Coding II

Ronierison Maciel

Agosto 2024

Quem sou eu?

Nome: Ronierison Maciel / Roni
Formação: Mestre em Ciência da Computação
Ocupação: Pesquisador, Professor e Desenvolvedor de Software
Hobbies: Jogar cartas, ficar com a família no final de semana conversando sobre diversos temas
Interesses: Carros, aprimoramento na área educacional, desenvolvimento de software, data science e machine learning
Email: ronierison.maciel@pe.senac.br
GitHub: `https://github.com/ronierisonmaciel`

Conteúdo

Objetivos da semana

Objetivos:

- **Compreender** os conceitos básicos da programação e a história do Python.
- **Configurar** o ambiente de desenvolvimento para começar a programar em Python.
- **Escrever** e executar o primeiro script em Python.

O que é programação?

Programação é o processo de escrever instruções que serão executadas por um computador. Componentes da Programação:

- **Código-Fonte:** As instruções que escrevemos.
- **Compilador/Interpretador:** Ferramenta que traduz o código para o computador.
- **Output:** O resultado das instruções executadas.

A Origem da linguagem Python

Por volta de 1989, **Guido van Rossum** trabalhava no CWI (Centrum Wiskunde & Informatica) e estava envolvido no desenvolvimento da linguagem ABC. A linguagem ABC era uma linguagem interpretada, ou seja, seu código era executado diretamente sem a necessidade de compilação prévia. Além disso, os blocos de código em ABC eram delimitados por indentação, uma característica que promove a legibilidade do código. ABC também oferecia tipos de dados de altíssimo nível, o que facilitava a programação para os usuários.

Limitações da linguagem ABC

Apesar de suas vantagens, a linguagem **ABC** tinha uma limitação significativa: não permitia extensões. Isso significava que os programadores não podiam adicionar novas funcionalidades ou adaptar a linguagem para atender a necessidades específicas que surgissem durante o desenvolvimento de projetos mais complexos.

Surgimento da necessidade de uma nova linguagem

Diante dessas dificuldades, surgiu a necessidade de uma linguagem de script semelhante à **ABC**, mas que permitisse extensões genéricas. **Guido** queria uma linguagem que combinasse a simplicidade e a clareza do ABC com a flexibilidade de extensões e o poder de manipulação do sistema que eles precisavam para o Amoeba.

```
1 HOW TO total sum list:
2     PUT 0 IN total
3     FOR number IN list:
4         ADD number TO total
5     RETURN total
6
```

Criação da linguagem Python

Durante os feriados de **Natal** de 1989, **Guido** começou a trabalhar nessa nova linguagem. Ele dedicou seu tempo disponível no ano seguinte ao desenvolvimento dessa linguagem, que eventualmente se tornaria o Python. A linguagem foi projetada para ser fácil de usar e aprender, incorporando a clareza do **ABC** e a robustez necessária para o Amoeba.

Python foi usado com sucesso crescente no projeto Amoeba, mostrando-se uma ferramenta poderosa e flexível. Em fevereiro de 1991, depois de pouco mais de um ano de desenvolvimento, Python foi publicado na USENET, tornando-se disponível para a comunidade de desenvolvedores.

Características e uso do Python

Python é uma linguagem de programação que apresenta muitas particularidades tornando-a utilizada e apreciada pela comunidade de desenvolvedores. Vamos explorar essas características e o impacto que elas têm no desenvolvimento de software.

- **Interpretada**: O código é executado diretamente pelo interpretador, sem necessidade de compilação prévia, facilitando a execução e teste rápido.
- **Orientada a Objetos**: Suporta a definição de classes e criação de objetos, organizando e estruturando o código de maneira modular e reutilizável.
- **Altíssimo Nível**: Abstrai complexidades do hardware, permitindo que os programadores foquem mais na lógica do problema e menos nos detalhes técnicos.

Características e uso do Python

- **Multiplataforma**: Executável em diversos sistemas operacionais (Windows, macOS, Linux) sem necessidade de modificações.
- **Interativa**: Oferece um modo interativo para executar comandos linha a linha, tornando o desenvolvimento e depuração mais dinâmicos e eficientes.

Atualmente

- Desenvolvimento Web,
- Ciência de Dados,
- Inteligência Artificial,
- Automação, entre outros.

Instalação do Python

Windows:

- Baixar o instalador em python.org
- Executar o instalador e marcar a opção "Add Python to PATH".
- Verificar a instalação usando o comando python –version no terminal.

Mac/Linux:

- Verificar a versão pré-instalada.
- Atualizar ou instalar usando gerenciadores de pacotes (brew, apt-get, etc.).
- Comando: python3 –version.

IDEs recomendadas

VSCode:

- Leve e flexível.
- Suporte para extensões e Python integrado.

PyCharm:

- Completa, com ferramentas de depuração avançadas.

Jupyter Notebook:

- Ideal para prototipação rápida e análise de dados.

Explorando o REPL

O que é REPL?:
Read, **E**val, **P**rint, **L**oop.

Ferramenta interativa do Python para testar comandos em tempo real. Comandos Básicos:

- Operações matemáticas: `2 + 2`
- Trabalhar com strings: `"Hello" * 3`
- Importar bibliotecas: `import math`

Explorando o IDLE

O que é o IDLE?:

- IDLE é um ambiente de desenvolvimento integrado simples que vem instalado com o Python, oferecendo uma interface gráfica para escrever e executar código Python.

Características do IDLE:

- **Editor de código:**
- Permite escrever, editar e salvar scripts Python.

Shell Interativo:

- Uma interface interativa que permite executar comandos Python em tempo real, semelhante ao REPL, mas com uma interface gráfica.

Continua...

Continuação

Depuração:

- Oferece uma ferramenta de depuração integrada para ajudar a encontrar e corrigir erros no código.

Como iniciar o IDLE: Iniciando:

- No Windows, você pode iniciar o IDLE pesquisando "IDLE" no menu Iniciar.
- No Mac ou Linux, pode ser iniciado a partir do terminal com o comando idle ou `import idlelib.idle`

Exemplo prático:

- **Escrevendo e Executando Código:**
- Digite um código simples, como print("Olá, mundo!"), no editor e execute-o para ver a saída na janela de shell interativa.

Criação de um script

Abra sua IDE ou editor de texto.

- Digite o código: print("Hello, World!").
- Salve o arquivo com extensão .py (ex.: hello.py).
- Executando o Script:
- Via terminal: python hello.py.
- Na IDE: Usando a funcionalidade de execução integrada.

Prática de laboratório

- **Tarefa 1:**

Configurar o ambiente de desenvolvimento seguindo as instruções.

- **Tarefa 2:**

Escrever e executar o script Hello, World!.

- **Tarefa 3:**

Explorar o REPL e testar comandos básicos.

- **Discussão:** Compartilhe dificuldades e soluções encontradas durante a configuração e execução.

Resumo da semana 1

Revisão:

- Conceitos básicos de programação.
- História e características do Python.
- Instalação do Python e configuração de ambiente.
- Criação e execução do primeiro script em Python.

Próximos passos:

- Continuar explorando o ambiente.
- Preparar-se para estudar tipos de dados e operadores na próxima semana.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Tutoriais online:

- Codecademy, W3Schools.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** os diferentes tipos de dados em Python.
- **Aprender** a utilizar operadores aritméticos, relacionais e lógicos.
- **Aplicar** esses conceitos na criação de expressões e scripts simples.

Tipos de dados primitivos

Tipos de dados são categorias que especificam quais tipos de valores uma variável pode armazenar.

Tipos de dados em Python:

- **Inteiros (int):** Números inteiros, positivos ou negativos.
- **Flutuantes (float):** Números decimais.
- **Strings (str):** Sequência de caracteres.
- **Booleanos (bool):** Verdadeiro (True) ou Falso (False).

Trabalhando com Inteiros e Flutuantes

Inteiros (int):

- **ex.** x = 10, y = -3
- Usado para contar, iterar, etc.

Flutuantes (float):

- **ex.** pi = 3.14, temperature = -5.5
- Usado para medições, cálculos precisos, etc.

Exemplo prático:

- a = 7
- b = 3.5
- c = a + b

Trabalhando com strings

Strings são sequências de caracteres delimitadas por aspas simples ('...') ou duplas ("...").

