





UNIP
UNIVERSIDADE PAULISTA

Roteiros

Pensamento Lógico Computacional com Python

  Instituto de Ciências Exatas e Tecnologia	Disciplina: Pensamento Lógico Computacional com Python Título da aula: Introdução à Lógica de Programação e Python	Roteiro 1
---	---	------------------

ROTEIRO DE AULA PRÁTICA – AULA 01: INTRODUÇÃO À LÓGICA DE PROGRAMAÇÃO E PYTHON

1. Objetivos da Aula

- Apresentar os conceitos básicos de lógica de programação (algoritmos, pseudocódigo e fluxogramas).
- Mostrar como esses conceitos se relacionam à linguagem Python.
- Praticar a escrita de código Python simples, empregando a função `print()`, a função `input()`, variáveis e strings (incluindo f-strings e strings multilinha).
- Desenvolver um pequeno exercício prático, culminando em um relatório com o resumo teórico e o código-fonte produzido.

2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório ou BYOD – *bring your own device*).
- Acesso a um interpretador Python (instalado localmente ou online, como [OnlineGDB](#), [Online Python](#) ou [OneCompiler](#)).
- Material de apoio (capítulo 1 do livro-texto introdutório sobre lógica de programação, algoritmos, pseudocódigo, fluxogramas e introdução ao Python).
- Editor de texto ou IDE (opcional) para organização do relatório final.

3. Estrutura da Aula

1. Abertura (10 minutos)

- **Apresentação do tema:** Relembrar rapidamente o que é lógica de programação, algoritmos, pseudocódigo, fluxogramas e por que essas ferramentas são importantes para se pensar em soluções computacionais.
- **Conexão com Python:** Ressaltar como Python, sendo simples e versátil, facilita o primeiro contato prático com a lógica de programação.

2. Revisão Conceitual (20 minutos)

- **O que é um algoritmo:** Conceito, importância e exemplos do dia a dia (como o exemplo do leite derramado ou o da criança calculando a média de números).
- **Pseudocódigo:** Mostrar um pequeno pseudocódigo de cálculo de média para fixar o conceito.
- **Fluxogramas:** Explicar os principais símbolos padronizados (ISO 5807) e relacioná-los à lógica de decisões e loops.
- **Breve histórico do Python:** Explicar por que Python foi criado, sua sintaxe com indentação e facilidade de uso.
- **Exemplos rápidos:** Mostrar pequenos trechos em pseudocódigo e como seriam em Python.

3. Demonstração Prática (30 minutos)

- **Acesso ao interpretador:** Orientar os alunos a abrir um dos interpretadores online ou uma IDE instalada localmente.

Primeiro contato com Python:

1. Usar a função `print()` para exibir mensagens simples.
2. Criar uma interação com o usuário via `input()`.

Exemplo do "Viajante do Tempo":

1. **Código 1** – Exibição de mensagem fixa com `print()` (apresentação do "Viajante do Tempo").
2. **Código 2** – Solicitar nome ao usuário, armazenar em variável, usar f-strings para personalizar a mensagem.
3. **Código 3** – Utilizar strings multilinha para contar brevemente a história do viajante.
 - **Comentando o código:** Mostrar como usar `#` para inserir comentários que facilitem a leitura e manutenção.

4. Atividade Prática (40 minutos)

▪ Desafio:

1. Cada aluno deve escrever um pequeno programa que:
 - Pergunte o nome do usuário.
 - Pergunte de qual ano a pessoa "vem" (simulando viagem no tempo).
 - Exiba uma mensagem personalizada usando os dados fornecidos, em pelo menos **duas** linhas (podem usar `print()` múltiplos ou strings multilinha).

2. Exemplo (mínimo esperado):

```
nome = input("Quem é você, viajante? ") ano =  
input("De que ano você vem? ")  
  
print(f"Olá, {nome}. Incrível saber que você veio  
diretamente de {ano}!")  
print(f"Espero que a tecnologia do ano {ano} seja tão  
avançada quanto imagino.")
```

3. Ampliação: Desafiar os alunos a inserirem mais interações, como perguntar o motivo da viagem ou qual é a maior invenção tecnológica que conhecem.

5. Encerramento e Orientações Finais (20 minutos)

- **Tempo para dúvidas:** Esclarecer eventuais problemas encontrados na implementação.
- **Proposta de continuação:** Sugerir que testem e alterem o código em casa, expandindo a lógica (ex.: fazer cálculos com o ano de viagem, usar condicionais para diferentes respostas etc.).
- **Entrega do relatório:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.

6. Relatório Final

Cada aluno deve produzir um relatório curto (1 a 2 páginas) contendo:

Resumo Teórico:

- Explicar, com palavras próprias, o que é lógica de programação e por que ela é importante.
- Mencionar brevemente o que é pseudocódigo e fluxograma e como ajudam na organização de ideias.
- Citar as vantagens de usar Python para aprender programação.

Código-Fonte Comentado:

- Inserir o *código-fonte completo* da atividade proposta.
- Comentar as principais linhas, ressaltando o uso de `print()`, `input()`, variáveis etc.


7. Critérios de Avaliação

Critério	Peso	Descrição
Qualidade do Resumo Teórico	2,0	Clareza, objetividade e demonstração de entendimento sobre lógica de programação, pseudocódigo, fluxograma e Python.
Estrutura e Organização do Código Funcionamento da Solução	3,0	O código deve estar indentado corretamente, usar nomes de variáveis adequados e conter comentários informativos (quando necessários).
	3,0	O programa deve rodar sem erros, solicitando informações ao usuário e apresentando as mensagens personalizadas.
Criatividade e Aprimoramentos	2,0	Adição de perguntas extras, uso de strings multilinha, personalização das mensagens e outras melhorias que demonstrem domínio do conteúdo.

Nota Final: Será a soma dos valores obtidos em cada critério. Alunos ou equipes que não cumprirem os requisitos mínimos de funcionamento do código ou não entregarem o relatório dentro do prazo terão sua nota diminuída proporcionalmente.



8. Conclusão

Este roteiro busca oferecer uma base sólida para que os alunos compreendam a lógica de programação e apliquem-na em Python de forma interativa. Ao final da aula, espera-se que cada estudante seja capaz de:

- 
- a) Reconhecer a importância de algoritmos, pseudocódigos e fluxogramas.
 - b) Criar pequenos programas em Python que leiam dados do usuário e exibam respostas personalizadas.
 - c) Documentar a solução desenvolvida em um relatório, evidenciando entendimento teórico e prático.

Bom estudo e boa prática de programação!



