

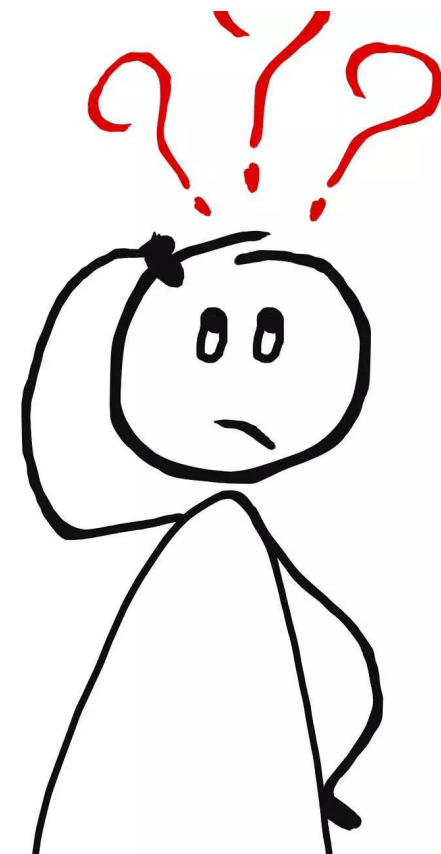
PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática
Lucas Sampaio Leite



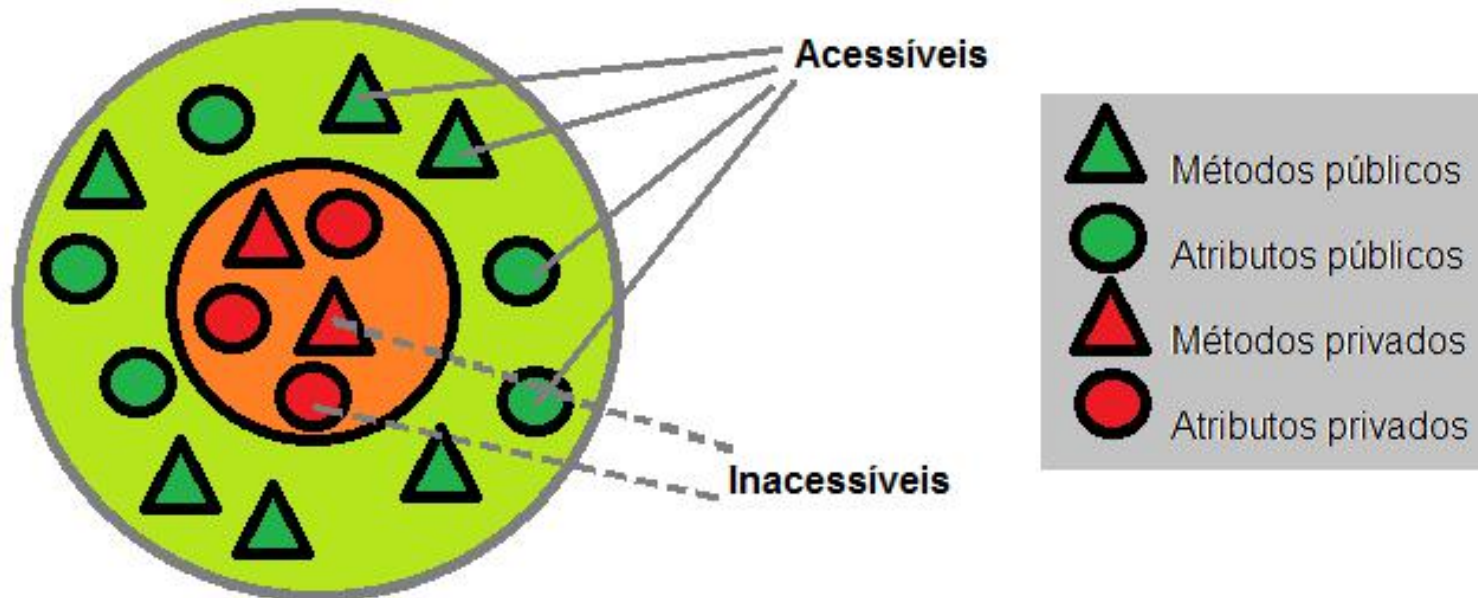
Revisão de Programação Orientada a Objetos com Python

- O que é encapsulamento dentro da programação orientada a objetos?



