("Hello")

hello():
```

```
Hello
```

```
def bye():
    print("Bye")

def bye()
```

```
Bye
```

hello() function နှင့် bye()function တို့၏ output ဖြစ်သည့် `Hello` နှင့် `Bye` ကို ***** ဖြင့် အပေါ်အအောက် wrap လုပ်မည့် decorator တစ်ခု ဆောက်မည်။

```
def my_decorator(func):
    def wrapper_func():
        print("***********")
        func()
        print("***********")
    return wrapper_func

@my_decorator
def hello():
    print("Hello")

@my_decorator
def bye():
    print("Bye")

hello()
```

```
***********
```

```
Hello ***********
```

```
bye()
```

```
***********
Bye
***********
```

ဤနည်းဖြင့် decorator တစ်ခုကို function များစွာအတွက် အသုံးပြုနိုင်သည်။

15.7.3 Single Decorator on Multiple Function Example-3

ကိန်းအလုံးရေ 100000000 ကို range() function ဖြင့် loop ပတ်လျှင် အချိန်မည်မျှကြာမည်နည်း။ ကိန်းအလုံးရေ 100000000 ရှိသည့် list တစ်ခုကို loop ပတ်လျှင် အချိန်မည်မျှကြာမည်နည်း။ နှစ်ခုစလုံးသည် ကြာအချိန်ကို သိလိုခြင်း ဖြစ်သောကြောင့် decorator တစ်ခု ရေးပြီး စမ်းသပ်မည်။

```
def range_loop():
    print ("This is range() loop.")
    for i in range(100000000):     # range() ကို ပတ်မည့် for loop
        i = i *2

range_loop()
```

```
This is range() loop.
```

```
def list_loop():
    print ("This is list loop.")
    for i in list(range(100000000)):    # list ကို ပတ်မည့် for loop
        i = i *2

list_loop()
```

```
This is list loop.
```

```
from time import time

def performance(func):
    def wrap_func(*args, **kwargs):
        t1 = time()                        # loop မစခင် အချိန်
        result = func (*args, **kwargs)
        t2 = time()                        # loop ပတ်ပြီးသည့် အချိန်
        print(f"It tooks {t2-t1} secs")  # loop ပတ်ရန် ကြာချိန်
        return result
    return wrap_func

@performance
```

```
def range_loop():
    print ("This is range() loop. ")
    for i in range(100000000):
        i = i *2

@performance
def list_loop():
    print ("This is list loop.")
    for i in list(range(100000000)):
        i = i *2

range_loop()
```

```
This is range() loop.
It tooks 3.961437463760376 secs
```

```
list_loop()
```

```
This is list loop.
It tooks 5.5374391078948975 secs
```

15.8 Decorator ကြောင့် မူလ(original) function ရဲ့ အချက်အလက်တွေကို ဖတ်မရခြင်း

decorator က function ရဲ့တချို့ အချက်အလက်တွေက မဖော်ပြတော့ပါဘူး။ decorator ကို သုံးလိုက်တာနဲ့ function နာမည် ၊ parameter ၊ docstring စတာတွေကို မဖော်ပြတော့ပါဘူး။ decorator က functionailty အသစ်တွေ ထပ်ထည့်ပေးပေမဲ့ မူလ function က အချက်အလက်တွေက မဖော်ပြတော့ပါဘူး။ decorator တွင် ပြီးတော့ control လုပ်သွားတာပါ။ မူလ function ရဲ နာမည်ကို ဖတ်ချင် .__name__ သုံးလို့ မရတော့ပါဘူး? ဥပမာလေးနဲ့ လေ့လာကြရအောင်။ "good morning" လို့ ဖော်ပြပေးမဲ့ function တစ်ခုကို အောက်မှာ ဖော်ပြထားပါတယ်။

```
def greet():
    'This is docstring for function greet' # python function docstri
ng
    return "good morning"

print(greet())
```

```
good morning
```

function ရဲ့နာမည်ကို __name__ နဲ့ ဖတ်ကြည့်ရအောင်

```
print(greet.__name__)
```

```
greet
```

function ရဲ့နာမည်က `greet` ဖြစ်ကြောင်း ဖော်ပြပေးသည်။ help(greet) ဖြင့် ဖတ်ကြည့်ရအောင်

```
help(greet)
```

```
Help on function greet in module __main__:
```

```
greet()
    This is docstring for function greet
```

အဲဒီ function ကို decorator နဲ့သုံးပြီးတော့ `good morning` ကို အကြီးစားလုံးပြောင်းကြည့်ရအောင်

```
def decorator(func):
    def inner():
        str1= func()
        return str1.upper()
    return inner

@decorator
def greet():
    return "good morning"

print(greet())
```

```
GOOD MORNING
```

အခု greet() function ကို decorator နဲ့ တွဲသုံးလိုက်ပါပြီ။ good morning ကို decorator မှာ ရေးထားတဲ့အတိုင်း အကြီးစာလုံးတွေနဲ့ ထုတ်ပေးပါပြီ။ function ကို __name__ နဲ့ ဖတ်လို့ မရတော့ပါဘူး။ help(greet) ကလည်း မူလ function က အချက်အလက်တွေကို မပြတော့ပါဘူး။

```
print(greet.__name__)
```

```
inner
```

```
help(greet)
```

```
Help on function inner in module __main__:
inner()
```

15.8.1 import functools

အရင်ကလို မူလ function တွေရဲ့ အချက်အလက်တွေကို ဖတ်ဘို့ functools ကို import လုပ်ရမည်။

@funtools.wrap(func) ကုဒ်တစ်ကြောင်းထည့်ပေးရမည်။

```
import functools
def decorator(func):
    @functools.wraps(func)
    def inner():
        str1= func()
        return str1.upper()
    return inner

@decorator
def greet():
    return "good morning"
```

```
print(greet())
```

အခု `print(greet.__name__)` ဆိုရင် မူလ function ကအချက်အလက်တွေကို ပြန်ဖတ်လို့ ရပါပြီ။

Decorator function ကို method မှာလည်း သုံးလို့ရသည်။

Method ဆိုတာ class တစ်ခုတွင်းမှာ ရှိတဲ့ function တွေပါ။ ဥပမာနဲ့ပြဘို့ Printing ဆိုတဲ့ class တစ်ခုတည်ဆောက်ပါမည်။ ထည့်ပေးတဲ့ နာမည်(string)ကို print ထုတ်ပေးရုံပါ။

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

    def print_name(self):
        print("Enter User Name is: ", self.name)

p = Printing("Python Lover")
p.print_name()
```

```
Enter User Name is:  Python Lover
```

အပေါ်က class ထဲမှာ ရှိတဲ့ print_name ဆိုတဲ့ method ကို decorate လုပ်မှာပါ။ method ကို decorate လုပ်မှာဖြစ်လို့ decorator function မှာ method ဆိုပြီးထည့်ပေးရမည်။ လုပ်ပေးမဲ့အလုပ်က ထည့်ပေးတဲ့ နာမည်က Python Lover ဆိုရင် "Hey my name is also same" ဆိုတဲ့ message ထုတ်ပေးရုံပါ။ ထည့်ပေးတဲ့ နာမည်နဲ့ မတူဘူးဆိုရင် နာမည်ကိုပဲ print ထုတ်ပေးမှာပါ။

```
def check_name(method):
    def inner(name_ref):
        if name_ref.name == "Python Lover":
            print("Hey my name is also same")
        else:
            method(name_ref)
    return inner

class Printing:
    def __init__(self,name):
        self.name = name

    @check_name
    def print_name(self):
        print("Enter User Name is: ", self.name)

p = Printing("Python Lover")
p.print_name()
```

```
Hey my name is also same
```

မတူတဲ့ နာမည် ထည့်ကြည့်ရအောင်

```
p = Printing("Java")
p.print_name()
```

```
Enter User Name is:  Java
```

