Curso de Python

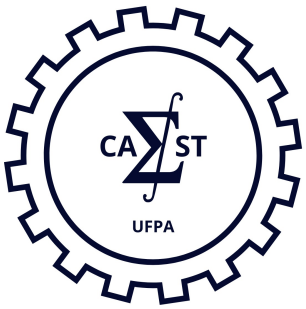
para Análise de Dados

Breno Cauã Rodrigues da Silva

Índice

1 Boas-vindas ao Universo da Análise de Dados com Python!

FAEST	CAEST	PYTHON



Seja muito bem-vindo(a) ao **Curso de Python para Análise de Dados**! Se você está aqui, provavelmente compartilha da curiosidade e do desejo de transformar dados brutos em informações valiosas e decisões estratégicas. Este curso foi meticulosamente planejado para ser sua porta de entrada nesse universo fascinante, guiando você desde os primeiros passos na programação com Python até a aplicação de técnicas essenciais de análise de dados.

Python se consolidou como uma das linguagens de programação mais influentes e requisitadas no campo da Ciência e Análise de Dados. Sua popularidade não é por acaso: a sintaxe intuitiva, a vasta coleção de bibliotecas especializadas e uma comunidade global ativa tornam o aprendizado mais acessível e o desenvolvimento mais eficiente. Aqui, você descobrirá como aproveitar todo esse potencial.

1.1 Sobre o Autor

i Conheça o Instrutor

Meu nome é Breno C R Silva, sou Bacharelando em Estatística pela Universidade Federal do Pará com foco em ***Estatística Descritiva, Probabilidade, Estatística Inferêncial, Modelos de Regressão e Classificação e, em especial, Análise de Séries Temporais e Análise de Sobrevivência****.*
Trabalho com Python e Análise de Dados há 3 anos, aplicando essas ferramentas em ***Previsão do ICMS, Modelagem Semiparamétrica em Dados Sujeitos a Censura Intervalar, Previsão de Concessão de Crédito*** *entre outros projetos.*
Criei este curso para compartilhar meu conhecimento e ajudar mais pessoas a descobrirem o poder dos dados. Espero que aproveitem a jornada!

1.2 Por Que Aprender Python para Análise de Dados?

Vivemos na era dos dados. Empresas, instituições de pesquisa e organizações de todos os tipos coletam volumes massivos de informações diariamente. A capacidade de analisar esses dados para extrair *insights*, identificar padrões, prever tendências e comunicar descobertas tornou-se uma habilidade crucial e altamente valorizada no mercado de trabalho.

Python, com seu ecossistema robusto (especialmente bibliotecas como Pandas, NumPy, Matplotlib e Seaborn), oferece as ferramentas perfeitas para realizar todo o ciclo de vida da análise de dados:

1. **Coleta e Limpeza:** Obter dados de diversas fontes e prepará-los para análise, tratando inconsistências e valores ausentes.
2. **Manipulação e Transformação:** Estruturar, filtrar, agregar e remodelar os dados para atender às necessidades da análise.
3. **Análise Exploratória (EDA):** Investigar os dados para entender suas características principais, descobrir relações e formular hipóteses.
4. **Visualização:** Criar gráficos e representações visuais claras e impactantes para comunicar os resultados.
5. **Comunicação:** Gerar relatórios e apresentações que traduzam as descobertas técnicas em informações acionáveis.

1.3 Objetivos de Aprendizagem do Curso

Ao concluir este curso, você estará apto(a) a:

- **Dominar os Fundamentos de Python:** Escrever código Python claro e eficiente, utilizando variáveis, tipos de dados, operadores, estruturas de controle (condicionais e loops) e funções.
- **Manipular Dados com Pandas:** Utilizar DataFrames e Series para carregar, limpar, transformar, filtrar, agregar e combinar conjuntos de dados de forma eficaz.
- **Realizar Cálculos Numéricos com NumPy:** Empregar arrays NumPy para operações matemáticas e estatísticas vetorizadas de alta performance.
- **Criar Visualizações com Matplotlib e Seaborn:** Gerar diversos tipos de gráficos (linhas, barras, dispersão, histogramas, boxplots, etc.) para explorar dados e comunicar resultados visualmente.
- **Aplicar o Processo de Análise de Dados:** Integrar as ferramentas aprendidas para realizar análises exploratórias básicas em conjuntos de dados reais ou simulados.
- **Desenvolver Raciocínio Analítico:** Interpretar os resultados das análises e das visualizações para extrair conclusões relevantes.

