# Decorators

July 14, 2022

Python Function Decorators

Decorators are a metaprogramming . Metaprogramming is about creating functions and classes whose main goal is to manipulate code (e.g., modifying, generating, or wrapping existing code).

[4]:
```
#three functions, which takes two numbers and process them
def add(x,y):
    return x + y
def multiple(x,y):
    return x * y
def raise_to(x,y):
    return x ** y
```

[5]:
```
add(10,20)
```

[5]:
```
30
```

[6]:
```
add(10,"JLP")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-3856d00c8c7b> in <module>
----> 1 add(10,"JLP")

<ipython-input-4-de9db409a36e> in add(x, y)
      1 #three functions, which takes two numbers and process them
      2 def add(x,y):
----> 3     return x + y
      4 def multiple(x,y):
      5     return x * y

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[14]:
```
def add(x,y):
    if type(x ) == int and type(y) == int:
        return x + y
    else:
        print("Invalid arguments")
```

```
add(10, "JLP")
add(10, 20)
```

```
Invalid arguments
```

[14]: 30

## 0.1 Decorator Function

[28]:
```
def ensure_int(f):
    def wrapper(x,y):
        if type(x) == int and type(y) == int:
            return f(x,y)
        else:
            print("Invalid arguments")
    return wrapper
```

### 0.1.1 Decorating Function

[40]:
```
#Applying Decorator

def add(x,y):
    return x + y

add = ensure_int(add) #passing our function to decorator function as arugument

add(10,'JLP')
add(10,20)
```

```
Invalid arguments
```

[40]: 30

## 0.2 Pie syntax or Syntactic Sugar

[41]:
```
##Piesyntax or Syntactic Sugar

@ensure_int
def multiple(x,y):
    return x * y

@ensure_int
def raise_to(x,y):
    return x**y
```

[45]:
```
# calling add function
```

```
#with invalid inputs
add(10,'JLP')

#with valid inputs
add(10,20)
```

```
Invalid arguments
```

[45]:
```
12
```

[47]:
```
#Calling Multiple Function

# with invalid inputs
multiple(10,'JLP')

#with valid inputs
multiple(10,2)
```

```
Invalid arguments
```

[47]:
```
20
```

[46]:
```
#Calling Raise to function

# with invalid inputs
raise_to(4,'JLP')

#with valid inputs
raise_to(4,3)
```

```
Invalid arguments
```

[46]:
```
64
```

### 0.3 Decorating Functions that Takes Arguments

[48]:
```
def decorate_it(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
```

[49]:
```
@decorate_it
def hello(name):
    print("Hello", name)
hello("Bob")
```

```
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-49-ffbf59cb3166> in <module>
      2 def hello(name):
      3     print("Hello", name)
----> 4 hello("Bob")

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

[30]:
```
#Usage of Variable length aruguments

def decorate_it(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        func(*args, **kwargs)
        print("After function call")
    return wrapper
```

[31]:
```
@decorate_it
def hello(name,age):
    print("Hello")
hello("Bob")
```

```
Before function call
Hello Bob 27
After function call
```

### 0.4 Returning Values from Decorated Functions :

[4]:
```
@decorate_it
def hello(name):
    return "Hello " + name
result = hello("Bob")
print(result)
```

```
Before function call
After function call
None
```

[5]:
```
#Decorator Retruning Values
def decorate_it(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        result = func(*args, **kwargs)
        print("After function call")
        return result
    return wrapper
```

[6]:
```
@decorate_it
def hello(name):
    return "Hello " + name
result = hello("Bob")
print(result)
```

```
Before function call
After function call
Hello Bob
```

### 0.5 Preserving Function Metadata :

When we create function each object has its own metadata, like **name**, **doc** etc.