Operações básicas:

- **Concatenação:** `greeting = "Hello" + " " + "World!"`
- **Repetição:** `laugh = "ha" * 3` → Resultado: `"hahaha"`
- **Indexação:** `first_letter = "Python"[0]` → Resultado: `"P"`

Exemplo prático:

- `name = "Roni"`
- `welcome_message = "Hello, " + name + "!"`

Tipos booleanos e comparações

Booleanos representam valores verdadeiros ou falsos: True ou False.

Operações lógicas:

- **Comparações:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Lógicos:** `and`, `or`, `not`

Exemplo prático:

- `is_adult = age >= 18`
- `can_vote = is_adult and citizenship == "Brazil"`

Operadores aritméticos

Lista de operadores:

- **Adição:** `(+) → 5 + 3 = 8`
- **Subtração:** `(-) → 9 - 4 = 5`
- **Multiplicação:** `(*) → 7 * 6 = 42`
- **Divisão:** `(/) → 8 / 2 = 4.0`
- **Divisão inteira:** `(//) → 8 // 3 = 2`
- **Módulo:** `(%) → 8 % 3 = 2`
- **Exponenciação:** `(**) → 2 ** 3 = 8`

Operadores relacionais

Lista de operadores:

- **Igual a:** `==`
- **Diferente de:** `!=`
- **Maior que:** `>`
- **Menor que:** `<`
- **Maior ou igual a:** `>=`
- **Menor ou igual a:** `<=`

Exemplo prático:

- `result = (5 > 3)` → `Resultado: True`
- `result = (7 == 8)` → `Resultado: False`

Operadores lógicos

Lista de operadores:

- E (`and`): Retorna True se ambos os operandos forem verdadeiros.
- Ou (`or`): Retorna True se pelo menos um dos operandos for verdadeiro.
- Não (`not`): Inverte o valor lógico.

Exemplo prático:

- `result = (5 > 3) and (8 > 6)` → `Resultado: True`
- `result = (7 > 10) or (5 < 2)` → `Resultado: False`
- `result = not (4 == 4)` → `Resultado: False`

Criando expressões combinadas

Combinação de operadores:

Operadores aritméticos, relacionais e lógicos podem ser combinados para criar expressões complexas.

Precedência de operadores:

- **Parênteses:** `( )`
- **Exponenciação:** `**`
- **Multiplicação, divisão, módulo:** `*`, `/`, `%`
- **Adição e subtração:** `+`, `-`
- **Comparações:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Lógicos:** `not`, `and`, `or`

Exemplo prático:

- `x = (5 + 2) * 3 > 10 and (4 % 2 == 0)`

Prática de laboratório

- **Tarefa 1:**

Criar variáveis usando diferentes tipos de dados.

- **Tarefa 2:**

Escrever expressões aritméticas e lógicas, e prever seus resultados.

- **Tarefa 3:**

Implementar um script que peça ao usuário dois números e mostre os resultados das operações aritméticas básicas entre eles.

Resumo da semana 2

Revisão:

- Tipos de dados primitivos em Python: inteiros, flutuantes, strings, booleanos.
- Uso de operadores aritméticos, relacionais e lógicos.
- Criação de expressões combinadas e compreensão da precedência de operadores.

Próximos passos:

- Aplicar o conhecimento em estruturas condicionais na próxima semana.

Recursos adicionais

Leitura recomendada:

- Documentação oficial de tipos de dados em Python: docs.python.org

Exercícios online:

- Exercícios sobre tipos de dados e operadores em plataformas como Codecademy, HackerRank.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Entender** como usar estruturas condicionais em Python para tomar decisões no código.
- **Aprender** a utilizar `if`, `elif`, e `else` para controlar o fluxo do programa.
- **Aplicar** esses conceitos em problemas práticos.

Introdução às estruturas condicionais

Estruturas condicionais permitem que seu programa execute determinadas partes do código com base em condições específicas.

- **Fluxo de Decisão:** Dependendo de uma condição (True ou False), diferentes blocos de código serão executados.
- **Importância:** Fundamental para criar programas dinâmicos que reagem de maneiras diferentes a diferentes entradas.

Bloco if

sintaxe básica:

```
if condição:
    # bloco de código a ser executado se a condição for verdadeira
```

Exemplo prático:

```
age = 18
if age >= 18:
    print("Você é maior de idade.")
```

Explicação: Se **age** for maior ou igual a 18, a mensagem "Você é maior de idade." será exibida.
Nota: A indentação é crucial em Python para definir blocos de código.

Usando else

sintaxe básica:

```
    if condição:
    # bloco de código 1
else:
    # bloco de código 2
```

Exemplo prático:

```
age = 16
if age >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

Explicação: Se a condição do **if** não for verdadeira, o bloco **else** será executado.

Bloco elif

sintaxe básica:

```
if condição1:
    # bloco de código 1
elif condição2:
    # bloco de código 2
else:
    # bloco de código 3
```

Exemplo prático:

```
score = 85
if score >= 90:
    print("Nota A")
elif score >= 80:
    print("Nota B")
else:
    print("Nota C")
```

Explicação: **elif** permite testar múltiplas condições em sequência, parando na primeira que for verdadeira.

Aninhando estruturas condicionais

Estruturas condicionais podem ser aninhadas dentro de outras para lidar com cenários mais complexos.

Exemplo prático:

```
age = 20
if age >= 18:
    if age >= 65:
        print("Você é um idoso.")
    else:
        print("Você é um adulto.")
else:
    print("Você é menor de idade.")
```

Explicação: O bloco **if** interno verifica uma segunda condição se a primeira for verdadeira.

Condições complexas

Condições podem ser combinadas usando operadores lógicos (and, or, not) para formar expressões mais complexas.

```
temperature = 22
is_sunny = True

if temperature > 20 and is_sunny:
    print("Ótimo dia para um passeio!")
elif temperature <= 20 or not is_sunny:
    print("Talvez ficar em casa seja uma boa ideia.")
else:
    print("Que dia estranho.")
```

Explicação: O programa decide o que exibir com base em duas condições diferentes.

Evitando erros comuns

- **Indentação:**

Regra: Cada bloco de código em Python deve ser corretamente indentado.

```
if condição:
print("Isso causará um erro!")  # Sem indentação correta
```

- **Comparação com = ao invés de ==:**

Erro Comum: Usar = (atribuição) ao invés de == (comparação) em uma condição.

```
if x = 5:  # Erro: deve ser '=='
    print("Cinco")
```

Prática de laboratório

- **Tarefa 1:**

Escreva um script que leia uma nota (0-100) e imprima o conceito (A, B, C, D, F) baseado em faixas de valores.

- **Tarefa 2:**

Crie um programa que peça ao usuário sua idade e decida se ele pode votar, beber e/ou se ele é aposentado.

- **Tarefa 3:**

Desenvolva um pequeno jogo de adivinhação onde o usuário deve acertar um número entre 1 e 10, com dicas baseadas na entrada do usuário.

Resumo da semana 3

Revisão:

- **Compreendemos** como usar estruturas condicionais if, elif, e else.
- **Exploramos** aninhamento de condições e combinações com operadores lógicos.
- **Aplicamos** esses conceitos em exemplos práticos.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre estruturas condicionais em plataformas como Codecademy, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** como usar estruturas de repetição para automatizar tarefas em Python.
- **Aprender** a utilizar os laços for e while para iterar sobre sequências e executar blocos de código repetidamente.
- **Entender** o uso de comandos de controle como break, continue e pass.

Introdução às estruturras de repetição

Estruturas de repetição permitem que um bloco de código seja executado múltiplas vezes com base em uma condição ou sobre uma sequência.

- **Importância:**

Essencial para automatizar tarefas repetitivas, processar listas de dados, e realizar iterações controladas.

O laço `for`

sintaxe básica:

```
for item in sequência:
    # bloco de código a ser repetido
```

Funcionamento:

O laço `for` itera sobre cada item em uma sequência (como listas, strings, tuplas).

Exemplo prático:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Explicação: O código imprime cada fruta na lista.

Iterando com range()

A função range() gera uma sequência de números, muito usada com for.

sintaxe:

```
for i in range(start, stop, step):
    # bloco de código a ser repetido
```

Exemplo prático:

```
for i in range(1, 6):
    print(i)
```

Explicação: Imprime números de 1 a 5.

O laço `while`

sintaxe básica

```
while condição:
    # bloco de código a ser repetido
```

Funcionamento:

O laço `while` repete o bloco de código enquanto a condição for verdadeira.

Exemplo prático:

```
count = 1
while count <= 5:
    print(count)
    count += 1
```

Explicação: O código imprime números de 1 a 5, incrementando count a cada iteração.

Comandos de controle: `break`, `continue`, `pass`

O **break** encerra o laço imediatamente.
Exemplo:

```
for i in range(1, 10):
    if i == 5:
        break
    print(i)
```

Explicação: O laço é interrompido quando i é igual a 5. Já o **continue** pula para a próxima iteração do laço.
Exemplo:

```
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Explicação: Quando i é 3, a impressão é pulada.

Comandos de controle: `break`, `continue`, `pass`

O **pass** não faz nada; é usado como placeholder.

Exemplo:

```
for i in range(1, 6):
    if i == 3:
        pass
    print(i)
```

Explicação: O pass não afeta o fluxo do laço.

Iterando sobre strings

Strings são iteráveis, o que significa que podemos iterar sobre cada caractere usando um laço for.

Exemplo prático:

```
1 word = "Python"
2 for letter in word:
3     print(letter)
```

Explicação: O código imprime cada letra da string "Python" em uma nova linha.

Laços aninhados

Um laço pode ser colocado dentro de outro, criando um laço aninhado, útil para trabalhar com matrizes ou listas de listas.

Exemplo prático:

```
1 for i in range(1, 4):
2     for j in range(1, 4):
3         print(f"i={i}, j={j}")
```

Explicação: Para cada valor de i, o laço interno itera j de 1 a 3.

Evitando loops infinitos

Um loop infinito ocorre quando a condição de um laço while nunca se torna falsa.

Exemplo:

```
1 while True:
2     print("Isso vai continuar para sempre... ou até você
    interromper.")
```

Dicas para Evitar:

- Verifique cuidadosamente as condições de saída.
- Garanta que a variável de controle seja corretamente atualizada.

Prática de laboratório

- **Tarefa 1**

Escreva um laço for que percorre uma lista de números e imprime apenas os números pares.

- **Tarefa 2**

Crie um script que solicite ao usuário um número e use um laço while para calcular o fatorial desse número.

- **Tarefa 3**

Desenvolva um jogo simples em que o usuário deve adivinhar um número aleatório entre 1 e 100, com dicas fornecidas após cada tentativa. Use while para repetir até que o usuário acerte.

Resumo da semana 4

Revisão:

- Compreendemos o uso de laços for e while para iterar sobre sequências e condições.
- Exploramos comandos de controle como break, continue, e pass.
- Praticamos a criação de laços aninhados e discutimos a importância de evitar loops infinitos.

Próximos passos:

- Na próxima semana, aprenderemos sobre funções em Python, que permitem modularizar e reutilizar código.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre loops em plataformas como LeetCode, HackerRank.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** o conceito de funções e sua importância em programação.
- **Aprender** a definir e chamar funções em Python.
- **Entender** o uso de parâmetros e valores de retorno.
- **Aplicar** o conceito de funções em problemas práticos.

Introdução às funções

Definição:

- Funções são blocos de código reutilizáveis que executam uma tarefa específica.

Importância:

- Facilita a organização e a modularização do código.
- Promove a reutilização e a manutenção eficiente do código.

Analogia:

- Pense em uma função como uma "receita": ela recebe ingredientes (parâmetros), executa um conjunto de instruções e pode retornar um resultado.

Definindo uma função

sintaxe básica:

```
def nome_da_funcao(param1, param2):
    # bloco de código a ser executado
    return valor_de_retorno
```

Componentes:

- `def`: Palavra-chave que indica a definição de uma função.
- `nome_da_funcao`: Nome da função.
- `param1, param2`: Parâmetros (opcionais).

Exemplo prático:

```
def saudacao(nome):
    return f"Olá, {nome}!"
```

Chamando uma função

sintaxe básica:

```
1 nome_da_funcao(argumento1, argumento2)
```

Funcionamento:

Quando uma função é chamada, o controle do programa passa para a função, os parâmetros são passados, e o bloco de código é executado.

```
1 mensagem = saudacao("Roni")
2 print(mensagem)
```

Parâmetros e argumentos

Definição:

- **Parâmetros:** Variáveis listadas na definição da função.
- **Argumentos:** Valores passados para os parâmetros na chamada da função.

Tios de Parâmetro:

- **Obrigatórios:** Devem ser fornecidos ao chamar a função.
- **Opcionais** (com valores padrão): Têm um valor padrão se não forem fornecidos.

```
1 def saudacao(nome, saudacao="Olá"):
2     return f"{saudacao}, {nome}!"
3
4 print(saudacao("Roni"))
5 print(saudacao("Roni", "Bom dia"))
```

Retornando valores de funções

Definição:

- Uma função pode retornar um valor usando a palavra-chave `return`.
- `return` finaliza a execução da função e retorna o valor especificado.

Exemlo prático:

```
1 def soma(a, b):
2     return a + b
3
4 resultado = soma(5, 3)
5 print(resultado)  # Saída: 8
```

Nota: Se uma função não tiver **return**, ela retorna **None** por padrão.

Escopo de variáveis

Definição:

- **Escopo Local:** Variáveis definidas dentro de uma função só existem dentro dessa função.
- **Escopo Global:** Variáveis definidas fora de qualquer função podem ser acessadas em qualquer lugar no código.

Exemplo prático:

```
1 x = 10  # Variável global
2
3 def funcao():
4     x = 5  # Variável local
5     print(x)
6
7 funcao()  # Saída: 5
8 print(x)  # Saída: 10
```

Nota: Evita conflitos de nomes e facilita a manutenção do código.

Funções Aninhadas e Closures

Definição:

- **Funções Aninhadas:** Uma função pode ser definida dentro de outra função.
- **Closure:** Uma função interna que se lembra das variáveis no escopo de sua função externa, mesmo depois que a função externa foi executada.

Exemplo prático:

```
1 def pai(num):
2     def filho(x):
3         return x + num
4     return filho
5
6 adicao = pai(10)
7 print(adicao(5))  # Saída: 15
```

Documentando funções

Docstrings:

- **Definição:** Docstrings são cadeias de caracteres que documentam uma função.
- **Uso:** Colocadas logo após a definição da função para descrever seu propósito e parâmetros.

Exemplo prático:

```
1  def saudacao(nome):
2      """
3      Esta função retorna uma saudação personalizada.
4
5      Parâmetros:
6      nome (str): Nome da pessoa a ser saudada.
7
8      Retorno:
9      str: Uma saudação na forma de string.
10     """
11     return f"Olá, {nome}!"
```

Prática de laboratório

Tarefa 1:

- Escreva uma função media que recebe três números e retorna a média aritmética deles.

Tarefa 2:

- Crie uma função fatorial que calcule o fatorial de um número usando um loop while.

Tarefa 3:

- Desenvolva uma função conversor_temperatura que converte graus Celsius para Fahrenheit e vice-versa, dependendo de um parâmetro adicional.

Resumo da semana 5

Revisão:

- Aprendemos a definir e chamar funções, a trabalhar com parâmetros e retornos.
- Discutimos escopo de variáveis e vimos a importância das docstrings.
- Aplicamos esses conceitos em exercícios práticos.

Próximos passos:

- Na próxima semana, aprenderemos sobre estruturas de dados mais avançadas, como listas e dicionários.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre funções em plataformas como Codecademy, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** o conceito de listas e sua importância como estrutura de dados em Python.
- **Aprender** a criar, acessar, modificar e iterar sobre listas.
- **Explorar** métodos e operações comuns em listas.
- **Aplicar** listas em problemas práticos.

Introdução às listas

Definição:

- Uma lista é uma coleção ordenada e mutável de itens em Python, que podem ser de tipos variados.

Importância:

- Listas são uma das estruturas de dados mais versáteis e amplamente usadas, permitindo armazenar e manipular múltiplos itens em uma única variável.

