EAD

EDUCAÇÃO A DISTÂNCIA

UNIDADE IV

Pensamento Lógico Computacional com Python

Prof. MSc. Tarcísio Peres

Conteúdo da unidade IV

- Técnicas de depuração em Python.
- Testes automatizados simples.
- Classes e objetos.
- Métodos e encapsulamento.

Introdução à depuração

- A depuração é um processo essencial no desenvolvimento de software, permitindo identificar e corrigir problemas no código.
- Existem diversas técnicas de depuração, desde métodos simples, como print(), até ferramentas avançadas de inspeção em tempo real.
- Cada técnica tem um propósito específico, dependendo da complexidade do erro e do ambiente de desenvolvimento.
- A depuração auxilia na criação de códigos mais robustos, garantindo que o programa funcione conforme esperado.
    - O termo "bug" tem uma origem histórica curiosa, sendo popularizado por um incidente envolvendo uma mariposa em um computador da década de 1940.
    - A correção de bugs, conhecida como debugging, tornou-se uma prática fundamental na programação moderna.

Depuração com print() – Simples, mas eficaz

- O uso de print() é uma das formas mais comuns e acessíveis de depuração em Python.
- Inserindo print() em pontos estratégicos, é possível inspecionar o valor de variáveis e verificar o fluxo do programa.
- A técnica é útil para detectar falhas em cálculos, verificar a execução de condições e acompanhar mudanças de estado.
- Para depuração eficiente, é recomendável incluir rótulos nos print(), como [DEBUG], para identificar claramente as mensagens de depuração.
- Embora eficaz para pequenos problemas, o uso excessivo de print() pode poluir a saída do console e dificultar a análise dos resultados.
    - Após corrigir o erro, os print() de depuração devem ser removidos ou comentados para evitar confusão na versão final do código.

Exemplo de depuração com print()

- Em um programa de simulação de viagem no tempo, print() pode ser usado para verificar valores importantes.
- Inserindo print(f"[DEBUG] Ano atual: {ano_atual}"), é possível garantir que o ano foi corretamente extraído do sistema.
- A conversão de entrada do usuário é verificada com print(f"[DEBUG] Ano destino: {ano_destino}"), prevenindo erros de tipo.
- O cálculo da energia necessária pode ser depurado com print(f"[DEBUG] Energia necessária: {energia_necessaria}").
    - Caso o programa apresente comportamento inesperado, as mensagens [DEBUG] ajudam a identificar em qual etapa ocorreu o erro.
    - O uso sistemático de print() facilita a localização de problemas, tornando a depuração acessível mesmo para iniciantes.

Exemplo de depuração com print()

```
from datetime import datetime
ano_atual = datetime.now().year
print(f"[DEBUG] Ano atual obtido: {ano_atual}")  # Depuração: Confirma o ano atual extraído do sistema.
entrada_ano = input("Insira o ano de destino: ")
try:
    ano_destino = int(entrada_ano)
    print(f"[DEBUG] Ano destino inserido pelo usuário: {ano_destino}")  # Depuração: Checa se a conversão para int ocorreu corretamente.
except ValueError:
                                    # Caso o usuário insira algo que não seja um número inteiro, informa o erro.
                                    print("[DEBUG] Erro na conversão do ano de destino, entrada inválida:", entrada_ano)
                                    print("Ano inválido. Encerrando o programa.")
                                    exit()
```

Exemplo de depuração com print()

```
diferenca = ano_destino - ano_atual
print(f"[DEBUG] Diferença temporal calculada: {diferenca}")  # Depuração: Verifica se a diferença está correta.
energia_necessaria = abs(diferenca) * 10
print(f"[DEBUG] Energia necessária calculada: {energia_necessaria}")  # Depuração: Confirma o valor calculado.
modo = input("Escolha o modo de viagem (instantaneo/gradual): ").strip().lower()
print(f"[DEBUG] Modo de viagem inserido: {modo}")  # Depuração: Garante que o valor lido é o esperado.
if modo == "instantaneo":
        print("[DEBUG] O modo de viagem selecionado é instantâneo, pulando etapas intermediárias.")
elif modo == "gradual":
                print("[DEBUG] O modo de viagem selecionado é gradual, preparando a sequência de saltos intermediários.")
else:
    # Caso o usuário insira um modo não reconhecido, informa o erro.
    print(f"[DEBUG] Modo desconhecido: {modo}, encerrando.")
    print("Modo de viagem inválido. Encerrando o programa.")
    exit()
```

Limitando a depuração com print()