1.4 A Quem Se Destina Este Curso?

Este material foi pensado com carinho para um público diversificado, incluindo:

- **Estudantes Universitários:** De cursos como Estatística, Ciência da Computação, Engenharias, Economia, Administração e áreas afins que desejam uma introdução prática à análise de dados com Python.
- **Profissionais em Transição de Carreira:** Pessoas que buscam adquirir habilidades em análise de dados para novas oportunidades no mercado.
- **Curiosos e Entusiastas:** Qualquer pessoa com interesse em aprender a programar e a trabalhar com dados. Acreditamos que a vontade de aprender é o principal pré-requisito!
- **Iniciantes em Programação:** Se você nunca programou antes, não se preocupe! Começaremos do básico.

1.5 Estrutura Detalhada do Curso

Nossa jornada será dividida em módulos progressivos, construindo seu conhecimento passo a passo:

1. **Módulo 1: Fundamentos Essenciais de Python:**
    - Introdução à linguagem e configuração do ambiente (Anaconda, Jupyter Notebook).
    - Sintaxe básica, variáveis, tipos de dados (números, strings, booleanos).
    - Operadores (aritméticos, comparação, lógicos).
    - Estruturas de dados nativas: Listas, Tuplas, Dicionários e Conjuntos (Sets).
    - Estruturas de controle: Condicionais (`if`, `elif`, `else`) e Laços de repetição (`for`, `while`).
    - Definição e uso de Funções para modularizar o código.
2. **Módulo 2: Computação Numérica com NumPy:**
    - Introdução aos arrays NumPy (ndarrays) e suas vantagens.
    - Criação, indexação e fatiamento de arrays uni e multidimensionais.
    - Operações matemáticas vetorizadas e broadcasting.
    - Funções estatísticas e manipulação de arrays.
3. **Módulo 3: Manipulação e Análise de Dados com Pandas:**
    - As estruturas chave: Series e DataFrames.
    - Leitura e escrita de dados (CSV, Excel, etc.).
    - Seleção e filtragem de dados (`loc`, `iloc`, boolean indexing).
    - Limpeza de dados: Tratamento de valores ausentes e duplicados.
    - Transformação de dados: Aplicação de funções, criação de novas colunas.
    - Agrupamento e agregação com `groupby()`.
    - Combinação de DataFrames (`merge`, `concat`, `join`).
4. **Módulo 4: Visualização de Dados com Matplotlib e Seaborn:**
    - Princípios da visualização de dados.
    - Criação de gráficos básicos e customização com Matplotlib.
    - Geração de gráficos estatísticos avançados e esteticamente agradáveis com Seaborn.
    - Integração com Pandas para visualização direta de DataFrames.
5. **Módulo 5: Projeto de Análise Exploratória de Dados (EDA):**
    - Aplicação integrada dos conhecimentos adquiridos em um projeto prático.
    - Passo a passo: Definição do problema, coleta/carregamento dos dados, limpeza, análise exploratória, visualização e comunicação dos resultados.

1.6 Nossa Metodologia de Ensino

Buscamos um equilíbrio entre teoria e prática, utilizando uma abordagem ativa:

- **Conteúdo Conceitual Claro:** Explicações diretas e objetivas dos conceitos fundamentais, utilizando analogias e exemplos simples.
- **Código Comentado e Exemplos Práticos:** Demonstrações passo a passo de como aplicar cada conceito e ferramenta em Python, com código funcional e comentado.
- **Foco na Resolução de Problemas:** Apresentação de cenários e problemas típicos da análise de dados para contextualizar o aprendizado.
- **Exercícios Progressivos:** Desafios práticos ao final das seções para você testar e consolidar seu entendimento.