Criando listas

sintaxe básica:

```
nome_da_lista = [item1, item2, item3, ...]
```

Exemplo prático:

```
frutas = ["maçã", "banana", "cereja"]
numeros = [1, 2, 3, 4, 5]
misturada = ["texto", 42, 3.14, True]
```

Características:

- **Ordenada:** A ordem dos itens é preservada.
- **Mutável:** Itens podem ser modificados, adicionados ou removidos.

Acessando itens em listas

Indexação:

- Cada item em uma lista tem uma posição (índice) que começa em 0.
- sintaxe: nome_da_lista[indice]

Exemplo prático:

```
1 frutas = ["maçã", "banana", "cereja"]
2 print(frutas[0])  # Saída: maçã
3 print(frutas[2])  # Saída: cereja
```

Indexação negativa:

- Permite acessar itens de trás para frente.
- Exemplo: frutas[-1] retorna "cereja".

Modificando listas

Modificando itens:

- Listas são mutáveis, o que significa que podemos modificar seus itens.
- sintaxe: nome_da_lista[indice] = novo_valor

Exemplo Prático:

```
1 frutas = ["maçã", "banana", "cereja"]
2 frutas[1] = "laranja"
3 print(frutas)  # Saída: ["maçã", "laranja", "cereja"]
```

Métodos comuns de listas

append(item):

- Adiciona um item ao final da lista.
- **Exemplo:** frutas.append("uva")

insert(indice, item):

- Insere um item em uma posição específica.
- **Exemplo:** frutas.insert(1, "abacaxi")

remove(item):

- Remove a primeira ocorrência de um item na lista.
- **Exemplo:** frutas.remove("banana")

pop(indice):

- Remove e retorna o item no índice especificado (ou o último item, se o índice não for fornecido).
- **Exemplo:** ultimo = frutas.pop()

Métodos comuns de listas

sort():

- Ordena a lista em ordem crescente.
- **Exemplo:** numeros.sort()

reverse():

- Inverte a ordem dos itens na lista.
- **Exemplo:** frutas.reverse()

Operações comuns com listas

Concatenando listas:

- **sintaxe:** lista1 + lista2

Exemplo:

```
1 lista1 = [1, 2, 3]
2 lista2 = [4, 5, 6]
3 lista3 = lista1 + lista2  # Saída: [1, 2, 3, 4, 5, 6]
```

Repetindo listas:

- **sintaxe:** lista * n

Exemplo:

```
1 frutas = ["maçã", "banana"]
2 nova_lista = frutas * 2  # Saída: ["maçã", "banana", "maçã",
      "banana"]
```

Operações comuns com listas

Verificando existência de itens:

- **sintaxe:** item in lista

Exemplo:

```
1 if "maçã" in frutas:
2     print("A lista contém maçã.")
```

Iterando sobre listas

Uso de for:

- Iteração sobre cada item da lista.
- **sintaxe:**

```
for item in lista:
    # bloco de código a ser executado para cada item
```

Exemplo prático:

```
frutas = ["maçã", "banana", "cereja"]
for fruta in frutas:
    print(fruta)
```

Saída

```
maçã
banana
cereja
```

List comprehensions

Definição:

- Uma maneira concisa de criar listas usando uma expressão.
- **sintaxe:** [expressao for item in lista if condicao]

Exemplo prático:

```
1 quadrados = [x**2 for x in range(10)]
2 print(quadrados)  # Saída: [0, 1, 4, 9, 16, 25, 36, 49, 64,
    81]
```

Explicação:

- A expressão x**2 é aplicada a cada x no intervalo de 0 a 9, gerando uma nova lista de quadrados.

Fatiamento de listas

Definição:

- Fatiamento permite acessar subpartes de uma lista.
- **sintaxe:** lista[inicio:fim:passo]

Exemplo prático:

```
1 numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 print(numeros[2:5])     # Saída: [2, 3, 4]
3 print(numeros[:4])      # Saída: [0, 1, 2, 3]
4 print(numeros[::2])     # Saída: [0, 2, 4, 6, 8]
```

Explicação:

- **inicio:** posição inicial (inclusiva).
- **fim:** posição final (exclusiva).
- **passo:** salto entre elementos.

Prática de laboratório

Tarefa 1:

- Crie uma lista de números de 1 a 10 e, em seguida, faça um loop para calcular a soma de todos os números.

Tarefa 2:

- Escreva um script que recebe uma lista de nomes e retorna uma nova lista contendo apenas os nomes que começam com uma vogal.

Tarefa 3:

- Desenvolva um programa que solicita ao usuário uma lista de números e, em seguida, remove todos os números duplicados da lista.

Resumo da semana 6

Revisão:

- **Compreendemos** como criar, acessar e modificar listas.
- **Exploramos** métodos comuns e operações em listas, como concatenação, repetição e fatiamento.
- **Aplicamos** o conceito de listas em exercícios práticos.

Próximos passos:

- Na próxima semana, aprenderemos sobre outras estruturas de dados importantes em Python: tuplas e dicionários.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre listas em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** o conceito de tuplas e dicionários e suas diferenças em relação a listas.
- **Aprender** a criar, acessar e modificar tuplas e dicionários.
- **Explorar** operações e métodos comuns em tuplas e dicionários.
- **Aplicar** esses conceitos em problemas práticos.

Introdução às tuplas

Definição:

- Uma tupla é uma coleção ordenada e imutável de itens em Python.

Características:

- **Imutável:** Após a criação, os itens não podem ser modificados.
- **Ordenada:** A ordem dos itens é preservada.

Quando usar:

- Quando os dados não precisam ser alterados e você deseja garantir sua integridade.

Criando tuplas

sintaxe básica:

```
nome_da_tupla = (item1, item2, item3, ...)
```

Exemplo prático:

```
coordenadas = (10.0, 20.0)
cores = ("vermelho", "verde", "azul")
misturada = ("texto", 42, 3.14, True)
```

Nota:

- Tuplas com um único item devem incluir uma vírgula: (item,)

Acessando itens em tuplas

Indexação:

- Semelhante às listas, cada item em uma tupla tem uma posição (índice) que começa em 0.
- **sintaxe:** nome_da_tupla[indice]

Exemplo prático:

```
1 cores = ("vermelho", "verde", "azul")
2 print(cores[0])  # Saída: vermelho
3 print(cores[2])  # Saída: azul
```

Indexação negativa:

- **Exemplo:** cores[-1] retorna "azul".

Métodos comuns de tuplas

count(item):

- Retorna o número de vezes que um item aparece na tupla.
- **Exemplo:** coordenadas.count(10.0)

index(item):

- Retorna o índice da primeira ocorrência do item na tupla.
- **Exemplo:** cores.index("verde")

Operações com tuplas

Concatenando tuplas:

- **sintaxe:** tupla1 + tupla2

Exemplo:

```
1 tupla1 = (1, 2, 3)
2 tupla2 = (4, 5, 6)
3 tupla3 = tupla1 + tupla2  # Saída: (1, 2, 3, 4, 5, 6)
```

Repetindo tuplas:

- **sintaxe:** tupla * n

Exemplo:

```
1 cores = ("vermelho", "verde")
2 nova_tupla = cores * 2  # Saída: ("vermelho", "verde", "
    vermelho", "verde")
```

Iterando sobre tuplas

Uso de for:

- Iteração sobre cada item da tupla.

sintaxe:

```
for item in tupla:
    # bloco de código a ser executado para cada item
```

Exemplo prático:

```
cores = ("vermelho", "verde", "azul")
for cor in cores:
    print(cor)
```

Saída

```
vermelho
verde
azul
```

Fatiamento de tuplas

Definição:

- Fatiamento permite acessar subpartes de uma tupla.
- **sintaxe:** tupla[inicio:fim:passo]

Exemplo prático:

```
1 numeros = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
2 print(numeros[2:5])     # Saída: (2, 3, 4)
3 print(numeros[:4])      # Saída: (0, 1, 2, 3)
4 print(numeros[::2])     # Saída: (0, 2, 4, 6, 8)
```

Introdução aos dicionários

Definição:

- Um dicionário é uma coleção desordenada de pares chave-valor em Python.

Características:

- **Mutável:** Pode-se adicionar, modificar ou remover itens.
- **Desordenado:** A ordem dos itens não é garantida.

Quando usar:

- Quando precisamos mapear valores a chaves únicas e de fácil acesso.

