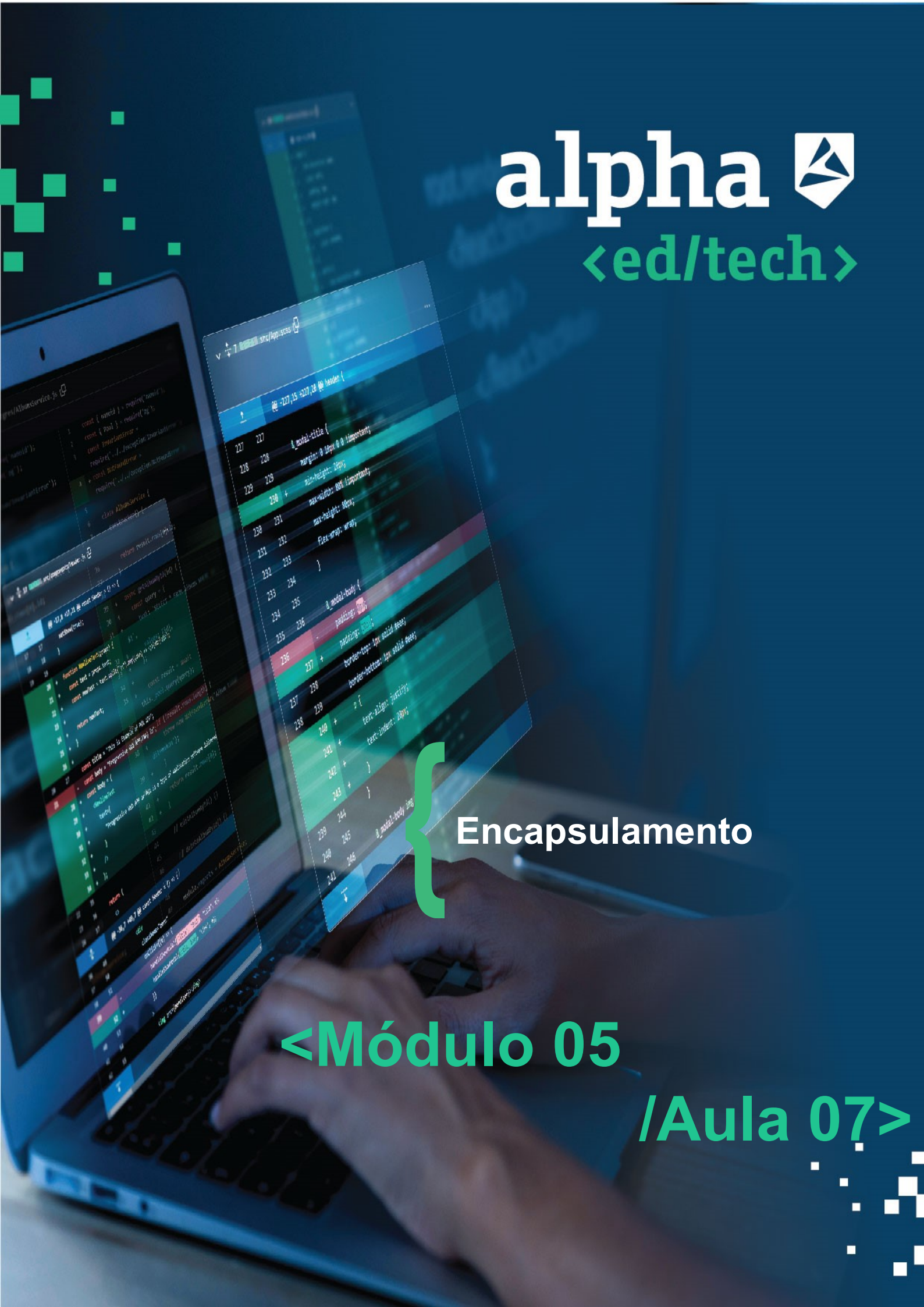
alpha
<ed/tech>
Encapsulamento
<Módulo 05
/Aula 07>

Encapsulamento

Motivação

•Existem dados que podem quebrar o funcionamento de uma classe. E se alguém **alterar eles diretamente**, sem os cuidados que o desenvolvedor planejou para o comportamento da classe;

•**Encapsulamento** permite **evitar** esse tipo de problema;

•A maioria do que é implementado em Python é apenas CONVENÇÃO.

