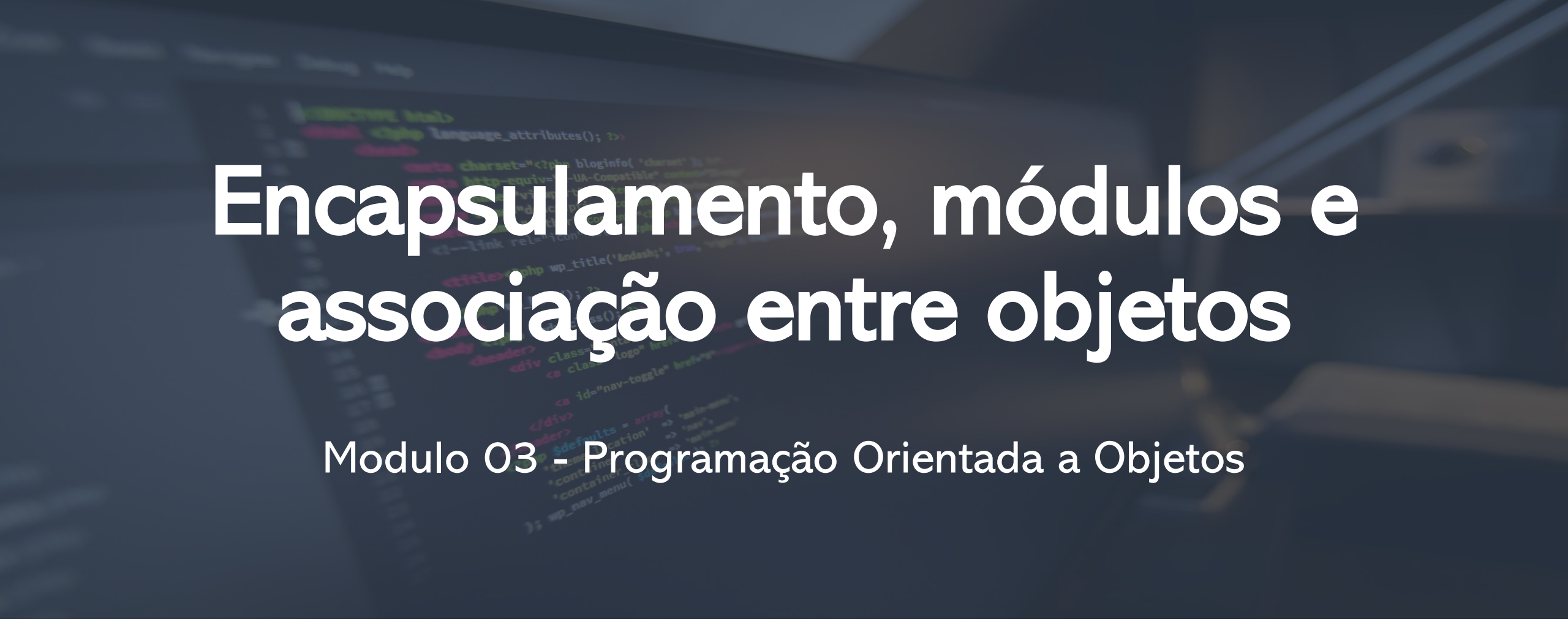


Encapsulamento, módulos e associação entre objetos

Modulo 03 - Programação Orientada a Objetos

ANTÔNIO DAVID VINISKI
antonio.david@pucpr.br
PUCPR

Agenda

- Encapsulamento
  - Visibilidade public
  - Visibilidade protected
  - Visibilidade private
- Métodos Getters e Setters
- Associação entre objetos
  - Agregação
  - Composição

Encapsulamento

Encapsulamento

- Permite que certas características ou propriedades dos objetos de uma classe não possam ser vistas ou modificadas externamente
- Ocultam-se as características internas do objeto.
  - outras classes só podem acessar os atributos de uma classe invocando os métodos **públicos**.
  - **restringe a visibilidade do objeto, mas facilita o reuso**.

Princípio de Encapsulamento

- Uma classe deve ocultar os seus detalhes de implementação das classes que a utilizam.
- Uma classe deve especificar quais dos seus membros (atributos e métodos) são visíveis e acessíveis por outras classes.
  - **atributo**: um dado do objeto.
  - **método**: uma operação (função) aplicável ao objeto.
- O conjunto de métodos de uma classe especificados como acessíveis por outras classes constitui a *interface* dessa classe.

