Decorators in Python

Motivation

By now, you already about functions in Python and how they behave. We have already seen that functions also act as objects in Python. Functions can be nested inside other functions. They can also be passed as arguments to another function and returned from another function.

But what if we want to add some more functionality to an existing function without altering the code in it?

For example, consider the following function:

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

Now when we call this function:

```
ordinary()
```

```
 I am ordinary
```

It prints out the required statement as expected.

Now, Let's say we want to add some more functionality, like some more print statements to this function, we can do something like this:

```
def ordinary():
    print("I got decorated")
    print("I am ordinary")
```

Now when we call this altered function:

```
ordinary()
```

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

It gives the output as expected.

But what if we don't want to alter the code inside this function?

Let's say the code is written by some other programmer for a very specific purpose. By altering the code inside the existing function, we are changing the purpose for which the original function was created.

So, how do we add functionality to an existing function without altering the code inside it?

This is where **decorators** come into picture!!

What are decorators?

Python has an interesting feature called decorators to **add functionality to an existing code**. This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.

In most basic sense, **a decorator takes in a function, adds some functionality and returns it**.

Functions and methods are called "callable" as they can be called.

In fact, any object which implements the special `__call__()` method is termed **callable**. So, basically, a **decorator is a callable that returns a callable**.

Let's add a decorator (additional functionality) to our `ordinary()` function without altering the original code inside it

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


def make_pretty(func): # `func()` passed as argument to another function `make_pretty()`
    def inner():
        print("I got decorated")
        func()
    return inner


ordinary()
```

```
I am ordinary
```

Let's decorate this ordinary function

```
pretty = make_pretty(ordinary)

pretty()
```

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

In this example, **`make_pretty()` is a decorator**. In the assignment step:

```
pretty = make_pretty(ordinary)
```

The function `ordinary()` got decorated and the returned function was given the name `pretty`.

We can see that the **decorator function added some new functionality to the original function**. This is similar to packing a gift. The decorator acts as a wrapper. **The nature of the object that got decorated (actual gift inside) does not alter**. But now, it looks pretty, since it got decorated.

Generally, we decorate a function and reassign it as:

```
ordinary = make_pretty(ordinary)
```

This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the `@` symbol along with the name of the decorator function and place it above the definition of the function to be decorated**.

For example:

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

is equivalent to

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

This is just a syntactic sugar to implement decorators :)

Decorating Functions with Parameters

The decorator we saw till now was simple and it only worked with functions that did not have any parameters.

What if we had functions that took in parameters?

```
def divide(a, b):
    return a/b
```

This function has two parameters, `a` and `b`. **We know it will give an error if we pass in `b` as `0`.**

```
divide(2, 5)
```

```
0.4
```

```
divide(2, 0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-16-45e818b2b9c5> in <module>
----> 1 divide(2, 0)

<ipython-input-14-7507bdc665d5> in divide(a, b)
      1 def divide(a, b):
----> 2     return a/b

ZeroDivisionError: division by zero
```

SEARCH STACK OVERFLOW

- Now let's make a decorator to check for this case that will cause the error.

[ ] ↳ *1 cell hidden*

- This new implementation will return None if the error condition arises.

[ ] ↳ *2 cells hidden*

- In this manner, we can decorate functions that take parameters.

  One important thing to note here is that parameters of the nested `inner()` function inside the decorator is the same as the parameters of functions it decorates. Taking this into account, now we can make **general decorators that work with any number of parameters**.

  ### Decorating Functions with unknown number of parameters

  Do you remember how can we make a function take any number of arguments without knowing how many?

  We can use `function(*args, **kwargs)`.

  In this way, **args will be the tuple of positional arguments** and **kwargs will be the dictionary of keyword arguments**.

  An example of such a decorator is as follows:

[ ] ↳ *1 cell hidden*

- Chaining Decorators in Python

Multiple decorators can be chained in Python.

This means, **a function can be decorated multiple times with different (or same) decorators**. We simply place the decorators above the desired function.

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner


@star
@percent
def printer(msg):
    print(msg)


printer("Hello")
```

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

- Explanation:

The above syntax of:

```
@star
@percent
def printer(msg):
    print(msg)
```

is equivalent to

```
def printer(msg):
    print(msg)
printer = star(percent(printer))
```

The order in which we chain decorators matter.

If we had reversed the order as:

[ ] ↳ *3 cells hidden*

Methods in a Python class

In Python classes, we have three types of methods:

1. Instance methods
2. Class methods
3. Static methods.

Whatever methods we implemented within our classes during the lecture were all **instance methods**. In this reading, we will be focusing on class methods and static methods.

Instance methods

Let's begin by writing a simple class that contains a simple examples Instance method

```
class MyClass:
    def method(self):
        return 'instance method called', self
```

That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, `self`, which points to an instance of `MyClass` when the method is called. But of course, instance methods can accept more than just one parameter.

Through the `self` parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.

Not only can they modify object state, instance methods can also access the class itself through the `self.__class__` attribute. This means **instance methods can also modify class state.**

Now, Let's talk about Class method and compare it with Instance method

Class methods

Class methods work with class variables and are accessible using the class name rather than its object.

Instead of accepting a `self` parameter, **class methods take a `cls` parameter that points to the class, and not the object instance**, when the method is called.

Because the class method only has access to this `cls` argument, **it can't modify object instance state**. That would require access to `self`. However, class methods can still modify class state that applies across all instances of the class.

Since all class objects share the class variables, class methods are used to **access and modify class variables**. Class methods are accessed using the class name and **can be accessed without creating a class object.**

Syntax

To declare a method as a class method, we use the **decorator `@classmethod`**. `cls` is used to refer to the class just like `self` is used to refer to the object of the class.

