Decorator

Python-ի դեկորատորները առաջին հայացքից կարող են բարդ թվալ, բայց իրականում դրանք հիմնված են շատ տրամաբանական սկզբունքների վրա:

- Ի՞նչ է դեկորատորը

Դեկորատորը ֆունկցիա է, որը թույլ է տալիս փոփոխել կամ ընդլայնել մեկ այլ ֆունկցիայի վարքագիծը՝ առանց դրա կոդը ուղղակիորեն փոխելու: Պատկերացրեք դեկորատորը որպես «փաթեթավորում» (wrapper) նվերի համար. նվերը մնում է նույնը, բայց փաթեթավորումը ավելացնում է նոր տեսք կամ լրացուցիչ հատկություններ:

- Ինչպե՞ս է այն սահմանվում

Դեկորատորը սահմանվում է որպես սովորական ֆունկցիա, որը որպես արգումենտ ընդունում է մեկ այլ ֆունկցիա: Օգտագործելիս կիրառվում է @ նշանը:

```
def my_decorator(func):
    def wrapper():
        print("Ֆունկցիայի կանչից առաջ")
        res = func()
        print("Ֆունկցիայի կանչից հետո")
        return res
    return wrapper

@my_decorator
def say_hello():
    print("Ողջույն!")

say_hello()
```

- Ի՞նչ է կատարվում իրականում

Երբ տեսնում եք @my_decorator, դա պարզապես կարճ ճանապարհ է գրելու հետևյալը.

```
say_hello = my_decorator(say_hello)
```

Այսինքն՝ մենք վերցնում ենք մեր say_hello ֆունկցիան, այն որպես արգումենտ փոխանցում ենք my_decorator-ին, և այն ինչ my_decorator-ը մեզ հետ կտա (իր ներսի wrapper-ը), կդառնա մեր նոր say_hello-ն:

- Ինչո՞ւ պետք է սահմանենք ներդրված (wrapper) ֆունկցիա

Սա ամենակարևոր պահն է: Եթե մենք չստեղծենք ներդրված wrapper ֆունկցիան, մենք չենք կարողանա ստեղծել նոր վարքագիծ և այն վերադարձնել որպես ֆունկցիա՝ հետագայում կանչելու համար:

➔ Նպատակը: Մենք ուզում ենք ստեղծել մի նոր «տրամաբանական բլոկ», որը կպարունակի թե՛ հին ֆունկցիան, թե՛ մեր ավելացրած նոր կոդը:

➔ Closure: Ներդրված ֆունկցիան «հիշում է» այն միջավայրը, որտեղ ստեղծվել է: Այն հիշում է func-ը, նույնիսկ երբ դեկորատորի աշխատանքն արդեն ավարտվել է:

Այսինքն եթե չգրենք ներդրված ֆունկցիա, ամբողջ տրամաբանությունը կխախտվի, չենք կարող հասկանալ նոր ֆունկցիային ինչ արգումենտ փոխանցենք:

- Ինչո՞ւ պետք է վերադարձնենք (return) ներդրված ֆունկցիան

Երբ մենք գրում ենք @my_decorator, Python-ը կատարում է հետևյալը՝ say_hello = my_decorator(say_hello):

➔ Եթե դեկորատորը չվերադարձնի wrapper-ը, ապա say_hello-ն կդառնա None (կամ այն, ինչ վերադարձնում է դեկորատորը), և մենք այլևս չենք կարողանա այն կանչել որպես ֆունկցիա:

➔ Մենք վերադարձնում ենք ֆունկցիայի հղումը, որպեսզի հետագայում, երբ ծրագրի մեջ կանչենք say_hello(), իրականում կանչվի մեր ստեղծած wrapper-ը:

- Քայլ առ քայլ գործընթացը

ա) Դեկորատորի սահմանումը

```
def my_decorator(func): # 1. func-ը հենց մեր say_hello ֆունկցիան է
    def wrapper():      # 2. Ստեղծում ենք նոր «տուփ» (ֆունկցիա)
        print("...")    # 3. Ավելացնում ենք նոր գործողություն
        res = func()          # 4. Կանչում ենք մեր իրական ֆունկցիան
        print("...")    # 5. Ավելացնում ենք ևս մեկ գործողություն
        return res      # 6 Վերադարձնում ենք այն ինչ որ սկզբնական
ֆունկցիան
    return wrapper      # 7. Վերադարձնում ենք այս նոր տուփը
```