Qualificadores de Visibilidade e Proteção de Membros

- Usados para definir a visibilidade de cada membro (atributo ou método) de uma classe, isto é, para proteger o acesso aos membros de uma classe.
- Visibilidade e qualificadores:
  - Público - **`public`**
    - Atributo ou método da classe pode ser acessado por todas as demais entidades do sistema
  - Protegido - **`protected`**
    - Atributo ou método da classe pode ser acessado somente por classes da mesma hierarquia e mesmo pacote
  - Privado - **`private`**
    - Atributo ou método da classe pode ser acessado somente por métodos da própria classe

Qualificadores Python - public

- Os membros de uma classe que são declarados públicos são facilmente acessíveis de qualquer parte do programa.
- Todos os membros de dados e membro de métodos de uma classe são públicos por padrão.

```
class Classe:
    def __init__(self, text: str, boleano: bool, inteiro: int,decimal: float):
        self.text: str = text
        self.boleano: bool = boleano
        self.inteiro: int = inteiro
        self.decimal: float = decimal
```

Qualificadores Python - public

- Podemos modificar os valores dos atributos diretamente

```
classe = Classe("Texto", True, 22, 12.79)
classe.text  = "Novo Texto"
classe.booleano  = False
```

- Na orientação a objetos, a convenção é que possamos modificar os valores dos atributos somente pelos métodos definidos especificamente para isso.
  - No caso dos atributos públicos, isso não acontece

Qualificadores Python - protected

- Os membros de uma classe declarados protegidos só são acessíveis a uma classe derivada dela.
- Os membros de dados de uma classe são declarados protegidos adicionando um único símbolo de sublinhado '_' antes do membro de dados dessa classe.

```
class ClasseProtected:
    def __init__(self, text: str, boleano: bool, inteiro: int,decimal: float):
        self._text: str = text
        self._boleano: bool = boleano
        self._inteiro: int = inteiro
        self._decimal: float = decimal
```

Qualificadores Python - private

- Os membros de uma classe declarados privados são acessíveis apenas dentro da classe; o modificador de acesso privado é o modificador de acesso mais seguro.
- Os membros de dados de uma classe são declarados privados adicionando um símbolo de sublinhado duplo '__' antes do membro de dados dessa classe.

```
class ClassePrivate:
    def __init__(self, text: str, boleano: bool, inteiro: int,decimal: float):
        self.__text: str = text
        self.__boleano: bool = boleano
        self.__inteiro: int = inteiro
        self.__decimal: float = decimal
```

Tipos de Acesso

- A escolha dos tipos de acesso é muito importante na POO.
  - Define o escopo dos atributos e métodos.
- Em geral, atributos são declarados privados
  - Métodos da própria classe são responsáveis por modificar e recuperar o estado dos atributos.
    - Tais métodos são públicos.
    - *Setters* e *getters*.
    - Garantem a estabilidade e segurança.
    - *Information-hiding.*

Módulos

Módulos

- Ao sair e entrar de novo no interpretador Python, as definições anteriores (funções e variáveis) são perdidas.
- Portanto, se quiser escrever um programa maior, será mais eficiente usar um editor de texto para preparar as entradas para o interpretador, e executá-lo usando o arquivo como entrada.
- Isso é conhecido como criar um script.
- Se o programa se torna ainda maior, é uma boa prática dividi-lo em arquivos menores, para facilitar a manutenção.
- Também é preferível usar um arquivo separado para uma função que você escreveria em vários programas diferentes, para não copiar a definição de função em cada um deles.

Módulos II

- Para permitir isso, Python tem uma maneira de colocar as definições em um arquivo e então usá-las em um script ou em uma execução interativa do interpretador.
- Tal arquivo é chamado de módulo; definições de um módulo podem ser importadas para outros módulos, ou para o módulo principal.
- Um módulo é um arquivo contendo definições e instruções Python.
- O nome do arquivo é o nome do módulo acrescido do sufixo .py.
- Dentro de um módulo, o nome do módulo (como uma string) está disponível como o valor da variável global **__name__**.

Módulos - Exemplo Escola

- Supondo que o cadastro de novos cursos só possa ser realizado por uma escola
  - Precisamos criar uma classe Escola que será responsável por criar os cursos.
- Para aumentar a flexibilidade do programa, podemos ter tanto a classe curso quanto a classe escola em arquivos separados.
- Vamos também criar o nosso arquivo principal (main.py), que será responsável por criar uma instancia de Escola e posteriormente criar os Cursos a partir da escola.

