```
# ---------------------------------------------------------------------
# encapsulation.py
# ---------------------------------------------------------------------
# Option-1 (using __setattr__)
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def circumference(self):
        return 2 * 3.14 * self.radius

    def __setattr__(self, name, value):
        if not isinstance(value, (int, float)):
            raise TypeError("radius should be a number")
        super().__setattr__(name, value)

# option-2 (getters and setters)
# single underscore attributes are only for internal implementation or use
# you are not suppose to access _ attributes directly.
# But in python you will be able to access the internal attributes of the class
# underscore attributes are called "private" attributes (it can be instance varaiable)
# or it can be internal function or method

# 100% encapsulation is not possible
# Python community assusmes that all users of python are "adults" (>25)
class Circle:
    def __init__(self, radius):
        # self.radius is property (getter and setter)
        self.radius = radius

    # this method is not supposed to be accessed outside the class
    def _some_thing(self):
        return "hello something"

    # getter method in python
    @property
    def radius(self):
        self._some_thing()  # calling internal method (starts with underscore)
        return self._radius

    # setter method in python
    @radius.setter
    def radius(self, value):
        if not isinstance(value, (int, float)):
            raise TypeError("only numbers are allowed for radius")
        if value < 0:
            raise ValueError("Only positive numbers are allowed")
        # underscore variable is an internal variable (private varaible)
        # _radius is internal variable
```

```
        # hiding the value of radius inside an internal attribute "_radius"
        self._radius = value

    def circumference(self):
        return 2 * 3.14 * self.radius

# 1. fname and lname should be min of 5 chars and max of 10 chars
# 2. age should be min of 25 and max of 60
# 3. pay should be min of $1000 and max should $15000
class Employee:
    def __init__(self, fname, lname, age, pay):
        self.fname = fname
        self.lname = lname
        self.age = age
        self.pay = pay

    def __setattr__(self, name, value):
        if name == "fname" or name == "lname":
            # all fname and lname validations
            if len(value) < 5:
                raise ValueError
            if len(value) > 10:
                raise ValueError
            # pass on the "fname" and "lname" to parent class __setattr__
            super().__setattr__(name, value)

        elif name == "age":
            # all "age" related validations
            if not isinstance(value, (int, float)):
                raise TypeError
            if value < 25:
                raise ValueError
            if value > 60:
                raise ValueError
            super().__setattr__(name, value)

        elif name == "pay":
            if not isinstance(value, (int, float)):
                raise TypeError
            if value < 1000:
                raise ValueError
            if value > 15000:
                raise ValueError
            super().__setattr__(name, value)



# option-2
class Employee:
```

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

@property
def fname(self):
    return self._fname

@fname.setter
def fname(self, value):
    # all fname and lname validations
    if len(value) < 5:
        raise ValueError
    if len(value) > 10:
        raise ValueError
    # hiding the fname in underscore variable (_fname)
    self._fname = value

@property
def lname(self):
    return self._lname

@lname.setter
def lname(self, value):
    # all fname and lname validations
    if len(value) < 5:
        raise ValueError
    if len(value) > 10:
        raise ValueError
    # hiding the fname in underscore variable (_fname)
    self._lname = value


@property
def age(self):
    return self._age

@age.setter
def age(self, value):
    # all "age" related validations
    if not isinstance(value, (int, float)):
        raise TypeError
    if value < 25:
        raise ValueError
    if value > 60:
        raise ValueError
    self._age = value
```

```
    @property
    def pay(self):
        return self._pay

    @pay.setter
    def pay(self, value):
        if not isinstance(value, (int, float)):
            raise TypeError
        if value < 1000:
            raise ValueError
        if value > 15000:
            raise ValueError
        self._pay = value

# solution-1
class Calculator:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    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 __setattr__(self, name, value):
        if not isinstance(value, (int, float)):
            raise TypeError
        super().__setattr__(name, value)


#solution-2
class Calculator:
    def __init__(self, a, b):
        self.a = a
        self.b = b

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

    @a.setter
    def a(self, value):
```

```
    print(f"setting a to {value}")
    if not isinstance(value, (int, float)):
        raise TypeError
    self._a = value

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

@b.setter
def b(self, value):
    print(f"setting b to {value}")
    if not isinstance(value, (int, float)):
        raise TypeError
    self._b = value


def add(self):
    return self.a + self.b

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

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