բ) Ֆունկցիայի «փաթեթավորումը»

Երբ Python-ը տեսնում է @my_decorator, նա անմիջապես կանչում է my_decorator(say_hello):

- func-ը դառնում է օրիգինալ say_hello-ն:
- wrapper ֆունկցիան ստեղծվում է և իր ներսում «հիշում է» func-ին (սա կոչվում է Closure):
- return wrapper տողը ասում է. «Հիմա say_hello անունը թող վերաբերվի ոչ թե հին ֆունկցիային, այլ այս նոր wrapper-ին»:

գ) Կանչը

Երբ վերջում գրում ենք say_hello(), մենք իրականում կանչում ենք wrapper()-ը:

1. Այն տպում է "Ֆունկցիայի կանչից առաջ":
2. Հետո այն կատարում է func()-ը (որը մեր օրիգինալ "Ողջույն!" տպող կոդն է):
3. Հետո տպում է "Ֆունկցիայի կանչից հետո":

Պատկերացնենք, որ մեր say_hello ֆունկցիան հիմա ընդունում է մեկ արգումենտ, որը մարդու անուն է:

```
def my_decorator(func):
    def wrapper():  # <--- Ուշադրություն. wrapper-ը արգումենտ չի ընդունում
        print("Ֆունկցիայի կանչից առաջ")
        res = func()      # <--- func-ն էլ կանչվում է առանց արգումենտի
```

```
        print("Ֆունկցիայի կանչից հետո")
        return res
    return wrapper

@my_decorator
def say_hello(name):  # Ավելացրինք 'name' արգումենտը

    print(f"Ողջույն, {name}!")

# Կանչում ենք ֆունկցիան
say_hello("Աննա")
```

Ի՞նչ տեղի կունենա: Մենք կստանանք սխալ՝

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

Ինչո՞ւ: Որովհետև @my_decorator գրելով՝ say_hello անունը այժմ հղվում է wrapper ֆունկցիայի վրա: Երբ մենք գրում ենք say_hello("Աննա"), մենք իրականում փորձում ենք "Աննա" արժեքը փոխանցել wrapper()-ին, բայց մենք wrapper-ը սահմանելիս նրան ոչ մի արգումենտ չէինք տվել: Այս խնդիրը լուծելու համար կարող ենք, wrapper ֆունկցիային ավելացնել մեկ արգումենտ, օրինակ՝

```
def my_decorator(func):
    def wrapper(name):  # wrapper ֆունկցիան ստանում է արգումենտ
        print("Ֆունկցիայի կանչից առաջ")
        res = func(name)      # <--- func-ը կանչվում է արգումենտով
        print("Ֆունկցիայի կանչից հետո")
        return res
    return wrapper

@my_decorator
def say_hello(name):  # Ավելացրինք 'name' արգումենտը
    print(f"Ողջույն, {name}!")

# Կանչում ենք ֆունկցիան
say_hello("Աննա")
```

Այստեղ ամեն ինչ աշխատում է նորմալ: Տեսնում ենք հետևյալ արդյունքը:

```
Ֆունկցիայի կանչից առաջ
Ողջույն, Աննա!
Ֆունկցիայի կանչից հետո
```