  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Pensamento Lógico Computacional com Python</p> <p>Título da aula: Estruturas de Controle em Python</p>	<p>Roteiro 2</p>
---	---	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 02: ESTRUTURAS DE CONTROLE EM PYTHON

1. Objetivos da Aula

- Apresentar as estruturas de controle fundamentais em Python, incluindo:
 - **Estruturas de decisão** (if, else, elif).
 - **Estruturas de repetição** (for e while).
- Explicar o conceito de **lógica booleana** e operadores lógicos (and, or, not).
- Demonstrar como utilizar os operadores de comparação (==, !=, >, <, >=, <=) e a **precedência de operadores**.
- Introduzir o uso de **comandos especiais** (break, continue) para controle de fluxo.
- Propor exercícios práticos para que os alunos dominem essas estruturas no contexto de pequenos programas.

2. Recursos Necessários

- **Computadores** ou dispositivos com acesso à internet e ao **interpretador Python** (local ou online).
- **Editor de texto** ou IDE (opcional, mas recomendado) para organizar e executar os códigos (PyCharm, VS Code, Jupyter etc.).
- **Material de apoio** (capítulo 2 do livro-texto: trechos de código de exemplo e slides sobre estruturas de controle).

3. Estrutura da Aula

1. Abertura (10 minutos)

- **Recapitulação da aula anterior:**

- Lembrar conceitos básicos de lógica de programação, pseudocódigo e fluxogramas.
- Destacar como Python executa código de forma sequencial e por que precisamos de "desvios" e "repetições".

Apresentação dos objetivos da aula:

- Explicar por que estruturas de decisão e repetição são fundamentais para tornar o programa mais inteligente e dinâmico.

Exposição Teórica (20 minutos)

- **Estruturas de Decisão (if, else, elif)**

- Conceito de **condição** (expressão booleana) e **fluxo de execução**.
- Operadores de comparação e operadores lógicos (==, !=, >, <, >=, <=, and, or, not).
- **Precedência de operadores** (ordem de avaliação).
- Exemplos rápidos:

```
if x > 0:
    print("x é positivo") elif x <
0:
    print("x é negativo") else:
    print("x é zero")
```

Estruturas de Repetição (for, while)

- **for:** Usado quando se sabe o número de iterações ou se quer percorrer uma sequência (lista, string, range etc.).

```
for i in range(5): print(i)
```

- **while:** Mantém o loop enquanto a condição for verdadeira. Cuidado com loops infinitos.

```
while condição:  
    # repete ações
```

Comandos break e continue

- **break:** Interrompe completamente o loop.
- **continue:** Pula para a próxima iteração, sem sair do loop.

Demonstração Prática (30 minutos)

- **Estruturas de Decisão**

1. **Exemplo 1:** Escolha de destino temporal (passado/futuro).

- Mostrar if, elif, else para tratar opções válidas e inválidas.

2. **Exemplo 2:** Detecção de ano bissexto.

- Demonstrar uso de if aninhado e operadores de comparação (% para divisibilidade).

Estruturas de Repetição

0. **Exemplo 3:** Contagem regressiva com `for` e uso de `time.sleep()`.

1. **Exemplo 4:** Laço `while` para insistir em uma entrada válida (ex.: escolha de destino).

Atividade Prática em Grupo (40 minutos)

▪ **Desafio A:** *Cadastro de tentativas de viagem*

0. O programa pergunta quantas vezes o usuário deseja "viajar no tempo".
1. Usa um **for** para iterar (de 1 até o número de viagens).
2. Em cada iteração, pergunta o ano alvo e se deve ser "passado" ou "futuro". Armazena essas escolhas numa lista.
3. No final, exibe um resumo das viagens cadastradas.

▪ **Desafio B:** *Verificação de permissão temporal*

0. O programa pergunta a idade do usuário em um loop **while** até que uma idade válida (≥ 0) seja informada.
1. Se a idade for menor que 18, imprimir "Acesso negado". Caso contrário, imprimir "Acesso permitido".

▪ **Desafio Extra** (opcional):

- Criar um minimenu de opções (por exemplo, "1 - Registrar evento histórico", "2 - Mostrar lista de eventos", "0 - Sair"), usando **while** com `break` quando a opção "0" for escolhida.

Encerramento e Orientações Finais (20 minutos)

▪ Momento de Dúvidas:

- Permitir que os alunos perguntem sobre sintaxe, lógica booleana, situações de erro etc.

Sugestões de Continuação:

- Recomendar exercícios adicionais, como criação de pequenos jogos ("adivinha o número") ou sistemas de login com senha usando **while**.

Entrega do relatório:

- Explicar como deve ser a estrutura do relatório, contendo resumo e códigos desenvolvidos.

4. Relatório Final

Cada aluno deve produzir um relatório sucinto (2 a 3 páginas) contendo:

Resumo Teórico

- Explicar, com suas palavras, como funcionam as estruturas de decisão (if, else, elif) e repetição (for, while) em Python.
- Mencionar a importância da lógica booleana, operadores e precedência.

Códigos-Fontes Comentados

- Inserir os *códigos completos* das atividades propostas (Desafio A e B, e opcional).
- Adicionar comentários que expliquem a lógica de cada parte relevante.

Breve Reflexão

- Descrever se houve dificuldades.
- Comentar possíveis aplicações práticas (por exemplo, jogos, validações de formulário etc.).

5. Critérios de Avaliação

Critério	Peso	Descrição
Clareza do Resumo Teórico	2,0	Demonstração de entendimento sobre if, else, elif, for, while, lógica booleana e precedência de operadores.
Funcionalidade do Código	3,0	Códigos sem erros de execução, estruturas de controle funcionando corretamente, tratamento mínimo de entradas inválidas etc.
Organização e Comentários do Código	2,0	Códigos bem indentados, variáveis com nomes adequados, comentários que facilitem a leitura e entendimento.
Criatividade e Completeness	3,0	Adição de funções extras (ex.: uso de break/continue), exemplos mais elaborados, melhorias na interação com o usuário etc.



Nota Final: Será a soma dos valores obtidos em cada critério (máximo de 10,0). Trabalhos que não cumprirem requisitos mínimos poderão sofrer decréscimos adicionais na pontuação.

6. Conclusão

Ao final desta aula, espera-se que os alunos:

- Entendam como **estruturas de decisão** tornam o programa capaz de reagir a condições e múltiplas possibilidades.
- Saibam aplicar **estruturas de repetição** para evitar duplicação de código e lidar com entradas sucessivas.
- Tenham noções sólidas de **lógica booleana** e do fluxo de execução em Python.
- Sintam-se seguros para criar programas com comportamento interativo, realizando diferentes caminhos ou repetindo tarefas conforme necessidade.

Boa prática e ótimas explorações nos “ciclos temporais” da programação!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Pensamento Lógico Computacional com Python</p> <p>Título da aula: Tipos de Dados e Variáveis em Python</p>	<p>Roteiro 3</p>
---	---	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 03: TIPOS DE DADOS E VARIÁVEIS EM PYTHON

1. Objetivos da Aula

- Explicar os **tipos de dados básicos** em Python: números inteiros (int), números de ponto flutuante (float), números complexos (complex) e strings (str).
- Apresentar as **variáveis** como contêineres de dados, explicando a tipagem dinâmica e as convenções de nomenclatura segundo o PEP 8.
- Demonstrar o uso de **operadores aritméticos**, lógicos, de comparação e de atribuição composta (+=, -= etc.).
- Mostrar como combinar essas ferramentas em situações práticas (por exemplo, cálculos de datas, manipulação de textos, formatação de mensagens).