Criando dicionários

sintaxe básica:

```
nome_do_dicionario = {chave1: valor1, chave2: valor2, ...}
```

Exemplo prático:

```
aluno = {"nome": "Roni", "idade": 25, "curso": "Engenharia"}
estoque = {"maçã": 10, "banana": 15, "laranja": 8}
```

Acessando e modificando itens em dicionários

Acessando valores:

- **sintaxe:** nome_do_dicionario[chave]

Exemplo:

```
1 print(aluno["nome"])  # Saída: Roni
```

Modificando valores:

- **sintaxe:** nome_do_dicionario[chave] = novo_valor

Exemplo:

```
1 aluno["idade"] = 26
```

Métodos comuns de dicionários

get(chave, valor_padrao):

- Retorna o valor associado à chave, ou um valor padrão se a chave não existir.
- **Exemplo:** aluno.get("cidade", "Não informado")

keys():

- Retorna uma lista das chaves do dicionário.
- **Exemplo:** aluno.keys()

values():

- Retorna uma lista dos valores do dicionário.
- **Exemplo:** aluno.values()

Métodos comuns de dicionários

items():

- Retorna uma lista de pares (chave, valor).
- **Exemplo:** aluno.items()

pop(chave):

- Remove e retorna o valor associado à chave.
- **Exemplo:** aluno.pop("curso")

Adicionando e removendo itens

Adicionando Itens:

- sintaxe: nome_do_dicionario[nova_chave] = valor

Exemplo:

```
1 aluno["cidade"] = "Recife"
```

Removendo itens:

- sintaxe: del nome_do_dicionario[chave]

Exemplo:

```
1 del aluno["idade"]
```

Iterando sobre ddicionários

Iteração sobre chaves:

sintaxe:

```
for chave in dicionario:
    # bloco de código a ser executado para cada chave
```

Iteração sobre pares chave-valor:

sintaxe:

```
for chave, valor in dicionario.items():
    # bloco de código a ser executado para cada par
```

Exemplo prático:

```
for chave, valor in aluno.items():
    print(f"{chave}: {valor}")
```

Saída

```
nome: Roni
curso: Engenharia
cidade: Recife
```

Dicionários aninhados

Definição:

- Dicionários podem conter outros dicionários, formando uma estrutura mais complexa.

Exemplo prático:

```
alunos = {
    "aluno1": {"nome": "Roni", "idade": 25},
    "aluno2": {"nome": "Maria", "idade": 22},
}
print(alunos["aluno1"]["nome"])  # Saída: Roni
```

Prática de laboratório

Tarefa 1:

- Crie uma tupla com as 12 horas de um relógio e itere sobre ela para exibir cada hora.

Tarefa 2:

- Escreva um script que solicita informações de um livro (título, autor, ano) e armazene-as em um dicionário.

Tarefa 3:

- Desenvolva um programa que recebe um dicionário de produtos e seus preços, e retorna o produto mais caro.

Resumo da semana 7

Revisão:

- **Compreendemos** como criar, acessar e modificar tuplas e dicionários.
- **Exploramos** métodos e operações comuns em tuplas e dicionários.
- **Aplicamos** esses conceitos em exercícios práticos.

Próximos passos:

- Na próxima semana, aprenderemos sobre entrada e saída de dados em Python, incluindo leitura e gravação de arquivos.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre tuplas e dicionários em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** como capturar e manipular a entrada de dados do usuário.
- **Aprender** a realizar operações de entrada e saída com arquivos.
- **Entender** o uso de funções integradas para formatação de saída.
- **Aplicar** esses conceitos em problemas práticos que envolvem interação com o usuário e manipulação de arquivos.

Entrada de dados com input()

Definição:

- A função input() captura a entrada do usuário como uma string.

sintaxe:

```
1 variavel = input("Mensagem para o usuário: ")
```

Exemplo prático:

```
1 nome = input("Digite seu nome: ")
2 print(f"Olá, {nome}!")
```

Nota:

- O valor retornado por input() é sempre uma string; conversões de tipo podem ser necessárias.

Convertendo entrada de dados

Conversões comuns:

- **Inteiro:** int()
- **Flutuante:** float()

Exemplo prático:

```
1 idade = int(input("Digite sua idade: "))
2 altura = float(input("Digite sua altura em metros: "))
3 print(f"Você tem {idade} anos e {altura} metros de altura.")
```

Entrada de dados validada

Validação de entrada:

- Garantir que os dados fornecidos pelo usuário sejam válidos antes de prosseguir.

Exemplo prático:

```
while True:
    idade = input("Digite sua idade: ")
    if idade.isdigit():
        idade = int(idade)
        break
    else:
        print("Por favor, insira um número válido.")
```

Nota:

- A função isdigit() verifica se uma string contém apenas dígitos.

Saída de dados com print()

Definição:

- A função print() exibe a saída de dados no console.

sintaxe:

```
1 print(valor1, valor2, ..., sep=' ', end='\n')
```

Exemplo prático:

```
1 print("Python", "é", "divertido", sep="-", end="!\n")
```

Saída: Python-é-divertido!

Formatação de saída

Métodos de Formatação:

- **Concatenação de Strings:**

```
nome = "Roni"
print("Olá, " + nome + "!")
```

Formatação com %:

```
print("Eu tenho %d anos." % 25)
```

Método .format():

```
print("Meu nome é {} e tenho {} anos.".format("Roni", 25))
```

F-strings (Python 3.6+):

```
nome = "Roni"
idade = 25
print(f"Meu nome é {nome} e tenho {idade} anos.")
```

Abrindo e fechando arquivos

Abertura de arquivos:

- A função open() abre um arquivo para leitura, gravação ou anexação.

sintaxe:

```
arquivo = open("nome_do_arquivo.txt", modo)
```

Modos comuns:

- **'r'**: Leitura (padrão).
- **'w'**: Escrita (sobrescreve o arquivo existente).
- **'a'**: Anexação (escreve ao final do arquivo).

Fechamento de arquivos:

- É importante fechar o arquivo após a operação.

sintaxe:

```
arquivo.close()
```

Leitura de arquivos

Métodos de Leitura:

- **read():** Lê o arquivo inteiro.

```
conteudo = arquivo.read()
```

- **readline():** Lê uma linha de cada vez.

```
linha = arquivo.readline()
```

- **readlines():** Lê todas as linhas e retorna uma lista.

```
linhas = arquivo.readlines()
```

Exemplo prático:

```
with open("exemplo.txt", "r") as arquivo:
    conteudo = arquivo.read()
    print(conteudo)
```

Nota: Usar with garante que o arquivo seja fechado automaticamente.

Escrita em arquivos

Métodos de escrita:

- **write(texto):** Escreve uma string no arquivo.

```
arquivo.write("Texto a ser escrito.")
```

- **writelines(lista_de_linhas):** Escreve uma lista de strings no arquivo.

```
arquivo.writelines(["Linha 1\n", "Linha 2\n"])
```

Exemplo prático:

```
with open("exemplo.txt", "w") as arquivo:
    arquivo.write("Olá, mundo!\n")
    arquivo.write("Esta é uma nova linha.")
```

Manipulação de arquivos com with

Uso de with:

- O uso de with é a maneira mais segura de abrir e manipular arquivos, garantindo que eles sejam fechados corretamente.

sintaxe:

```
with open("nome_do_arquivo.txt", "modo") as arquivo:
    # operações com o arquivo
```

Exemplo prático:

```
with open("dados.txt", "a") as arquivo:
    arquivo.write("Dados adicionados ao arquivo.\n")
```

Prática de laboratório

Tarefa 1:

- Escreva um script que solicita ao usuário uma lista de compras e as salva em um arquivo de texto.

Tarefa 2:

- Desenvolva um programa que lê um arquivo de texto contendo números, um por linha, e calcula a soma desses números.

Tarefa 3:

- Crie um sistema simples de registro de notas, onde o usuário insere o nome do aluno e a nota, e o programa salva essas informações em um arquivo.

Resumo da semana 8

Revisão:

- **Compreendemos** como capturar entrada do usuário e realizar saídas formatadas.
- **Exploramos** a leitura e escrita de arquivos em Python, incluindo o uso seguro de with.
- **Aplicamos** esses conceitos em exercícios práticos.

