

# Estrutura de dados

## Orientação a Objetos

Prof<sup>a</sup>. Dr<sup>a</sup>. Alana Moraes

# Última aula

- Conceitos Principais
- Ciclo de vida de um objeto
- Diferença entre classe e objeto



# Orientação a Objetos

- Vamos partir do exemplo final da aula:
  - Modele um funcionário. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário (double), a data de entrada no banco (String) e seu RG (String).
  - Crie um método receberAumento que aumenta o salário do funcionário de acordo com o parâmetro passado como argumento.
  - Crie também um método calculaGanhoAnual, que não recebe parâmetro algum, devolvendo o valor do salário multiplicado por 12.
  - Por fim, crie uma aplicação teste para verificar sua classe.

# Planejamento OO

1. Identifique uma classe que deve existir no seu problema
  - a. Pense nos atributos
  - b. Implemente o construtor
  - c. Implemente métodos adicionais
2. Verifique se há outras classes
  - a. Repita o processo
3. Implemente o arquivo teste.py



# Encapsulamento

- Não é seguro acessar o atributo salário do objeto diretamente.
- É necessário encapsular tais informações e usar os métodos responsáveis por encapsular o acesso ao objeto.
- Então, para melhorarmos a classe Funcionario, devemos restringir o acesso a salário, tornando-o privado, adicionando dois caracteres *underscore* (`__`).
- Em algumas linguagens como Java, a palavra **private** define o atributo como privado e é chamado como modificador de visibilidade.

# Encapsulamento

- Nós continuamos a ter acesso aos atributos, ainda que eles tenham mudado de nome — o Python adicionou a classe antecedido por `_`.
- Ao escrevermos ***func.\_Funcionario\_\_salario***, o Python informará ao desenvolvedor que o atributo ***\_\_salario*** não deve ser acessado.

A ação de tornar privado o acesso aos atributos, no mundo Orientado a Objetos, chamamos de **encapsulamento**.

# Encapsulamento

- Encapsular é fundamental para que seu sistema seja suscetível a mudanças: não precisamos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está encapsulada.
- O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

# Exercício

```
class Retangulo:  
    def __init__(self, x, y):  
        self.__x = x  
        self.__y = y  
        self.__area = x * y  
  
    def obter_area(self):  
        return self.__area
```

Assumindo que a classe foi carregada corretamente, podemos executar o seguinte código:

```
r = Retangulo(7,6)  
r.area = 7  
r.obter_area()
```

Qual é o resultado da execução? Se tiver com dúvida, faça o teste!



# Exercício

Voltem ao exemplo da classe Conta, Círculo ou Retângulo da última aula e a reestruture para torná-la encapsulada.



# Getters e Setters

- O underscore \_ alerta que ninguém deve modificar, nem mesmo ler, o atributo em questão.
  - Temos um problema: como fazer para mostrar o saldo de uma Conta, já que não devemos acessá-lo para leitura diretamente?
  - Precisamos então arranjar uma maneira de fazer esse acesso.
  - Sempre que precisamos arrumar uma maneira de fazer alguma coisa com um objeto
- A convenção para esses métodos em muitas linguagens orientadas a objetos é colocar a palavra **get** ou **set** antes do nome do atributo.

# Getters e Setters

```
class Conta:
    def __init__(self, titular, saldo):
        self._titular = titular
        self._saldo = saldo

    def get_saldo(self):
        return self._saldo

    def set_saldo(self, saldo):
        self._saldo = saldo
```

```
    def get_titular(self):
        return self._titular

    def set_titular(self, titular):
        self._titular = titular
```

# Exemplo Funcionário

Imagine que agora no seu projeto você precisa adicionar um nome e um id ao Setor.



# Properties - Getter

- Existe outra forma de acessar as variáveis privadas sem quebrar o encapsulamento
  - Na linguagem Python, os métodos que dão acesso são nomeados como properties.
  - Desta forma, indicaremos para o Python nossa intenção de ter acesso ao objeto.
  - A declaração de uma property é feita com o uso do caractere @.
- Se esquecermos de adicionar `__` ao atributo nome e torná-lo privado, receberemos uma mensagem de erro quando tentarmos acessá-lo no console.
- Exemplo: gostaria de criar uma classe cliente que sempre mostrasse o nome maiúsculo e eu já tenho método que faz isso na minha classe.

# Properties - Getter

```
class Cliente:
```

```
    def __init__(self, nome):  
        self.__nome = nome
```

```
    @property
```

```
    def nome(self):  
        return self.__nome.primeiraMaiuscula()
```

```
cliente = Cliente("alana")  
print(cliente.nome)
```

# Properties - Setter

- Da mesma forma como fazemos isso para um getter, faremos para um setter.

class Cliente:

# init e getter

@nome.setter

def nome(self, nome):

print("chamando setter nome()")

self.\_\_nome = nome

cliente = Cliente("alana")

cliente.nome = "giovanni"

# Properties - Setter

- Da mesma forma como fazemos isso para um getter, faremos para um setter.

class Cliente:

# init e get

Mesmo nome da propriedade criada

@nome.setter

def nome(self, nome):

print("chamando setter nome()")

self.\_\_nome = nome

cliente = Cliente("alana")

cliente.nome = "giovanni"



# Exercício

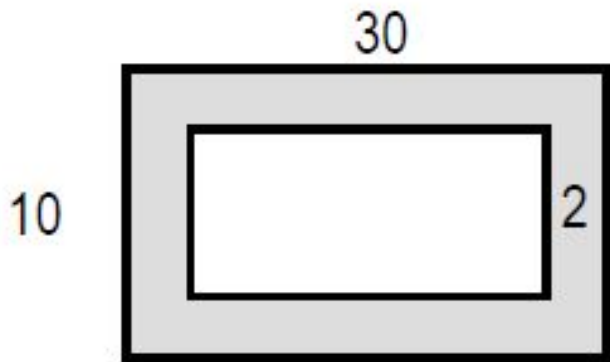
Agora meu Cliente além do nome terá um endereço.

Adicione este atributo, ajuste seu construtor, gets e sets, e, por fim, crie propriedades para acessar o endereço.



# Exercício da Aula

- Implemente um programa que calcule a área de uma moldura.
- Planeje suas classes, pense no encapsulamento, crie propriedades e teste a classe.



# Dúvidas?



[alanamm.prof@gmail.com](mailto:alanamm.prof@gmail.com)