Revisão de Programação Orientada a Objetos com Python

- Encapsulamento é um dos pilares da Programação Orientada a Objetos (POO). Ele consiste em restringir o acesso direto aos dados internos de um objeto, fornecendo meios controlados de acesso e modificação por meio de métodos (funções dentro das classes).



Revisão de Programação Orientada a Objetos com Python

- Objetos podem ser vistos como cápsulas, dentro das quais estão as implementações dos comportamentos e o espaço de memória para armazenamento das propriedades.
- O espaço de memória e as implementações ficam protegidos dentro da cápsula e não podem ser acessados diretamente pelo código externo a essa cápsula.
- Tudo que o código externo pode ver são as interfaces do objeto. A interface é um canal, através do qual, o objeto oferece serviços.

Revisão de Programação Orientada a Objetos com Python

- O encapsulamento ajuda a garantir que o estado do objeto se mantenha consistente.
 - Por exemplo, no caso do objeto conta, o atributo saldo só deve ser modificado por meio dos métodos creditar e debitar, que fazem parte da interface.
 - Do contrário, seria possível aumentar ou reduzir o valor do saldo de maneira aleatória (sem operações de crédito e débito) e o extrato da conta ficaria inconsistente. Portanto, nenhum objeto externo deve conseguir acessar diretamente o saldo.

Revisão de Programação Orientada a Objetos com Python

```
c1 = Conta("Lucas")  
c1.depositar(1000)  
c1.saldo = 10000  
c1.sacar(50)
```



```
[Lucas] Depósito de R$ 1000.00 realizado. Novo saldo: R$ 1000.00  
[Lucas] Saque de R$ 50.00 realizado. Novo saldo: R$ 9950.00
```

Se permitirmos acesso direto aos atributos internos de uma classe, como saldo, corremos o risco de colocar o objeto em um estado inconsistente, ou seja, um estado que viola as regras de funcionamento do sistema.

Revisão de Programação Orientada a Objetos com Python

- Encapsulamento em Python:
 - Python não tem modificadores de acesso (como private, protected, public), mas possui convenções para indicar o nível de acesso desejado:

Prefixo	Acesso sugerido	Exemplo
Nenhum	Público	<code>self.nome</code>
<code>_</code>	Protegido e não deve ser acessado diretamente	<code>self._idade</code>
<code>__</code>	Privado e deve ser acessado apenas por métodos da classe	<code>self.__senha</code>

Revisão de Programação Orientada a Objetos com Python

- Encapsulamento em Python:
 - Python não tem modificadores de acesso (como private, protected, public), mas possui convenções para indicar o nível de acesso desejado:

Prefixo	Acesso sugerido	Exemplo
Nenhum	Público	<code>self.nome</code>
<code>_</code>	Protegido e não deve ser acessado diretamente	<code>self._idade</code>
<code>__</code>	Privado e deve ser acessado apenas por métodos da classe	<code>self.__senha</code>

Em linguagens como Java e C#, modificadores de acesso controlam de forma efetiva a visibilidade dos atributos e métodos, permitindo restringir o acesso externo e preservar a integridade do objeto.

Revisão de Programação Orientada a Objetos com Python

- Aplicando os prefixos:

```
class Conta:
    def __init__(self, titular, saldo):
        self.titular = titular
        self._saldo = saldo
        self.__senha = "1234"

    def depositar(self, valor):
        if valor > 0:
            self._saldo += valor

    def set_senha(self, senha_atual, nova_senha):
        if senha_atual == self.__senha:
            self.__senha = nova_senha
            print("Senha alterada com sucesso.")
        else:
            print("Senha atual incorreta. Não foi possível alterar a senha.")
```

Revisão de Programação Orientada a Objetos com Python

- Aplicando os prefixos:

```
conta = Conta("Maria", 1000)
print(conta.titular)          # Maria
print(conta._saldo)           # Possível, mas não recomendado
conta.set_senha("1234", "novaSenha123" ) #Senha alterada com sucesso.
print(conta._Conta__senha)    # Funciona (mas não deve ser usado)
print(conta.__senha)           # Erro!
```

Atributos com `__` sofrem name mangling (são renomeados internamente para evitar acesso accidental): `obj.__senha` vira `obj._Conta__senha`.

Revisão de Programação Orientada a Objetos com Python

- Assim como os atributos, também pode-se definir métodos de uso interno na classe — ou seja, métodos que não fazem parte da interface pública do objeto.
- Métodos privados são utilizados para organizar e auxiliar a implementação de outros métodos, mas não devem ser acessados diretamente por quem usa a classe, já que podem ser modificados ou removidos a qualquer momento sem aviso, quebrando a compatibilidade.

