```
class Point:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def move(self, dx, dy):
        self.a += dx
        self.b += dy

# -------------------------------------------------
# Attribute Lookup
# 1. p.a
# 2. p.__getattribute__("a")
# 3. getattr(p, "a")

# Attribute Assignment
# 1. p.a = -1
# 2. p.__setattr__("a", -1)
# 3. setattr(p, "a", -1)
# -------------------------------------------------

class Employee:
    # class attribute (variable)
    company = "HP"

    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def email(self):
        return f"{self.fname}.{self.lname}@company.com"

# -------------------------------------------------
# Attribute Lookup
# 1. e.fname
# 2. e.__getattribute__("fname")
# 3. getattr(e, "fname")

# Attribute Assignment
# 1. e.fname = "STEVE"
# 2. e.__setattr__("fname", "STEVE")
# 3. setattr(e, "fname", "STEVE")
# -------------------------------------------------
# class decorator
def attach_count(cls):  # memory address of the entire class will be passed
    setattr(cls, "count", 0)   # setattr(Point, "count", 0)
    return cls
```

```
@attach_count    # Point = attach_count(Point)
class Point:
    def __init__(self, a, b):
        print("__init__")
        self.a = a
        self.b = b
        Point.count = Point.count + 1


def attach(cls):
    # defining methods
    def move(self, dx, dy):
        self.a += dx
        self.b += dy

    def greet(self):
        return "hello world"

    # attaching the methods to the class
    setattr(cls, "__move__", move)
    setattr(cls, "__greet__", greet)

    # return the modified class
    return cls


@attach
class Point:
    def __init__(self, a, b):
        self.a = a
        self.b = b


def attach_init(cls):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    setattr(cls, "__init__", __init__)
    return cls

@attach_init            # Point = attach_init(Point)
class Point:
    def move(self, dx, dy):
        self.a += dx
        self.b += dy
```

```
def intercept(cls):
    def __setattr__(self, name, value):
        if value < 0:
            raise ValueError("Number should be positive")
        # giving call to parent class __setattr__ (object)
        object.__setattr__(self, name, value)
    setattr(cls, "__setattr__", __setattr__)
    return cls


def intercept(cls):
    # getting the memory address of object class's __setattr__ method
    orig_setattr = getattr(cls, "__setattr__")
    def new_setattr(self, name, value):
        if value < 0:
            raise ValueError("Positive values Only")
        # calling __setattr__ of object class
        orig_setattr(self, name, value)
    setattr(cls, "__setattr__", new_setattr)
    return cls

@intercept
class Point:
    def __init__(self, a, b):
        self.a = a
        self.b = b


def log(func):
    def wrapper(*args, **kwargs):
        print(f"You called {func.__name__}")
        return func(*args, **kwargs)
    return wrapper


def logging(cls):
    for func_name, func_address in cls.__dict__.items():
        if callable(func_address):
            if not func_name == "__init__":
                setattr(cls, func_name, log(func_address))
    return cls


@logging
class Calculator:
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

```
    def add(self):      # add = log(add)
        return self.a + self.b

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

    def mul(self):      # log(mul)
        return self.a * self.b

    def div(self):  # log(div)
        return self.a / self.b

# parameterized decorator
def log(message):
    def _log(func):
        def wrapper(*args, **kwargs):
            print(f"{message}")
            return func(*args, **kwargs)
        return wrapper
    return _log


@log("hey there you called add function")
def add(a, b):
    return a + b

@log("hello there i am doing subtraction operation")
def sub(a, b):
    return a - b



def filter_message(message_type):
    def _filter_message(cls):
        orig_log = getattr(cls, "log")
        def new_log(self, message):
            if message_type in message:
                # calling original log (after filteration)
                orig_log(self, message)
        setattr(cls, "log", new_log)
        return cls
    return _filter_message



@filter_message("error")
class ConsoleLogger:
    def log(self, message):
        print(message)
```

```
@filter_message("info")
class TextLogger:
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message)
        self.file.write("\n")
        self.file.flush()
```