# Lista de Exercícios: Dominando Classes com Encapsulamento e Properties

Aqui estão 10 exercícios, do fácil ao difícil, onde cada um exige a implementação completa de atributos encapsulados com @property e @*.setter.

---

## Nível Fácil

### Exercício 1: Classe Pessoa

**Objetivo:** Criar uma classe com validações simples nos setters para tipo e valor.

- **Requisitos:**
  1. Crie uma classe Pessoa com os atributos protegidos _nome e _idade.
  2. Crie uma @property para nome.
  3. Crie um @nome.setter que valide se o valor é uma string e não está vazio.
  4. Crie uma @property para idade.
  5. Crie um @idade.setter que valide se a idade é um número inteiro e positivo.
  6. No __init__, use os setters para atribuir os valores iniciais (ex: self.nome = nome).
  7. Instancie um objeto Pessoa com dados válidos e depois tente alterar nome para um valor vazio e idade para um valor negativo para testar as validações.

### Exercício 2: Classe Círculo

**Objetivo:** Usar um setter para validar dados numéricos e ter métodos que usam a property.

- **Requisitos:**
  1. Crie uma classe Circulo com um atributo protegido _raio.
  2. Crie uma @property para raio.
  3. Crie um @raio.setter que valide se o valor do raio é um número positivo.
  4. Crie um método calcular_area() que use a propriedade self.raio para retornar a área do círculo ($A=\pi \cdot r2$).

5. Instancie um círculo, teste a alteração do raio para um valor válido e um inválido, e imprima a área.

### Exercício 3: Classe Carro

**Objetivo:** Usar métodos para interagir com uma propriedade booleana através de seu setter.

- **Requisitos:**
  1. Crie uma classe Carro com o atributo protegido _ligado (booleano).
  2. Crie uma @property para ligado.
  3. Crie um @ligado.setter que valide se o valor recebido é um booleano (True ou False).
  4. Crie um método ligar() que simplesmente faz self.ligado = True.
  5. Crie um método desligar() que simplesmente faz self.ligado = False.
  6. Instancie um carro, use os métodos ligar() e desligar() e imprima o status da propriedade ligado após cada ação.

---

## Nível Intermediário

### Exercício 4: Classe ContaBancaria

**Objetivo:** Ter um setter que depende de uma condição (saldo) e métodos que usam esse setter.

- **Requisitos:**
  1. Crie uma classe ContaBancaria com _titular e _saldo protegidos.
  2. Crie properties e setters para ambos. O setter de _titular deve validar que é uma string. O setter de _saldo deve validar que é um número.
  3. Crie um método depositar(valor) que valida se o valor é positivo e então usa o setter para atualizar o saldo (self.saldo += valor).
  4. Crie um método sacar(valor) que verifica duas coisas: se o valor do saque é positivo e se self.saldo é suficiente. Se for, use o setter para atualizar o saldo (self.saldo -= valor).
  5. Instancie uma conta, faça depósitos e saques para testar a lógica.

### Exercício 5: Classe Produto

**Objetivo:** Gerenciar múltiplos atributos com validações independentes.

- **Requisitos:**
  1. Crie uma classe Produto com atributos protegidos _nome, _preco e _estoque.
  2. Crie properties e setters para todos os três atributos.
  3. O setter de nome deve garantir que é uma string não vazia.
  4. O setter de preco deve garantir que é um número maior ou igual a zero.
  5. O setter de estoque deve garantir que é um número inteiro maior ou igual a zero.
  6. Instancie um produto e teste as validações de todos os setters.

### Exercício 6: Classe Retangulo com Propriedade Calculada

**Objetivo:** Introduzir uma propriedade que é apenas um "getter" (somente leitura) e calculada a partir de outras.

- **Requisitos:**
  1. Crie uma classe Retangulo com _largura and _altura protegidos.
  2. Crie properties e setters para largura e altura, garantindo que ambos sejam sempre valores positivos.
  3. Crie uma **terceira** @property chamada area.
  4. Esta propriedade area deve **apenas** ter um getter (o método com @property). Ele deve calcular e retornar a área (self.largura * self.altura).
  5. Esta propriedade area **não deve** ter um setter.
  6. Instancie um retângulo, altere sua largura e altura, e observe como o valor de area se atualiza automaticamente.

