```python
# Class decorators take class defnition as an argument
# Attach an attribute to the class
def attach_count(cls):
    cls.count = 0   # Creates an attribute "count" and attaches the attribute
    return cls

@attach_count       # Demo = attach_count(Demo)
class Demo:
    def spam(self):
        print("hello spam")
        self.count += 1     # Demo.count += 1
# ================================================================================
# Attaches greet function to the class
def attach_greet(cls):
    def greet(self):
        return "hello world"
    cls.greet = greet
    return cls
# ================================================================================
@attach_greet       # Demo = greet(Demo)
class Demo:
    def spam(self):
        return "demo spam"
# ================================================================================
# Attaching a class attribute using class decorator
def prices(cls):
    print('attaching class attribute')
```

```
    # creates a class attribute by name "apple" on the class ShoppingCart
    cls.apple = {"iphone": 900, "ipad": 2800, "imac": 4500}
    return cls

@prices     # ShoppingCart = prices(ShoppingCart)
class ShoppingCart:
    def demo(self):
        print(self.apple)
# ================================================================================
# Attaching an instance method to the class using class decorator
def attach_init(cls):
    def wrapper(self, a, b):
        self.a = a
        self.b = b
    cls.__init__ = wrapper
    return cls

@attach_init        # Arithmetic = attach_init(Arithmetic)
class Arithmetic:
    def add(self):
        return self.a + self.b

    def sub(self):
        return self.a - self.b

    def mul(self):
        return self.a * self.b
# ================================================================================
def singleton(cls):
    instances = { }
    def wrapper(*args, **kwargs):
        if cls.__name__ not in instances:
            instances[cls.__name__] = cls(*args, **kwargs)
            print(instances)
            return instances[cls.__name__]
        raise Exception(f"Only one instance can be created for class {cls.__n
    return wrapper
# ================================================================================
def intercept(cls):
    # Redefining the original __setattr__ method
    def __setattr__(self, name, value):
        if value < 0:
            raise ValueError("cannot set a negative value")
        # Calling __setattr__ of object class which sets the value.
        object.__setattr__(self, name, value)
```

```
73      return cls
74
75  @intercept
76  class Point:  # Point = intercept(Point)
77      def __init__(self, a, b):
78          self.a = a
79          self.b = b
80  # ================================================================================
81  # Normal function decorator.
82  def log(func):
83      def wrapper(*args, **kwargs):
84          print('Calling decorator')
85          return func(*args, **kwargs)
86      return wrapper
87
88  # Class decorators should take class as an argument and modified that class a
89  def logging(cls):
90      for key, value in cls.__dict__.items():
91          if callable(value):
92              setattr(cls, key, log(value))
93      return cls
94
95  # All the methods inside the class will be applied with the logging decorator
96  @logging
97  class Arithmetic:       # Arithmetic = logging(Arithmetic)
98      def __init__(self, a, b):
99          self.a = a
100         self.b = b
101
102     def add(self):
103         return self.a + self.b
104
105     def sub(self):
106         return self.a - self.b
107
108     def mul(self):
109         return self.a * self.b
110 # ================================================================================
```

Give feedback