

מעטרים או מקשטים בפייתון

זהו כלי חזק ואלגנטי שמאפשר לשנות או להרחיב את השימוש בפונקציות או פעולות קיימות מבלי לשנות את הקוד של הפונקציה עצמה. לרוב רק מתכנתי פייתון ותיקים ומנוסים ברמה גבוהה מבינים את הטכניקה הזו.

זוהי דרך מצוינת להוסיף יכולות לפונקציות שונות שעוסקות בנושאי תזמון, ניהול זיכרון מטמון, רישום פעילות (לוג) או וידוא זהות או אמיתות של נתונים או אנשים (הרשאות).

Decorators are a powerful and elegant feature in Python that allows you to modify or extend the behavior of functions or methods without changing their actual code. They are an excellent way to apply reusable functionality across multiple functions, such as timing, caching, logging, or authentication

A Python decorator is a function that takes in a function and returns it by adding some functionality.

על מנת להבין את המנגנון הזה נראה מספר מקרים של איך עובדות פונקציות בפייתון. (הסבר בשלבים)

אפשר לעשות השמה של פונקציה למשתנה ואז לקרוא לה על ידי שימוש במשתנה.

שלב 1

```
def plus_one(number):  
    return number + 1
```

```
add_one = plus_one # after this, the variable add_one is a function  
add_one(5)
```

התוצאה של הפעולה תהיה 6. שימו לב שההשמה של plus_one למשתנה הייתה ללא סוגריים ואז את המשתנה מפעילים עם סוגריים, כלומר המשתנה למעשה הפך לפונקציה בעצמו.

שלב 2

בנוסף, אפשר בפייתון להגדיר פונקציה בתוך פונקציה, (פונקציות מקוננות) לדוגמה:

```
def plus_one(number):  
    def add_one(number):  
        return number + 1  
    result = add_one(number)  
    return result  
x=plus_one(4)  
print(x)
```

הפונקציה plus_one קוראת לפונקציה add_one ומקבלת ממנה החזר לתוך המשתנה result. הפעולה נעשית על הערך 4 ולכן מוחזר הערך 5. שימו לב שאת ההוספה של אחד עושים בתוך add_one כי לפונקציה פנימית יש גישה למשתנים ופרמטרים של העוטפת אותה. לכן המשתנה number בתוך add_one הוא הפרמטר של plus_one.

```
def plus_one(number):  
    def add_one():  
        return number + 1  
    result = add_one()  
    return result  
x=plus_one(4)  
print(x)
```

שלב 3

אפשר בפייתון להעביר פונקציה כפרמטר לפונקציה אחרת, למשל:

```
def plus_one(number):  
    return number + 1  
  
def function_call(function):  
    number_to_add = 5  
    return function(number_to_add)  
  
print(function_call(plus_one))
```

כאן הפונקציה `function_call` מקבלת כפרמטר פונקציה של הוספת 1 (`plus_one`). היא מחזירה ערך על ידי הפעלת הפונקציה שקיבלה כפרמטר על משתנה שהגדירה. התוצאה תהיה 6 כמובן (שגם יודפס, אפשר לנסות זאת בבית).

שלב 4

בפייתון אפשר גם להחזיר פונקציה כערך מוחזר מפונקציה אחרת, למשל:

```
def hello_function():  
    def say_hi():  
        return "Hi"  
    return say_hi  
hello = hello_function()  
hello()
```

הפונקציה `say_hi` מחזירה את המחרוזת "Hi". היא מוגדרת בתוך פונקציה אחרת: `hello_function` שמחזירה את `say_hi`. בתוכנית הראשית מגדירים משתנה `hello` שמקבל את הערך המוחזר מ `hello_function` שזו בעצם הפונקציה: `say_hi`. לאחר

מכּן אפשר לקרוא למשתנה hello כפונקציה, כי הוא מכיל את say_hi (פונקציה).
כתוצאה מהמהלך הזה מוחזר הערך "Hi" (אפשר להדפיס את התוצאה ולוודא שזה קורה - נסו בבית).

שלב 5

כפי שלמדנו, לפונקציה פנימית ישנה גישה למשתנים ולפרמטרים של הפונקציה שמכילה אותה. זה נקרא closure. לדוגמה:

```
def print_message(message):  
    def message_sender():  
        print(message) # the parameter of print_message  
    message_sender()  
print_message("Some random message")
```

בתוכנית הראשית מפעילים את הפונקציה print_message עם פרמטר שהוא מחרוזת, בתוך print_message מוגדרת פונקציה פנימית message_sender שגם קוראים לה. המחרוזת מודפסת כי היא מועברת כפרמטר ל print_message ולפונקציה הפנימית (המדפיסה) message_sender יש גישה לפרמטר כי היא פנימית.

כעת לאחר שהבנו את כל השלבים הנ"ל והיכולות של פייתון, אפשר לגשת לכתיבת פונקציה "מעטרת" שתשנה משפט מאותיות קטנות לגדולות. נשתמש בטכניקה של הגדרת פונקציה בתוך פונקציה. נקרא לפנימית "פונקצית מעטפת".

```
def uppercase_decorator(function):  
    def wrapper():  
        func = function()  
        make_uppercase = func.upper()  
        return make_uppercase  
  
    return wrapper
```

הפונקציה המעטרת (uppercase_decorator) מקבלת פונקציה כפרמטר. בפונקצית המעטפת (שם מוסיפים את הפונקציונליות) יוצרים משתנה func שמקבל את ההחזר של הפונקציה שהתקבלה כפרמטר function. מפעילים את פונקצית הבילט אין upper על המשתנה, והערך המתקבל לתוך המשתנה make_uppercase. הפונקציה המעטרת מחזירה את פונקצית המעטפת (שהיא מחזירה את המחרוזת כאותיות גדולות).

