Orientação a Objetos com Python

Introdução

WOMAKERS CODE

Clarisse Simões

Software Engineer @ Microsoft

https://linktr.ee/clarissesimoes

Orientação a Objetos com Python

Conteúdo do Módulo

1. Namespaces, Pacotes e Escopos
2. Classes e Objetos
3. Entendendo o contexto de Orientação a Objetos
4. Construtores e Destrutores
5. Atributos de visibilidade e encapsulamento
6. Herança
7. Classes abstratas e a biblioteca ABC
8. Pseudo-Interfaces
9. Lidando com erros e exceções

Orientação a Objetos com Python

Material Suplementar

1. Slides & Quizzes – Plataforma
2. GitHub: WoMakersCode/back-end-python (github.com)
3. Listas de Exercicícios
4. Aula de Mentoria ao vivo

Orientação a Objetos com Python

Módulos e *Namespaces*

WOMAKERS CODE

Módulos e *Namespaces*

- Módulos são locais onde você define os nomes e funções que quer quer fiquem visíveis para o resto do sistema.
- Falando tecnicamente, um módulo é um “espaço que serve para a declaração de nomes”, ou seja, um *namespace*.
- Em um módulo podem ser definidos componentes reutilizáveis em outros arquivos Python. Ex: variáveis, funções, classes, etc.
- Variáveis que são definidas dentro de um *namespace* são chamadas de **atributos.**

meu_modulo.py

```
versao = ‘0.1.1’

def mostrar_mensagem(texto):
    print(texto)
```

app.py

```
# importar um módulo na forma de um namespace à parte
import meu_modulo

mensagem = 'Usando versão ' + meu_modulo.versao
meu_modulo.mostrar_mensagem(mensagem)
```

namespace

Orientação a Objetos com Python

Pacotes

WOMAKERS CODE

Instalando Pacotes em seu ambiente

```
# Instalar um pacote individualmente
pip install colorama

# Instalar uma lista de pacotes
pip install -r requirements.txt

# requirements.txt
colorama
```

Orientação a Objetos com Python

Escopos

WOMAKERS CODE

Escopos de variáveis

```
variavel_global = 'global teste'
def minha_funcao():
  global variavel_global
  variavel_local = 'local_teste'
  variavel_global = 'outro valor'
```

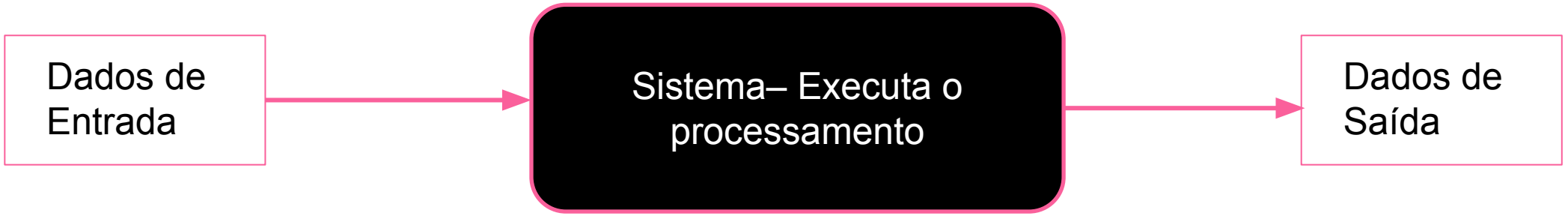
- Variáveis declaradas dentro de uma função não podem ser acessadas fora dela. Neste caso, dizemos que a variável é **local** porque ela só existe dentro do seu escopo, que é a delimitado pela função onde é declarada.
- Variáveis declaradas fora de qualquer função são chamadas de **globais**. Elas se encontram em um escopo que é acessível em qualquer parte do seu script e também por outros módulos.
- Uma aplicação comum de variáveis globais é o armazenamento de valores constantes no programa, que ficam acessíveis para a todas as funções.
- Para alterar variáveis globais dentro de funções, precisamos indicar a função que estamos querendo alterar a variável do escopo global. Caso contrário, outra variável de mesmo nome é criada dentro do escopo da função e é alterada apenas localmente.