Բայց մենք ուզում ենք մեր գրած դեկորատորը կախել ցանկացած ֆունկցիայի վրա: Մենք չգիտենք արդյոք, որ ֆունկցիան քանի հատ և ինչ տիպի արգումենտներ է ստանում, որպեսզի կարողանանք փոխել դեկորատորի կոդը(արգումենտների քանակը), հետևաբար պետք է գրել այնպիսի ունիվերսալ կոդ, այնպես որ մենք կախված չլինենք ֆունկցիային փոխանցվող արգումենտների քանակից և տիպերից: Այս դեպքում կարող ենք ներդրված ֆունկցիան սահմանել այնպես, որ այն ստանա անսահման քանակությամբ արգումենտներ, և այդ արգումենտները փոխանցել իրական ֆունկցիային, այսպես մենք ստիպված չենք լինում փոփոխել դեկորատորի կոդը ֆունկցիայի արգումենտների տեսակներից ելնելով:

```
def my_decorator(func):
    def wrapper(*args, ** kwargs):  # wrapper ֆունկցիան ստանում է անսահման քանակությամբ positional և keyword արգումենտներ
        print("Ֆունկցիայի կանչից առաջ")
        res = func(*args, ** kwargs)      # func-ը կանչվում է իրական արգումենտներով
        print("Ֆունկցիայի կանչից հետո")
        return res
    return wrapper

@my_decorator
def say_hello(name):
    print(f"Ողջույն, {name}!")

# Կանչում ենք ֆունկցիան
say_hello("Աննա")
```

Այս կանչի ընթացքում wrapper ֆունկցիային, որպես positional արգումենտ է գնում ‘Աննա’ տողը, ֆունկցիայի մեջ այն լինելու է args-ի մեջ հետևյալ տեսքով՝ (‘Աննա’,), իսկ kwargs-ը լինելու է դատարկ բառարան: Իսկ ինչ կլինի, երբ ֆունկցիան կանչենք այսպես:

```
say_hello(name = "Աննա")
```

Այս դեպքում, մեր ֆունկցիային փոխանցում ենք միայն keyword արգումենտ, հետևաբար wrapper ֆունկցիան որպես արգումենտներ կստանա, kwargs = {‘name’ : ‘Աննա’}, args = ():

Երբ մենք ֆունկցիան «փաթաթում» ենք դեկորատորով, մենք իրականում փոխարինում ենք մեր օրիգինալ ֆունկցիան wrapper-ով: Դրա պատճառով կորում են ֆունկցիայի մետատվյալները (անունը, docstring-ը և annotation-ները):
Եկեք տեսնենք դա գործնականում:
1. Ֆունկցիան առանց դեկորատորի
Եկեք ստեղծենք մի սովորական ֆունկցիա և տեսնենք, թե ինչ է ցույց տալիս help()-ը:

```
def say_hello(name: str):
    """Այս ֆունկցիան ողջունում է օգտատիրոջը:"""
```

```
    print(f"Ողջույն, {name}!")

help(say_hello)
```

Արդյունքը կլինի ճիշտ.

```
say_hello(name: str)
Այս ֆունկցիան ողջունում է օգտատիրոջը:
```

---

2. Ֆունկցիան դեկորատորով
Հիմա կիրառենք մեր դեկորատորը, և տեսնենք, թե ինչպես է Python-ը «խառնվում իրար»:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        """Ես wrapper ֆունկցիան եմ:"""
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def say_hello(name):
    """Այս ֆունկցիան ողջունում է օգտատիրոջը:"""
    print(f"Ողջույն, {name}!")

help(say_hello)
```

Արդյունքը կզարմացնի.

```
wrapper(*args, **kwargs)
Ես wrapper ֆունկցիան եմ:
```

Ի՞նչ պատահեց: Քանի որ դեկորատորը վերադարձրեց wrapper ֆունկցիան, Python-ը հիմա կարծում է, թե say_hello-ն հենց այդ wrapper-ն է: Օրիգինալ անունը և նկարագրությունը («Այս ֆունկցիան ողջունում է...») կորան:

---

3. Ինչպե՞ս կարող ենք լուծել այս խնդիրը:
Մենք կարող ենք wrapper ֆունկցիան ձևափոխել այնպես, որ լինի օրիգինալ ֆունկցիայի նման:

```
import inspect

def my_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"Գործարկվում է {func.__name__} ֆունկցիան...")
        result = func(*args, **kwargs)
        print(f"{func.__name__} ավարտվեց:")
        return result
```

```
    wrapper.__name__ = func.__name__ #Փոխում ենք ֆունկցիայի անունը
    wrapper.__doc__ = func.__doc__ #Փոխում ենք ֆունկցիայի docstring-ը
    wrapper.__signature__ = inspect.signature(func) # Փոխում ենք
ֆունկցիայի արգումենտները և annotation-ները
    return wrapper

@my_decorator
def say_hello(name):
    """Այս ֆունկցիան ողջունում է օգտատիրոջը:"""
    print(f"Ողջույն, {name}!")

help(say_hello)
print(f"Ֆունկցիայի իրական անունը: {say_hello.__name__}")
```

Արդյունքը հիմա կլինի կատարյալ.