הפעלה אפשרית של 'קישוט' פונקציה שמחזירה מחרוזת כך:

```
def say_hi():  
    return 'hello world'  
  
decorate = uppercase_decorator(say_hi)  
decorate()
```

הפונקציה אותה "מעטרים" (או מוסיפים לה פונקציונליות) היא: say_hi שמחזירה מחרוזת. הפונקציונליות שמוסיפים לה זה הפיכת המחרוזת לאותיות גדולות, ולכן מה שנקבל כהחזר יהיה:

HELLO WORLD

השתמשנו בפונקציה המעטרת שכתבנו קודם: upper_decorator (ראו קודם)

יש בפייתון דרך קלה יותר להפעיל את הפונקציה המעטרת לאחר שכתבנו אותה או אחרי שייבאנו אותה כ module בעזרת import על ידי תו השטרודל:

```
@uppercase_decorator
def say_hi():
    return 'hello world'
```

```
say_hi()
```

הקוד הזה שקול בדיוק לקוד שכתבנו קודם. כן, התו @ מסמל את העיטור כשהוא מוצמד לפונקציה המעטרת.

אז אם נשים הכל ביחד זה יראה כך:

```
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase
    return wrapper
```

```
@uppercase_decorator
def say_hi():
    return 'hello world'
```

```
say_hi() ==> HELLO WORLD
```

אפשר לקשט פונקציה עם מספר פונקציות כאשר הן מתבצעות מלמטה למעלה, למשל נגדיר עוד פונקציה שמפצלת מחרוזת למילים ומחזירה רשימה עם המילים הללו.

```
def split_string(function):  
    def wrapper():  
        func = function()  
        splitted_string = func.split()  
        return splitted_string  
    return wrapper
```

וכעת נקשט בעזרתה את מה שבנינו עד עכשיו כך:

```
@split_string  
@uppercase_decorator  
def say_hi():  
    return 'hello world'  
  
temp = say_hi()  
print(temp)      ==> ['HELLO', 'WORLD']
```

נסו להחזיר משפט אחר מהפונקציה `say_hi` לבדוק עם זה עובד באופן כללי. שימו לב שהסדר שלהן משנה כי אי אפשר להפעיל `upper` על רשימה, רק על מחרוזת.

כמובן שהמנגנון עובד גם עם פונקציות שדורשות פרמטרים. הפרמטרים יוגדרו בפונקצית המעטפת ויועברו לפונקציה המעוטרת בזמן הקריאה שלה.

בדוגמה הבאה נראה תוכנית מעטרת שמוסיפה שתי פקודות (המודגשות) לפונקציה קיימת עם פרמטרים:

```
def decorator_with_arguments(function):  
    def wrapper_accepting_arguments(arg1, arg2):  
        str1 = arg1*5 + ' ' + arg2*3  
        print(str1)  
    function(arg1, arg2)  
    return wrapper_accepting_arguments
```

```
@decorator_with_arguments  
def cities(city_one, city_two):  
    print("Cities I love are {0} and {1}".format(city_one,  
                                                city_two))  
  
cities("Paris", "London")
```

והנה דוגמה מהעולם האמיתי.

נניח שיש לנו פונקציה שמוחקת משתמש ממערכת המחשב שנראית כך:

```
def delete_user(user_id):  
    # Logic to delete a user  
    return f"User {user_id} has been deleted."
```

לא נתעכב על האלגוריתם של המחיקה עצמה, אבל אנחנו רוצים שרק משתמש שהוא מנהל מיחשוב (admin) יוכל להשתמש בה, מבלי לשנות אותה.

אפשר לבנות פונקציה מעטרת כך:


```
def admin_required(func):
    def wrapper(*args, **kwargs):
        if user_is_admin():
            return func(*args, **kwargs)
        else:
            raise PermissionError("Permission
                                   denied")
    return wrapper
```

ואז נוסיף לקוד המקורי את `@admin_required` כך:

```
@admin_required
def delete_user(user_id):
    # Logic to delete a user
    return f"User {user_id} has been deleted."
```

ואז במידה והמשתמש לא יזוהה כ `admin` התנאי :

```
if user_is_admin()
```

לא יתקיים והפונקציה המעטרת הוסיפה את הפונקציונליות של הרמת דגל של שגיאה על העדר הרשאה אם המשתמש אינו מנהל נתונים.

לסיכום אפשר להגיד ש:

Basically, a decorator takes in a function, adds some functionality and returns that function.

דוגמה אחרונה פשוטה למי שעדיין לא בטוח מה קורה כאן:

```
def make_pretty(func):
    def inner()
        print("I got decorated")
        func()
    return inner
```

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

```
ordinary()
```

prints:

```
I got decorated  
I am ordinary
```

הפונקציה `ordinary` מקבלת קישוט על ידי `.make_pretty`.
`make_pretty` מקבלת את `ordinary` כפרמטר, בעזרת הפונקציה
הפנימית `inner`, מוסיפה הדפסה של שורה שגם מפעילה אותה על ידי
הפקודה `func()` כי זה מייצג את `ordinary` ובסוף מחזירה את `inner`
שעכשיו מכילה גם את ההדפסה של השורה הנוספת וגם את
הפונקציות המקוריות של `ordinary`.