You can use any other name instead of `cls`, but `cls` is used as per convention, and we will continue to use this convention here.

Just like instance methods, all class methods have at least one argument, `cls`.

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls
```

Here, we have used the `@classmethod` decorator to define `getTeamName` as a class method and later, we will call that method using the class name.

Now, Let's talk about Static methods

Static methods

Static methods are methods that are usually limited to class only and not their objects. They have **no direct relation to class variables or instance variables**. They are used as **utility functions** inside the class or when we do not want the inherited classes to modify a method definition.

This type of method takes **neither a `self` nor a `cls` parameter**, but of course it's free to accept an arbitrary number of other parameters.

Static methods do not know anything about the state of the class

This means that they cannot modify class attributes. Therefore a **static method can neither modify object state nor class state**. Static methods are restricted in what data they can access - and they're primarily a way to namespace your methods. The purpose of a static method is to use its parameters and produce a useful result.

Static methods can be accessed using the class name or the object name.

Syntax

To declare a method as a static method, we use the **decorator** `@staticmethod` . It does not use a reference to the object or class, so we **do not have to use** `self` **or** `cls` . We can pass as many arguments as we want and use this method to perform any function without interfering with the instance or class variables.

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Intuition behind Instance, Class and Static methods

Let's take a look at how these methods behave in action when we call them. We'll start by creating an instance of the class and then calling the three different methods on it.

The above `MyClass` is set up in such a way that each method's implementation returns a tuple containing information for us to trace what's going on, and which parts of the class or object the method can access.

Here's what happens when we call an instance method:

```
obj = MyClass()
obj.method()
```

```
('instance method called', <__main__.MyClass at 0x7fadb0338a90>)
```

This confirms that the **instance method has access to the object instance** via the `self` argument. When the method is called, Python replaces the `self` argument with the instance object, `obj` .

Instance methods can also access the class itself through the `self.__class__` attribute. This makes instance methods powerful in terms of access restrictions as they can modify state on the object instance and on the class itself.

- Let's try out the class method now:

```
[ ] ↳ 1 cell hidden
```

Calling classmethod() shows us it **doesn't have access to the `MyClass` instance object**, but only to the `class MyClass`.

Notice how **Python automatically passes the class as the first argument to the function when we call `MyClass.classmethod()`**. The `self` parameter on instance methods works the same way.

- Let's call the static method now:

```
[ ] ↳ 1 cell hidden
```

- Did you see how we called `staticmethod()` on the object and were able to do so successfully?

Behind the scenes Python simply enforces the access restrictions by not passing in the `self` or the `cls` argument when a static method gets called.

This confirms that **static methods can neither access the object instance state nor the class state**. They work like regular functions but belong to the class's (and every instance's) namespace.

Now, let's take a look at what happens when we attempt to call these methods on the class itself, without creating an object instance beforehand:

```
[ ] ↳ 4 cells hidden
```

- # A Realistic Example to understand when to use these special method types

Now, Let's go over a slightly more realistic example to understand when to use these special method types.

Let's create a class `Pizza` which we'll use to store the ingredients used in each instance of `Pizza` object

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.ingredients!r})')

Pizza(['cheese', 'tomatoes'])
```

```
 Pizza(['cheese', 'tomatoes'])
```

But there are many variations of `Pizza` available

```
Pizza(['mozzarella', 'tomatoes'])
Pizza(['mozzarella', 'tomatoes', 'ham', 'mushrooms'])
Pizza(['mozzarella'] * 4)
```

We can give the users of our `Pizza` class a better interface for creating the pizza objects. A nice and clean way to do that is by using **class methods** for the different kinds of pizzas we can create:

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])
```

Note how we use the `cls` argument in the `margherita` and `prosciutto` methods instead of calling the `Pizza()` constructor directly.

- Now, Let's create some pizzas using these class methods:

[ ] ↳ *3 cells hidden*

- Now, When To Use Static Methods?

Let's say we need to compute the circular area of all the `Pizza` objects that we create. We can use **static method** for this.

[ ] ↳ *1 cell hidden*

▸ As you can see:

First, we modified the constructor and `__repr__` to accept an extra `radius` argument.

We also added an **`area()` instance method** that calculates and returns the pizza's area.

Instead of calculating the area directly within `area()`, using the well-known circle area formula, we factored that out to a **separate `circle_area()` static method**.

Let's use this now:

[ ] ↳ *4 cells hidden*

▸ But, why is that useful?

Flagging a method as a static method is not just a hint that a method won't modify class or instance state, this restriction is also **enforced by the Python runtime**.

Techniques like this allow you to communicate clearly about parts of your class architecture so that new development work is naturally guided to happen within these set boundaries. Of course, it would be easy enough to defy these restrictions. But in practice they often help avoid accidental modifications going against the original design.

Using static methods and class methods are ways to communicate developer intent while enforcing that intent enough to avoid most slip of the mind mistakes and bugs that would break the design. Writing some of your methods that way can provide maintenance benefits and make it less likely that other developers use your classes incorrectly.

Static methods also have benefits when it comes to writing test code.

Because the `circle_area()` method is completely independent from the rest of the class it's much easier to test.

So, we don't have to worry about setting up a complete class instance before we can test the method in a unit test. We can just test it like we would be testing a regular function. This makes future maintenance easier.

↳ *1 cell hidden*

▾ Just to Summarize

- Instance methods need a class instance and can access the instance through `self`.
- Class methods don't need a class instance. They can't access the instance (`self`) but they have access to the class itself via `cls`.

- Static methods don't have access to `cls` or `self`. They work like regular functions but belong to the class's namespace.
- Static and class methods communicate and, to a certain degree, can enforce developer

This was all about class and static methods.