```
# logging.py
# ---------------------------------------------------------------------
from abc import ABC, abstractmethod

# design spec
# Logger is an abstract class with abstract method
# which ever class inherits from "Logger" class, they have to implement "log" method
class Logger(ABC):
    @abstractmethod
    def log(self, message):
        pass

# ConsoleLogger "is-a" logger
class ConsoleLogger(Logger):
    def log(self, message):
        print(message)

# TextLogger "is-a" logger
class TextLogger(Logger):
    def __init__(self, file_object):
        self.file_object = file_object

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

# CSVLogger "is-a" logger
class CSVLogger(Logger):
    def __init__(self, file_object):
        self.file_object = file_object

    def log(self, message):
        from csv import writer
        w = writer(self.file_object)
        words = message.split()
        w.writerow(words)
        self.file_object.flush()

# HTMLLogger "is-a" logger
class HTMLLogger(Logger):
    def __init__(self, file_object):
        self.file_object = file_object

    def log(self, message):
        pass
        # you have to write the logic to write the contents to HTML file

# JSONLogger "is-a" logger
```

```
class JSONLogger(Logger):
    def __init__(self, file_object):
        self.file_object = file_object

    def log(self, message):
        pass  # <--------- write the logic to put things to JSON file

# XMLLogger "is-a" logger
class XMLLogger(Logger):
    def __init__(self, file_object):
        self.file_object = file_object

    def log(self, message):
        pass  # <--------- write the logic to put things to XML file


class MixinFilteredLogger:
    def __init__(self, pattern):
        self.pattern = pattern

    def log(self, message):
        if self.pattern in message:
            # this statement will not give a call to object class
            # since this class is used along with multiple inheritance
            # super will try to call log method which is present in next parent
            # in multiple inheritance
            super().log(message)


# Multiple Inheritance
class FilteredConsoleLogger(MixinFilteredLogger, ConsoleLogger):
    def __init__(self, pattern):
        MixinFilteredLogger.__init__(self, pattern)


class FilteredTextLogger(MixinFilteredLogger, TextLogger):
    def __init__(self, pattern, file):
        MixinFilteredLogger.__init__(self, pattern)
        TextLogger.__init__(self, file)


# Inheritance
# Composition
# Decorator pattern
# Multiple Inheritance


# SpamLogger "is-a" logger
# You will not be able to create an instance of this class
# because it has not implemented mandatory method "log"
#class SpamLogger(Logger):
#    def spam(self):
#        return "spam"
#
# ------------------------------------------------------------------
# Inheritance (IS-A) relationship (Bad Design)
# ------------------------------------------------------------------
# class FilteredConsoleLogger(ConsoleLogger):
#    def __init__(self, pattern):
#        self.pattern = pattern

#    # redefinig log method in child class because
#    # the parent class log method is not filtering the things
#    def log(self, message):
#        # extra functionality is it does filtering
#        if self.pattern in message:
#            super().log(message)  # call "log" method in parent class


# class FilteredTextLogger(TextLogger):
#    def __init__(self, file_object, pattern):
#        self.pattern = pattern
#        super().__init__(file_object)

#    def log(self, message):
#        if self.pattern in message:
#            super().log(message)


# class FilteredCSVLogger(CSVLogger):
#    def log(self, message):
#        pass    # <------ Logic for filtering and passing the original string back to log
method of CSVLogger


# class FilteredHTMLLogger(HTMLLogger):
#    def log(self, message):
#        pass    # <------ Logic for filtering and passing the original string back to log
method of CSVLogger


# class FilteredJSONLogger(JSONLogger):
#    def log(self, message):
#        pass   # <---- filteratioon logic and call back original log method


# class FilteredXMLLogger(XMLLogger):
#    def log(self, message):
```

```
#       pass

# -------------------------------------------------------
# Composition (HAS-AS) relationship
# -------------------------------------------------------

# to the constructor or to the __init__ method
# you have to pass an object instance of a class that has implemented
# "log" method!!!
class FilteredLogger:
   def __init__(self, pattern, logger):
       self.logger = logger   # ----> Dependency inject (FilteredLogger depends on logger
object)
       self.pattern = pattern

   # polymorphic function
   def log(self, message):
       if self.pattern in message:
           # this could call "log" method of either ConsoleLogger or TextLogger or
CSVLogger
           # or HTMLLogger or JSONLogger or XMLLogger
           # it depends on which logger that is being passed as constructor argumet to "
FilteredLogger" class
           self.logger.log(message)    # ----> which class "log" method is called??
## -------------------------------------------------------
#
info = {"fname": "steve", "lname": "jobs", "age": 26}

# I will make Employee object to behave like a dict
# i am going to implement a method by name "get"
# this is called "duck" typing
class Employee:
   def __init__(self, fname, lname, age):
       self.fname = fname
       self.lname = lname
       self.age = age

   def get(self, key):
       return self.__dict__[key]

   def __getitem__(self, key):
       return self.__dict__[key]

# this function assumes that you are storing employee details in a dict object
# email function takes "dict like object"
def email(emp_info):
   first_name = emp_info["fname"]
   last_name = emp_info["lname"]
   return f"{first_name}.{last_name}@company.com"
```

```
#
class Point:
    def __init__(self, a, b):
        super().__setattr__("a", a)
        super().__setattr__("b", b)

    def __setattr__(self, name, value):
        raise Exception


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

    @property
    def a(self):
        print("getter")
        return self._a


    @property
    def b(self):
        print("getter")
        return self._b
```