# Decorator:-

Decorator syntax, detailed in PEP 318, is a clean way of adding extra functionality to functions by using the "@" symbol. The intent behind the addition is to ensure the function name is referenced to once when defining a function and decorating it — which is, essentially, a single definition — to make code clearer. Aside from syntax, decorators are important as they allow us to re-use certain repetitive pieces of code in each function without having to write the same thing in each one. Common uses for decorators are to log information when a function is called, to perform checks like user authentication or to check the supplied arguments. In any case, it's always handy to move a portion of your code into a decorator if you find that many of your functions require the same functionality and you are tending to repeat yourself in each one.

## Writing a basic decorator

In terms of writing them, decorators could be defined as a function which accepts a function and returns a new one. For our first decorator, we're going to write one which prints when the function was called. Let's start by writing a function:

```
def func():
print('calling func')
```

Now, for the decorator, we said that it must be a function that accepts a function, so that must be in its parameter list. It also must return a new one, so we'll need to define a new function inside the decorator. Let's start with an example and we'll discuss how it works later:

```
def log_calls(func):
def wrapper():
name = func.__name__
print(f'before {name} was called')
func()
print(f'after {name} was called')
return wrapper
```

Let's "decorate" our original function by calling the decorator on it:

```
func = log_calls(func)
```

And when we call our decorated `func`:

```
>>> func()
before func was called
calling func
after func was called
```

As you can see, the code in the `wrapper` function was executed. This is because when we called the decorator, the function returned is actually `wrapper`. The special thing about `wrapper` is that it was defined within the scope of `log_calls`. This means that the argument `func` in the parameter list is accessible to the `wrapper` function, meaning it can be called. This is how decorators work.

## "Syntactic Sugar"

The phrase "syntactic sugar" is something always associated with decorators. As we mentioned at the start, the decorator syntax uses the `@` symbol above function definitions to decorate them. This behaviour is the syntactic sugar because writing `@decorator` above the function definition is the same as calling `f = decorator(f)` after it.

## Passing arguments

Having this behaviour available to us is all well and good, but most functions we write take arguments. Let's write a function that takes arguments and decorate it:

```
@log_calls
def func_with_args(a, b):
    return a + b
```

Now, calling the function:

```
>>> func_with_args(1, 2)
TypeError: wrapper() takes 0 positional arguments but 2
were given
```

An exception was raised and it is clear why: the function `wrapper` we return doesn't accept any arguments, but `func_with_args` does. So how do we solve this problem? Well, all we need to do is simply amend our decorator and let it accept arguments. The best way is to use `*args` and `**kwargs` to ensure that any function with any kind of arguments will work:

```
def log_calls(func):
    def wrapper(*args, **kwargs):
        name = func.__name__
        print(f'before {name} was called')
        func(*args, **kwargs)
        print(f'after {name} was called')
    return wrapper
```

Let's try again:

```
>>> f1(1,2)
before func_with_args was called
after func_with_args was called
```

That works, but we have another problem: our function didn't return its value. This is because the wrapper function calls `func(*args, **kwargs)`, but doesn't set its return value to anything. This is fixed easily by changing this line to `r =`

`func(*args, **kwargs)` and then at the end of `wrapper`, add `return r` to return this value. Making this change, our function works as expected:

```
>>> f1(1,2)
before func_with_args was called
after func_with_args was called
3
```

## A slightly more realistic example

So, now that we know how decorators work and how to write them, let's move onto writing a slightly more practical example (*because knowing when you call your function is **so** useful!*). This time, let's write a function that formats the result of a function into the form `£00.00`. This could be useful in a program that works with money as integers or floats but displays all values in this form. To start, the decorator:

```
def money_format(func):
    def wrapper(*args, **kwargs):
        r = func(*args, **kwargs)
        formatted = '£{:.2f}'.format(r)
        return formatted
    return wrapper
```

And a quick "tax-adder" function:

```
@money_format
def add_tax(value, tax_percentage):
    return value * (1 + (tax_percentage / 100))
```

Let's test our decorated function:

```
>>> add_tax(52, 20)
'£62.40'
>>> add_tax(10, 10)
'£11.00'
```

## Decorators with arguments

A core benefit of using decorators is that they offer better code reusability. Taking this a step further, it is possible to write decorators which accept arguments as well as the function they decorate, making them even more generic.

Let's stick with the idea of formatting function return values into a suitable format, but this time we will create a decorator called `prefix` which, unsurprisingly, prefixes the return value. Decorators with arguments have 3 "levels": the *argument level*, the *function level* and the *wrapper level*. We've already seen the last 2 earlier in the article and each "level" is its own function. Thus, it only makes sense that including all 3 levels requires 3 functions:

1. The outer function is the one that accepts the arguments. It returns a decorator.
2. The middle function is the decorator which is returned by the outer function: it accepts a function and returns the wrapper function to decorate the supplied one.
3. The inner function is the wrapper function. It should usually accept an arbitrary number of arguments which are supplied to the function being decorated. This is where the extra functionality is added.

Let's see an example now of how we would write this more complex decorator:

```
def prefix(value):
    def decorator(func):
        def wrapper(*args, **kwargs):
            r = func(*args, **kwargs)
            return str(value) + str(r)
        return wrapper
    return decorator
```

Now let's create a function to decorate:

```
>>> @prefix('£')
... def generate_bonus():
...     return 250
...
>>> generate_bonus()
'£250'
```

## Digression: Decorators as classes

Although functions are usually used to write decorators, it is possible to write decorators using classes since they can have a `__call__` method which makes them behave like functions anyway. Let's take a look at an example of the `money_format` decorator we wrote earlier as a class:

```
class MoneyFormat:
    def __init__(self, func):
        self.func = function    def
__call__(self, *args, **kwargs):
        r = self.func(*args, **kwargs)
        return '£{:.2f}'.format(r)
```

As you can see, the code that adds the functionality goes in the `__call__` method and `__init__` takes the function and sets it

## @functools.wraps

Often when working in Python, we need to debug our code. This often involves calling `print` at various points in the code to see what functions were called. We might look at the `__name__` and `__doc__` attributes:

```
>>> @money_format
... def double(x):
...     """Doubles a number."""
...     return x * 2
```

```
...
>>> double
<function money_format.<locals>.wrapper at
0x...>
>>> double.__name__
'wrapper'
>>> double.__doc__
>>>
```

This is where problems arise: when using decorators, the wrapped function's signature such as its `__name__` are lost and replaced by that of the wrapper function. To avoid this, `functools.wraps` comes into play. It can be used as a decorator (yes, another one!) that decorates the wrapper function to preserve the wrapped function's signature. Let's see how we can use it:

```
from functools import wrapsdef
money_format(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        r = func(*args, **kwargs)
        formatted = '£{:.2f}'.format(r)
        return formatted
    return wrapper
```

When writing decorators as classes, however, this does not work as the `__call__` method is defined before any instances have been created:

```
import functoolsclass MoneyFormat:
    def __init__(self, function):
        self.f = function    # the error occurs
in the next line
    @functools.wraps(self.f)
    def __call__(self, *args, **kwargs):
```

```
        return '£{:.2f}'.format(self.f(*args,
**kwargs))NameError: name 'self' is not defined
```

A solution is to use `functools.update_wrapper` inside `__init__` to give `self` the necessary attributes of the function:

```
import functoolsclass MoneyFormat:
    def __init__(self, function):
        self.f = function
        functools.update_wrapper(self, self.f)
def __call__(self, *args, **kwargs):
        return '£{:.2f}'.format(self.f(*args,
**kwargs))
```

## Summary: What have we learnt?

In this article covering how to write decorators in Python, we've covered:

- a few common uses for decorators
- how to write a simple decorator
- how to use the `@decorator` syntax and the fact that is is equivalent to calling `func = decorator(func)`
- how to pass arguments to a decorator
- how to write decorators as classes
- why `functools.wraps` is useful when writing decorators