Python Decorators

Decorators are a metaprogramming *feature that lets you modify the language itself. A decorator is basically a wrap function. It takes a function as an argument and returns a replacement function. In general, a decorator is used to modify the behavior of function or classes and perform some additional actions on it.*

Even though it is the same underlying concept, we have two different kinds of decorators in Python:

1. *Function decorators*
2. *Class decorators*

Features of decorator in python :

1. *Decorator act as a wrapper means that when we pack an actual gift using a wrapper, the gift is present inside the wrapper all we are providing additional decoration of a gift. Similarly, when we use decorators in python programming we are providing additional functionality to our original function without any alteration.*

2. *Decorator help in exception handlings such as divide by zero scenarios, login password checking scenario, and many more*

Key reasons to use decorators are :

1. Improve code readability

2. Extend functionality of a function

Decorators in the real world :

Think of a gun which is wrapped with a silencer. The silencer modifies the behavior of the gun and performs some additional action like suppressing the noise. The silencer extends the functionality of gun without modifying it. It makes it easy to turn on and turn off the additional features.

Simple Decorator in Python :

Decorators works on four concepts (Python Functions as First Class Objects):

- ✓ *Nested function : one function inside another function and so on, as many we want.*
- ✓ *Python functions are capable of returning another function as output.*
- ✓ *The function can be assigned to a variable*
- ✓ *Python functions can also be used as an input parameter to the function*

syntax:

```
@my_decorator
def my_function()
```

Here my_decorator is decorator applied to my_function,a special symbol '@' is used to apply decorator to function.

@my_decorator is also called as syntantic sugar,which is same as

```
my_function = my_decorator(my_function)
```

Let's look at an example,

Below we have 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
```

When we call add() function.

```
add(10,20)
output:
30
```

The output is as expected.What if the user doesn't provide numbers

```
add(10, 'JLP')

output:

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

We get a ***TypeError exception.****So its becomes user responsibility to provide correct values. One way of fixing is using* ***try-exception block*** *, Another way of ensuring type is, by checking type of inputs before processing in function.*

```
def add(x,y):

    if type(x ) == int and type(y) == int:

        return x + y
    else:

        print("Invalid arguments")

add(10, "JLP")

output:

Invalid arguments

add(10, 20)

output:

30
```

But these two approach are not scalable.

Suppose you have 200 such functions in your code base,and its hell lot of work to modify each function to put try-except or type check code.

That's where decorators come to our rescue.

Let's see how decorators make our life easy.

A decorator function, which validates,input parameters as integers

```
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
```

*Here **ensure_int(f)** is decorator function, which only takes callables, in this case, function **'f'**.It creates **wrapper( ) function**, which takes **same arguments** as function 'f'. It validates the input parameters as integer and calls the **original function 'f'**, if parameters are integers,otherwise it prints a message.After this processing, ensure_int **returns wrapper** function.*

Decorating function :

Now let's use this decorator on a function We will go back to, three functions which takes two numbers and process them

Ex 1:

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

We will check, how these functions works,after applying decorator.

```
add = ensure_int(add)
add(10,'JLP')
Output : Invalid arguments
add(10,20)
Output : 30
```

Works as expected right ? that's the power of decorators.

Pie syntax or Syntactic Sugar :

The way we decorated add( ) above is a little clunky. First of all, you end up typing the name add three times. In addition, the decoration gets a bit hidden away below the definition of the function.

Instead, we use decorators in a simpler way with the @ symbol, sometimes called the ***"pie" syntax or Syntactic Sugar***

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

```
# call add with invalid inputs
add(10,'JLP')       #Output : Invalid arguments
# call add with valid inputs
multiple(10,20)     #Output : 30

# call multiple with invalid inputs
multiple(10,'JLP')      #Output : Invalid arguments
# call multiple with valid inputs
multiple(10,2)      #Output : 20

# call raise_to with invalid inputs
raise_to(4,'JLP')       #Output : Invalid arguments
# call raise_to with valid inputs
raise_to(4,3)           #Output : 64
```

How decorator works :

@ is special symbol denotes a decorator.Whenever Python iterpreter sees @ symbol it identifies that a decorator is being applied, and

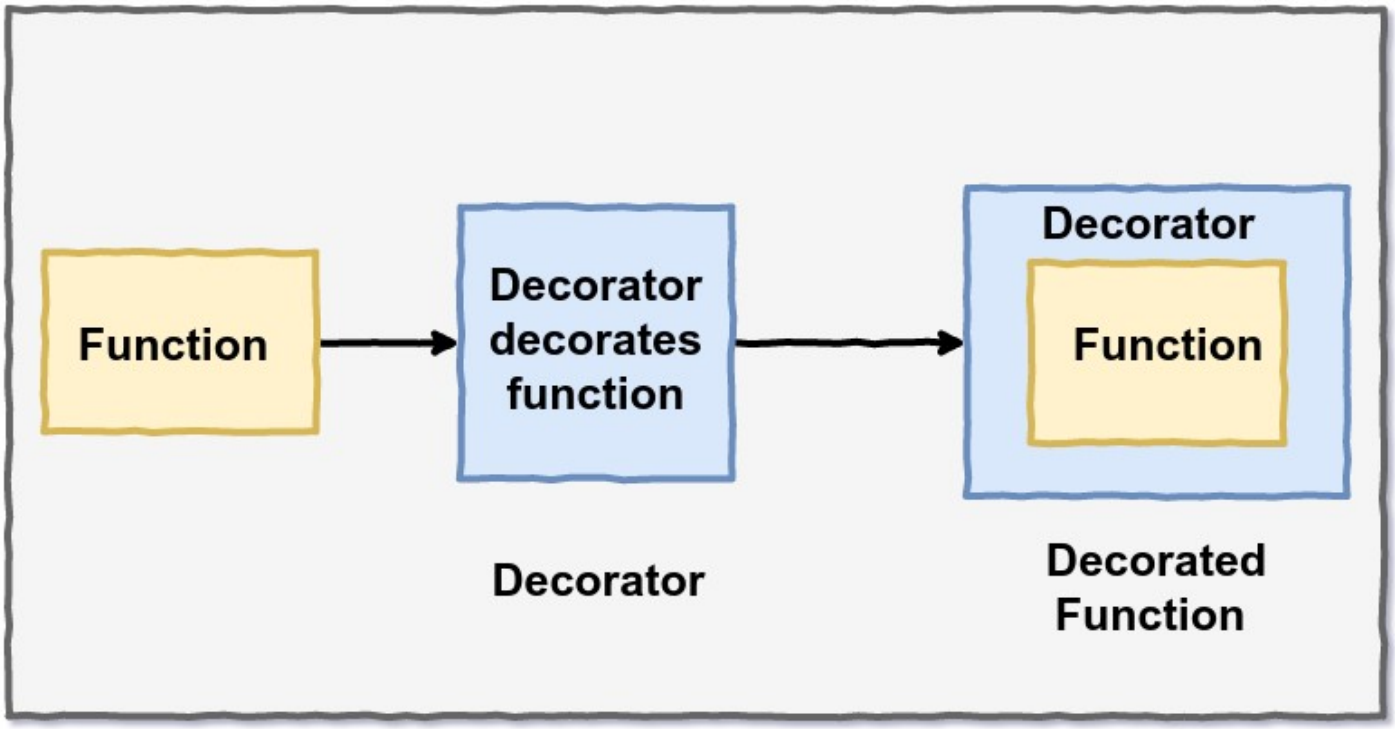
step 1: it compiles add( ) / multiple( ) / raise_to( ) in to a function object.

step 2: Python then passes compiled function object to def ensure_int(f), By design ensure_int takes and return function.

step 3: Python binds the return value to original function name

Whatever function you pass to decorater, you get a new function that includes the extra statements that decorater adds. A decorator doesn't actually have to run any code from func, but decorater calls func part way through so that you get the results of func as well as all the extras.

Simply put, a decorator is a function that takes a function as input, modifies its behavior and returns it



Decorating Functions that Takes Arguments :

Let's say you have a function hello( ) that accepts an argument and you want to decorate it.

```
def decorate_it(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
@decorate_it
def hello(name):
    print("Hello", name)
hello("Bob")    # Prints _wrapper() takes 0 positional arguments but 1 was given
```

Unfortunately, running this code raises an error. Because, the inner function wrapper() does not take any arguments, but we passed one argument.

*The solution is to include ***args and **kwargs** in the inner wrapper function. The use of *args and **kwargs is there to make sure that any number of input arguments can be accepted.*

Let's rewrite the above example.

```
def decorate_it(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        func(*args, **kwargs)
        print("After function call")
    return wrapper
@decorate_it
def hello(name):
    print("Hello", name)
hello("Bob")
# Prints Before function call    # Prints Hello Bob        # Prints After function call
```

Returning Values from Decorated Functions :

What if the function you are decorating returns a value? Let's try that quickly on above function

```
@decorate_it
def hello(name):
    return "Hello " + name
result = hello("Bob")
print(result)
# Prints Before function call
# Prints After function call
# Prints None
```

Because the decorate_it( ) doesn't explicitly return a value, the call hello("Bob") ended up returning None.

To fix this, you need to make sure the wrapper function returns the return value of the inner function.

Let's rewrite the above example.

```
def decorate_it(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        result = func(*args, **kwargs)
        print("After function call")
        return result
    return wrapper
@decorate_it
def hello(name):
    return "Hello " + name
result = hello("Bob")
print(result)
# Prints Before function call    # Prints After function call    # Prints Hello Bob
```

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( )

Copying decorator metadata is an important part of writing decorators.

When you apply a decorator to a function, important metadata such as the name, doc string, annotations, and calling signature are lost.