2. Recursos Necessários

- Computadores/dispositivos com **Python** instalado ou acesso a interpretadores online.
- **Editor de texto** ou IDE (Visual Studio Code, PyCharm, Jupyter Notebook etc.).
- **Material de apoio** (capítulo 3 do livro-texto: Exemplos de códigos-fonte em Python envolvendo variáveis, operadores e manipulação de dados).

3. Estrutura da Aula

1. Abertura (10 minutos)

- **Recapitulação das aulas anteriores:** Retomar brevemente o que é lógica de programação, estruturas de controle e como Python lida com fluxos de execução.
- **Apresentação dos objetivos:** Destacar a importância de conhecer os tipos de dados e variáveis para manipular informações em qualquer programa.

Exposição Teórica (30 minutos)

- **Tipos Numéricos:**

- 1. **int:** Inteiros de precisão arbitrária. Exemplos simples e limites em outras linguagens.

- 2. **float:** Números de ponto flutuante (padrão IEEE 754), questões de precisão (ex.: $0.1 + 0.2$).

- 3. **complex:** Forma $a + bj$, usos em cálculos avançados (engenharia, Fourier etc.).

Operações Aritméticas e Funções Matemáticas:

- Soma, subtração, multiplicação, divisão, módulo, exponenciação.
- Módulo `math` para operações mais complexas (`sqrt`, `sin`, `cos`, `radians` etc.).

Strings (classe `str`):

- Definição (`""` ou `''`), imutabilidade, fatiamento (`[início:fim:passo]`).
- Métodos importantes (`lower`, `upper`, `replace`, `strip`, `split` etc.).
- Formatação (f-strings, `.format()`).

Variáveis:

- Tipagem dinâmica e referência de objetos.
- Convenções do PEP 8 (snake_case, nomes descritivos etc.).
- Atribuição (=) vs. comparação (==).

Operadores:

- **Atribuição** (simples e composta: =, +=, -=, etc.).
- **Lógicos** (and, or, not).
- **Comparação** (==, !=, >, <, >=, <=).
- **Identidade** (is, is not).
- **Associação** (in, not in).

Exemplos Rápidos:

- Cálculos de datas (diferença entre anos).
- Manipulação simples de strings (inversão, substituição, formatação).

Demonstrações Práticas (20 minutos)

- **Exemplo 1:** Cálculo de intervalo entre ano atual e ano de destino (usando datetime.now().year).
- **Exemplo 2:** Inversão de uma string (mensagem criptografada) usando slicing [::-1].
- **Exemplo 3:** Mini "conversor de unidades de tempo" (anos → meses, dias, horas, minutos, segundos).
- **Exemplo 4:** Exibição de década e século a partir de um ano fornecido.

Atividade Prática (40 minutos)

- **Desafio A:** *Ferramenta de Cálculo de Salto Temporal*

0. Pedir ao usuário: ano atual (opcionalmente obtido via datetime), ano de destino.

1. Calcular e mostrar a diferença.

2. Exibir em qual década e século se encontra o destino.

3. **float:** Números de ponto flutuante (padrão IEEE 754), questões de precisão (ex.: 0.1 + 0.2).

4. **complex:** Forma $a + bj$, usos em cálculos avançados (engenharia, Fourier etc.).

Operações Aritméticas e Funções Matemáticas:

- Soma, subtração, multiplicação, divisão, módulo, exponenciação.
- Módulo math para operações mais complexas (sqrt, sin, cos, radians etc.).

Strings (classe str):

- Definição ("" ou ""), imutabilidade, fatiamento ([início:fim:passo]).
- Métodos importantes (lower, upper, replace, strip, split etc.).
- Formatação (f-strings, .format()).

Variáveis:

- Tipagem dinâmica e referência de objetos.
- Convenções do PEP 8 (snake_case, nomes descritivos etc.).
- Atribuição (=) vs. comparação (==).

Operadores:

- **Atribuição** (simples e composta: =, +=, -= etc.).
- **Lógicos** (and, or, not).
- **Comparação** (==, !=, >, <, >=, <=).
- **Identidade** (is, is not).
- **Associação** (in, not in).

Exemplos Rápidos:

- Cálculos de datas (diferença entre anos).
- Manipulação simples de strings (inversão, substituição, formatação).

Demonstrações Práticas (20 minutos)

- **Exemplo 1:** Cálculo de intervalo entre ano atual e ano de destino (usando `datetime.now().year`).
- **Exemplo 2:** Inversão de uma string (mensagem criptografada) usando slicing `[::-1]`.
- **Exemplo 3:** Mini "conversor de unidades de tempo" (anos → meses, dias, horas, minutos, segundos).
- **Exemplo 4:** Exibição de década e século a partir de um ano fornecido.

Atividade Prática (40 minutos)

- **Desafio A:** *Ferramenta de Cálculo de Salto Temporal*

0. Pedir ao usuário: ano atual (opcionalmente obtido via `datetime`), ano de destino.
1. Calcular e mostrar a diferença.
2. Exibir em qual década e século se encontra o destino.

- **Desafio B:** *Manipulação de Strings para Mensagem Secreta*

0. Solicitar ao usuário uma frase.

1. Limpar espaços extras (strip), normalizar para minúsculas (lower) e substituir uma ou duas palavras-chave por sinônimos.

2. Inverter a string para manter "segredo".

3. Mostrar o resultado final.

- **Desafio C** (Opcional): *Números Complexos no Plano*

0. Solicitar coordenadas de ponto inicial (parte real e imaginária).

1. Executar uma rotação (pedir ângulo) e um fator de escala.

2. Exibir novo ponto, magnitude e ângulo em relação ao eixo x.

Encerramento e Orientações Finais (20 minutos)

- **Dúvidas:** Incentivar perguntas sobre manipulação de dados, operadores, strings etc.

- **Próximos Passos:** Antecipar o uso de funções (capítulo 4) e estruturas de dados mais complexas (capítulo 5).

- **Entrega do Relatório:** Explicar a formatação e prazo.

4. Relatório Final

Cada aluno deve produzir um relatório curto (2 a 3 páginas), contendo:

Resumo Teórico:

- Explicar em suas palavras os tipos de dados (int, float, complex, str) e por que variáveis são importantes.

- Descrever brevemente os operadores e suas finalidades.

- Comentar a convenção de nomes (PEP 8) em Python.

Códigos Desenvolvidos:

- Inserir o *código completo* das atividades propostas (Desafios A, B e C, se houver).
- Adicionar comentários essenciais que expliquem a lógica de cada trecho.

Conclusão:

- Descrever se a aula ajudou no entendimento de dados e variáveis.
- Possíveis aplicações práticas no contexto de viagens no tempo, conversão de dados ou manipulação textual.

5. Critérios de Avaliação

Critério	Peso	Descrição
Clareza e Organização	2,0	Texto bem estruturado, com escrita coesa e uso correto das convenções (PEP 8).
Domínio dos Conceitos	3,0	Demonstra entendimento de tipos numéricos, strings, operadores e variáveis em Python.
Implementação dos Códigos	3,0	Códigos funcionais e sem erros, uso adequado de operadores (aritméticos, lógicos, atribuição).
Criatividade / Inovação	2,0	Abordagens originais nos desafios (ex. formatação criativa, mensagens adicionais, mini-aplicações extras).



Nota Final: Soma dos valores obtidos em cada critério, totalizando 10 pontos.

6. Conclusão

Ao final desta aula, os alunos deverão:

- Dominar a distinção entre **int**, **float**, **complex** e **str**, aplicando-os conforme a necessidade.
- Entender o **modelo de variáveis** em Python (tipagem dinâmica, referências de memória).
- Utilizar **operadores** de forma efetiva (aritméticos, lógicos, comparação, atribuição).
- Aplicar essas ferramentas para **manipular dados** reais e strings em cenários diversos, inclusive no contexto lúdico de viagens no tempo.

Boas explorações e ótimos estudos rumo às próximas dimensões da programação em Python!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Pensamento Lógico Computacional com Python</p> <p>Título da aula: Funções em Python</p>	<p>Roteiro 4</p>
---	--	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 04: FUNÇÕES EM PYTHON

1. Objetivos da Aula

- **Compreender o papel das funções** na organização e reutilização de código (programação estruturada e funcional).
- **Aprender a criar, chamar e retornar valores** em funções, entendendo a passagem de parâmetros e escopo de variáveis.
- **Explorar exemplos práticos** (no contexto de viagem no tempo) para ilustrar como funções facilitam modularidade e manutenção de software.
- **Aplicar boas práticas** de programação, definindo funções que realizem tarefas específicas e retornem valores significativos.

2. Recursos Necessários

- **Computadores ou dispositivos** com acesso ao interpretador Python (local ou online).
- **Editor de texto ou IDE** (VS Code, PyCharm, Jupyter Notebook etc.).
- Material de apoio (capítulo 4 do livro-texto) com **exemplos de código** que envolvem definições e chamadas de função.

3. Estrutura da Aula

Abertura (10 minutos)

- **Recapitulação rápida:** Relembrar o que já foi visto sobre tipos de dados, variáveis e estruturas de controle.
- **Objetivos:** Destacar a relevância das funções para evitar repetição de código e promover clareza no programa.

Exposição Teórica (20 minutos)

- **Conceito de Função**
 - Analogia com "receitas de culinária": um bloco de instruções que pode ser reutilizado.
 - Sintaxe básica (`def nome_da_funcao(parametros): ...`).

Passagem de Parâmetros

- Parâmetros vs. argumentos.
- Comportamento com objetos imutáveis (`int`, `str`, `tuple`) e mutáveis (`list`, `dict`).

Retorno de Valores

- Uso do `return`, a possibilidade de retornar múltiplos valores (tuplas, dicionários).
- Funções que não retornam valor (retornam `None`).

Escopo de Variáveis

- Escopo local vs. global.
- Palavras-chave `global` e `nonlocal`.

Boas Práticas

- Funções pequenas, com tarefas específicas.
- Documentação e nome de funções coerentes.

Demonstração Prática (30 minutos)

- **Exemplo 1:** Função que converte século numérico para romano (retomando a lógica de capítulo anterior).
 - Explicar a estrutura e mostrar a chamada.
- **Exemplo 2:** Função que calcula energia requerida para viagem temporal (recebe distância em anos, retorna custo).
- **Exemplo 3:** Função que exibe saudação personalizada (recebe nome do viajante, retorna string de boas-vindas).

Atividade Prática (40 minutos)

- **Desafio A:** *Organizador de Viagem no Tempo*
 1. Pedir ao usuário o ano de destino;
 2. Usar **funções** separadas para:
 - Preparar a viagem (cálculo de diferença e energia).
 - Executar a viagem (verificar se a energia é suficiente).
 - Confirmar a chegada (exibir resultado final).
 3. Mostrar o fluxo completo e modular.
- **Desafio B:** *Calculadora de Tempo de Viagem*
 0. Solicitar distância (anos) e velocidade (anos/segundo).
 1. Uma função `calcular_tempo(anos, velocidade)` que retorne segundos;
 2. Imprimir em minutos e horas, como desejar.

- **Desafio C** (Opcional): *Verificador de Paradoxos*

0. Crie uma função que receba parâmetros como `encontra_si_mesmo`, `interfere_ancestral`, `altera_evento_crucial`.

1. Retorne dicionário com "paradoxo": True/False, "razao": "...".
2. Teste cenários distintos e exiba as mensagens adequadas.

Encerramento e Orientações Finais (20 minutos)

- **Sessão de Dúvidas:** Permitir que os alunos questionem detalhes de sintaxe, escopo etc.
- **Próximos Passos:** Antecipar uso de **estruturas de dados mais avançadas** (capítulo 5) e possíveis integrações com POO.
- **Entrega do Relatório:** Esclarecer o formato e prazo.

4. Relatório Final

Cada aluno deve produzir um relatório sintético (2 a 3 páginas) contendo:

Resumo Teórico

- Conceituação de função, parâmetros, retorno.
- Diferença entre variáveis de escopo local e global.

Códigos Desenvolvidos

- Inserir os *códigos-fonte completos* das atividades (Desafios A e B, e opcionalmente C).
- Comentar as partes relevantes das funções, mostrando claramente a entrada e saída (retorno).

Reflexão

- Benefícios de usar funções (organização, reuso, clareza).
- Dificuldades encontradas e soluções.

5. Critérios de Avaliação



Critério	Peso	Descrição
Clareza do Resumo Teórico	2,0	Mostra entendimento de como criar e chamar funções, passagem de parâmetros, retorno de valores e escopo de variáveis.
Implementação Correta do Código	3,0	Códigos funcionais, uso adequado de def..., return, parâmetros, teste de casos básicos (cenários de entrada e saída).
Boas Práticas e Organização	3,0	Legibilidade, indentação, nome de funções e variáveis coerentes, modularidade (funções pequenas e objetivas).
Originalidade / Abrangência	2,0	Abordagem criativa nos desafios, exemplos adicionais, integração com temas anteriores (ex.: paradoxos, cálculos de data etc.).

Nota Final: Soma dos valores obtidos em cada critério, totalizando 10 pontos.

6. Conclusão

Ao término desta aula, espera-se que os alunos compreendam: **Por que as funções** são centrais para evitar duplicação de código e facilitar manutenção. **Como definir e chamar funções** em Python, incluindo parâmetros opcionais, retorno de múltiplos valores e manipulação de escopo. **Importância do encapsulamento**: Funções como "blocos lógicos" que organizam processos complexos (simulações de viagem no tempo). **Aplicações práticas** no dia a dia da programação (não apenas no contexto lúdico de viagem temporal).

Boas práticas e excelente desenvolvimento de códigos mais organizados e modulares!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Pensamento Lógico Computacional com Python</p> <p>Título da aula: Manipulação de Listas e Dicionários</p>	<p>Roteiro 5</p>
---	--	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 05: MANIPULAÇÃO DE LISTAS E DICIONÁRIOS

1. Objetivos da Aula

- **Compreender as estruturas de dados** listas e dicionários em Python.
- **Aprender operações fundamentais:** criação, acesso, inserção, remoção, pesquisa e iteração sobre listas e dicionários.
- **Aplicar métodos e funcionalidades** (por exemplo, append, remove, sort, keys, values etc.) a cenários reais (simulação de viagens no tempo).
- **Entender a utilidade de cada estrutura:** listas para sequências ordenadas, dicionários para mapeamento (chave-valor).
- **Explorar possibilidades de aninhamento** (listas de dicionários, dicionários de listas, dicionários aninhados).

2. Recursos Necessários

- Computadores ou dispositivos com **Python** instalado ou acesso a um interpretador online.
- **Editor de texto ou IDE** (VS Code, PyCharm, Jupyter Notebook etc.).
- Exemplos práticos de manipulação de **listas** e **dicionários** (capítulo 5 do livro-texto).

3. Estrutura da Aula

1. Abertura (10 minutos)

- **Revisão Rápida:** Lembrar da importância das estruturas de dados para organizar informações.
- **Objetivos Específicos:** Destacar que listas e dicionários são pilares da linguagem Python e formam a base de muitas aplicações (inclusive de viagem no tempo!).

Exposição Teórica (20 minutos)

- **Listas**
 - Criação e mutabilidade (ex.: `minha_lista = []` ou `list(range(5))`).
 - Acesso a elementos por **índice** (0, 1, 2...).
 - Métodos principais (`append`, `remove`, `sort`, `reverse`, `insert` etc.).
 - Operações úteis (`len()`, `min()`, `max()`).
 - **Fatiamento** (slicing): `[inicio:fim:passo]`.

Dicionários

- Conceito chave-valor (ex.: `meu_dict = {"chave": "valor"}`).
- Acesso aos dados por chaves, e não por índices.
- Criação, modificação, exclusão de pares (`dic["nova_chave"] = valor`).
- Métodos (`keys`, `values`, `items`, `get` etc.).
- **Dicionários aninhados** (ex.: `{ano: {"evento": "...", "local": "...}}`).
- **Comparação:** Quando usar listas vs. dicionários, exemplos práticos.

Demonstrações Práticas (30 minutos)

▪ Exemplo 1: Lista de Destinos Temporais

1. Pedir ao usuário quantos anos (destinos) serão inseridos.
2. Inserir cada ano em uma lista.
3. Exibir, ordenar, remover ano etc.

Exemplo 2: Dicionário de Informações do Viajante

0. Criar um dicionário vazio.
1. Solicitar nome, idade, ano de origem.
2. Exibir o dicionário.

Exemplo 3: Dicionários Aninhados

0. Receber eventos históricos de diversos anos.
1. Armazená-los em {ano: {"nome_evento":..., "localizacao":...,...}}.
2. Exibir e consultar anos específicos.

Atividade Prática (40 minutos)

▪ Desafio A: *Agenda Temporal*

0. Criar um dicionário no qual cada **chave** seja um ano e o **valor** seja uma lista de compromissos.
1. Permitir inserir compromissos para cada ano, consultar e remover se necessário.
2. Exibir a agenda final.

▪ Desafio B: *Tradutor de Gírias Temporais*

0. Criar um dicionário mapeando palavras modernas → expressões antigas (ou vice-versa).
1. Receber uma frase do usuário, fatiar em palavras e traduzir usando o dicionário.

2. Lidar com pontuações (string.punctuation) e maiúsculas/minúsculas.

- **Desafio C** (Opcional): *Inventário de Épocas*

0. Criar uma lista de dicionários, em que cada item da lista represente uma época com chaves como {"ano":..., "acontecimentos": [...], "comentarios": ...}.

1. Permitir adicionar novas épocas, listar todas, remover etc.

Encerramento e Orientações Finais (20 minutos)

- **Momento de Dúvidas:** Esclarecimentos sobre problemas encontrados, métodos avançados, boas práticas.
- **Sustentação Teórica:** Reforçar que listas e dicionários são básicos, mas poderosos.
- **Entrega do Relatório:** Avisar como cada desafio deve ser documentado.

4. Relatório Final

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

Resumo Teórico

- Explicar em suas palavras:
 - O que são listas, como funcionam índices, métodos e fatiamento.
 - O que são dicionários, como funcionam chaves e valores, principais métodos.

Códigos Desenvolvidos

- Inserir *códigos completos* dos Desafios A, B (C se houver).
- Comentar partes importantes, destacando como ocorrem inserções, remoções, buscas etc.

Conclusão

- Experiência prática: o que foi mais fácil, o que foi desafiador, aplicação no contexto de viagens no tempo.

5. Critérios de Avaliação

Critério	Peso	Descrição
Clareza no Resumo Teórico	2,0	Demonstração de entendimento sobre listas (mutabilidade, índices, métodos) e dicionários (mapeamento chave-valor, métodos, uso).
Implementação Correta do Código	3,0	Códigos funcionais, manipulação adequada de listas e dicionários, tratamento básico de exceções ou casos especiais (ex.: remoções).
Organização e Boas Práticas	3,0	Código limpo, indentação correta, nomes de variáveis claríssimos. Fatiamento adequado, uso de métodos e iterações onde necessário.
Criatividade / Completeness	2,0	Soluções originais nos desafios, possibilidade de extensão (ex.: dicionários aninhados), funções extras de consulta ou manipulação.

Nota Final: Soma dos valores obtidos em cada critério, totalizando 10 pontos.



6. Conclusão

Ao final desta aula, os alunos deverão:

- **Manusear listas** com confiança, utilizando métodos, slicing e operações de ordenação.
- **Manusear dicionários** para mapeamento eficiente, inclusive aninhados, quando necessário.
- **Saber escolher** entre listas e dicionários conforme a necessidade do projeto.

- **Aprimorar** a capacidade de estruturar dados em Python para cenários mais complexos (ex.: agenda temporal, tradutor).