Orientação a Objetos com Python

Entendendo o contexto de Orientação a Objetos

WOMAKERS CODE

Modelagem de um sistema orientado a objetos



WOMAKERS CODE

Modelagem de um sistema orientado a objetos

- A programação orientada a objetos cria **modelos** do mundo em que os dados são operados.
- Os modelos possuem classes que representam **atores** do mundo real, e como eles interagem entre si.
- Durante a fase de modelagem, você examina uma descrição de um **domínio** e tenta analisar os atores e as **regras de negócio**.
- **Atores** atuam no domínio e executam uma ação. Por exemplo, um carro (ator) acelera (ação).
- Atores costumam atuar sobre de **dados**, que são a entrada necessária para executar uma ação.
- O que os atores fazem para executar essa ação é o **comportamento**.
- Atores podem interagir uns com os outros para chegar a um resultado tangível.
- O resultado da atuação e interação entre os atores sobre os dados gera uma **saída** para o nosso programa.

Modelagem é o processo de identificar os **atores**, os **dados** necessários e o tipo de **interação** que está ocorrendo. Para modelar um sistema, é necessário conhecer suas **regras de negócio**.

Exemplo Estacionamento: Requisitos

- O estacionamento é um pátio de apenas um andar. Ele possui 10 vagas.
- Há 5 vagas para carros e 5 vagas para motos. Vagas para carro são maiores do que as vagas para motos.
- Carros e motos são identificados por suas placas.
- Vagas são identificadas por um número. Cada vaga tem um número identificador único.
- Carros só podem ser estacionado em vagas específicas para carros.
- Motos preferencialmente são estacionadas em vagas de motos, mas se não houver mais vagas exclusivas de motos disponíveis, motos podem ser estacionadas em vagas de carros.
- É preciso ter controle sobre qual carro está em qual vaga para agilizar a saída quando o dono vem buscar.
- É preciso saber o número de vagas livres de carro e de moto para que o estacionamento saiba se pode novos carros e motos.

Exemplo Estacionamento

Atores

Estacionamento

Vaga

Carro

Moto

Exemplo Estacionamento: Requisitos

- O estacionamento é um pátio de apenas um andar. Ele possui 10 vagas.
- Há 5 vagas para carros e 5 vagas para motos. Vagas para carro são maiores do que as vagas para motos.
- Carros e motos são identificados por suas placas.
- Vagas são identificadas por um número. Cada vaga tem um número identificador único.
- Carros só podem ser estacionado em vagas específicas para carros.
- Motos preferencialmente são estacionadas em vagas de motos, mas se não houver mais vagas exclusivas de motos disponíveis, motos podem ser estacionadas em vagas de carros.
- É preciso ter controle sobre qual carro está em qual vaga para agilizar a saída quando o dono vem buscar.
- É preciso saber o número de vagas livres de carro e de moto para que o estacionamento saiba se pode novos carros e motos.

Exemplo Estacionamento

Estacionamento

vagas_de_carro
vagas_de_moto
carro_para_vaga
moto_para_vaga
total_vagas_livres_carro
total_vagas_livres_moto

Vaga

id
tipo

Carro

placa

Moto

placa

Exemplo Estacionamento: Requisitos

- O estacionamento é um pátio de apenas um andar. Ele possui 10 vagas.
- Há 5 vagas para carros e 5 vagas para motos. Vagas para carro são maiores do que as vagas para motos.
- Carros e motos são identificados por suas placas.
- Vagas são identificadas por um número. Cada vaga tem um número identificador único.
- Carros só podem ser estacionados em vagas específicas para carros.
- Motos preferencialmente são estacionadas em vagas de motos, mas se não houver mais vagas exclusivas de motos disponíveis, motos podem ser estacionadas em vagas de carros.
- É preciso ter controle sobre qual carro está em qual vaga para agilizar a saída quando o dono vem buscar.
- É preciso saber o número de vagas livres de carro e de moto para que o estacionamento saiba se pode novos carros e motos.

Exemplo Estacionamento

Estacionamento

vagas_de_carro
vagas_de_moto
carro_para_vaga
moto_para_vaga
total_vagas_livres_carro
total_vagas_livres_moto

Vaga

id
tipo
livre

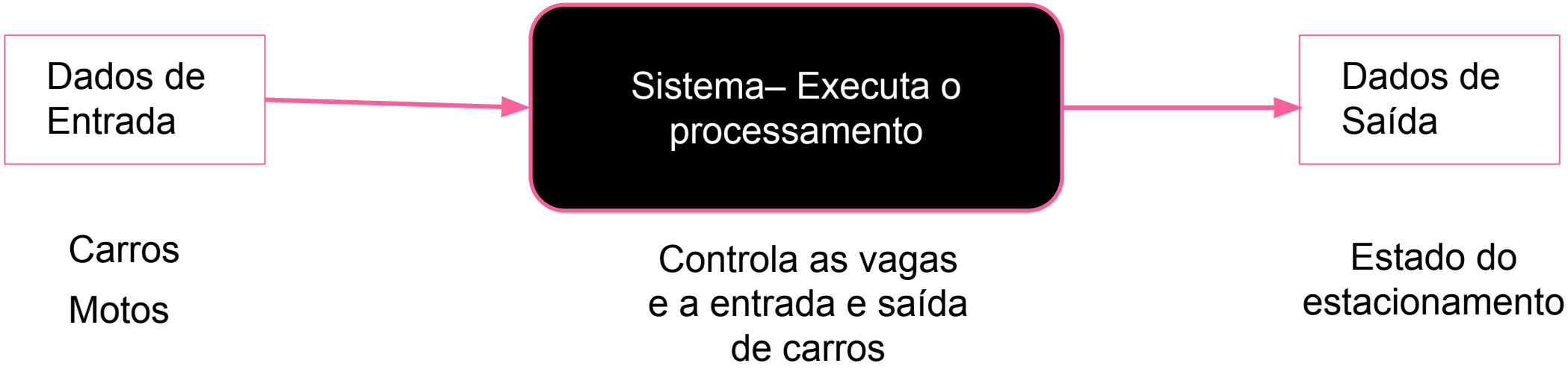
Carro

placa

Moto

placa

Modelagem de um sistema orientado a objetos



WOMAKERS CODE

Exemplo Estacionamento

Estacionamento

```
vagas_de_carro
vagas_de_moto
carro_para_vaga
moto_para_vaga
total_vagas_livres_carro
total_vagas_livres_moto

estacionar_carro(carro)
estacionar_moto(moto)
remover_carro(carro)
remover_moto(moto)
estado_do_estacionamento()
```

Vaga

```
id
tipo
livre
placa

ocupar()
desocupar()
```

Carro

```
placa
estacionado

estacionar()
sair_da_vaga()
```

Moto

```
placa
estacionado

estacionar()
sair_da_vaga()
```

Orientação a Objetos com Python

Atributos de visibilidade e encapsulamento

WOMAKERS CODE

Encapsulamento em Python

- Em Python, todas os atributos e métodos declarados em uma classe são **públicos**, ou seja, podem ser acessados por códigos externos à classe.
- **Isso não quer dizer que eles devam ser usadas por quem instancia um objeto daquela classe.**
- Alguns atributos e métodos só existem na classe para seu funcionamento interno. Se forem alterados, podem gerar mal funcionamento e bugs no código.

```
class Quadrado:
    def __init__(self, medida):
        self.altura = medida
        self.largura = medida

    def area(self):
        return altura * largura

quadrado = Quadrado(2)
quadrado.altura = 3 # não é mais um quadrado
```

Encapsulamento em Python

- Para indicar ao usuário quais os atributos e métodos que ele não deve alterar na classe, nós utilizamos **convenções** em seus nomes.
- Existem duas convenções que são utilizadas em Python para se iniciar nomes de métodos e atributos.