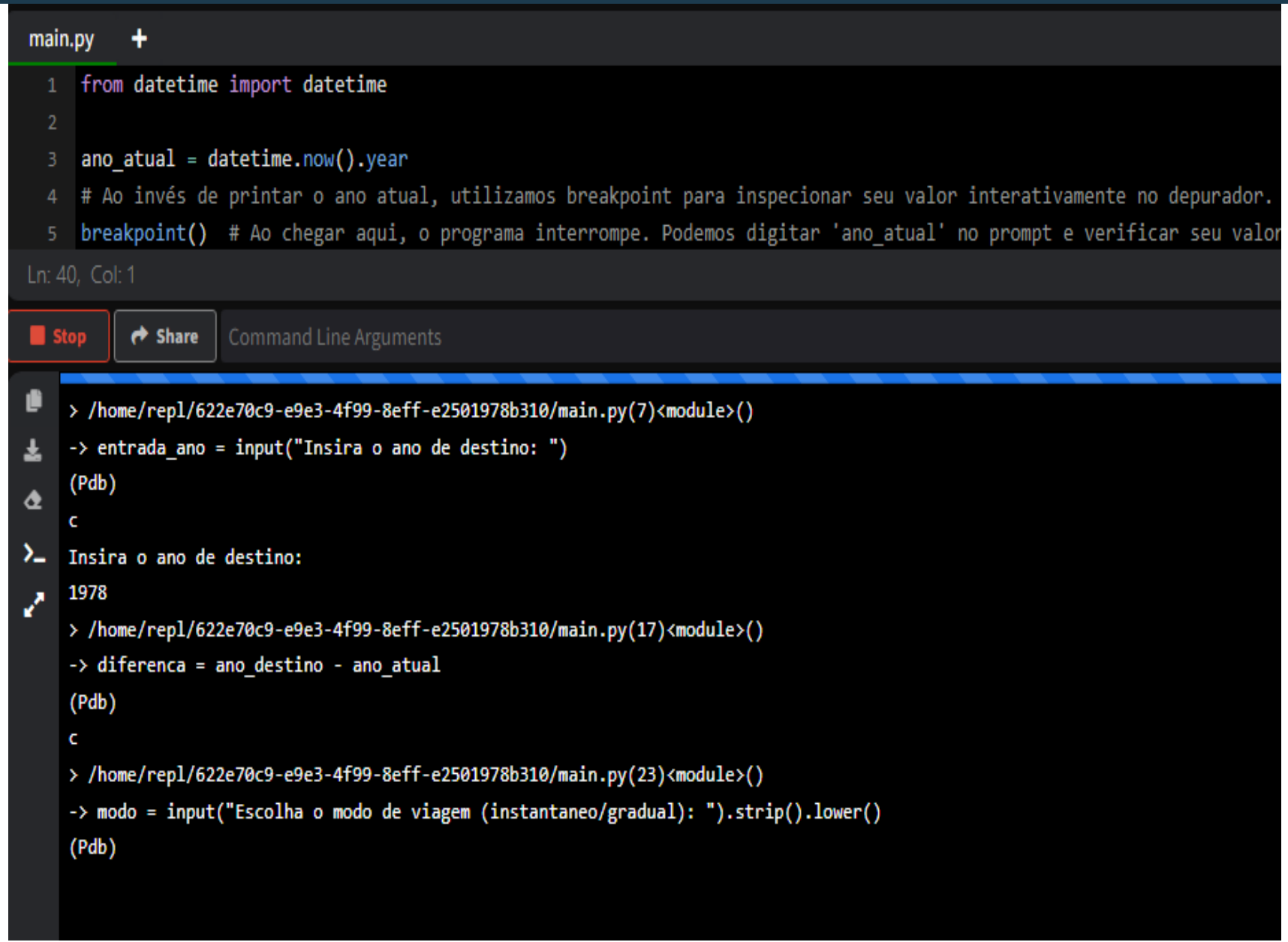
- O uso excessivo de print() pode tornar a saída confusa, dificultando a identificação do erro.
- Projetos maiores exigem um método mais organizado para registrar mensagens de depuração sem poluir o console.
- Para melhorar a clareza, pode-se implementar um controle condicional para ativar e desativar mensagens de depuração.
- Uma abordagem comum é definir uma variável DEBUG = True e exibir mensagens apenas quando ativada.
- if DEBUG: print("[DEBUG] Mensagem") permite desativar rapidamente a depuração alterando apenas um valor.
    - Outra opção é utilizar a biblioteca logging, que oferece níveis de mensagem como DEBUG, INFO e ERROR.
    - Métodos mais avançados de depuração, como pdb, evitam a necessidade de modificar o código com múltiplos print().

Depuração avançada com o pdb (Python Debugger)

- O módulo pdb permite inspecionar e manipular variáveis em tempo real durante a execução.
- Comandos do pdb incluem n para avançar linha a linha, s para entrar em funções e c para continuar a execução.
- Breakpoints são definidos com breakpoint() (Python 3.7+) ou import pdb; pdb.set_trace() em versões anteriores.
- Durante a execução interativa, é possível verificar valores digitando os nomes das variáveis diretamente no console.
- A depuração com pdb é útil quando o erro não é óbvio e ocorre em condições específicas que são difíceis de reproduzir.
    - IDEs modernas, como PyCharm e VS Code, integram depuradores gráficos que oferecem uma interface visual para inspeção de código.

Exemplo de depuração com pdb

- Pdb em ação



Fonte: autoria própria.

Tratamento de exceções como estratégia de depuração

- O uso de try-except permite capturar e tratar erros antes que o programa falhe inesperadamente.
- Exceções comuns, como ValueError, podem ser identificadas e tratadas para evitar interrupções abruptas na execução.
- Mensagens de erro personalizadas ajudam o usuário a entender o que deu errado e como corrigir o problema.
- Para depuração detalhada, a biblioteca traceback permite exibir a pilha de chamadas, ajudando a localizar a origem do erro.
- Combinar try-except com pdb e logging cria um fluxo de depuração mais eficiente e seguro.

```
try:
    ano_destino = int(input("Insira o ano de destino: "))
except ValueError as e:
    print(f"Erro: entrada inválida. Detalhes: {e}")
```

Monitoramento contínuo com logging

- O módulo logging permite registrar eventos do programa em diferentes níveis de severidade: DEBUG, INFO, WARNING, ERROR e CRITICAL.
- Diferentemente do print(), o logging pode gravar mensagens em arquivos ou enviá-las para sistemas externos de monitoramento.
- O uso de logging.debug() substitui print("[DEBUG] Mensagem"), organizando melhor as mensagens de depuração.
- Registros podem ser ativados ou desativados facilmente, tornando o código mais limpo e gerenciável.

```
import logging
logging.basicConfig(level=logging.DEBUG,
format="%(asctime)s - %(levelname)s - %(message)s")
logging.debug("Iniciando o programa")
```

Interatividade

Qual das alternativas abaixo é correta?

a) A técnica de depuração com print() permite modificar diretamente o estado das variáveis em tempo de execução, tornando-a ideal para testes dinâmicos.

b) O comando breakpoint() pode ser utilizado para interromper a execução de um programa, permitindo inspecionar variáveis e testar diferentes entradas antes de continuar.

c) O depurador pdb não permite a navegação interativa no código, sendo apenas uma ferramenta para registrar erros em arquivos de log.

d) O uso de try-except em Python impede que qualquer erro ocorra no programa, tornando desnecessária a análise de mensagens de erro ou rastreamento da execução.

e) A função logging() substitui a necessidade de usar print(), pois registra automaticamente todas as variáveis e eventos do programa sem necessidade de configuração adicional.

Resposta

Qual das alternativas abaixo é correta?

a) A técnica de depuração com print() permite modificar diretamente o estado das variáveis em tempo de execução, tornando-a ideal para testes dinâmicos.

b) O comando breakpoint() pode ser utilizado para interromper a execução de um programa, permitindo inspecionar variáveis e testar diferentes entradas antes de continuar.

c) O depurador pdb não permite a navegação interativa no código, sendo apenas uma ferramenta para registrar erros em arquivos de log.

d) O uso de try-except em Python impede que qualquer erro ocorra no programa, tornando desnecessária a análise de mensagens de erro ou rastreamento da execução.

e) A função logging() substitui a necessidade de usar print(), pois registra automaticamente todas as variáveis e eventos do programa sem necessidade de configuração adicional.

Introdução aos testes automatizados

- Testes automatizados são fundamentais no desenvolvimento de software, permitindo validar automaticamente funcionalidades e evitar erros inesperados.
- Entre os tipos mais comuns de testes automatizados, estão os testes unitários, que verificam pequenas partes do código, como funções ou métodos, de forma isolada.
- O principal benefício dos testes unitários é detectar erros cedo no desenvolvimento, garantindo que mudanças futuras não introduzam novos problemas.
- Testes bem implementados oferecem maior confiança no código, tornando a manutenção e a evolução do software mais seguras e previsíveis.
    - A escrita de testes é uma prática essencial em projetos complexos, evitando que falhas se propaguem para outras partes do sistema.
    - Python oferece diversas ferramentas para testes automatizados, incluindo frameworks como unittest e pytest, que facilitam a criação e execução dos testes.
    - A automação de testes reduz a necessidade de verificações manuais, acelerando o ciclo de desenvolvimento.

O papel dos testes unitários

- Testes unitários validam o funcionamento de partes isoladas do código, garantindo que cada função ou método opere conforme esperado.
- Cada teste unitário deve focar em um único comportamento, facilitando a identificação de falhas e a correção de erros.
- Esses testes são importantes para evitar que alterações no código impactem funcionalidades já desenvolvidas, prevenindo regressões.
- A estrutura dos testes unitários geralmente inclui entrada de dados, execução da função e comparação do resultado com o esperado.
    - Em Python, o uso de assert ou de métodos específicos de bibliotecas de testes permite verificar se a saída obtida é a correta.
    - Em projetos de grande porte, a existência de testes unitários bem definidos pode reduzir significativamente o tempo gasto na depuração e na manutenção.

