# Decorator

בשיעורים הקודמים ראינו שפונקציות בפייתון הן אובייקטים ממעלה ראשונה- אפשר להעביר אותן כארגומנט לפונקציה ,להפעיל אותן בתוך הפונקציה שקיבלה אותה, לשמור במשתנה, ולהחזיר אותן מפונקציה:

def high\_ord\_func(x: int , func: func) -> int:

    return func(x)+x

func = lambda a : a\*\*2

print(high\_ord\_func(2, func))

print(type(func))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
6  
<class 'function'>

עוד משהו שניתן לעשות , משום שפונקציות הן אובייקט בפני עצמו ניתן להגדיר פונקציה בתוך פונקציה אחרת:

def out\_f():

    def in\_f():

        print("I'm in the inner function")

    return in\_f

f = out\_f()

f()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
I'm in the inner function

עכשיו כשיישרנו קו נראה את הדוגמא הבאה:

def outer\_function(func: func):

    def wrapper():

        print("started")

        func()

        print("ended")

    return wrapper

def a\_function():

    print("I'm a function")

יצרנו פונקציה שמקבלת כפרמטר פונקציה אחרת ועוטפת אותה ב"פונקציית מעטפת" ומחזירה את הפונקציית מעטפת.  
יש שתי דרכים מקובלות להפעיל את הפונקציה:

outer\_function(a\_function)()  
#or:  
f\_wrapper = outer\_function(a\_function)

f\_wrapper()

מה שעשינו עכשיו הוא הבסיס לתבנית העיצוב קשטן על פונקציות- לקחנו אובייקט קיים והוספנו לו התנהגויות מבלי לשנות אותו.  
בפייתון יש עוד אפשרות להפעיל את הפונקציה לעיל מבלי להכניס אותה לתוך פונקציה אחרת, וזה ע"י הגדרת הפונקציה העוטפת עם @ ושם הפונקציה:

@outer\_function

def a\_function():

    print("I'm a function")

a\_function()  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
started   
I'm a function   
ended

מה שקורה מתחת לפני השטח הוא שלקחנו את a\_function ושמנו לה את הערך של outer\_function(a\_function)   
כלומר : a\_function= outer\_function(a\_function), וכל זה מבלי לבצע את ההשמה בפועל.   
ובכך הרחבנו את ההנהגות של הפונקציה מבלי לשנות את ערכה המקורי.  
  
בעיה שיכולה לעלות היא מה קורה אם הפונקציה המקורית מקבלת ארגומנטים? אם ננסה להפעיל למשל את outer\_function על פונקציה שיש לה פרמטרים האם נקבל שגיאה? והתשובה היא ככל הנראה, כי כאשר אנחנו מפעילים את decoration על פונקציה אנחנו משנים את המצביע של הפונקציה (השם שלה) לפונקציית המעטפת שמתקבלת, ואם פונקציית המעטפת לא מקבלת פרמטרים אז היא תזרוק שגיאה,

@outer\_function

def param\_function(num: int):

    print(f"{num} is a number")

param\_function(3)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**---------------------------------------------------------------------------**   
**TypeError**   
Traceback (most recent call last)  **<ipython-input-32-380132969f8b>** in <module>   
 2 **def** param\_function**(**num**:** int**):**   
 3 print**(f"{num} is a number")**   
**----> 4** param\_function**(3)**  **TypeError**: wrapper() takes 0 positional arguments but 1 was given