Na programação orientada a objetos (POO), uma **classe é uma estrutura fundamental que representa um modelo para criar objetos (instanciar um Objeto a partir da Classe)**. Ela funciona como um plano ou projeto que define atributos (características) e métodos (comportamentos) que os objetos desse tipo terão.

Python é uma linguagem Orientada a Objetos e as **variáveis são objetos**. Em Python, uma classe é uma estrutura que permite organizar e estruturar dados e funcionalidades relacionadas. Classe serve como um modelo para criar objetos, que são instâncias dessa classe.

Definindo uma Classe

```python
class Carro:
    def __init__(self, modelo, cor):
        self.modelo = modelo
        self.cor = cor
```

Aqui, criamos a classe `Carro` com atributos `modelo` e `cor`. O método `__init__` é um construtor que inicializa os atributos quando um objeto é criado.

Criando Objetos (Instâncias) a partir de uma Classe

```python
meu_carro = Carro("Sedan", "Prata")
```

Agora, `meu_carro` é uma instância da classe `Carro` com modelo "Sedan" e cor "Prata".

Atributos e Métodos

```python
class Carro:
    def __init__(self, modelo, cor):
        self.modelo = modelo
        self.cor = cor
    def acelerar(self):
        print(f"O carro {self.modelo} está acelerando!")
meu_carro = Carro("Sedan", "Prata")
meu_carro.acelerar()
```

Encapsulamento

`acelerar` é um método da classe que pode ser chamado em instâncias.
No exemplo, imprime uma mensagem.

Visibilidade dos Atributos:

```python exemplo050702.py
class Carro:
    def __init__(self, modelo, cor):
        self._modelo = modelo  # Atributo protegido
        self.__cor = cor       # Atributo privado
    def obter_cor(self):
        return self.__cor

meu_carro = Carro("Sedan", "Prata")
print(meu_carro.obter_cor())
```

O uso de um underscore ou dois indica o nível de encapsulamento (protegido ou privado).

Membro protegido

•Um membro protegido é um atributo ou método da classe que se deseja esconder do usuário, mas não de quem herda a classe;
•Convenção: utilizar `_` no início do nome do membro para indicar que ele é protegido;
•Não deveria ser acessado e nem modificado no escopo fora da classe ... mas o Python permite acessar e modificar;
•Pode ser acessado e modificado na classe que herda esta classe como pai.

```python exemplo050703.py
class Funcionario:
    def __init__(self, id:int, nome:str) -> None:
        self.id = id
        self._nome = nome
```

```
x = Funcionario(1, "Paulo Marcotti")
print(x.id)
print(x._nome)
```

Encapsulamento

Membro protegido na classe que herda

```python exemplo050704.py
class Funcionario:
    def __init__(self, id:int, nome:str) -> None:
        self.id = id
        self._nome = nome

class Chefe(Funcionario):
    def mostrar_bonus(self) -> None:
        print(f"{self._nome} vai receber R$ 1000 de bônus!")

x = Funcionario(1, "Paulo Marcotti")
print(x.id)
print(x._nome) # acessado protected!!!

y = Chefe(2, "Kenji Taniguchi")
y.mostrar_bonus()
```

Membro privado (private)