Exemplo prático de testes unitários

```
def calcular_diferenca(ano_atual, ano_destino):
      return ano_destino - ano_atual
def calcular_energia(anos):
      return abs(anos) * 10
def validar_modo_viagem(modo):
      return modo.strip().lower() in ["instantaneo", "gradual"]
```

Para testar essas funções, utilizamos assert, que verifica se a saída obtida corresponde à esperada:

```
assert calcular_diferenca(2024, 2050) == 26
assert calcular_energia(30) == 300
assert validar_modo_viagem("instantaneo") == True
```

- Se um assert falhar, um erro será gerado, indicando que a função não está retornando o resultado esperado.
- Esse método simples já garante um nível básico de verificação, ajudando a identificar problemas rapidamente.

Framework unittest para testes automatizados

- O unittest é um framework embutido no Python que permite estruturar testes de forma organizada e reaproveitável.
- Testes em unittest são agrupados em classes, nas quais cada método representa um caso de teste específico.
- O método setUp() pode ser usado para configurar variáveis antes da execução dos testes, evitando repetições de código.
- As funções de asserção do unittest, como assertEqual() e assertTrue(), ajudam a verificar se os resultados obtidos estão corretos.
- A execução dos testes gera um relatório detalhado, indicando quais testes passaram.

```
import unittest class
TesteFuncoesTempo(unittest.TestCase):
    def test_calcular_diferenca(self):
            self.assertEqual(calcular_diferenca(2024,
2050), 26)
    def test_calcular_energia(self):
    self.assertEqual(calcular_energia(30), 300)
def test_validar_modo_viagem(self):
    self.assertTrue(validar_modo_viagem("instantaneo"))
```

Execução e relatórios com unittest

- Para executar os testes, basta executar o arquivo Python que contém a classe de testes.
- O unittest executa cada método de teste e gera um relatório indicando o número total de testes e quais passaram ou falharam.
- Quando um teste falha, o framework exibe detalhes do erro, incluindo a saída obtida e a esperada.
- O relatório permite identificar rapidamente quais funcionalidades apresentam problemas e precisam de correção.
- O uso de unittest melhora a organização dos testes e facilita a manutenção do código ao longo do tempo.
    - Testes unitários automatizados são uma peça fundamental na construção de sistemas confiáveis e de fácil depuração.
    - Ferramentas como pytest oferecem uma abordagem alternativa para testes, sendo mais simples e flexíveis para desenvolvedores.

Testes automatizados com pytest

- O pytest é um framework mais leve e flexível que unittest, permitindo escrever testes de forma mais intuitiva.
- Diferentemente do unittest, ele não requer a criação de classes, funcionando com funções comuns de teste.
- Qualquer função cujo nome comece com test_ é automaticamente reconhecida pelo pytest como um caso de teste.
- O pytest exibe um relatório simplificado, indicando quais testes passaram e onde ocorreram falhas.

```
import pytest
def test_calcular_diferenca():
        assert calcular_diferenca(2024, 2050) == 26
def test_calcular_energia():
        assert calcular_energia(30) == 300
def test_validar_modo_viagem():
        assert validar_modo_viagem("instantaneo") ==
        True
```

Benefícios dos testes automatizados

- Testes automatizados garantem que novas implementações não quebrem funcionalidades existentes, prevenindo erros em produção.
- A execução regular de testes reduz o tempo de depuração e melhora a confiabilidade do sistema.
- Frameworks de teste como unittest e pytest permitem validar o comportamento do código de forma eficiente e organizada.
- A automação de testes permite detectar falhas antes que o software seja entregue ao usuário final.

  - Empresas que adotam testes automatizados reduzem custos de manutenção e garantem maior estabilidade do código.
  - Testes contínuos fazem parte de práticas como Integração Contínua (CI), automatizando verificações sempre que há mudanças no código.

Interatividade

Qual das alternativas abaixo é correta?

a) O unittest é um framework externo ao Python que precisa ser instalado separadamente para permitir a criação de testes automatizados.

b) O uso de assert nos testes automatizados impede que o programa seja executado se qualquer teste falhar, exigindo a correção antes da execução completa do código.

c) O pytest permite a execução automática de funções de teste sem a necessidade de organizar os testes em classes, tornando o processo mais simplificado.

d) Os testes unitários exigem que cada função testada seja alterada para incluir chamadas diretas a assert, garantindo que apenas funções modificadas possam ser testadas.

e) Testes unitários são apenas úteis para identificar erros de sintaxe, sem impacto na validação da lógica do programa.

Resposta

Qual das alternativas abaixo é correta?

a) O unittest é um framework externo ao Python que precisa ser instalado separadamente para permitir a criação de testes automatizados.

b) O uso de assert nos testes automatizados impede que o programa seja executado se qualquer teste falhar, exigindo a correção antes da execução completa do código.

c) O pytest permite a execução automática de funções de teste sem a necessidade de organizar os testes em classes, tornando o processo mais simplificado.

d) Os testes unitários exigem que cada função testada seja alterada para incluir chamadas diretas a assert, garantindo que apenas funções modificadas possam ser testadas.

e) Testes unitários são apenas úteis para identificar erros de sintaxe, sem impacto na validação da lógica do programa.