Revisão de Programação Orientada a Objetos com Python

```
class Conta:
    def __init__(self, titular, saldo):
        self.titular = titular
        self._saldo = saldo
        self.__senha = "1234"

    def sacar(self, valor):
        if self._valor_valido(valor) and valor <= self._saldo:
            self._saldo -= valor
            print(f"Saque de R$ {valor} realizado com sucesso.")
        else:
            print("Saque não realizado: valor inválido ou saldo insuficiente.")

    def _valor_valido(self, valor):
        return valor > 0
```

O método `_valor_valido` é protegido e não faz parte da interface pública. Ele existe apenas para uso interno e facilita a reutilização de código dentro da própria classe.

Revisão de Programação Orientada a Objetos com Python

- Em várias linguagens de programação, como Java e C#, é comum definir métodos específicos para acessar (get) e modificar (set) os atributos de uma classe.
- Esses métodos ajudam a controlar o acesso e garantir a integridade dos dados.

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

    def get_saldo(self):
        return self._saldo

    def get_limite(self):
        return self._limite

    def set_limite(self, limite):
        if limite < 0:
            print('Limite não pode ser negativo!')
        else:
            self._limite = limite
```


Revisão de Programação Orientada a Objetos com Python

- A comunidade Python critica o uso excessivo de métodos get e set para acessar e modificar atributos de uma classe.
 - Esses métodos acabam tornando o código mais verboso e difícil de ler.

```
conta1.creditar(conta2.get_saldo() + conta3.get_saldo())  
conta2.debitar(conta2.get_saldo())  
conta3.debitar(conta3.get_saldo())
```

Simplificar traria de volta o problema de acessar direto os atributos da classe.

Revisão de Programação Orientada a Objetos com Python

- Python resolve esse problema com um padrão de projeto de software conhecido como decorators.
- O @property é um decorador que transforma um método em um atributo de acesso controlado.
 - Ele permite encapsular o acesso a atributos internos (normalmente precedidos por _) de forma elegante e sem alterar a interface do objeto.
 - Permite ler e modificar atributos como se fossem públicos, mas com validações por trás.
- Melhora a legibilidade do código em comparação com métodos get_ e set_.

Revisão de Programação Orientada a Objetos com Python

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

    @property
    def titular(self):
        return self._titular

    @titular.setter
    def titular(self, novo_titular):
        if isinstance(novo_titular, str) and novo_titular.strip() != "":
            self._titular = novo_titular
        else:
            print("Nome do titular inválido. Deve ser uma string não vazia.")
```


Revisão de Programação Orientada a Objetos com Python

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

    @property
    def titular(self):
        return self._titular

    @titular.setter
    def titular(self, novo_titular):
        if isinstance(novo_titular, str) and novo_titular.strip() != "":
            self._titular = novo_titular
        else:
            print("Nome do titular inválido. Deve ser uma string não vazia.")
```

```
conta = Conta("Lucas", 1000)
print(conta.titular)
```

```
conta.titular = "Maria"
print(conta.titular)
```

```
conta.titular = ""
```



```
Lucas
Maria
Nome do titular inválido. Deve ser uma string não vazia.
```

Revisão de Programação Orientada a Objetos com Python

- Polimorfismo é um dos pilares da programação orientada a objetos e refere-se à capacidade de objetos diferentes responderem a uma mesma interface (ou método), mesmo que de formas distintas.
 - Métodos distintos de saque para cada tipo de conta (ex: Poupança, ContaCorrente, etc.)
- No Python, isso significa que diferentes classes podem implementar um mesmo método com comportamentos específicos. Assim, podemos tratar objetos de classes distintas de forma uniforme, tornando o código mais genérico, reutilizável e flexível.

Revisão de Programação Orientada a Objetos com Python

- Imagine que estamos desenvolvendo um sistema de pagamentos que pode receber diferentes formas de pagamento: cartão de crédito, boleto e Pix.
- Cada forma de pagamento processa o pagamento de maneira diferente, mas todas devem ter um método chamado pagar.

Soluções de pagamentos



Crédito



Débito



Boleto



PIX

Revisão de Programação Orientada a Objetos com Python

```
class Pagamento:
    def pagar(self, valor):
        raise NotImplementedError("Este método deve ser implementado pelas subclasses.")