Atributos e métodos que têm seus nomes iniciados com **_ (*underscore*)** são **protegidos** e não devem ser acessados pelo mundo externo a não ser que o usuário saiba exatamente o que está fazendo, ou seja, ainda pode existir algum caso de uso em que faça sentido ter acesso a esse método/atributo, mas não é o mais comum.

Atributos e métodos que têm seus nomes iniciados com **__ (*underscore* duplo)** são **privados** e não devem ser acessados pelo mundo externo de forma nenhuma.

Propriedades

Propriedades nos dão acesso a variáveis que se parecem com atributos, mas na verdade usam métodos por trás dos panos.

```python
class Quadrado:
  def __init__(self, medida):
    self.altura = medida
    self.largura = medida

  @property
  def altura(self):
    return self.__medida

  @altura.setter
  def altura(self, valor):
    # executa algum tipo de validação
    self.__medida = valor

  @property
  def largura(self):
    return self.__medida

  @largura.setter
  def largura(self, valor):
    # executa algum tipo de validação
    self.__medida = valor

  def area(self):
    return self.largura * self.altura

quadrado = Quadrado(2)
quadrado.altura = 3
quadrado.largura = 2
```

altura e **largura** são propriedades criadas com o decorator **@property**. Esses métodos são chamados **getter** porque retornam o valor da propriedade.

Propriedades

Propriedades nos dão acesso a variáveis que se parecem com atributos, mas na verdade usam métodos por trás dos panos.

```python
class Quadrado:
  def __init__(self, medida):
    self.altura = medida
    self.largura = medida

  @property
  def altura(self):
    return self.__medida

  @altura.setter
  def altura(self, valor):
    # executa algum tipo de validação
    self.__medida = valor

  @property
  def largura(self):
    return self.__medida

  @largura.setter
  def largura(self, valor):
    # executa algum tipo de validação
    self.__medida = valor

  def area(self):
    return self.largura * self.altura

quadrado = Quadrado(2)
quadrado.altura = 3
quadrado.largura = 2
```

O método **setter** altera o valor da propriedade.

Propriedades

```python
class Quadrado:
  def __init__(self, medida):
    self.altura = medida
    self.largura = medida

  @property
  def altura(self):
    return self.__medida

  @altura.setter
  def altura(self, valor):
    # executa algum tipo de validação
    self.__medida = valor

  @property
  def largura(self):
    return self.__medida

  @largura.setter
  def largura(self, valor):
    # executa algum tipo de validação
    self.__medida = valor

  def area(self):
    return self.largura * self.altura

quadrado = Quadrado(2)
quadrado.altura = 3
quadrado.largura = 2
```

As propriedades podem ser acessadas como fossem atributos comuns, mas na verdade os métodos **getter** e **setter** estão sendo chamados.

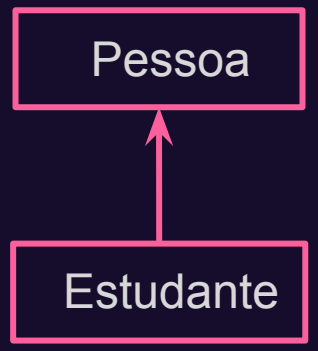
Orientação a Objetos com Python

Herança e *Mixins*

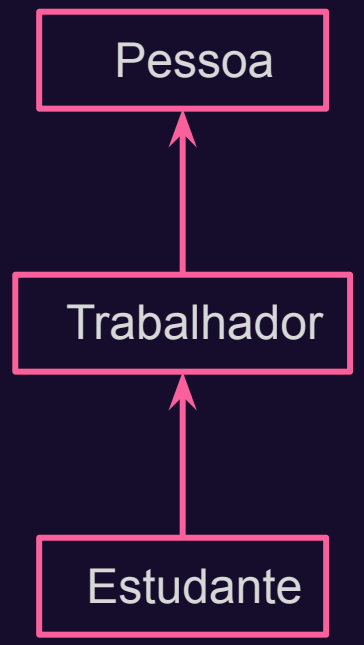
WOMAKERS CODE

Herança

- **Herança** é um conceito comum a todas as linguagens de programação que possuem orientação a objetos
- Com a herança, uma classe pode **herdar** os métodos e atributos de outra classe.
- A herança cria uma relação de uma classe **“é”** do mesmo tipo de outra. Por exemplo: um estudante é uma pessoa.
- A relação de “é” se opõe a outro tipo de relação, que é o **“tem”**. Por exemplo: um estudante tem livros.