ဒါက decorator function ကို method မှာသုံးတဲ့နည်းပါ။

15.9 Class decorator အကြောင်း

Call method အကြောင်းအနည်းငယ် ရှင်းပြပါမည်။ အပေါ်မှာ ရေးခဲ့တဲ့ Printing class ကို p() ဆိုပြီးတော့ ခေါ်သုံးလို့ မရပါဘူး။

```
p = Printing("Java")
p()
```

```
TypeError: 'Printing' object is not callable
```

P() က ခေါ်သုံးလို့ရတဲ့ `callable object` မဟုတ်ဘူးဆိုပြီးတော့ **TypeError** ပေးပါလိမ့်မည်။ အဲတော့ ခေါ်သုံးလို့ရအောင် အနည်းငယ် ပြင်ရေးရပါမည်။ `print_name` ဆိုတဲ့ `method name` နေရာမှာ __call__ ဆိုတဲ့ special method လို့ ပြောင်းလိုက်ရင် call method ကို သုံးလို့ရပါပြီ။

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

    def __call__(self):
        print("Enter User Name is: ", self.name)

p = Printing("Python Lover")
p()
```

```
Enter User Name is:  Python Lover
```

__call__ method ကို အနည်းငယ် ရင်းနှီးသွားရင် Class decorator ဥပမာကို ဆက်လို့ရပါပြီ။

15.9.1 Class Decorator Example-1

```
def greet():
    return "good morning"

print(greet())
```

```
good morning
```

အပေါ်က greet() function ရဲ့ output ဖြစ်တဲ့ good morning ကို အကြီးစာလုံးတွေနဲ့ ပေါ်အောင်

class decorator နဲ့ decorate လုပ်ပါမည်။

```
class Decorator:
    def __init__(self, func):
        self.func = func

    def __call__(self):
        str1 = self.func()
        return str1.upper()

@Decorator
def greet():
    return "good morning"

print(greet())
```

```
GOOD MORNING
```

15.9.2 Class Decorator Example-2

ဒီတစ်ခါ decorate လုပ်ပေးမှာက function မဟုတ်ပါဘူး။ class ကနေ decorate လုပ်ပေးမှာပါ။

အောက်ပါ happy_hr ဆိုတဲ့ function တစ်ခုက 'Happy Hours' ဆိုတဲ့ စာတန်းကို print ထုတ်ပေးမှာပါ။

```
def happy_hr():
    print ('Happy Hours')

happy_hr()
```

```
'Happy Hours'
```

အဲဒါကို Bar class နှင့် decorate လုပ်ကြည့်ရအောင်။

```
class Bar(object):
    def __init__(self, func):
        self.func = func

    def __call__(self):
        print ('Bar going to Open')
        self.func()
        print ('Bar Closed')

@Bar
def happy_hr():
    print ('Happy Hours')

happy_hr()
```

```
Bar going to Open
Happy Hours
```

```
Bar Closed
```

15.9.3 Class Decorator Example-3

အပေါ်မှာ တည်ဆောက်ခဲ့တဲ့ ကိန်း(၁)လုံးကို တခြားကိန်း (၁)လုံးနဲ့ စားတဲ့ function ပါ။ ပိုင်းခြေက သုည သို့မဟုတ် သုညနဲ့စားမယ်ဆိုရင် ZeroDivisionError: division by zero ပေးမှာပါ။ အဲလို ZeroDivisionError မပေးအောင်လို့ message ထုတ်ပေး Class decorator တစ်ခုကို ဥပမာ အဖြစ် ရှင်းပြထားသည်။

```
def div(a,b):
    return(a/b)

print(div(10,0))
```

ZeroDivisionError: division by zero

အောက်မှာ Check_denominator ဆိုတဲ့ class ရေးပြီးတော့ မူလ function ကို decorate လုပ်ပါမည်။

```
class Check_denominator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        if args[1] == 0:
            return "You can't devide, change the input"
        else:
            self.func(*args, **kwargs)

@Check_denominator
def div(a,b):
    return(a/b)

print(div(10,0))
```

```
You can't devide, change the input
```

-End-