כדי לתקן את זה נצטרך לשנות את הפונקציה העוטפת.  
משום שהיא זו שמוחזרת מהפונקציה אנחנו רוצים שגם לה יהיה את האפשרות לקבל פרמטרים.  
אבל אנחנו לא יודעים כמה ארגומנטים , אם בכלל, הפונקציה אמורה לקבל, אז איך נגדיר פונקציית מעטפת שאמורה לקבל פרמטרים אם אנחנו לא יודעים כמה ארגומנטים הפונקציה שאותה היא עוטפת אמורה לקבל?   
אז זהו שיש כלי שימושי מאוד שגם אותו ראינו בשיעורים הקודמים, והוא אופרטור כוכבית, או בשמו המקצועי \*args \*\*kwargs , תזכורת: הפרמטר '\*' מאפשר לנו להגדיר בחתימת הפונקציה מספר בלתי מוגבל של פרמטרים שיכנסו ל-tuple והפרמטר '\*\*' עוטף את הארגומנטים במילון בתנאי שהם מתקבלים בצורה של מפתח וערך:

def my\_sum(\*args):

    result = 0

    for x in args:

        result += x

    return result

list1 = [1, 2, 3, 4, 5]

print(my\_sum(1,2,3,4,5))  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

15  
  
def f(\*\*kwargs):

    print(kwargs)

    print(type(kwargs))

    for key, val in kwargs.items():

        print(key, '->', val)

f(e=1, r=2,j=3)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{'e': 1, 'r': 2, 'j': 3}  
<class 'dict'>  
e -> 1  
r -> 2  
j -> 3

אז אם נגדיר את חתימת הפונקציה של הפונקציה העוטפת עם \*args \*\*kwargs , ניתן לפונקציה אפשרות לקבל מספר בלתי מוגבל של ארגומנטים, כך הפונקציה תוכל לקבל לתוכה כל סוג של אובייקט אפשרי:

def outer\_function(func: func):

    def wrapper(\*args , \*\*kwargs):

        print("started")

        func(\*args, \*\*kwargs)

        print("ended")

    return wrapper

@outer\_function

def param\_function(num: int):

    print(f"{num} is a number")

param\_function(3)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
started   
3 is a number   
ended

לפעמים הפונקציה שנרצה לעטוף מחזירה ערך, גם כאן נצטרך לשנות את הפונקציה העוטפת כדי להתאימה לפונקציה הנעטפת:

def new\_outer\_function(func):

    def wrapper(\*args , \*\*kwargs):

        print("Enter your name with no capital letters:")

        return\_val = func(\*args, \*\*kwargs)

        #do somthing

        return return\_val

    return wrapper

@new\_outer\_function

def change\_name\_to\_upper():

    """Changes the first character of the name to upper case """

    first\_name= input("Your first name is: ")

    last\_name= input("Your last name is: ")

    fname\_list=list(first\_name)

    lname\_list=list(last\_name)

    fname\_list[0]= fname\_list[0].upper()

    lname\_list[0]= lname\_list[0].upper()

    return "".join(fname\_list)+" "+"".join(lname\_list)

print(change\_name\_to\_upper())  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Enter your name with no capital letters:  
Your first name is:  
tom  
Your last name is:   
pythonovitch  
Tom Pythonovitch

שני קשטנים לאותה פונקציה-

לפעמים נדרש להפעיל שני decorators לפונקציה מסויימת.   
נסתכל בדוגמא הבאה, נניח יש לנו פונקציית logging, כלומר פונקציית שכותבת לנו קבצי log, קבצי log הם קבצים שמשמים לצורך תיעוד של פעולות של פונקציות, מחלקות תכניות וכו':

def my\_logger(orig\_func):

    import logging

    logging.basicConfig(filename='{}.log'.format(orig\_func.\_\_name\_\_), level=logging.INFO)

    def wrapper(\*args, \*\*kwargs):

        logging.info(

            'Ran with args: {}, and kwargs: {}'.format(args, kwargs))

        return orig\_func(\*args, \*\*kwargs)

    return wrapper

הפונקציה מייצרת קובץ log בשם של הפונקציה אותה היא קיבלה כפרמטר, וכותבת לתוך הקובץ את הארגומנטים שנשלחו לפונקציה.   
  