Próximos passos:

- Na próxima semana, começaremos a explorar o conceito de Programação Orientada a Objetos (POO) em Python.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre entrada e saída de dados em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** as operações fundamentais para manipulação de arquivos em Python.
- **Aprender** a ler, escrever e manipular diferentes tipos de arquivos.
- **Explorar** operações avançadas como leitura e escrita de arquivos CSV e JSON.
- **Aplicar** esses conceitos em problemas práticos envolvendo manipulação de dados.

Abertura de arquivos

Revisão:

- A função open() é usada para abrir arquivos para leitura, gravação ou anexação.

sintaxe:

```
1 arquivo = open("nome_do_arquivo.txt", "modo")
```

Modos comuns:

- **'r'**: Leitura (padrão).
- **'w'**: Escrita (sobrescreve o arquivo existente).
- **'a'**: Anexação (adiciona ao final do arquivo).
- **'b'**: Modo binário (para arquivos binários como imagens).

Fechamento:

- Use arquivo.close() para fechar o arquivo após a operação.

Manipulando arquivos CSV

O que é um arquivo CSV:

- **CSV** (Comma-Separated Values) é um formato de arquivo usado para armazenar dados tabulares.

Leitura de arquivos CSV:

- **Uso do módulo csv:**

```
import csv

with open("dados.csv", "r") as arquivo:
    leitor = csv.reader(arquivo)
    for linha in leitor:
        print(linha)
```

Escrita em arquivos CSV:

- **Exemplo:**

```
with open("dados.csv", "w", newline='') as arquivo:
    escritor = csv.writer(arquivo)
    escritor.writerow(["Nome", "Idade", "Cidade"])
    escritor.writerow(["Roni", 25, "Recife"])
```

Manipulando arquivos JSON

O que é um arquivo JSON:

- **JSON** (JavaScript Object Notation) é um formato leve de troca de dados.

Leitura de arquivos JSON:

- **Uso do módulo json:**

```
1 import json
2
3 with open("dados.json", "r") as arquivo:
4     dados = json.load(arquivo)
5     print(dados)
```

Escrita em arquivos JSON:

- **Exemplo:**

```
1 dados = {"nome": "Roni", "idade": 25, "cidade": "Recife"}
2
3 with open("dados.json", "w") as arquivo:
4     json.dump(dados, arquivo)
```

Manipulação de arquivos binários

O que é um Arquivo Binário:

- Arquivos binários são qualquer arquivo que não seja de texto, como imagens, vídeos, executáveis, etc.

Leitura de Arquivos Binários:

- **Exemplo:**

```
1 with open("imagem.jpg", "rb") as arquivo:
2     conteudo = arquivo.read()
```

Escrita em Arquivos Binários:

- **Exemplo:**

```
1 with open("copia_imagem.jpg", "wb") como arquivo:
2     arquivo.write(conteudo)
```

Operações com arquivos e diretórios

Uso do módulo os:

- **os.path.exists(caminho):** Verifica se um arquivo ou diretório existe.
- **os.remove(caminho):** Remove um arquivo.
- **os.mkdir(caminho):** Cria um novo diretório.
- **os.listdir(caminho):** Lista todos os arquivos e diretórios em um diretório.

Exemplo prático:

```
1 import os
2
3 if not os.path.exists("novo_diretorio"):
4     os.mkdir("novo_diretorio")
5
6 arquivos = os.listdir(".")
7 print(arquivos)
```

Prática de laboratório

Tarefa 1:

- Escreva um script que lê um arquivo CSV contendo dados de alunos (nome, idade, nota) e calcula a média das notas.

Tarefa 2:

- Desenvolva um programa que lê um arquivo JSON com informações de um catálogo de produtos e permite ao usuário buscar um produto pelo nome.

Tarefa 3:

- Crie um script que copia uma imagem de um diretório para outro e renomeia a cópia.

Resumo da semana 9

Revisão:

- **Compreendemos** as operações fundamentais de manipulação de arquivos, incluindo leitura e escrita de arquivos CSV e JSON.
- **Exploramos** a manipulação de arquivos binários e operações com diretórios.
- **Aplicamos** esses conceitos em exercícios práticos.

Próximos Passos:

- Na próxima semana, começaremos a estudar Programação Orientada a Objetos (POO) em Python, uma abordagem poderosa para organizar e estruturar seu código.

Objetivos da semana

Objetivos:

- **Compreender** os conceitos fundamentais da Programação Orientada a Objetos (POO).
- **Aprender** a definir classes, objetos, atributos e métodos em Python.
- **Explorar** conceitos como encapsulamento, herança e polimorfismo.
- **Aplicar** POO para resolver problemas práticos e organizar o código de maneira eficiente.

O que é Programação Orientada a Objetos (POO)?

Definição:

- A Programação Orientada a Objetos é um paradigma de programação que usa "objetos" – que são instâncias de classes – para modelar e representar entidades do mundo real ou conceitos.

Principais conceitos:

- **Objeto:** Uma instância de uma classe.
- **Classe:** Um molde ou definição para criar objetos.
- **Atributo:** Propriedade ou característica de um objeto.
- **Método:** Função associada a um objeto.

Nota:

- Organiza o código de maneira modular e reutilizável, facilitando a manutenção e expansão.

Definindo classes e objetos

Classe:

- Uma classe define a estrutura e o comportamento de objetos.

sintaxe:

```
class NomeDaClasse:
    # atributos e métodos
```

Objeto:

- Um objeto é uma instância de uma classe.

sintaxe:

```
objeto = NomeDaClasse()
```

Exemplo prático:

```
class Carro:
    pass

meu_carro = Carro()
```

Atributos de Instância

Definição:

- Atributos de instância são variáveis associadas a cada objeto individualmente.

Inicializando Atributos com __init__():

- O método __init__() é o construtor da classe e é chamado automaticamente quando um objeto é criado.

sintaxe:

```
1 class Carro:
2     def __init__(self, marca, modelo):
3         self.marca = marca
4         self.modelo = modelo
```

Continua na próxima página...

Continuação

Exemplo prático:

```
1 class Carro:
2     def __init__(self, marca, modelo):
3         self.marca = marca
4         self.modelo = modelo
5 
6 meu_carro = Carro("Toyota", "Corolla")
7 print(meu_carro.marca, meu_carro.modelo)  # Saída: Toyota
    Corolla
```

Métodos

Definição:

- Métodos são funções definidas dentro de uma classe, que descrevem os comportamentos dos objetos.

sintaxe:

```
class NomeDaClasse:
    def metodo(self):
        # código do método
```

Exemplo prático:

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

    def exibir_info(self):
        print(f"Carro: {self.marca} {self.modelo}")

meu_carro = Carro("Toyota", "Corolla")
meu_carro.exibir_info()  # Saída: Carro: Toyota Corolla
```

Atributos e métodos de classe

Atributos de classe:

- Pertencem à classe e são compartilhados por todas as instâncias.

sintaxe:

```
class Carro:
    rodas = 4  # atributo de classe
```

Métodos de classe:

- Definidos com o decorador `@classmethod`.

Exemplo:

```
class Carro:
    rodas = 4

    @classmethod
    def exibir_rodas(cls):
        print(f"Todos os carros têm {cls.rodas} rodas.")