class CartaoCredito(Pagamento):
    def pagar(self, valor):
        print(f"Pagando R${valor:.2f} com cartão de crédito.")

class Boleto(Pagamento):
    def pagar(self, valor):
        print(f"Gerando boleto no valor de R${valor:.2f}.")

class Pix(Pagamento):
    def pagar(self, valor):
        print(f"Transferindo R${valor:.2f} via Pix.")
```

Revisão de Programação Orientada a Objetos com Python

```
def processar_pagamento(pagamento: Pagamento, valor):  
    pagamento.pagar(valor)  
  
pagamentos = [  
    CartaoCredito(),  
    Boleto(),  
    Pix()  
]  
  
for forma in pagamentos:  
    processar_pagamento(forma, 150.00)
```



Pagando R\$150.00 com cartão de crédito.
Gerando boleto no valor de R\$150.00.
Transferindo R\$150.00 via Pix.

Revisão de Programação Orientada a Objetos com Python

```
def processar_pagamento(pagamento: Pagamento, valor):  
    pagamento.pagar(valor)  
  
pagamentos = [  
    CartaoCredito(),  
    Boleto(),  
    Pix()  
]  
  
for forma in pagamentos:  
    processar_pagamento(forma, 150.00)
```

A anotação pagamento:
Pagamento é uma sugestão de tipo
(type hint) que indica que esse
parâmetro deve seguir a classe
base Pagamento.



Pagando R\$150.00 com cartão de crédito.
Gerando boleto no valor de R\$150.00.
Transferindo R\$150.00 via Pix.

Revisão de Programação Orientada a Objetos com Python

```
def processar_pagamento(pagamento: Pagamento, valor):  
    pagamento.pagar(valor)  
  
pagamentos = [  
    CartaoCredito(),  
    Boleto(),  
    Pix()  
]  
  
for forma in pagamentos:  
    processar_pagamento(forma, 150.00)
```

O Python é uma linguagem de tipagem dinâmica e duck typing, ou seja, o importante não é o tipo declarado, mas se o objeto tem o comportamento esperado (neste caso, se tem um método pagar).



Pagando R\$150.00 com cartão de crédito.
Gerando boleto no valor de R\$150.00.
Transferindo R\$150.00 via Pix.

Revisão de Programação Orientada a Objetos com Python

- Cada classe implementa o método pagar de forma diferente.
- A função processar_pagamento não precisa saber qual é a forma de pagamento.
- Isso é polimorfismo: usar o mesmo método (pagar) com comportamentos diferentes, dependendo do objeto.

Revisão de Programação Orientada a Objetos com Python

- Uma classe abstrata é uma classe que não pode ser instanciada diretamente.
- Ela serve como modelo ou contrato para outras classes, e normalmente define métodos que devem ser implementados pelas suas subclasses.
 - Em outras palavras: ela define o que deve ser feito, mas não como será feito.
- Por que usar classe abstrata?
 - Garante que todas as subclasses implementem certos métodos.
 - Fornece uma estrutura comum para objetos relacionados.
 - Ajuda a aplicar o polimorfismo com segurança.

Revisão de Programação Orientada a Objetos com Python

- Voltando ao exemplo anterior:

```
class Pagamento:
    def pagar(self, valor):
        raise NotImplementedError("Este método deve ser implementado pelas subclasses.")

class CartaoCredito(Pagamento):
    def pagar(self, valor):
        print(f"Pagando R${valor:.2f}")

class Boleto(Pagamento):
    def pagar(self, valor):
        print(f"Gerando boleto no valor de R${valor:.2f}.")

class Pix(Pagamento):
    def pagar(self, valor):
        print(f"Transferindo R${valor:.2f} via Pix.")
```

Usar `raise NotImplementedError` simula uma classe abstrata, mas não impede que a classe base seja instanciada nem obriga formalmente a sobrescrita dos métodos.

Revisão de Programação Orientada a Objetos com Python

- Python oferece o módulo abc (Abstract Base Classes) para definir classes abstratas.