- **Projetos Integradores:** Aplicação do conhecimento em projetos que simulam situações reais de análise.

1.7 Pré-requisitos e Ferramentas

1.7.1 Pré-requisitos

- **Familiaridade básica com o uso de computadores:** Saber navegar em pastas, usar um navegador de internet e instalar programas simples.
- **Curiosidade e Vontade de Aprender:** A motivação é seu maior trunfo!
- **Nenhuma experiência prévia em programação é exigida.** O curso foi desenhado para iniciantes.

1.7.2 Ferramentas Recomendadas

Para acompanhar o curso, você precisará de um ambiente Python configurado. Recomendamos fortemente a instalação da **Distribuição Anaconda**, que já inclui Python, o Jupyter Notebook e as principais bibliotecas que usaremos.

- **Anaconda:** Baixe em anaconda.com/download. Siga as instruções de instalação para seu sistema operacional.
- **Jupyter Notebook/JupyterLab:** Ambiente interativo ideal para aprender e experimentar com código e visualizações. Vem com o Anaconda.
- **Alternativas Online (sem instalação):**
  - Google Colaboratory: Ambiente Jupyter Notebook gratuito na nuvem, oferecido pelo Google.
  - JupyterLite (via Jupyter.org): Executa um ambiente JupyterLab diretamente no seu navegador.
  - Programiz Online Compiler: Útil para testar pequenos trechos de código Python rapidamente.
- **Editor de Código (Opcional Avançado):** Se preferir, pode usar editores como VSCode com a extensão Python, mas o Jupyter é mais indicado para o formato do curso.

> Dica de Instalação
>
> Recomendamos fortemente o uso do **Anaconda**, pois ele simplifica a gestão das bibliotecas (pacotes) que usaremos ao longo do curso. Se encontrar dificuldades na instalação, procure tutoriais específicos para seu sistema operacional ou utilize as alternativas online.

1.8 Recursos Adicionais e Comunidade

O aprendizado não termina aqui! Explore estes recursos para aprofundar seus conhecimentos:

- **Documentação Oficial:**
  - Python
  - NumPy
  - Pandas
  - Matplotlib
  - Seaborn
- **Comunidades Online:** Stack Overflow, Reddit (r/learnpython, r/datascience), fóruns específicos.

- **Plataformas de Aprendizado:** Kaggle Learn, DataCamp, Coursera, edX.
- **Sites de Apoio (mencionados anteriormente):**
  - Python Academy
  - Python Examples
  - Hashtag Treinamentos (YouTube)

Estamos muito animados para começar esta jornada de aprendizado com você! Prepare-se para mergulhar no mundo da programação Python e descobrir como os dados podem contar histórias incríveis. Vamos lá!

2 Primeiros Passos

2.1 Introdução ao Python

Neste primeiro capítulo, daremos os passos iniciais no mundo da programação com Python. Vamos entender o que é essa linguagem, por que ela se tornou tão popular (especialmente para análise de dados) e como preparar nosso ambiente para começar a codificar.

2.1.1 O Que Torna Python Especial?

Python é frequentemente descrita como uma linguagem de programação **poderosa**, **versátil** e, acima de tudo, **legível**. Mas o que isso significa na prática?

1. **Linguagem Interpretada:** Diferente de linguagens compiladas (como C++ ou Java), onde o código fonte é traduzido para código de máquina antes da execução, o código Python é executado linha por linha por um programa chamado **interpretador**. Isso facilita o desenvolvimento e o teste, pois você pode executar pequenos trechos de código rapidamente.
2. **Alto Nível:** Python abstrai muitos detalhes complexos do hardware do computador (como gerenciamento de memória). