```
# -------------------------------------------------------------------------------------
# Container Protocols
# -------------------------------------------------------------------------------------
# 1. __len__   (len())
# 2. __contains__   (in)
# 3. __getitem__ (indexing)
# 4. __setitem__    (assign value using indexing)
# 5. __delitem__    (delete using indexing)
class Point(object):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __len__(self):      # this function should return an integer and its a rule!!!
        print("running __len__")
        return 3

    def __contains__(self, item): # this function should return a Boolean.. its a rule
        print("running __contains__")
        if item in self.__dict__.values():
            return True
        return False

    def __getitem__(self, index):
        print("running __getitem__")
        if index == 0:
            return self.a
        elif index == 1:
            return self.b
        elif index == 2:
            return self.c
        else:
            raise IndexError("Index out of range")

    def __setitem__(self, index, value):
        print("running __setitem__")
        if index == 0:
            self.a = value
        elif index == 1:
            self.b = value
        elif index == 2:
            self.c = value
        else:
            raise IndexError("Index out of range")


class PositivePoint(Point):
    # over-riding
    def __setitem__(self, index, value):
        if not isinstance(value, (int, float)):
            raise ValueError("Only numbers are allowed")
        if value < 0:
            raise ValueError("Only positive values are allowed")
        # this will call Point class __setitem__
        super().__setitem__(index, value)


# value of "a" -> 0-100
# value of "b" -> 0-50
# value of "c" -> 0-10
# p[0] setting "a"
# p[1] setting "b"
# p[2] setting "c"
# --------------------------------------------------------------------------------------------
class RangePoint(Point):
    def __setitem__(self, index, value):
        if index == 0:
            if value in range(0, 101):
                super().__setitem__(index, value)
            else:
                raise ValueError("Only permissible values are between 0-100")
        elif index == 1:
            if value in range(0, 51):
                super().__setitem__(index, value)
            else:
                raise ValueError("Only permissible values are between 0-50")
        elif index == 2:
            if value in range(0, 11):
                super().__setitem__(index, value)
            else:
                raise ValueError("Only permissible values are between 0-10")
        else:
            raise IndexError("Index out of range")

# --------------------------------------------------------------------------------------------
class Point:
    def __init__(self, *values):
        self.points = [ ]
        for value in values:
            self.points.append(value)

    def __len__(self):
        return len(self.points)     # self.points.__len__()

    def __contains__(self, value):
        if value in self.points:
            return True
        return False

    def __getitem__(self, index):
        return self.points[index]  # internally it calls __getitem__ on list class

    def __setitem__(self, index, value):
        # internally it calles __setitem__ on list object
        self.points[index] = value      # self.points.__setitem__(index) = value

# ------------------------------------
# Attribute Protocol  # __setattr__ and __getattribute__
# ------------------------------------
class Point:
    def __init__(self, a, b):
        self.a = a  # object.__setattr__("a", "1")
        self.b = b      # object.__setattr__("b", '2')


    # over-riding __setattr__ method in child class
    def __setattr__(self, name, value):
        print("running __setattr__")
        if not isinstance(value, (int, float)):
            raise TypeError("Only numbers are allowed")
        if value < 0:
            raise ValueError("Only positives values are allowed")
        print(f"handing over the value {value} and attr {name} back to parent class")
        super().__setattr__(name, value)


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


    def reset(self):
        self.a = 0
        self.b = 0


    def __len__(self):
        return 2


    def __contains__(self, value):
        if value == self.a or value == self.b:
            return True
        return False
```

```
class Employee(object):
    def __init__(self, fname, lname, age):
        self.fname = fname # __setattr__("fname", "steve")
        self.lname = lname  # __setattr__("lname", "jobs")
        self.age = age      # __setattr__("age", 26)

    def __setattr__(self, name, value):
        print("running __setattr__")
        if name == "fname" or name == "lname":
            # this will call "object" class __setattr__ method
            super().__setattr__(name, value.upper())
        elif name == "age":
            super().__setattr__(name, value)



class Employee(object):
    def __init__(self, fname, lname, age):
        self.fname = fname # __setattr__("fname", "steve")
        self.lname = lname  # __setattr__("lname", "jobs")
        self.age = age      # __setattr__("age", 26)

    def __setattr__(self, name, value):
        print("running __setattr__")
        if name == "fname" or name == "lname":
            # this will call "object" class __setattr__ method
            super().__setattr__(name, value[::-1])
        elif name == "age":
            super().__setattr__(name, value)


class Calculator:
    def __init__(self, a, b):
        self.a = a   # __setattr__("a", "2")
        self.b = b  # __setattr__("b", 3)

    # over-riding (intercepting the values of "a" and "b" before setting)
    def __setattr__(self, name, value):
        if not isinstance(value, (int, float)):
            raise TypeError
        # handing over the values back to parent (object) class __setattr__
        super().__setattr__(name, value)


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

# validate fname, lname and pay of an employee
# 1. fname and lname should not be greater than 5 characters
# 2. pay should not be less than $500

class Employee:
    def __init__(self, fname, lname, pay):
        self.fname = fname  # __setattr__("fname", "sandeep")
        self.lname = lname  # __setattr__("lname", "suryaprasad")
        self.pay = pay      # __setattr__("pay", -8000)


    def __setattr__(self, name, value):
        if name == "fname" or name == "lname":
            if len(value) > 5:
                raise ValueError("Max allowed len for fname and lname is 5")
            else:
                super().__setattr__(name, value)
        elif name == "pay":
            if value < 500:
                raise ValueError("Min pay is $500")
            else:
                super().__setattr__(name, value)


    def email(self):
        ...


    def hike(self, percentage):
        ...
# -----------------------------------------------------------------------------------------

class Employee:
    def __init__(self, fname, lname, pay):
        self.fname = fname
        self.lname = lname
        self.pay = pay

    def __setattr__(self, name, value):
        # restricting the user from adding new attribute to the class
        if name not in ("fname", "lname", "pay"):
            raise AttributeError(f"attribute {name} cannot be set")
        # if the attribute that the user is setrting is either "fname" or "lname"
        # or "pay", handover the name of the attribute and the value back to
        # parent class (object) __setattr__ method
        super().__setattr__(name, value)

# making class completely immutable
class Employee:
    # __init__ method will be called only once while creating insance
    def __init__(self, fname, lname, pay):
        print("running __init__")
        super().__setattr__("fname", fname)
        super().__setattr__("lname", lname)
        super().__setattr__("pay", pay)

    # completely over-riding parent class __setattr__ method in child class
    def __setattr__(self, name, value):
        print("RUNNING __SETATTR__")
        raise AttributeError("No")

# Comparsion Protocol
# __lt__, __gt__, __eq__, __ne__, __le__, __ge__
class Employee:
    def __init__(self, name, age, pay):
        self.name = name
        self.age = age
        self.pay = pay

    def __lt__(self, other):
        print("running __lt__")
        if self.age < other.age:   # 26.__lt__(29)
            return True
        return False

    def __gt__(self, other):
        print("running __gt__")
        if self.age > other.age:
            return True
        return False

    def __eq__(self, other):
        print("running __eq__")
        if self.age == other.age:
            return True
        return False
    # return True if self.age == other.age else False

e1 = Employee("alex", 26, 1000)
e2 = Employee("tammy", 21, 900)
e3 = Employee("bill", 21, 1200)

employees = [e1, e2, e3]




# Number Protocol
# __add__, __sub__, __mul__, __div__
class Employee:
    def __init__(self, name, age, pay):
        self.name = name
        self.age = age
        self.pay = pay

    def __add__(self, other):
        result_age = self.age + other.age
        result_name = f"{self.name} {other.name}"
        result_pay = self.pay + other.pay
        return Employee(result_name, result_age, result_pay)

    def __sub__(self, other):
        ...

    def __mul__(self, other):
        ...