```
from abc import ABC, abstractmethod
```

Classe abstrata
herda de ABC

```
class Pagamento(ABC):
```

```
    @abstractmethod
```

Método abstrato

```
    def pagar(self, valor):  
        pass
```

```
    def log_transacao(self, valor):
```

```
        print(f"[LOG] Transação registrada no valor de R${valor:.2f}")
```

Método concreto

Revisão de Programação Orientada a Objetos com Python

- Implementando os métodos abstratos nas subclasses:

```
class CartaoCredito(Pagamento):
    def pagar(self, valor):
        print(f"Pagando R${valor:.2f} com cartão de crédito.")
        self.log_transacao(valor)

class Boleto(Pagamento):
    def pagar(self, valor):
        print(f"Gerando boleto no valor de R${valor:.2f}.")
        self.log_transacao(valor)

class Pix(Pagamento):
    def pagar(self, valor):
        print(f"Transferindo R${valor:.2f} via Pix.")
        self.log_transacao(valor)
```

Revisão de Programação Orientada a Objetos com Python

```
def processar_pagamento(pagamento: Pagamento, valor):  
    pagamento.pagar(valor)  
  
pagamentos = [  
    CartaoCredito(),  
    Boleto(),  
    Pix()  
]  
  
for forma in pagamentos:  
    processar_pagamento(forma, 150.00)
```




```
Pagando R$150.00 com cartão de crédito.  
[LOG] Transação registrada no valor de R$150.00  
Gerando boleto no valor de R$150.00.  
[LOG] Transação registrada no valor de R$150.00  
Transferindo R$150.00 via Pix.  
[LOG] Transação registrada no valor de R$150.00
```


Revisão de Programação Orientada a Objetos com Python

```
def processar_pagamento(pagamento: Pagamento, valor):  
    pagamento.pagar(valor)  
  
pagamentos = [  
    CartaoCredito(),  
    Boleto(),  
    Pix()  
]  
  
for forma in pagamentos:  
    processar_pagamento(forma, 150.00)
```

Tentar instanciar uma classe abstrata (ex: `Pagamento()`), resulta em erro: “Can't instantiate abstract class `Pagamento` with abstract method `pagar`”



```
Pagando R$150.00 com cartão de crédito.  
[LOG] Transação registrada no valor de R$150.00  
Gerando boleto no valor de R$150.00.  
[LOG] Transação registrada no valor de R$150.00  
Transferindo R$150.00 via Pix.  
[LOG] Transação registrada no valor de R$150.00
```

Revisão de Programação Orientada a Objetos com Python

- Em Python, classe abstrata + métodos abstratos = interface (se não houver nenhum método implementado/concreto).
- Python não tem uma palavra-chave específica chamada interface, mas podemos criar interfaces usando classes abstratas com somente métodos abstratos.
- Interfaces definem o que uma classe deve fazer.
- Essas interfaces funcionam por meio do módulo ABC (Abstract Base Classes).

Revisão de Programação Orientada a Objetos com Python

- Exemplo de interface em Python:

```
from abc import ABC, abstractmethod

class IPagamento(ABC):
    @abstractmethod
    def pagar(self, valor):
        pass
```


Revisão de Programação Orientada a Objetos com Python

- Implementações da Interface:

```
class CartaoCredito(IPagamento):  
    def pagar(self, valor):  
        print(f"Pagando R${valor:.2f} com cartão de crédito.")  
  
class Boleto(IPagamento):  
    def pagar(self, valor):  
        print(f"Gerando boleto no valor de R${valor:.2f}.")  
  
class Pix(IPagamento):  
    def pagar(self, valor):  
        print(f"Transferindo R${valor:.2f} via Pix.")
```

Revisão de Programação Orientada a Objetos com Python

```
def processar_pagamento(pagamento: IPagamento, valor):  
    pagamento.pagar(valor)  
  
pagamentos = [  
    CartaoCredito(),  
    Boleto(),  
    Pix()  
]  
  
for forma in pagamentos:  
    processar_pagamento(forma, 150.00)
```



Pagando R\$150.00 com cartão de crédito.
Gerando boleto no valor de R\$150.00.
Transferindo R\$150.00 via Pix.

Exercícios

1. Aplique ao projeto desenvolvido na lista de exercícios 01 o conceito de encapsulamento usando @property.
2. Substitua a Classe Conta por uma ClasseAbstrata, não permitindo a sua instanciación.
3. Crie uma interface chamada Imposto que defina um método calcular_imposto(), que deve ser implementado tanto pela ContaCorrente quanto pela Poupanca. O imposto para a conta corrente será 0,5% do saldo, e para a poupança, será 0,3% do saldo.

Dúvidas



Outros conceitos de POO serão revisitados à medida que explorarmos o uso de frameworks de programação web.

PROGRAMAÇÃO WEB II

Curso Técnico Integrado em Informática
Lucas Sampaio Leite