Metadata is used by functions like help( ) or dir( )

[7]:
```
@decorate_it

def hello( ):
    '''function that greets'''          #Metadata_Docstring
    print("Hello world")

print(hello.__name__)
print(hello.__doc__)
print(hello)
```

```
wrapper
None
<function decorate_it.<locals>.wrapper at 0x007BD468>
```

[8]:
```
from functools import wraps
def decorate_it(func):
    @wraps(func)      #apply the @wraps decorator
    def wrapper( ):
        print("Before function call")
        func( )
        print("After function call")
    return wrapper
```

[9]:
```
@decorate_it

def hello( ):
    '''function that greets'''          #Metadata_Docstring
    print("Hello world")

print(hello.__name__)
print(hello.__doc__)
print(hello)
```

```
hello
function that greets
<function hello at 0x067717C8>
```

### 0.6 Unwrapping a Decorator :

[10]:
```
original_hello = hello.__wrapped__
original_hello( )
```

```
Hello world
```

### 0.7 Nesting Decorators :

[12]:
```
from functools import wraps

def double_it(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * 2
    return wrapper

def square_it(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * result
    return wrapper
```

[13]:
```
@double_it
@square_it

def add(a,b):
    return a + b #5 --> squared - 25 --> Double -- 50

print(add(2,3))
```

```
50
```

[14]:
```
@square_it
@double_it

def add(a,b):
    return a + b

print(add(2,3))
```

```
100
```

### 0.8 Applying Decorators to Built-in Functions :

[15]:
```
double_the_sum = double_it(sum)
print(double_the_sum([1,2]))
```

6

# 1 Real World Examples :

**You'll notice that they'll mainly follow the same pattern that you've learned so far :**

[16]:
```
from functools import wraps

def decorator(func):
    @wraps(func)                                    #To Preserve Metadata
    def wrapper_decorator(*args, **kwargs):         #For variable aruguments
        # Do something before                       #Add on To Original Function
        value = func(*args, **kwargs)               #To retrun value from decorator␣
 ↪function
        # Do something after
        return value
    return wrapper_decorator
```

## 1.1 Debugger :

**Let's create a @debug decorator that will do the following, whenever the function is called**

**Print the function's name**

**Print the values of its arguments**

**Run the function with the arguments**

**Print the result**

**Return the modified function for use**

[17]:
```
from functools import wraps

def debug(func):
    @wraps(func)

    def wrapper(*args, **kwargs):
        print('Running function:', func.__name__)
        print('Positional arguments:', args)
        print('keyword arguments:', kwargs)
        result = func(*args, **kwargs)
        print('Result:', result)
```

```
        return result

    return wrapper
```

[18]:
```
@debug

def hello(name):
    return "Hello " + name
hello("Bob")
```

```
Running function: hello
Positional arguments: ('Bob',)
keyword arguments: {}
Result: Hello Bob
```

[18]:
```
'Hello Bob'
```

[19]:
```
#You can also apply this decorator to any built-in function like this :

sum = debug(sum)
sum([1, 2, 3])
```

```
Running function: sum
Positional arguments: ([1, 2, 3],)
keyword arguments: {}
Result: 6
```

[19]:
```
6
```

## 2 Timer :

**The following @timer decorator reports the execution time of a function. It will do the following:**

**Store the time just before the function execution (Start Time)**

**Run the function**

**Store the time just after the function execution (End Time)**

**Print the difference between two time intervals**

**Return the modified function for use**

[2]:
```
import time
from functools import wraps

def timer(func):
    @wraps(func)
```

```
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print("Finished in {:.3f} secs".format(end-start))
        return result

    return wrapper
```

[3]:
```
@timer
def countdown(n):
    while n > 0:
        n -= 1
countdown(1000)
countdown(1000000)
```

```
Finished in 0.000 secs
Finished in 0.137 secs
```

### 2.1 Counter :

**An decorator to count how many times the function is called.**

[32]:
```
def count(func):
    def wrapper(*args, **kwargs):
        wrapper.counter += 1
        return func(*args,**kwargs)
    wrapper.counter = 0
    return wrapper
```

**This function don't have any implementation, this example is to demonstrate, how variable number of arguments are handled in decorator functions.**

[34]:
```
@count
def my_func(*args,**kwargs):
    pass

# call with multiple integer values
my_func(1,2,3,4)

# call with a string
my_func("1,2,3,4")

# call with an integer
my_func(2)

# check the counter
my_func.counter
```

[34]: 3

Decorators in Python ensures that your code is DRY(Don't Repeat Yourself).