•Um membro privado é um atributo ou método da classe que você deseja esconder do usuário e de quem herda a classe;
•Convenção: utiliza `__` no início do nome do membro para indicar;
•Não deveria ser acessado e nem modificado de fora do escopo da classe, "mas pode" (de forma menos convencional);
•Não deveria ser acessado e modificado dentro do escopo da classe que herda, "mas pode" (também, de forma menos convencional);
•É menos convencional pois o Python renomeia membros que o nome comece com `__` para `_NomeDaClasse__nomeDoMembro` quando usado fora da classe que declarou o membro privado;
•Observe que o MyPy reclama porque esse atributo não aparece explicitamente no código, ou seja, não tente acessá-lo!.

```python exemplo050705.py
class Funcionario:
    def __init__(self, id:int, nome:str) -> None:
        self.id = id
        self.__nome = nome
x = Funcionario(1, "Paulo Marcotti")
print(x.id)
# Forma menos convencional
print(x._Funcionario__nome)
# "Funcionario" has no attribute "_Funcionario__nome" [attr-defined]
```

Encapsulamento

Membro privado na classe que herda

Note que o MyPy reclama porque esse atributo não aparece explicitamente no código, ou seja, não tente acessá-lo!.

```python exemplo050706.py
class Funcionario:
    def __init__(self, id:int, nome:str) -> None:
        self.id = id
        self._nome = nome

class Chefe(Funcionario):
    def mostrar_bonus(self) -> None:
        print(f"{self._nome} vai receber R$ 1000 de bônus!")

x = Funcionario(1, "Paulo Marcotti")
print(x.id)print(x._nome) # acessado protected!!!

y = Chefe(2, "Kenji Taniguchi")
y.mostrar_bonus()
```

Como tentar proteger um atributo

•Nada de convenções, não confio no meu usuário (desenvolvedor que vai utilizar minha classe)!
•Usar o decorator @property
•Permite definir getters e setters.

```python exemplo050707.py
from typing import NoReturn
class Funcionario:
    def __init__(self, id:int, nome:str) -> None:
        self.id = id
        self._nome = nome

    @property
    def nome(self) -> str:
        return self._nome

    @nome.getter
    def nome(self) -> NoReturn:
        raise Exception("Não pode acessar o nome!")
x = Funcionario(1, "Paulo Marcotti")
print(x.id)
# Forma menos convencional
print(x._nome)
# Exception: Não pode acessar o nome!
print(x.nome)
```

Encapsulamento

Encapsulamento

Alinhado à visibilidade de atributos (propriedades do objeto) e métodos (comportamentos do objeto) existe a técnica de esconder a complexidade da implementação, de tal forma que o desenvolvedor que for utilizar a classe especificada não precisa conhecer a implementação.

Classes em Python oferecem uma estrutura poderosa para organizar código, promovendo a reutilização e facilitando a manutenção.

O encapsulamento na orientação a objetos refere-se à prática de esconder a implementação interna de um objeto e expor apenas a interface necessária para interagir com ele.

Isso significa que detalhes complexos de implementação são mantidos ocultos do usuário ou de outros desenvolvedores que utilizam a classe. O encapsulamento é alcançado principalmente por meio do uso de modificadores de acesso, como público, privado e protegido, para controlar o acesso aos membros da classe (atributos e métodos).

Um exemplo prático de encapsulamento é o uso de métodos getters e setters para acessar e modificar os atributos de uma classe.

Considere a seguinte classe em Python:

```python exemplo050708.py
class ContaBancaria:
    def __init__(self, saldo):
        self._saldo = saldo  # Atributo protegido
    def get_saldo(self):
        return self._saldo

    # Método getter
    def depositar(self, valor):
        if valor > 0:
            self._saldo += valor
    def sacar(self, valor):
        if valor > 0 and valor <= self._saldo:
            self._saldo -= valor
```

Neste exemplo, o atributo `_saldo` é marcado como protegido (convenção de nomenclatura com um sublinhado), indicando que não deve ser acessado diretamente de fora da classe. Em vez disso, métodos como `get_saldo`, `depositar` e `sacar` são fornecidos para interagir com o saldo da conta.

Ao encapsular o atributo `_saldo` e fornecer métodos específicos para interação, a implementação interna torna-se oculta, permitindo que outros desenvolvedores utilizem a classe sem precisar entender ou manipular diretamente os detalhes de implementação. Isso promove a segurança e a manutenção do código, uma vez que as mudanças internas podem ser feitas sem afetar os usuários externos da classe.