Módulos – criando um curso

- Como é responsabilidade da Escola criar um Curso, no arquivo escola.py, devemos ter acesso a classe Curso
  - Para isso, no início do arquivo escola.py, importamos do modulo curso para ter acesso a classe Curso

```
import curso
```

- Para acessarmos a classe Curso, basta utilizarmos o nome do módulo, seguido de um ponto (.) e o nome da classe.

```
import curso

poo = curso.Curso("Programação Orientada a Objetos", "07/10/2023", "Antonio David Viniski",24,1000.00)
```

Módulos - Exemplo Escola

- Um módulo pode conter tanto instruções executáveis quanto definições de funções e classes.
- Essas instruções servem para inicializar o módulo.
- Eles são executados somente na primeira vez que o módulo é encontrado em uma instrução de importação.
- Existe uma variante da instrução import que importa definições de um módulo diretamente para o espaço de nomes do módulo importador. Por exemplo:

```
from curso import Curso
```

- Isso não coloca o nome do módulo de onde foram feitas as importações no espaço de nomes local somente a classe Curso estará acessível, se houvesse outras classes, elas não poderiam ser utilizadas.
- Existe ainda uma variante que importa todos os nomes definidos em um módulo:

```
from curso import *
```

Exercício 1

- Considerando a aplicação da conta, crie uma classe Banco que é responsável por criar e armazenar as contas do banco.
  - Separe a aplicação em três arquivos, banco.py, conta.py e main.py
  - O arquivo banco.py implementa a classe Banco, que possui os atributos privados **nome** e **codigo** e os métodos **criaConta** e **listaContas**
  - O arquivo main.py será responsável por instanciar o objeto do tipo Banco e chamar os métodos **criaConta** e **listaContas**

Associação entre Classes

Tipos de associação

- Agregação
  - O objeto da classe associada existe de forma independente
- Composição
  - A existência de uma classe depende de outra classe

Tipos de associação - Agregação

- Supondo que precisamos de mais informações do professor do Curso, como nome, formação, e-mail.
  - Adicionar essas informações na classe Curso não é viável, pois toda alteração dos dados de professor exigiria uma alteração em todos os cursos que esse professor ministra
  - Com isso, torna-se necessária a criação da classe Professor
- Vamos implementar a agregação de Professor em curso.

Tipos de associação – Agregação - solução

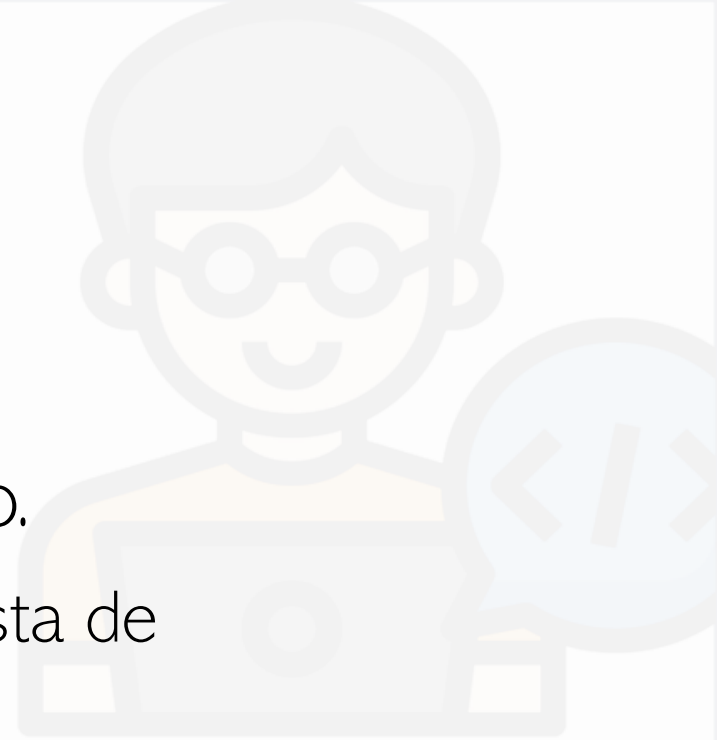
```
class Professor:
    def __init__(self, nome: str, formacao: str, email: str):
        self.__nome: str = nome
        self.__formacao: str = formacao
        self.__email: str = email
```

```
class Curso:
    def __init__(self, nome: str, data: str, professor: Professor, horas: int, valor: float):
        self.__nome: str = nome
        self.__data: str = data
        self.__professor: Professor = professor
        self.__horas: int = horas
        self.__valor: float = valor

antonio = Professor("Antônio David Viniski", "Doutor em Informática", "antonio.david@pucpr.br")

curso = Curso("Programação Orientada a Objetos", "07/10/2023", antonio, 20, 1000.00)
curso2 = Curso("Banco de Dados", "31/01/2024", antonio, 24, 1200.00)
```

Tipos de associação - Composição



- Agora vamos pensar na ementa de um curso.
  - Todo curso possui uma ementa, ou seja, uma lista de conteúdos que serão abordados
  - Toda ementa é criada exclusivamente para um curso, ou seja, ela vai existir somente associada a um curso
  - A ementa tem uma data de criação e também a data de modificação

Tipos de associação – Composição - Ementa

- Vamos criar a ementa do curso de Programação Orientada a Objetos
  - Precisamos de um método para adicionar itens a ementa
  - Precisamos também remover itens da ementa
  - Precisamos também atualizar a data de modificação toda vez que uma alteração é feita na ementa.