For example, the metadata in our example would look like this :

```
@decorate_it
def hello( ):
    '''function that greets'''      #Metadata_Docstring
    print("Hello world")
print(hello.__name__)
# Prints wrapper
print(hello.__doc__)
# Prints None
print(hello)      # Prints <function decorate_it.<locals>.wrapper at 0x02E15078>
```

*To fix this, apply the **@wraps** decorator from the functools library to the underlying wrapper function*

Technical Detail:

*The **@functools.wraps** decorator uses the function **functools.update_wrapper( )** to update special attributes like __name__ and __doc__ that are used in the introspection.*

```
from functools import wraps
def decorate_it(func):
    @wraps(func)
    def wrapper( ):
        print("Before function call")
        func( )
        print("After function call")
    return wrapper
@decorate_it
def hello( ):
    '''function that greets'''
    print("Hello world")
print(hello.__name__)
# Prints hello
print(hello.__doc__)
# Prints function that greets
print(hello)
# Prints <function hello at 0x02DC5BB8>
```

This can also be fixed by assigning wrapper attributes, with hello( ) function attributes but code looks ugly , elegant way of doing it is only by using functools wraps() decorator Whenever you define a decorator, do not forget to use @wraps, otherwise the decorated function will lose all sorts of useful information

Unwrapping a Decorator :

Even if you've applied a decorator to a function, you sometimes need to gain access to the original unwrapped function, especially for debugging or introspection.
Assuming that the decorator has been implemented using @wraps, you can usually gain access to the original function by accessing the __wrapped__ attribute.

Let's see for above example :

```
original_hello = hello.__wrapped__
original_hello( )
# Prints Hello world
```

Nesting Decorators :

You can have more than one decorator for a function. To demonstrate this let's write two decorators:

1. *double_it() that doubles the result*
2. *square_it() that squares the result*

```
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
#You can apply several decorators to a function by stacking them on top of each other.
@double_it
@square_it
def add(a,b):
    return a + b
print(add(2,3))  # Prints 50
```

Here, the addition of 2 and 3 was squared first then doubled. So you got the result 50.

result = ((2+3)^2)*2

= (5^2)*2

= 25*2

= 50

Execution order of decorators :

The decorator closest to the function (just above the def) runs first and then the one above it. The decorators are applied in reverse order,first my_dec1 is applied, and output of my_dec1 is supplied as argument to my_dec2, (further my_dec2 return value is supplied to my_dec3 so on n decortors) The final value is again binded to original function reference

Let's try reversing the decorator order:

```
@square_it
@double_it
def add(a,b):
    return a + b
print(add(2,3))
# Prints 100
```

Here, the addition of 2 and 3 was doubled first then squared. So you got the result 100.

result = ((2+3)*2)^2

= (5*2)^2

= 10^2

= 100

Applying Decorators to Built-in Functions :

You can apply decorators not only to the custom functions but also to the built-in functions.

The following example applies the double_it() decorator to the built-in sum() function.

```
double_the_sum = double_it(sum)
print(double_the_sum([1,2]))
# Prints 6
```

Real World Examples :

Let's look at some real world examples that will give you an idea of how decorators can be used.

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

```
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
```

***Notice Decorator function signature**

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*

```
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
```

Let's apply our @debug decorator to a function and see how this decorator actually works.

```
@debug
def hello(name):
    return "Hello " + name
hello("Bob")
# Prints Running function: hello
# Prints Positional arguments: ('Bob',)
# Prints keyword arguments: {}
# Prints Result: Hello Bob
```

You can also apply this decorator to any built-in function like this :

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

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*

```
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

@timer   #Let's apply this @timer decorator to a function.

def countdown(n):

    while n > 0:

        n -= 1
```

```
countdown(10000) # Prints Finished in 0.005 secs

countdown(1000000) # Prints Finished in 0.178 secs
```

Counter :

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

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

Here count( ) is the decorator function,creates a wrapper around func( ),and it has counter which incremented each time the func() is called.
This function don't have any implementation, this example is to demonstrate, how variable number of arguments are handled in decorator functions.

```
@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
Output : 3
```

Summing it up :

You can now use Python Decorators to make your code cleaner and better in general , here you have the key points.

- ✓ *To define a decorator, you need to write a function that accepts a function as a single parameter. Inside that function, create another function: this is the altered function, and your decorator should return just that. This internal function, the "wrapper" may call the original function and perform additional operations*
- ✓ *Apply a decorator by adding this syntax just above the function you want to decorate: @decorator*
- ✓ *The decorator may accept parameters if that's the case just go with @decorator('1', 'blah')*
- ✓ *You may apply multiple decorators, one per line. However, Python will apply them in bottom-up order.*
- ✓ *They can be reused.*
- ✓ *Decorator's return value replaces the function, i.e., existingFunc.if you want to run the function that we decorated, then make the decorator return it*
- ✓ *Using decorators in Python also ensures that your code is DRY(Don't Repeat Yourself).*
- ✓ *Decorators have several use cases such as:*
    a. *Authorization in Python frameworks such as Flask and Django*
    b. *Logging,*
    c. *Run time check,*
    d. *Synchronization,*
    e. *Type checking*
    f. *Debugging*

Till now we have seen function as decartor,which is most common use of decorator. However class and instance can also be decorators we will look into themin advanced topics To learn more about Python decorators check out [Python's Decorator Library]

----- ♡Happy Pythoning ! -----