e1 = Employee("alex", 26, 1000)
e2 = Employee("tammy", 21, 900)
e3 = Employee("bill", 21, 1200)


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

    def __add__(self, other):
        return Point(self.a + other.a, self.b + other.b)

    def __sub__(self, other):
        return Point(self.a - other.a, self.b - other.b)

    def __mul__(self, other):
        return Point(self.a * other.a, self.b * other.b)

# ----------------------------------------------------------------------
# truthiness protocol
# ----------------------------------------------------------------------
# 1. First python tries to call __bool__ method on an object, if it is not
# implemented, then it will try to check if __len__ is implemented
# 2. Based on the return value of either __bool__ or __len__, the truthiness
# of an objected is decided
# 3. If both __bool__ and __len__ are not implemented, the object always evaluates to
# boolean True
# 4. If both __bool__ and __len__ are implemented, the the first preference goes to
# __bool__
```

```
class Employee:
    def __init__(self, name, age, pay):
        self.name = name
        self.age = age
        self.pay = pay

    # employee object is evaluated to boolean True if the age of the employee
    # is greater than 18, else it will be evaluated to boolean False
    def __bool__(self):
        print("running __bool__")
        if self.age < 18:
            return False
        return True



#e = Employee("steve", 18, 1000)
#
#if e:       # trying to evaluate the boolean value of employee object
#    print("Hi")
#else:
#    print("Bye")
#


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

    # fallback mechanism (backup function)
    def __len__(self):
        print("runnin __len__")
        if self.a == 0 and self.b == 0:
            return 0
        elif self.a < 0 and self.b < 0:
            return 0
        return 1


#    def __bool__(self):
#        print("running __bool__")
#        if self.a == 0 and self.b == 0:
#            return False
#        elif self.a < 0 and self.b < 0:
#            return False
#        return True
```

```
p1 = Point(1, 2)    # evaluates to boolean True
p2 = Point(1, 0)    # evaluates to booelan True
p3 = Point(0, 2)    # evaluates to booelan True
p4 = Point(0, 0)    # evaluates to boolean False
p5 = Point(1, -2)   # evaluates to boolean True
p6 = Point(-1, 2)   # evaluates to boolean True
p7 = Point(-1, -2) # evaluates to boolean False

if p6:
    print("hi")
else:
    print("bye")


# ----------------------------------------------------------------------
# Function protocol
# ----------------------------------------------------------------------
# if any object if it has defined __call__, then it is a callable object

# Greetings has implemented function protocol by implementing __call__ method
class Greetings:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def greet(self, name):
        return f"hello {name}"

    # Greetings object is callable because we have implemented __call__ method
    # inside Greetings class
    def __call__(self, name):
        print("running __call__")
        print(self.a, self.b)
        return f"hello {name}"



def func(): # func is a variable and it is pointing to a function object
    return "hello func"


# Squares object is a callable, because we have implemented __call__ method
class Squares:
    def __call__(self, numbers: list) -> list:
       return [ number ** 2 for number in numbers ]

class Evens:
    def __call__(self, number: int) -> bool:
        return True if number % 2 == 0 else False
```

```
# Decorator??
# decorator is a callable which accepts one more callable and adds extra
# functionality to the existing func or callable
# function implementation of a decorator
def log(func):  # returns reference or memory address of wrapper function
    def wrapper(*args, **kwargs):
        print(f"you called {func.__name__}")
        return func(*args, **kwargs)   # add(1, 2)  add(a=1, b=2)
    return wrapper   # memory address or reference of wrapper function



class Log:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):  # __call__ acts as a wrapper function
        print(f"You are calling {self.func.__name__}")
        return self.func(*args, **kwargs)


# function implementation of a time decorator
def time(func):
    def wrapper(*args, **kwargs):
        from time import time
        start = time()
        result = func(*args, **kwargs)
        end = time()
        print(f"execution time: {end-start}")
        return result
    return wrapper

# class implementation of a decorator
class Time:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        from time import time
        start = time()
        result = self.func(*args, **kwargs)
        end = time()
        print(f'executinon time : {end-start}')
        return result
```

```
class Record:
    def __init__(self, func):
        self.func = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"{self.func.__name__} is called {self.count} times")
        return self.func(*args, **kwargs)


# add = log(add)  # add is no more pointing to original add but now
# it is pointing to memory address of wrapper function
@Record   # add = Record(add)
def add(a, b):
    from time import sleep
    sleep(3)
    return a + b


@Record     # sub = Record(sub)
def sub(a, b):
    return a - b


@Record     # mul = Recrod(mul)
def mul(a, b):
    return a * b


items = ['bv', 'aw', 'dt', 'cu']


def get_last_letter(item):
    return item[-1]


class GetLastLetter:
    def __call__(self, item):
        return item[-1]


s1 = sorted(items, key=GetLastLetter())   # g('bv'), g('aw'), g('dt'), g('cu')
s2 = sorted(items, key=get_last_lette)
s3 = sorted(items, key=lambda item: item[-1])


class Arithmetic:
```

```
@Time
def add(self, a, b):
    return a + b
@Time
def sub(self, a, b):
    return a - b
@Time
def mul(self, a, b):
    return a * b
```