---

## Nível Difícil

### Exercício 7: Classe Termometro com Conversão

**Objetivo:** Criar setters que, ao serem alterados, modificam um atributo base, e getters que calculam valores derivados.

- **Requisitos:**
  1. Crie uma classe Termometro com um único atributo protegido: _temperatura_celsius.
  2. Crie uma @property e @temperatura_celsius.setter para temperatura_celsius. O setter pode validar se é um número.
  3. Agora, crie uma **segunda** @property chamada temperatura_fahrenheit.
  4. O getter de temperatura_fahrenheit deve calcular e retornar a temperatura em Fahrenheit a partir de self._temperatura_celsius (F = C * 9/5 + 32).
  5. O setter de temperatura_fahrenheit deve receber um valor em Fahrenheit, convertê-lo para Celsius (C = (F - 32) * 5/9) e **atribuir o resultado a self._temperatura_celsius**.
  6. Instancie um termômetro, defina a temperatura em Celsius, leia em Fahrenheit. Depois, defina em Fahrenheit e leia em Celsius para ver se a conversão funciona nos dois sentidos.

### Exercício 8: Classe Funcionario com Setter Condicional

**Objetivo:** A lógica do setter depende do estado atual do próprio atributo.

- **Requisitos:**
  1. Crie uma classe Funcionario com _nome e _salario protegidos.
  2. Crie uma property e setter para nome.
  3. Crie uma @property para salario.
  4. Crie um @salario.setter com uma lógica especial: um novo salário só pode ser definido se for **maior** que o _salario atual. Não é permitido reduzir o salário de um funcionário.
  5. Crie um método promover(aumento_percentual) que calcula um novo salário com base no percentual de aumento e usa o setter (self.salario = ...) para aplicá-lo.
  6. Instancie um funcionário, tente dar um aumento usando o método promover, e

depois tente atribuir um salário menor diretamente para testar a validação do setter.

### Exercício 9: Classe Playlist

**Objetivo:** Gerenciar uma lista de forma protegida e ter propriedades que derivam informações dela.

- **Requisitos:**
  1. Crie uma classe Playlist com _nome e _musicas (uma lista) protegidos.
  2. Crie property/setter para nome.
  3. Crie um método adicionar_musica(musica) para adicionar uma música à lista _musicas. Não crie um setter para a lista _musicas diretamente.
  4. Crie uma @property somente leitura (sem setter) chamada tamanho que retorna o número de músicas na playlist (len(self._musicas)).
  5. Crie uma @property somente leitura chamada musicas que retorna uma **cópia** da lista de músicas (return self._musicas.copy()). Isso evita que a lista original seja modificada externamente.
  6. Instancie uma playlist, adicione algumas músicas e verifique o tamanho e a lista de musicas.

### Exercício 10: Classe Data com Validação Cruzada

**Objetivo:** O desafio final. Os setters precisam conhecer o estado de outros atributos para validar corretamente.

- **Requisitos:**
  1. Crie uma classe Data com os atributos protegidos _dia, _mes e _ano.
  2. Crie properties e setters para dia, mes e ano.
  3. A lógica de validação dos setters deve ser interdependente:
     - ano: Deve ser um número maior que 0.
     - mes: Deve ser um número entre 1 e 12.
     - dia: A validação do dia depende do mês e do ano. Ex: dia 31 é inválido para o mes 4 (abril). dia 29 só é válido para o mes 2 se o ano for bissexto.
  4. (Dica: você pode criar um método auxiliar _eh_bissexto(ano) para ajudar na validação).
  5. Instancie uma data e teste rigorosamente: tente mudar o dia para 31 em um mês de 30 dias; tente mudar para o dia 29 em um fevereiro não bissexto e depois em um bissexto.