Introdução à Programação Orientada a Objetos (POO)

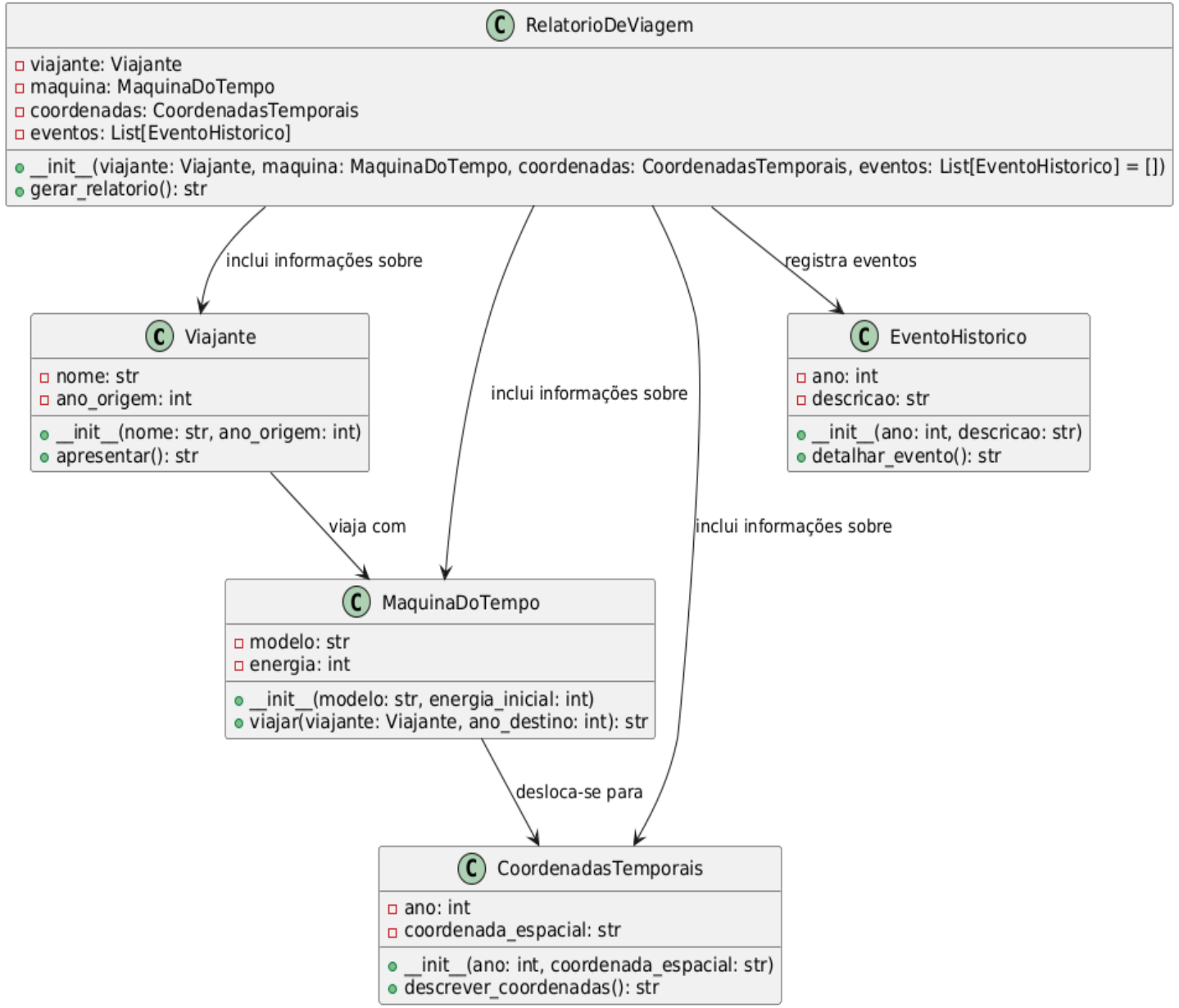
- A POO é um paradigma de programação que organiza o código em torno de objetos, que representam entidades do mundo real ou conceitos abstratos.
- Cada objeto é uma instância de uma classe, que define as características e comportamentos comuns a um grupo de objetos.
- O conceito de objetos na POO é mais abrangente do que no mundo real, permitindo modelar tanto objetos físicos quanto seres vivos e conceitos abstratos.
- Classes são estruturas que agrupam atributos (dados) e métodos (comportamentos), proporcionando modularidade e reutilização do código.
    - A modelagem em POO facilita a organização de sistemas complexos, tornando-os mais compreensíveis e escaláveis.
    - A UML (Unified Modeling Language) é frequentemente usada para representar classes e suas relações antes da implementação no código.