Carro.exibir_rodas()  # Saída: Todos os carros têm 4 rodas.
```

Conceitos avançados de POO

Encapsulamento

- O encapsulamento esconde detalhes internos de um objeto e protege seus dados.

Atributos Privados:

- Atributos podem ser tornados privados prefixando seus nomes com dois underscores `__`.

Exemplo:

```
class ContaBancaria:
    def __init__(self, saldo):
        self.__saldo = saldo

    def exibir_saldo(self):
        print(f"Saldo: R$ {self.__saldo}")

conta = ContaBancaria(1000)
conta.exibir_saldo()  # Saída: Saldo: R$ 1000
```

Herança

- Herança permite que uma classe herde atributos e métodos de outra classe, promovendo a reutilização de código.
- **Classe base e classe derivada:**

sintaxe:

```
1 class ClasseBase:
2     # atributos e métodos
3
4 class ClasseDerivada(ClasseBase):
5     # atributos e métodos adicionais
```

Continua...

Continuação

Exemplo prático:

```
class Veiculo:
    def __init__(self, marca):
        self.marca = marca

    def mover(self):
        print("O veículo está se movendo.")

class Carro(Veiculo):
    def __init__(self, marca, modelo):
        super().__init__(marca)
        self.modelo = modelo

    def exibir_info(self):
        print(f"Carro: {self.marca} {self.modelo}")

meu_carro = Carro("Toyota", "Corolla")
meu_carro.exibir_info()  # Saída: Carro: Toyota Corolla
```

Polimorfismo

Polimorfismo permite que métodos em diferentes classes compartilhem o mesmo nome, mas tenham comportamentos diferentes.
Continua...

Continuação

Exemplo prático:

```
1  class Animal:
2      def som(self):
3          pass
4
5  class Cachorro(Animal):
6      def som(self):
7          print("O cachorro late.")
8
9  class Gato(Animal):
10     def som(self):
11         print("O gato mia.")
12
13 animais = [Cachorro(), Gato()]
14 for animal in animais:
15     animal.som()
16 # Saída:
17 # O cachorro late.
18 # O gato mia.
```

Método __str__()

Definição:

- O método __str__() define o que será exibido quando o objeto for convertido em string ou impresso.

Exemplo prático:

```
1  class Carro:
2      def __init__(self, marca, modelo):
3          self.marca = marca
4          self.modelo = modelo
5
6      def __str__(self):
7          return f"{self.marca} {self.modelo}"
8
9  meu_carro = Carro("Toyota", "Corolla")
10 print(meu_carro)  # Saída: Toyota Corolla
```

Prática de laboratório

Tarefa 1:

- Crie uma classe Pessoa com atributos nome e idade. Crie uma instância da classe e exiba as informações.

Tarefa 2:

- Desenvolva uma classe Funcionario que herda de Pessoa e adicione um atributo salario. Crie um método que exibe o salário do funcionário.

Tarefa 3:

- Crie um sistema simples de gerenciamento de veículos, onde diferentes tipos de veículos (como carro e moto) herdam de uma classe Veiculo. Implemente polimorfismo para os métodos de movimento de cada veículo.

Resumo da semana 10

Revisão:

- **Compreendemos** os conceitos fundamentais da Programação Orientada a Objetos (POO) em Python.
- **Exploramos** classes, objetos, atributos, métodos e conceitos avançados como herança e polimorfismo.
- **Aplicamos** POO em exercícios práticos, organizando o código de maneira modular e eficiente.

Próximos Passos:

- Na próxima semana, vamos aprofundar a utilização de POO com mais exemplos práticos, incluindo o uso de herança múltipla e tratamento de exceções em POO.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre POO em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** o conceito de módulos e pacotes em Python.
- **Aprender** a criar e importar módulos para organizar e reutilizar código.
- **Explorar** pacotes e a estrutura de diretórios em projetos Python.
- **Aplicar** esses conceitos para modularizar projetos e gerenciar dependências de forma eficiente.

O que é um módulo?

Definição:

- Um módulo é um arquivo contendo código Python (funções, classes, variáveis) que pode ser importado e reutilizado em outros scripts ou programas.

Vantagens:

- Facilita a organização do código.
- Promove a reutilização e a manutenção do código.
- Reduz a complexidade dos programas ao dividir o código em partes menores e gerenciáveis.

Exemplo:

- Arquivo `meu_modulo.py` com funções que podem ser usadas em diferentes projetos.

Importando módulos

Importando módulos:
sintaxe:

```
1 import nome_do_modulo
```

Exemplo prático:

```
1 import math
2 print(math.sqrt(16))  # Saída: 4.0
```

Importando funções específicas:
sintaxe

```
1 from nome_do_modulo import nome_da_funcao
```

Exemplo:

```
1 from math import sqrt
2 print(sqrt(16))  # Saída: 4.0
```

Criando seu próprio módulo

Criando um módulo:

- Um módulo é simplesmente um arquivo Python (.py) contendo funções, classes e variáveis que podem ser importadas.

Exemplo:

- Crie um arquivo chamado meu_modulo.py com o seguinte código:

```python
def saudacao(nome):
    return f"Olá, {nome}!"

def soma(a, b):
    return a + b
```

Usando o módulo:

- **Importação:**

```python
import meu_modulo
print(meu_modulo.saudacao("Roni"))  # Saída: Olá, Roni!
print(meu_modulo.soma(5, 3))        # Saída: 8
```

Usando __name__ == "__main__"

Definição:

- __name__ é uma variável especial em Python que define o contexto em que um módulo está sendo executado.
- Se __name__ == "__main__" for True, o módulo está sendo executado diretamente e não importado.

Por que usar?:

- Permite que um módulo execute código de teste quando executado diretamente, mas oculte esse código quando importado como um módulo.

Exemplo Prático:

```
1 def saudacao(nome):
2     return f"Olá, {nome}!"
3
4 if __name__ == "__main__":
5     print(saudacao("Roni"))
```

Introdução aos pacotes

O que é um Pacote?

- Um pacote é uma coleção de módulos organizados em uma estrutura de diretórios.
- Cada diretório contendo um arquivo __init__.py é tratado como um pacote em Python.

Vantagens:

- Organiza melhor módulos relacionados.
- Facilita a manutenção e o desenvolvimento de projetos maiores.
- Permite a criação de hierarquias de módulos.

Exemplo de pacote:

- **Estrutura de diretórios:**

```
meu_pacote/
    __init__.py
    modulo1.py
    modulo2.py
```

Criando e usando pacotes

Criando um pacote:

- Crie um diretório com um arquivo __init__.py vazio ou com código de inicialização.
- Adicione módulos (arquivos .py) dentro desse diretório.

Estrutura de exemplo:

- **Diretório meu_pacote com os arquivos:**

```
meu_pacote/
    __init__.py
    calculadora.py
    saudacoes.py
```

Arquivo `calculadora.py`

```
def soma(a, b):
    return a + b
```

Continua...

Continuação

Arquivo saudacoes.py

```
1 def saudacao(nome):
2     return f"Olá, {nome}!"
```

Usando o Pacote:

- **Importação:**

```
1 from meu_pacote.calculadora import soma
2 from meu_pacote.saudacoes import saudacao
3
4 print(soma(5, 3))          # Saída: 8
5 print(saudacao("Roni"))    # Saída: Olá, Roni!
```

Importando de pacotes

Importando módulos de um pacote:
sintaxe:

```
1 import pacote.modulo
```

Exemplo:

```
1 import meu_pacote.calculadora
2 print(meu_pacote.calculadora.soma(5, 3))  # Saída: 8
```

Importando funções específicas:
sintaxe:

```
1 from pacote.modulo import funcao
```

Exemplo:

```
1 from meu_pacote.saudacoes import saudacao
2 print(saudacao("Roni"))  # Saída: Olá, Roni!
```

Gerenciando dependências e pacotes externos

Instalando Pacotes Externos com pip

- pip é o instalador de pacotes padrão para Python, usado para instalar e gerenciar pacotes Python que não são distribuídos com o Python padrão.

Instalando um Pacote:
sintaxe:

```
pip install nome_do_pacote
```

Exemplo:

```
pip install requests
```

Usando um pacote externo:

```
import requests
response = requests.get("https://api.github.com")
print(response.status_code)
```

Criando e distribuindo seus próprios pacotes

Estrutura Básica para um Pacote Distribuível:
Diretório do projeto:

```
meu_projeto/
    meu_pacote/
        __init__.py
        modulo1.py
        modulo2.py
    setup.py
```

Arquivo setup.py:

```python
from setuptools import setup, find_packages

setup(
    name="meu_pacote",
    version="0.1",
    packages=find_packages(),
    install_requires=[],
)
```

Continua...

Distribuindo o pacote:
Passos:

1. Execute `python setup.py sdist` para criar um pacote distribuível.
2. Distribua através do PyPI ou use `pip install` para instalá-lo localmente.

Prática de laboratório

Tarefa 1:

- Crie um módulo chamado operacoes.py que contenha funções básicas de matemática (adição, subtração, multiplicação, divisão). Importe e use esse módulo em um script separado.

Tarefa 2:

- Desenvolva um pacote chamado meu_calculador com dois módulos: aritmetica.py (com funções aritméticas) e geometria.py (com funções para cálculo de área e perímetro). Importe e use as funções desses módulos em um script separado.

Tarefa 3:

- Instale um pacote externo (por exemplo, requests) usando pip e use-o para fazer uma solicitação HTTP a um site de sua escolha.

Resumo da semana 12

Revisão:

- **Compreendemos** o conceito de módulos e pacotes em Python.
- **Exploramos** como criar e usar módulos e pacotes para organizar e reutilizar código.
- **Aprendemos** a instalar pacotes externos com pip e a distribuir nossos próprios pacotes.
- **Aplicamos** esses conceitos em exercícios práticos para modularizar projetos e gerenciar dependências.

Próximos passos:

- Na próxima semana, exploraremos o tratamento de exceções em Python para melhorar a robustez e a confiabilidade do código.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre módulos e pacotes em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.

Objetivos da semana

Objetivos:

- **Compreender** o conceito de exceções em Python e por que elas ocorrem.
- **Aprender** a usar blocos `try`, `except`, `else`, e `finally` para capturar e tratar exceções.
- **Explorar** a criação de exceções personalizadas para situações específicas.
- **Aplicar** o tratamento de exceções em exemplos práticos para melhorar a robustez e confiabilidade do código.

Introdução ao tratamento de exceções

O que é uma Exceção?

- Uma exceção é um evento que ocorre durante a execução de um programa e interrompe o fluxo normal do código.
- Exceções ocorrem geralmente devido a erros como divisão por zero, tentativa de acessar um índice inexistente em uma lista, ou erro de sintaxe.

Importância:

- Capturar e tratar exceções é essencial para evitar que o programa quebre inesperadamente e para lidar de forma controlada com situações de erro.

Exemplo de Exceção:

```
1 numero = int("texto")  # Gera um ValueError
```

Usando blocos try e except

Bloco try e except:

- O bloco try permite que você teste um bloco de código que pode gerar uma exceção.
- O bloco except permite que você capture e trate a exceção.

sintaxe:

```
try:
    # código que pode gerar uma exceção
except TipoDeExcecao:
    # código que será executado se ocorrer uma exceção
```

Exemplo prático:

```
try:
    numero = int(input("Digite um número: "))
    resultado = 10 / numero
    print(f"Resultado: {resultado}")
except ZeroDivisionError:
    print("Erro: Divisão por zero não é permitida.")
except ValueError:
    print("Erro: Por favor, insira um número válido.")
```

Bloco `else`:

- Executado se nenhum erro for encontrado no bloco try.

sintaxe:

```
try:
    # código que pode gerar uma exceção
except TipoDeExcecao:
    # código que será executado se ocorrer uma exceção
else:
    # código que será executado se não ocorrer uma exceção
```

Bloco `finally`:

- Sempre executado, independentemente de ocorrer uma exceção ou não. Usado para liberar recursos ou executar código de limpeza.

sintaxe:

```
try:
    # código que pode gerar uma exceção
except TipoDeExcecao:
    # código que será executado se ocorrer uma exceção
finally:
    # código que sempre será executado
```

Exemplo prático:

```
try:
    arquivo = open("dados.txt", "r")
    conteudo = arquivo.read()
except FileNotFoundError:
    print("Erro: Arquivo não encontrado.")
else:
    print(conteudo)
finally:
    arquivo.close()
    print("Arquivo fechado.")
```

Capturando múltiplas exceções

Múltiplas Exceções:

- Você pode capturar diferentes tipos de exceções no mesmo bloco try usando vários blocos except.

sintaxe:

```
try:
    # código que pode gerar exceções
except TipoDeExcecao1:
    # código para tratar TipoDeExcecao1
except TipoDeExcecao2:
    # código para tratar TipoDeExcecao2
```

Exemplo prático:

```
try:
    numero = int(input("Digite um número: "))
    resultado = 10 / numero
except (ZeroDivisionError, ValueError) as e:
    print(f"Ocorreu um erro: {e}")
```

Criando exceções personalizadas

Definição:

- Você pode criar suas próprias exceções em Python para lidar com situações específicas do seu programa.

sintaxe:

```
1 class MinhaExcecao(Exception):
2     pass
```

Continua...

Continuação

Exemplo prático:

```
1  class SaldoInsuficienteError(Exception):
2      def __init__(self, mensagem="Saldo insuficiente para
       realizar a operação."):
3          self.mensagem = mensagem
4          super().__init__(self.mensagem)
5
6  def sacar(saldo, valor):
7      if valor > saldo:
8          raise SaldoInsuficienteError()
9      saldo -= valor
10     return saldo
11
12 try:
13     saldo_atual = 100
14     saldo_atual = sacar(saldo_atual, 150)
15 except SaldoInsuficienteError as e:
16     print(e)
```

Melhores práticas para tratamento de exceções

Seja específico:

- Capture exceções específicas em vez de usar um except genérico. Isso ajuda a evitar a captura de exceções inesperadas e facilita a depuração.

Evite silenciar exceções:

- Não ignore exceções sem registrar ou lidar com elas. Isso pode mascarar problemas importantes no código.

Use finally para limpeza:

- Sempre libere recursos (como arquivos ou conexões de rede) no bloco finally para garantir que eles sejam liberados, independentemente de ocorrer uma exceção.

Documente exceções:

- Documente as exceções que uma função pode lançar, para que os usuários do código saibam como lidar com possíveis erros.

Exemplo prático de tratamento de exceções em um projeto

Projeto: Sistema de Gerenciamento de Biblioteca (SGB):

- **Descrição:** Um sistema simples que permite o empréstimo e devolução de livros, com controle de estoque.

Cenários de exceção:

- Tentativa de emprestar um livro que não está disponível.
- Devolução de um livro que não foi emprestado.
- Tentativa de acessar um livro que não existe no sistema.

Continua...

Exemplo de código:

```
class LivroIndisponivelError(Exception):
    pass

class LivroNaoEncontradoError(Exception):
    pass

class Biblioteca:
    def __init__(self):
        self.livros = {"Python Básico": 3, "POO Avançado": 2}

    def emprestar_livro(self, titulo):
        if titulo not in self.livros:
            raise LivroNaoEncontradoError(f"Livro '{titulo}' não encontrado.")
        if self.livros[titulo] == 0:
            raise LivroIndisponivelError(f"Livro '{titulo}' está indisponível.")
        self.livros[titulo] -= 1
        print(f"Livro '{titulo}' emprestado com sucesso.")

    def devolver_livro(self, titulo):
```

```
20          if titulo not in self.livros:
21              raise LivroNaoEncontradoError(f"Livro '{titulo}'
        não encontrado.")
22          self.livros[titulo] += 1
23          print(f"Livro '{titulo}' devolvido com sucesso.")
24
25  biblioteca = Biblioteca()
26  try:
27      biblioteca.emprestar_livro("Python Básico")
28      biblioteca.emprestar_livro("Python Avançado")
29  except (LivroIndisponivelError, LivroNaoEncontradoError) as
        e:
30      print(e)
```

Prática de laboratório

Tarefa 1:

- Crie um programa que leia números de um arquivo e calcule a média. Implemente tratamento de exceções para lidar com possíveis erros, como arquivo não encontrado ou dados inválidos.

Tarefa 2:

- Desenvolva um sistema simples de login que trate exceções como usuário não encontrado e senha incorreta. Crie exceções personalizadas para essas situações.

Tarefa 3:

- Implemente um sistema de carrinho de compras que capture exceções relacionadas a produtos fora de estoque e preços inválidos. Utilize blocos try, except, else, e finally para garantir o funcionamento robusto do sistema.

Resumo da semana 13

Revisão:

- Compreendemos o conceito de exceções e como elas afetam o fluxo de um programa.
- Exploramos o uso de blocos `try`, `except`, `else`, e `finally` para capturar e tratar exceções.
- Aprendemos a criar exceções personalizadas para lidar com situações específicas.
- Aplicamos o tratamento de exceções em exemplos práticos para melhorar a robustez e a confiabilidade do código.

Próximos Passos:

- Na próxima semana, concluiremos o curso com uma revisão geral dos principais conceitos abordados e discutiremos práticas recomendadas para escrever código Python de alta qualidade.

Recursos adicionais

Leitura recomendada:

- Documentação oficial do Python: docs.python.org

Exercícios online:

- Exercícios sobre tratamento de exceções em plataformas como HackerRank, LeetCode.

Comunidades:

- Stack Overflow, Reddit - r/learnpython.