Decorators have several use cases such as:

Authorization in Python frameworks such as Flask and Django

Logging

Run time check

Synchronization

Type checking

Debugging

Python Generators

<b>A Python generator is a kind of an iterable, like a Python list or a python tuple. It genera

[2]:
```
#square() function that squares an input list of numbers
def square(numbers):
    result = []
    for n in numbers:
        result.append(n ** 2)
    return result

numbers = [1, 2, 3, 4, 5]
squared_numbers = square(numbers)

print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

[3]:
```
#Let's turn this function into a generator.
#Instead of storing the squared numbers into a list, you can yield values one␣
 ↪at a time without storing them
def square(numbers):
    for n in numbers:
        yield n ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = square(numbers)
# It returns an object but does not start execution immediately.
print(squared_numbers)
```

```
<generator object square at 0x000001F4665C8660>
```

[4]:
```
#A generator object doesn't hold numbers in memory.
#Instead, it computes and yields one result at a time.
#It does this only when you ask for the next value using the next() function
```

```
print(next(squared_numbers))
print(next(squared_numbers))
print(next(squared_numbers))
print(next(squared_numbers))
```

```
1
4
9
16
```

[5]:
```
# Once the function yields, the function is paused and the control is␣
↪transferred to the caller.
# Local variables and theirs states are remembered between successive calls.
print(next(squared_numbers))
```

```
25
```

[6]:
```
# Finally, when the function terminates, StopIteration is raised automatically␣
↪on further calls.
print(next(squared_numbers))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-6-ee4db9c24dc2> in <module>
----> 1 print(next(squared_numbers))

StopIteration:
```

[9]:
```
#In reality, you don't need to call the next() function.
#Instead, you can use a for loop
def square(numbers):
    for n in numbers:
        yield n ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = square(numbers)

for n in squared_numbers:
    print(n)
```

```
1
4
9
16
25
```

[10]:
```python
#Generators vs. Lists-Runtime Comparison
import random
import timeit
from math import floor

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def data_list(n):
    result = []
    for i in range(n):
        result.append(random.choice(numbers))
    return result

def data_generator(n):
    for i in range(n):
        yield random.choice(numbers)

t_list_start = timeit.default_timer()
rand_list = data_list(1_000_000)
t_list_end = timeit.default_timer()

t_gen_start = timeit.default_timer()
rand_gen = data_generator(1_000_000)
t_gen_end = timeit.default_timer()

t_gen = t_gen_end - t_gen_start
t_list = t_list_end - t_list_start

print(f"List creation took {t_list} Seconds")
print(f"Generator creation took {t_gen} Seconds")

print(f"The generator is {floor(t_list / t_gen)} times faster")
```

```
List creation took 0.5300700000007055 Seconds
Generator creation took 3.840000135824084e-05 Seconds
The generator is 13803 times faster
```

This shows how a generator is way faster to create. This is because when you create a list, all the numbers have to be stored in memory. But when you use a generator, the numbers aren't stored anywhere, so it's lightning-fast.

[11]:
```python
#Generator Expression
squared_numbers = (n ** 2 for n in [1, 2, 3, 4, 5])
print(squared_numbers)
```

```
<generator object <genexpr> at 0x000001F466682970>
```

[12]:
```python
for n in squared_numbers:
    print(n)
```

```
1
4
9
16
25
```

[27]:
```python
#yeild vs Return
def mygenerator():
    print('First item')
    yield 10

    print('Second item')
    yield 20

    print('Last item')
    yield 30

gen = mygenerator()
print(next(gen))
print(next(gen))
print(next(gen))
```

```
First item
10
Second item
20
Last item
30
```

[28]:
```python
def mygenerator():
    print('First item')
    yield 10

    return

    print('Second item')
    yield 20

    print('Last item')
    yield 30

gen = mygenerator()
print(next(gen))
print(next(gen))
```

```
First item
10
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-28-e715d8959665> in <module>
     13 gen = mygenerator()
     14 print(next(gen))
---> 15 print(next(gen))

StopIteration:
```

### 2.1.1 To - DO :

Write a Python program to make a chain of function decorators (bold, italic, underline etc.).

Make a decorator which calls a given function twice. You can assume the functions don't return anything important, but they may take arguments

Imagine you have a list called books, which several functions in your application interact with. Write a decorator which causes your functions to run only if books is not empty

Write a generator which yeilds infinite fibonaci series

Write a generator which yeilds infinite prime numbers