Boas explorações nos tempos e espaços da programação!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Pensamento Lógico Computacional com Python</p> <p>Título da aula: Entrada e Saída de Dados em Python</p>	<p>Roteiro 6</p>
---	---	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 06: ENTRADA E SAÍDA DE DADOS EM PYTHON

1. Objetivos da Aula

- **Compreender os conceitos** de entrada (input) e saída (output) de dados em um programa.
- **Aprender a usar** as funções input() (leitura via teclado) e print() (exibição no console) para criar programas interativos.
- **Explorar a leitura e escrita de arquivos** (textos e JSON) para persistência de dados.
- **Entender boas práticas** (uso de blocos with, tratamento de exceções com try/except etc.) que tornam o código mais robusto.

2. Recursos Necessários

- Computadores ou dispositivos com **Python** instalado ou acesso a interpretadores online.
- **Editor de texto ou IDE** (VS Code, PyCharm, Jupyter Notebook etc.).
- Programas/exemplos práticos de leitura/saída de dados, incluindo manipulação de arquivos (capítulo 6 do livro-texto).

3. Estrutura da Aula

Abertura (10 minutos)

- **Contextualização:** Relembrar a importância de E/S em qualquer aplicação (capturar dados + apresentar resultados).
- **Objetivos:** Destacar que Python facilita bastante as tarefas de input, print e manipulação de arquivos.

Exposição Teórica (20 minutos)

- **Entrada de dados**
 - Uso de `input()` para capturar strings.
 - Conversão para outros tipos (`int`, `float`, `bool`), e tratamento de exceções (`try/except`).
 - Possíveis refinamentos (`strip`, `split`, validação de formato etc.).

Saída de dados

- Uso de `print()`, argumentos múltiplos, parâmetros `sep` e `end`.
- Formatação de strings (`f-strings`, `.format`, `%`).

Leitura e escrita em arquivos

- Função `open()` e modos (`r`, `w`, `a`, `x` etc.).
- Contexto gerenciado (`with open(...) as arquivo:`).
- Leitura (`read`, `readlines`), escrita (`write`).
- Codificações (`encoding='utf-8'`).

Uso de JSON

- Módulo json: funções dump, dumps, load, loads.
- Armazenar e recuperar dados estruturados (dicionários, listas) em arquivos.

Demonstração Prática (30 minutos)

▪ Exemplo 1:

1. Receber ano de destino e objetivo de viagem via input().
2. Exibir resumo personalizado com print().

Exemplo 2:

0. Criar função para salvar "registros de viagens" em um arquivo texto (w, a).
1. Criar função para ler o arquivo e listar cada registro.

Exemplo 3:

0. Manter "configurações" em formato JSON.
1. Mostrar como carregar e salvar (backup) esses dados.

Atividade Prática (40 minutos)

▪ Desafio A: *Cadastro Interativo*

0. Criar um programa que pergunta nome, idade, cidade (ou outras infos).
1. Usa input() com validação.
2. Exibe resultados formatados com print().

▪ Desafio B: *Gerenciador de Viagens Temporais*

0. Permite inserir registros (ano visitado + descrição).
1. Salva em um arquivo texto (registros.txt) com append.
2. Opção de ler e exibir todos os registros (readlines).

- **Desafio C** (Opcional): *Backup/Restore de Configuração em JSON*

0. Define um dicionário de configurações (ex.: { "modo": "futuro", "energia": 3000 }).

1. Usa json.dump para salvar no arquivo config.json.

2. Carrega de volta com json.load e exibe no console.

Encerramento e Orientações Finais (20 minutos)

- **Dúvidas:** Esclarecer problemas com codificação, permissões de arquivo, exceções etc.

- **Reforço:** Mostrar que entrada e saída de dados é fundamental para programas práticos e integrados.

- **Entrega do Relatório:** Orientar formatação e prazo.

4. Relatório Final

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

Resumo Teórico

- Conceituar "entrada de dados" (teclado, input, validações) e "saída de dados" (print, formatações).

- Explicar como funciona "leitura/escrita de arquivos" em Python e possíveis modos (r, w, a, json).

Códigos Desenvolvidos

- Inserir *códigos completos* dos desafios (A, B e opcional C).

- Comentar partes-chave (abertura de arquivo, manipulação de strings, conversões de tipo etc.).

Conclusão

- Relatar se foi simples ou desafiador manipular E/S.
- Possíveis aplicações práticas (logs, configuração persistente, relatórios etc.).

5. Critérios de Avaliação

Critério	Peso	Descrição
Entendimento do Conceito de E/S	2,0	Clareza ao explicar input, print, leitura e escrita em arquivos, JSON.
Implementação Correta do Código	3,0	Códigos funcionais, uso de try/except (quando necessário), abertura e fechamento de arquivos adequadamente.
Uso de Boas Práticas	3,0	Escopo bem definido, funções para separar lógica, formatações de saída (f-strings), manipulação correta de exceções.
Criatividade / Completeness	2,0	Extensão dos exemplos (ex.: filtrar dados, contagem regressiva, JSON com múltiplas chaves etc.).

Nota Final: Soma dos valores obtidos em cada critério, totalizando 10 pontos.



6. Conclusão

Ao término desta aula, os alunos devem:

- **Diferenciar** com segurança os conceitos de entrada e saída de dados.
- **Utilizar** `input()` para capturar informações, tratando conversões e exceções.
- **Dominar** `print()` para exibir resultados, formatação e concatenação de dados.
- **Manipular arquivos** (texto/JSON) para persistir e recuperar informações de forma confiável.
- **Integrar** esses conhecimentos para criar aplicações interativas e duradouras (com logs, backups, relatórios).



Boas práticas e excelente desenvolvimento de códigos que interajam ativamente com usuários e sistemas!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Pensamento Lógico Computacional com Python</p> <p>Título da aula: Depuração e Teste de Algoritmos</p>	<p>Roteiro 7</p>
--	--	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 07: DEPURAÇÃO E TESTE DE ALGORITMOS

1. Objetivos da Aula

- **Entender a importância** das etapas de **depuração** (debug) e **teste** no desenvolvimento de software.
- **Aprender técnicas de depuração** em Python, desde o uso de `print()` até o emprego de ferramentas como o depurador `pdb` e depuradores integrados às IDEs.
- **Familiarizar-se com testes automatizados** (unitários) usando `assert`, `unittest` ou `pytest` para garantir que as funções do código funcionem conforme esperado.
- **Desenvolver boas práticas** de debugging e testagem, reduzindo falhas e tornando o código mais robusto.

2. Recursos Necessários

- Computadores/dispositivos com **Python** instalado ou acesso a interpretadores online (recomendável Python 3.7+ para uso de `breakpoint()`).
- **Editor de texto ou IDE** (VS Code, PyCharm, Jupyter Notebook etc.) que facilite a depuração visual (opcional).
- Exemplos de códigos contendo potenciais bugs e funções a serem testadas (ex.: funções de cálculo de diferenças, manipulações de dados etc.).