```
say_hello(name)
Այս ֆունկցիան ողջունում է օգտատիրոջը:
Ֆունկցիայի իրական անունը: say_hello
```

Բայց python-ը տրամադրում է գործիք, որը ամեն ինչ կանի մեր փոխարեն, այն նույնպես դեկորատոր է: Ահա թե ինչպես կարող ենք կիրառել այն

```
import functools

def my_decorator(func):
    @functools.wraps(func) #Այս դեկորատորը ամեն ինչ անում է, մեր
փոխարեն
    def wrapper(*args, **kwargs):
        print(f"Գործարկվում է {func.__name__} ֆունկցիան...")
        result = func(*args, **kwargs)
        print(f"{func.__name__} ավարտվեց:")
        return result
    return wrapper
```

Ամբողջական և ճիշտ օրինակ (functools.wraps-ով)
Լավագույն պրակտիկան միշտ functools.wraps օգտագործելն է, որպեսզի ֆունկցիան չկորցնի իր անունը և տվյալները:

```python
import functools

def logger(func):
        @functools.wraps(func) # Պահպանում է օրիգինալ ֆունկցիայի անունը և տվյալները
    def wrapper(*args, **kwargs):
        print(f"Գործարկվում է {func.__name__} ֆունկցիան...")
          result = func(*args, **kwargs) # Փոխանցում ենք բոլոր հնարավոր արգումենտները
        print(f"{func.__name__} ավարտվեց:")
        return result
    return wrapper

@logger
def add(a, b):
    """Գումարում է երկու թիվ"""
    return a + b

print(add(5, 10))
print(add.__doc__) # Շնորհիվ @wraps-ի, սա կտպի "Գումարում է երկու թիվ"
```

Կիրառությունը
Պատկերացրեք՝ ունեք 100 հատ ֆունկցիա և ուզում եք իմանալ, թե յուրաքանչյուրը քանի վայրկյանում աշխատեց:

- Առանց դեկորատորի: Պետք է 100 ֆունկցիայի մեջ էլ ձեռքով գրեք ժամանակը չափող կոդ:
- Դեկորատորով: Գրում եք մեկ դեկորատոր և @timer ավելացնում բոլոր 100 ֆունկցիաների գլխին:

Խնդիրներ

1. Գրել դեկորատոր, որը կչափի և կտպի ֆունկցիայի կատարման տևողությունը վայրկյաններով:

2. Գրել դեկորատոր, որը ֆունկցիայի կանչից առաջ և հետո կտպի հաղորդագրություն

Օրինակ.

```
Function started
... ֆունկցիայի աշխատանքը ...
Function finished
```

3. Գրել դեկորատոր, որը ֆունկցիան կանչելիս կտպի՝

- Ֆունկցիայի անունը
- Փոխանցված բոլոր արգումենտները
- Ֆունկցիայի վերադարձրած արդյունքը

4. Գրել դեկորատոր, որը կհաշվի, թե քանի անգամ է ֆունկցիան կանչվել

5. Գրել դեկորատոր, որը թույլ կտա ֆունկցիան կանչել ոչ ավել, քան վայրկյանը 1 անգամ: Եթե ավելի հաճախ են կանչում, պետք է տպի զգուշացում:

7. Գրել դեկորատոր, որը կպահի (cache) ֆունկցիայի հաշվարկած արդյունքները: Եթե նույն արգումենտներով ֆունկցիան կրկին կանչվի, այն պետք է վերադարձնի պահպանված արժեքը՝ առանց ֆունկցիայի մարմինը նորից աշխատեցնելու: