Week-3

1. What is a decorator in Python?

Python offers a unique feature called decorators.

Let's start with an analogy before getting to the technical definition of the decorators. When we mention the word "decorator", what enters your mind? Well, likely something that adds beauty to an existing object. An example is when we hang a picture frame to a wall to enhance the room.

Decorators in Python add some feature or functionality to an existing function without altering it.

Let's say we have the following simple function that takes two numbers as parameters and divides them.

```
def divide(first, second):
    print ("The result is:", first/second)
```

Now if we call this function by passing the two values 16 and 4, it will return the following output:

```
divide(16, 4)
```

The output is:

```
The result is: 4.0
```

```
def divide(first, second):
    print ("The result is:", first/second)

divide(16, 4)
```

```
The result is: 4.0
```

What will happen if we pass the number 4 first, and 16 after? The answer will be 0.25. But we don't want it to happen. We want a scenario where if we see that first < second, we swap the numbers and divide them. But we aren't allowed to change the function.

Let's create a decorator that will take the function as a parameter. This decorator will add the swipe functionality to our function.

```
def swipe_decorator(func):
    def swipe(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)

    return swipe
```

Now we have generated a decorator for the divide() function. Let's see how it works.

```
divide = swipe_decorator(divide)
divide(4, 16)
```

The output is:

```
The result is: 4.0
```

We have passed the function as a parameter to the decorator. The decorator "swiped our values" and returned the function with swiped values. After that, we invoked the returned function to generate the output as expected.

```
# Func refrencinf this divide function
def divide(first, second):
    print ("The result is:", first/second)

def swipe_decorator(func):
    def swipe(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)
    return swipe


divide(4, 16) # Undecorated

divide = swipe_decorator(divide) # Decorated

# Decorated
divide(4, 16)
```

```
The result is: 0.25
The result is: 4.0
```

Start coding or generate with AI.

```
def divide(first, second):
    print ("The result is:", first/second)

divide(4,10) # Bigger/smaller
```

```
 The result is: 0.4
```

```
# Another way of doing the same thing

def swipe_decorator(func):
    def swipe(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)
    return swipe

@swipe_decorator # divide = swipe_decorator(divide)
def divide(first, second):
    print ("The result is:", first/second)



divide(4, 16)
```

```
 The result is: 4.0
```

2. How can you determine whether a class is a subclass of another class?

[ ] ↳ *2 cells hidden*

3.What does Python's MRO (Method Resolution Order) mean?

[ ] ↳ *3 cells hidden*

4. What’s the meaning of single and double underscores in Python variable and method names

[ ] ↳ *11 cells hidden*

5. What is the difference between OOP and SOP?

Object Oriented Programming	Struct
Object-Oriented Programming is a type of programming which is based on objects rather than just functions and procedures	Provic
Bottom-up approach	Top-d
Provides data hiding	Does
Can solve problems of any complexity	Can s
Code can be reused thereby reducing redundancy	Does

6. Can you call the base class method without creating an instance?

Yes, you can call the base class without instantiating it if:

- It is a static method
- The base class is inherited by some other subclass

7. What are the limitations of inheritance?

- Increases the time and effort required to execute a program as it requires jumping back and forth between different classes.
- The parent class and the child class get tightly coupled.
- Any modifications to the program would require changes both in the parent as well as the child class
- Needs careful implementation else would lead to incorrect results

8. What is the difference between range() and xrange()?

- range() creates a static list that can be iterated through while checking some conditions. This is a function that returns a list with integer sequences.
- xrange() is same in functionality as range() but it does not return a list, instead it returns an object of xrange(). xrange() is used in generators for yielding.

range()	xrange()
In Python 3, xrange() is not supported; instead, the range() function is used to iterate in for loops.	The xrange() function is used in P
It returns a list.	It returns a generator object as it
It takes more memory as it keeps the entire list of iterating numbers in memory.	It takes less memory as it keeps

9. How to override the way objects are printed?

Use the `__str__` and the `__repr__` dunder methods.

Here's an example that demonstrates how an instance from the Person class can be nicely formatted when printed to the console.

[ ] ↳ *1 cell hidden*

10. What is the difference between a class method, a static method and an instance method?

Let's begin by writing a (Python 3) class that contains simple examples for all three method types:

```
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'
```

Instance Methods

The first method on `MyClass`, called `method`, is a regular instance method. 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. This makes instance methods powerful in terms of access restrictions—they can freely modify state on the object instance and on the class itself.

Class Methods

Let's compare that to the second method, `MyClass.classmethod`. I marked this method with a `@classmethod` decorator to flag it as a class method. 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.

Since 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.

`Static Methods`

The third method, `MyClass.staticmethod` was marked with a `@staticmethod6` decorator to flag it as a static method.

This type of method doesn't take a `self` or a `cls` parameter, although, of course, it can be made to accept an arbitrary number of other parameters.

As a result, a static method cannot modify object state or class state. Static methods are restricted in what data they can access—they're primarily a way to namespace your methods.

Let's See Them in Action!

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.

`MyClass` was set up in such a way that each method's implementation returns a tuple containing information we can use to trace what's going on and which parts of the class or object that method can access.

```
# Class

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'
```

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

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

```
('instance method called', <__main__.MyClass at 0x7faa53c48640>)
```

This confirms that, in this case, the instance method called method has access to the object instance (printed as `<MyClass instance>`) via the self argument.

When the method is called, Python replaces the `self` argument with the instance object, `obj`.

We could ignore the syntactic sugar provided by the `obj.method()` **dot-call syntax** and pass the instance object manually to get the same result:

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

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

```
    ('instance method called', 10)
```

```
type(obj)
```

```
    __main__.MyClass
```

Let's try out the **class method** next:

```
obj.classmethod()
```

```
    ('class method called', __main__.MyClass)
```

Calling `classmethod()` showed us that it doesn't have access to the `<MyClass instance>` object, but only to the `<class MyClass>` object, representing the class itself (**everything in Python is an object, even classes themselves**).

Notice how Python automatically passes the class as the first argument to the function when we call `MyClass.classmethod()`. Calling a method in Python through the **dot syntax** triggers this behavior. The `self` parameter on instance methods works the same way.

Please note that naming these parameters `self` and `cls` is just a convention. You could just as easily name them `the_object` and `the_class` and get the same result. All that matters is that they're positioned first in the parameter list for that particular method.

Time to call the **static method** now:

```
obj.staticmethod()
```

```
    'static method called'
```

Did you see how we called `staticmethod()` on the object and were able to do so successfully? Some developers are surprised when they learn that it's possible to call a static method on an object instance.

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 using the dot syntax

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' (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:

```
# Class Method
print(MyClass.classmethod())
# Static method
print(MyClass.staticmethod())
#Instance Method
print(MyClass.method())
```

```
('class method called', <class '__main__.MyClass'>)
static method called
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-b561d87f2a57> in <module>
      4 print(MyClass.staticmethod())
      5 #Instance Method
----> 6 print(MyClass.method())

TypeError: method() missing 1 required positional argument: 'self'
```

We were able to call `classmethod()` and `staticmethod()` just fine, but attempting to call the instance method `method()` failed with a `TypeError`.

This is to be expected. This time we didn't create an object instance and tried calling an instance function directly on the class blueprint itself. This means there is no way for Python to populate the `self` argument and therefore the call fails with a `TypeError exception`.

This should make the distinction between these three method types a little more clear

Key Takeaways

- 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' namespace.
- Static and class methods communicate and (to a certain degree) enforce developer intent about class design. This can have definite maintenance benefits.

```
Start coding or generate with AI.
```