Classes e objetos na POO

- Classes definem um conjunto de atributos e métodos que seus objetos devem possuir, funcionando como moldes para a criação de instâncias.
- Um objeto é uma instância específica de uma classe, possuindo valores concretos para os atributos definidos na classe.
- A instância de uma classe ocorre em tempo de execução, utilizando o construtor __init__ para inicializar os atributos do objeto.
- Objetos de uma mesma classe podem ter características diferentes, mas compartilham os mesmos comportamentos definidos nos métodos da classe.
    - A separação entre classes e objetos permite que um código seja reutilizado e adaptado para diferentes situações sem duplicação desnecessária.
    - A criação de instâncias ocorre dinamicamente, garantindo que novos objetos possam ser criados conforme necessário sem modificar a estrutura da classe.

Exemplo de Diagrama de Classe em UML

- Diagrama de Classe



Fonte: autoria própria.

Atributos e métodos em classes Python

- Atributos são variáveis associadas a um objeto, armazenando características que diferenciam uma instância de outra.
- Métodos são funções dentro da classe que definem os comportamentos dos objetos e permitem a interação entre eles.
- Atributos podem ser públicos, privados ou protegidos, dependendo da necessidade de encapsulamento e controle de acesso.
- Métodos especiais, como __init__, são usados para definir o comportamento padrão da classe ao instanciar objetos.
    - Métodos de instância operam diretamente sobre os atributos do objeto e modificam seu estado conforme necessário.
    - Métodos estáticos e de classe permitem realizar operações sem necessidade de uma instância específica, sendo úteis para manipulações globais.

Exemplo de classe em Python – Viajante do tempo

- O __init__ inicializa os atributos do objeto com valores fornecidos na criação da instância.
- O método apresentar() retorna uma descrição textual do viajante, demonstrando a utilização dos atributos do objeto.
- Para criar um objeto dessa classe, basta chamar Viajante("Alice", 2024), criando uma instância com nome e ano de origem definidos.
- A abordagem orientada a objetos permite organizar a lógica do programa de maneira intuitiva, separando diferentes responsabilidades.
- Objetos dessa classe podem interagir com outros componentes do sistema, como máquinas do tempo e coordenadas temporais.

```
class Viajante:
    def __init__(self, nome, ano_origem):
        self.nome = nome
        self.ano_origem = ano_origem
    def apresentar(self):
        return f"Viajante {self.nome}, oriundo do ano
        {self.ano_origem}."
```

Criando e manipulando objetos

- A criação de um objeto ocorre chamando a classe como uma função, passando argumentos necessários ao método __init__.

```
viajante = Viajante("Alice", 2024)
```

- Cada objeto criado possui atributos independentes, permitindo a coexistência de múltiplas instâncias da mesma classe.
- Métodos da classe podem ser chamados diretamente a partir do objeto, como viajante.apresentar(), que exibe a apresentação do viajante.
- O Python gerencia automaticamente a alocação e desalocação de memória dos objetos, removendo instâncias que não são mais utilizadas.
    - Para garantir a persistência dos dados, além da execução do programa, é possível salvar os atributos do objeto em arquivos ou bancos de dados.
    - O encapsulamento de atributos protege as informações do objeto, permitindo controle sobre como os dados podem ser acessados e modificados.

Exemplo: Composição e interação entre classes

- O código do viajante do tempo demonstra a interação entre diferentes classes, simulando uma jornada temporal estruturada.
- A classe MaquinaDoTempo é responsável por realizar os deslocamentos temporais e gerenciar o consumo de energia.
- A classe CoordenadasTemporais armazena informações sobre o ano e a posição espacial do destino da viagem.
- A classe EventoHistorico registra acontecimentos observados durante a jornada, permitindo análise detalhada dos períodos visitados.
    - A classe RelatorioDeViagem reúne informações das demais classes, consolidando-as em um formato compreensível para o usuário.
    - A composição de classes é uma técnica fundamental para a construção de sistemas complexos, garantindo organização e reusabilidade.

Exemplo: Composição e interação entre classes

```
class CoordenadasTemporais:
    def __init__(self, ano, coordenada_espacial):
        self.ano = ano
        self.coordenada_espacial = coordenada_espacial
    def descrever_coordenadas(self):
        return f"Coordenadas: Ano {self.ano}, Posição Espacial:
{self.coordenada_espacial}"
```

```
class EventoHistorico:
    def __init__(self, ano, descricao):
        self.ano = ano
        self.descricao = descricao
    def detalhar_evento(self):
        return f"No ano {self.ano}: {self.descricao}"
```

Exemplo: Composição e interação entre classes

```
class RelatorioDeViagem:
    def __init__(self, viajante, maquina, coordenadas, eventos=[]):
        self.viajante = viajante
        self.maquina = maquina
        self.coordenadas = coordenadas
        self.eventos = eventos
    def gerar_relatorio(self):
        relatorio = []
        relatorio.append(self.viajante.apresentar())
                            relatorio.append(f"Máquina do tempo utilizada:
                            {self.maquina.modelo}, Energia atual:
                            {self.maquina.energia}")

                            relatorio.append(self.coordenadas.descrever_coordenadas())
```