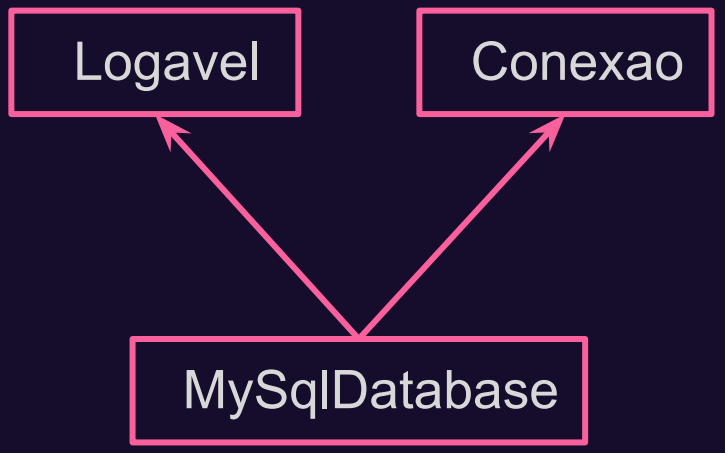
Benefícios do uso da Herança



- A herança permite modelar relações do mundo real.
- **Herança** é mais uma ferramenta que permite o **reuso de código**, já que o programador não precisa escrever as mesmas funcionalidades várias vezes em classes diferentes.
- **Herança** permite adicionar novas funcionalidades sem modificar uma classe que já existe, o que pode ser feito criando uma classe **derivada** (ou classe **filha**).
- **Herança** funciona de forma **transitiva**, ou seja, se a classe B herda todas as funcionalidades da classe A, todas as classes que herdam de B também ganham automaticamente todas as funcionalidades da classe A.
- **Exemplo**: um professor é um trabalhador, que por sua vez é uma pessoa. Logo, a classe **Professor** vai herdar todos os métodos e propriedades de **Trabalhador**, que por sua vez também herda de **Pessoa**.
- Em Python, todas as classes herdam implicitamente da classe **object**, que fornece métodos comuns que podem ser sobrescritos como o **__init__**, o **__str__** e outros.≈

Herança Múltipla (*Mixins*)

- Uma classe pode herdar de múltiplas classes em Python.
- Essa funcionalidade não existe em algumas outras linguagens de programação orientadas a objetos (C#, Java) por ser controversa
- Herança múltipla pode fazer o código ficar muito mais complicado do que o necessário.
- O caso de uso mais legítimo é na criação de um framework.
- Ao trabalhar com Django, vocês podem ver casos onde uma classe vai herdar de duas ou mais classes.



Orientação a Objetos com Python

Classes Abstratas e Biblioteca ABC

WOMAKERS CODE

Classes Abstratas e Biblioteca ABC

- Uma **classe abstrata** pode ser considerada como um modelo para criar outras classes.
- Uma classe abstrata precisa ter um ou mais métodos/propriedades abstratos que devem ser implementados pelas classes filhas.
- Python tem a biblioteca **ABC** (***Abstract Base Classes***) que fornece a funcionalidade de classes abstratas.
- A classe abstrata precisa herdar de ABC e métodos abstratos são marcados com o *decorator* **@abstractmethod**.
- As classes abstratas são usadas para definir a **API** de suas classes filhas.

```python
from abc import ABC, abstractmethod

class ClasseAbstrata(ABC):
  @abstractmethod
  def metodo_abstrato(self):
      pass

  @property
  @abstractmethod
  def propriedade_abstrata(self):
    pass

class ClasseDerivada(ClasseAbstrata):
  # sobrescrevendo o método abstrato
  def metodo_abstrato(self):
     print("Boa noite")

  # sobrescrevendo a propriedade abstrata
  @property
  def propriedade_abstrata(self):
     return 'Noite'
```