3. Estrutura da Aula

1. Abertura (10 minutos)

- **Contextualização:** Por que depurar e testar? Consequências de bugs (falhas em lógica, paradoxos em "viagens no tempo" etc.).
- **Objetivos:** Destacar as vantagens de localizar erros mais cedo e validar algoritmos de forma sistemática.

Exposição Teórica (20 minutos)

- **Depuração em Python**
 - Uso de `print()` como método básico (pros e contras).
 - Introdução ao **depurador pdb**: breakpoints, inspeção de variáveis, execução passo a passo.
 - Depuradores integrados em IDEs (PyCharm, VS Code).
- **Teste de Algoritmos**
 - Conceito de **teste unitário**: testar pequenas partes do código isoladamente.
 - Ferramentas em Python: `assert`, `unittest`, `pytest`.
 - Boas práticas: testes repetíveis, independentes, e automáticos.
- **Integração da Depuração e Teste**
 - Depurar quando algo falha nos testes.
 - Uso contínuo durante o desenvolvimento (TDD - Test-Driven Development, se desejado).

Demonstrações Práticas (30 minutos)

1. Exemplo de Depuração

- Código com erro lógico.
- Inserção de `print()` para localizar falhas.
- Uso de `breakpoint()` (ou `pdb.set_trace()`) para inspeção interativa no terminal.

- Mostrar como avançar passo a passo no depurador (n, s, c).

Exemplo de Testes

- Criação de funções simples (ex.: cálculo de anos, energia).
- Testes rudimentares com assert.
- Estruturação em unittest ou pytest para rodar vários testes e relatório final.

Atividade Prática (40 minutos)

- **Desafio A:** *Depurando o "Módulo de Viagem"*

1. O professor fornece um código "bugado" que faz cálculos de datas ou outras lógicas.
2. O aluno deve inserir print() ou usar pdb para localizar e corrigir o problema.
3. Justificar o raciocínio de onde encontrou o bug.

- **Desafio B:** *Criando testes unitários*

0. Escolher 2 a 3 funções do código (ex.: cálculo de diferença de anos, validação de modos).
1. Escrever testes usando assert ou unittest que verifiquem comportamento esperado em vários cenários.
2. Executar os testes e interpretar resultado.

- **Desafio C** (Opcional): *Automatizar com pytest*

0. Separar funções em um arquivo.
1. Criar arquivo de testes seguindo padrão test_*.py.
2. Rodar pytest e analisar relatório (pass, fail).

Encerramento e Orientações Finais (20 minutos)

- **Sessão de Dúvidas:** Sobre pdb, teste unitário, possíveis problemas de lógica.
- **Enfatizar Importância:** Menos bugs em produção, maior confiança no sistema, facilidade de manutenção.
- **Entrega do Relatório:** Orientar formatação (descrição de bugs encontrados, outputs de testes etc.).

4. Relatório Final

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

Resumo Teórico

- Diferenciar "depuração" e "teste automatizado"; por que ambos são necessários.
- Ferramentas em Python (pdb, unittest, pytest).

Procedimento Prático

- Explicar como depurou (ex.: prints, uso de breakpoint() ou debugger IDE).
- Descrever o conjunto de testes criados (testes básicos, cenários, entradas, saídas esperadas).

Resultados

- Mostrar as falhas (bugs) encontradas, o que foi corrigido.
- Exibir o resultado final dos testes (todos passaram, algum falhou etc.).

Conclusão

- Reflexão sobre a experiência de localizar e corrigir erros.
- Benefícios de manter testes para garantir estabilidade futura do código.

5. Critérios de Avaliação

Critério	Peso	Descrição
Entendimento de Depuração	2,0	Demonstra uso correto de print, pdb ou debugger IDE para isolar bugs; clareza ao explicar processo de identificar falhas.
Implementação de Testes	3,0	Testes básicos (assert, unittest, pytest) cobrindo as funções principais; coerência nos cenários de teste e checagens adequadas.
Organização e Clareza do Relatório	3,0	Código legível, relatório bem estruturado, descrição clara de bugs, testes, outputs etc.
Correção e Completude das Soluções	2,0	Bugs devidamente corrigidos; testes rodam sem falhas inesperadas; respostas/pontos identificados são coerentes com o problema proposto



Nota Final: Soma dos valores obtidos em cada critério, totalizando 10 pontos.

6. Conclusão

Ao término desta aula, os alunos deverão:

- **Dominar técnicas de depuração** simples (print) e avançadas (pdb, IDEs).
- **Compreender o valor dos testes automatizados** na confiabilidade do software.
- **Criar e rodar testes** para verificar funções e corrigir falhas logicamente.
- **Aprimorar a qualidade** do código, tornando-o mais estável e seguro contra regressões ou novas funcionalidades problemáticas.

Boas práticas de depuração e sucesso nos testes de seus algoritmos!

  <p>Instituto de Ciências Exatas e Tecnologia</p>	<p>Disciplina: Pensamento Lógico Computacional com Python</p> <p>Título da aula: Introdução à Programação Orientada a Objetos (POO) em Python</p>	<p>Roteiro 8</p>
---	---	-------------------------

ROTEIRO DE AULA PRÁTICA – AULA 08: INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS (POO) EM PYTHON

1. Objetivos da Aula

- **Entender os princípios da POO:** classes, objetos, atributos, métodos.
- **Aprender a criar e instanciar classes** em Python, encapsulando dados e comportamentos.
- Explorar conceitos de **encapsulamento**, visibilidade (público, protegido, privado) e métodos especiais (`__init__`).
- **Demonstrar** como a POO organiza melhor o código, tornando-o modular e expansível.

2. Recursos Necessários

- Computadores ou dispositivos com **Python** instalado ou acesso a um interpretador online.
- **Editor de texto ou IDE** (VS Code, PyCharm, Jupyter etc.) para organizar o projeto orientado a objetos.
- Exemplos simples de **diagrama UML** (ver seções abaixo).

3. Estrutura da Aula

Abertura (10 minutos)

- **Motivação:** Explicar por que a POO facilita a modelagem de problemas complexos (objetos do "mundo real" → objetos no código).
- **Objetivos:** Apresentar os conceitos de classe, objeto, atributos, métodos, encapsulamento e como Python implementa POO.

Exposição Teórica (20 minutos)

- **Conceitos Fundamentais**
 - **Classe** vs. **objeto** (ex.: "Caneta" como classe, "uma caneta específica" como objeto).
 - **Atributos** (dados de um objeto) e **métodos** (comportamentos).
 - **Encapsulamento:** Proteção e organização dos dados (público, protegido `_`, privado `__`).

Ciclo de Vida

- Criação de objetos (instanciação), uso e destruição (GC em Python).
- Relação com persistência (salvar estado em arquivo/banco).