חוץ ממנה נניח שיש לנו פונקציה שבודקת כמה זמן רצה פונקציה מסוימת:

def my\_timer(orig\_func):

    import time

    def wrapper(\*args, \*\*kwargs):

        t1 = time.time()

        result = orig\_func(\*args, \*\*kwargs)

        t2 = time.time() - t1

        print(f'{ orig\_func.\_\_name\_\_} ran in: { t2} sec')

        return result

    return wrapper

אם נרצה להפעיל את שתי הפונקציות כקשטן לפונקציה נוכל לכתוב אותם אחד אחרי השני מעל הפונקציה שאותה אנחנו רוצים לקשט :

@my\_timer

@my\_logger

def display\_info(name, age):

    print('display\_info ran with arguments ({}, {})'.format(name, age))

display\_info('Tom', 30)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
display\_info ran with arguments (Tom, 30)   
wrapper ran in: 0.0012555122375488 sec

הבעיה שקיבלנו תוצאה שלא רצינו.  
 ציפינו שהפונקציה time תפעל על הפונקציה המקורית ולא על ה-wrapper .  
מה שקורה מאחורי הקלעים זה שהפונקציה logger עוטפת את הפונקציה המקורית ואז הtimer עוטף את הפונקציה שמתקבלת מהקשטן של הlogger:

display\_info = my\_logger(display\_info)  
#display\_info is the logger wrapper now  
display\_info = my\_timer(display\_info)  
#display\_info is timer wrapper on the logger wrapper

ואם נשנה את הסדר של ה-decorators ,כלומר נבצע את הtimer ואז את ה-logger (*הערה:* אם מריצים מהג'ופיטר צריך לצאת ממנו ולהריץ מחדש, אחרת הוא ישתמש בקובץ log שכבר פתוח).  
נקבל שנוצר לנו קובץ .log חדש בשם wrapper ,ולא כשם הפונקציה שרצינו להפעיל עליה את הקשטן.  
אז איך אפשר לפתור את זה?   
מתי שמפעילים כמה decorators מומלץ תמיד לשמר את הפונקציה המקורית שעליה הם פועלים, איך עושים את זה? ע"י יבוא של מודול מיוחד כמובן: from functool import wraps .  
ואז בכל מקום שיש לנו פונקציית wrapper נוסיף נעטוף אותה בקשטן @wraps() ובסוגריים את הפונקציה שאותה אנחנו מנסים לקשט.   
מה שאמור לקרות זה שהפונקציה תקשט את הקשטן כך שישמר את שם הפונקציה שאותה הוא מקשט, וכך בכל פעם שנפעיל עליו "קשטן מקושט" הפונקציה המקורית תיהיה פתוחה לעוד קשטנים:

# Decorators

from functools import wraps

def my\_logger(orig\_func):

    import logging

    logging.basicConfig(filename='{}.log'.format(orig\_func.\_\_name\_\_), level=logging.INFO)

**@wraps(orig\_func)** # <---------- we added this

    def wrapper(\*args, \*\*kwargs):

        logging.info(

            'Ran with args: {}, and kwargs: {}'.format(args, kwargs))

        return orig\_func(\*args, \*\*kwargs)

    return wrapper

def my\_timer(orig\_func):

    import time

**@wraps(orig\_func)** # <---------- we added this  
    def wrapper(\*args, \*\*kwargs):

        t1 = time.time()

        result = orig\_func(\*args, \*\*kwargs)

        t2 = time.time() - t1

        print('{} ran in: {} sec'.format(orig\_func.\_\_name\_\_, t2))

        return result

    return wrapper

@my\_logger

@my\_timer

def display\_info(name, age):

    print('display\_info ran with arguments ({}, {})'.format(name, age))

display\_info('Tom Pythonovitch', 50)

ושוב: אם מריצים בג'ופיטר , אם הפעלתם את הקודים הקודמים, יש להפעיל את החלק הזה בטעינה מחודשת של הקובץ ipynb כי אז הוא יכתוב לקובץ .log האחרון שהוא פתח.