Exemplo: Composição e interação entre classes

```
if self.eventos:
        relatorio.append("Eventos observados durante a viagem:")
        for e in self.eventos:
                relatorio.append(" - " + e.detalhar_evento())
else:
        relatorio.append("Nenhum evento notável foi registrado nesta época.")
return "\n".join(relatorio)
```

Interatividade

Qual das alternativas abaixo é correta?

a) Em Python, objetos são criados automaticamente assim que uma classe é definida, sem necessidade de instanciação explícita.

b) O método __init__ é um método especial de Python que deve ser chamado manualmente sempre que um objeto for criado a partir de uma classe.

c) A Programação Orientada a Objetos (POO) permite modelar elementos do mundo real em código, utilizando classes para definir atributos e métodos que descrevem o comportamento dos objetos.

d) A linguagem UML é utilizada exclusivamente para definir a sintaxe das classes em Python, sendo um requisito obrigatório para a implementação de programas orientados a objetos.

e) Na POO, os objetos permanecem na memória indefinidamente após serem criados, independentemente da execução do programa.

Resposta

Qual das alternativas abaixo é correta?

a) Em Python, objetos são criados automaticamente assim que uma classe é definida, sem necessidade de instanciação explícita.

b) O método __init__ é um método especial de Python que deve ser chamado manualmente sempre que um objeto for criado a partir de uma classe.

c) A Programação Orientada a Objetos (POO) permite modelar elementos do mundo real em código, utilizando classes para definir atributos e métodos que descrevem o comportamento dos objetos.

d) A linguagem UML é utilizada exclusivamente para definir a sintaxe das classes em Python, sendo um requisito obrigatório para a implementação de programas orientados a objetos.

e) Na POO, os objetos permanecem na memória indefinidamente após serem criados, independentemente da execução do programa.

Encapsulamento e organização do código na POO

- O encapsulamento é um dos princípios fundamentais da Programação Orientada a Objetos (POO), permitindo ocultar detalhes internos de um objeto e expor apenas uma interface pública.
- Esse conceito protege os atributos de um objeto contra acesso indevido, garantindo que eles sejam manipulados apenas por métodos apropriados.
- Métodos públicos servem como interface para interação externa, permitindo acesso controlado aos dados encapsulados na classe.
- A separação entre atributos privados e métodos públicos facilita a manutenção e evolução do código, evitando dependências desnecessárias.
    - O encapsulamento reforça a segurança dos dados, impedindo que variáveis críticas sejam modificadas diretamente por código externo.

Atributos públicos, protegidos e privados

- Atributos públicos podem ser acessados e modificados diretamente por qualquer parte do código, mas isso pode gerar inconsistências se não for bem controlado.
- Atributos protegidos, identificados com um sublinhado _, indicam que devem ser utilizados apenas dentro da classe ou suas subclasses.
- Atributos privados, iniciados com __, são inacessíveis diretamente de fora da classe, garantindo maior proteção contra modificações acidentais.
- O name mangling em Python impede o acesso direto a atributos privados, tornando-os acessíveis apenas dentro da classe em que foram definidos.
    - O uso adequado desses níveis de visibilidade permite um controle rigoroso sobre os dados, garantindo que eles sejam manipulados corretamente.
    - O encapsulamento adequado evita erros lógicos causados por acesso indevido aos atributos internos dos objetos.

Métodos públicos como interface de acesso

- Métodos públicos controlam o acesso aos atributos privados, permitindo que os dados internos sejam manipulados de forma segura.
- A criação de métodos de acesso (getters) e modificação (setters) garante que atributos privados sejam alterados apenas sob condições predefinidas.

```
class CoordenadasTemporais:
        def __init__(self, ano):
            self.__ano = ano
        def get_ano(self):
            return self.__ano
        def set_ano(self, novo_ano):
            if novo_ano > 0:
                self.__ano = novo_ano
            else:
                raise ValueError("O ano deve ser
                maior que zero.")
```

  - Esse controle previne que valores inválidos sejam atribuídos aos atributos internos da classe, garantindo consistência nos dados.

Métodos de instância e a manipulação de objetos

- Métodos de instância são definidos dentro de uma classe e operam sobre atributos específicos de um objeto, utilizando o parâmetro self.
- Esses métodos permitem que cada instância da classe tenha comportamentos próprios baseados nos dados armazenados em seus atributos.

```
class Viajante:
        def __init__(self, nome, ano_origem):
            self.nome = nome
            self.ano_origem = ano_origem
        def apresentar(self):
            return f"Viajante {self.nome}, oriundo do ano
            {self.ano_origem}.“
```

  - Ao chamar apresentar(), o método retorna uma descrição personalizada do objeto, demonstrando como ele pode operar sobre seus próprios atributos.
  - Métodos de instância ajudam a organizar a lógica do programa dentro das classes, evitando a necessidade de manipulação direta dos atributos.