Exemplos de Python

- Sintaxe básica: `class NomeDaClasse: __init__, self` etc.
- Métodos de instância, de classe (`@classmethod`), estáticos (`@staticmethod`).

Demonstração Prática (30 minutos)

1. Exemplo 1: Criação de uma classe Viajante (ou outra temática):

- Atributos como `nome`, `ano_origem`.

- Método apresentar().
- Instanciação e uso do objeto.

2. Exemplo 2: Modelagem de múltiplas classes interagindo (ex.: MaquinaDoTempo, CoordenadasTemporais etc.).

- Mostrar como objetos se relacionam (passar um objeto como parâmetro para outro método).
- Exibir encapsulamento: atributo privado `__energia`.

3. Exemplo 3: Implementar um método `@staticmethod` ou `@classmethod` para mostrar variações de métodos.

Atividade Prática (40 minutos)

- **Desafio A:** *Criar Classes Básicas*

1. Cada aluno define uma classe simples (ex.: ContaBancaria, Produto, Carro etc.).
- 2. Atributos** no `__init__` e métodos básicos (ex.: depositar, sacar, exibir_informacoes).
3. Instanciar objetos e demonstrar funcionamento.

- **Desafio B:** *Sistema de Viagem no Tempo Simplificado*

0. Classes: Viajante, MaquinaDoTempo, EventoHistorico.
1. Permitir "viajar" e "registrar evento".
2. Exibir relatório final usando métodos de cada classe.

- **Desafio C** (Opcional): *Encapsulamento e Acesso*

0. Usar `__`atributo privado.
1. Criar getters/setters para manipulação segura (ex.: validar valores).

2. Demonstrar tentativas de acesso indevido e como o encapsulamento protege.

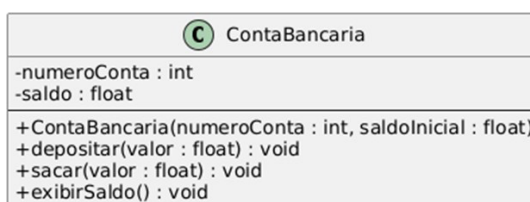
Encerramento e Orientações Finais (20 minutos)

- **Momento de Dúvidas:** Sobre construção de classes, atributos privados, métodos especiais.
- **Valor da POO:** Reforçar que POO organiza melhor o código, facilita manutenção e reuso.
- **Entrega do Relatório:** Orientar como documentar as classes criadas, instâncias, exemplos de uso.

Exemplos de Diagrama UML

Abaixo, seguem dois exemplos simples de diagramas UML, que podem ilustrar como visualizar classes e relacionamentos na POO.

Diagrama 1: Classe Simples (ContaBancaria)

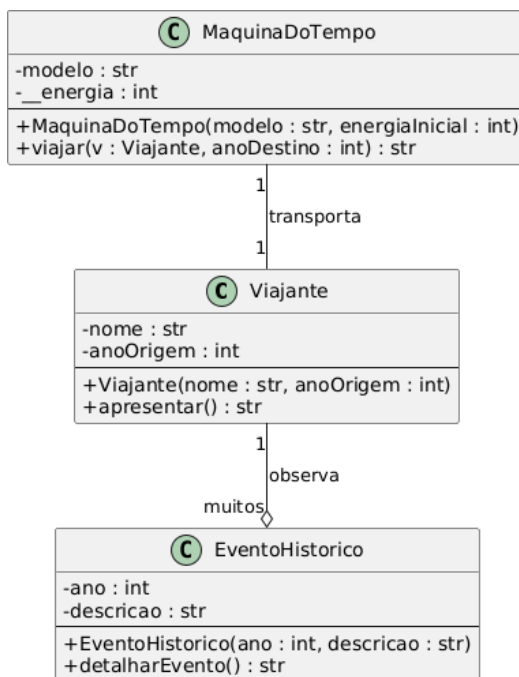


Explicação:

- ContaBancaria é a classe.
- Atributos privados:
 - `-numeroConta : int`
 - `-saldo : float`
- Métodos públicos (com o símbolo +):

- ContaBancaria(numeroConta, saldoInicial) é o construtor (___init___em Python).
- depositar(valor), sacar(valor), exhibirSaldo().

Diagrama 2: Múltiplas Classes (Viajante, MaquinaDoTempo, EventoHistorico)



Explicação:

- **Viajante** tem atributos nome, anoOrigem e métodos apresentar().
- **MaquinaDoTempo** tem atributos modelo, __energia e método viajar().
- **EventoHistorico** tem ano, descricao e método detalharEvento().
- O relacionamento Viajante "1" --o "muitos" EventoHistorico : "observa" indica que um Viajante pode observar vários EventosHistóricos.
- O relacionamento MaquinaDoTempo "1" -- "1" Viajante : "transporta" (apenas ilustrativo) mostra que a máquina transporta o viajante.

Os diagramas são simbólicos para ilustrar como classes e relações podem ser representadas em UML.

4. Relatório Final

Cada aluno deve produzir um relatório (2 a 3 páginas) contendo:

Resumo Teórico

- Conceitos de classe, objeto, atributos, métodos, encapsulamento, instância.
- Diferenças entre métodos de instância, de classe e estáticos.

Descrição das Classes Criadas

- Mostrar atributos e métodos principais.
- Ilustrar (caso queira) com **diagrama UML** simples.

Exemplos de Uso

- Trechos de código demonstrando instanciação e chamadas de métodos.
- Resultados exibidos, comportamento observado.

Conclusão

- Reflexão sobre como a POO facilita a organização e extensibilidade do código.
- Planos de como evoluir as classes criadas (ex.: adicionar herança, polimorfismo).

5. Critérios de Avaliação

Critério	Peso	Descrição
Entendimento de Conceitos de POO	2,0	Clareza em classe vs. objeto, atributos, métodos, encapsulamento.
Implementação Correta das Classes	3,0	Código funcional, uso de <code>__init__</code> , métodos coerentes, convenções de nome e boas práticas.
Demonstração de Encapsulamento	3,0	Exemplo de atributo privado/protegido, getters/setters ou métodos de acesso, evitando acessos diretos indevidos.
Organização e Clareza do Relatório	2,0	Explicações claras, exemplos de instâncias e resultados, discussões sobre melhorias.

Nota Final: Soma dos valores obtidos em cada critério, totalizando 10 pontos.

6. Conclusão

Ao final desta aula, os alunos deverão:

- **Criar e instanciar classes** em Python, compreendendo bem `_init_` e `self`.
- **Encapsular dados e métodos** de modo coerente, mantendo boa organização interna e uma interface limpa.
- **Dominar conceitos fundamentais de POO** (classes, objetos, encapsulamento) e estar prontos para avançar em herança e polimorfismo.
- **Perceber a importância** de modelar sistemas complexos via POO, dividindo responsabilidades e facilitando a manutenção.

Boas criações de classes e experimentações no mundo da Orientação a Objetos com Python!