Métodos de classe e métodos estáticos

- Métodos de classe (@classmethod) operam no contexto da classe em vez de uma instância específica, sendo úteis para modificar atributos globais.
- São úteis quando se deseja criar instâncias padronizadas ou modificar atributos compartilhados entre todas as instâncias.
- Métodos estáticos (@staticmethod) não dependem de atributos da classe ou da instância, funcionando como funções auxiliares dentro da classe.
- Encapsulam lógica que não depende do estado do objeto, evitando a necessidade de criar instâncias apenas para chamar uma função auxiliar.

```
class MaquinaDoTempo:
    @classmethod
    def modelo_padrao(cls):
        return cls("Chronos-1000",
                   energia_inicial=5000)
class MaquinaDoTempo:
    @staticmethod
    def calcular_custo(diferenca_anos):
        return diferenca_anos * 10
```

Encapsulamento na prática: Classe MaquinaDoTempo

- A classe MaquinaDoTempo encapsula a lógica de viagem temporal, protegendo atributos.

```
class MaquinaDoTempo:
        def __init__(self, modelo, energia_inicial):
                self.modelo = modelo self.__energia = energia_inicial
        def viajar(self, viajante, ano_destino):
                diferenca = abs(ano_destino - viajante.ano_origem)
                custo = diferenca * 10
                if custo <= self.__energia:
                        self.__energia -= custo
                        viajante.ano_origem = ano_destino
                        return f"Viajante transportado para o ano
              {ano_destino}. Energia restante: {self.__energia}."
                else:
                        return "Energia insuficiente."
```

  - O atributo __energia é privado, impedindo que seu valor seja alterado diretamente por código externo.
  - O método viajar() garante que apenas condições válidas permitam a alteração da linha temporal.

Benefícios do encapsulamento na POO

- O encapsulamento protege os dados da classe contra modificações indevidas, garantindo a integridade do sistema.
- A separação entre atributos internos e métodos públicos permite alterações internas sem impactar o código externo.
- Métodos públicos atuam como interfaces de comunicação, permitindo que apenas operações seguras sejam realizadas sobre os atributos privados.
- A modularização promovida pelo encapsulamento facilita a reutilização de código em diferentes partes do programa.
    - O isolamento de responsabilidades melhora a organização do código, tornando-o mais fácil de entender e manter.
    - Em sistemas complexos, o encapsulamento reduz a ocorrência de erros causados por acesso indevido a atributos internos.

Interatividade

Qual das alternativas abaixo é correta?

a) O encapsulamento impede o acesso a atributos privados em Python, tornando impossível a sua leitura ou modificação, mesmo dentro da própria classe.

b) Métodos estáticos em Python exigem acesso direto a atributos de instância e classe, pois não podem ser definidos sem depender de self ou cls.

c) O método @classmethod permite que um método seja associado à classe em vez de uma instância específica, permitindo que operações sejam realizadas no contexto geral da classe.

d) Atributos protegidos em Python são completamente inacessíveis fora da classe, mesmo que sejam herdados por subclasses.

e) Em Python, a definição de um método dentro de uma classe exige que o primeiro parâmetro seja cls, independentemente de ser um método de instância, classe ou estático.

Resposta

Qual das alternativas abaixo é correta?

a) O encapsulamento impede o acesso a atributos privados em Python, tornando impossível a sua leitura ou modificação, mesmo dentro da própria classe.

b) Métodos estáticos em Python exigem acesso direto a atributos de instância e classe, pois não podem ser definidos sem depender de self ou cls.

c) O método @classmethod permite que um método seja associado à classe em vez de uma instância específica, permitindo que operações sejam realizadas no contexto geral da classe.

d) Atributos protegidos em Python são completamente inacessíveis fora da classe, mesmo que sejam herdados por subclasses.

e) Em Python, a definição de um método dentro de uma classe exige que o primeiro parâmetro seja cls, independentemente de ser um método de instância, classe ou estático.

Referências

- ALVES, E. *Estruturas de dados com Python*. São Paulo: Novatec, 2022
- FURTADO, A. L. *Python*: programação para leigos. Rio de Janeiro: Alta Books, 2021.
- NILO, L. E. *Introdução à programação com Python*. São Paulo: Novatec, 2019.
- RAMALHO, L. *Fluent Python*. 2. ed. O'Reilly Media, 2022.
- RAMALHO, L. *Python fluente*: programação clara, concisa e eficaz. São Paulo: Novatec, 2015.
- SWEIGART, A. *Automatize tarefas maçantes com Python*: programação prática para verdadeiros iniciantes. São Paulo: Novatec, 2015.
- VAN ROSSUM, G.; DRAKE, F. L. *Python reference manual*. PythonLabs, 2007.

ATÉ A PRÓXIMA!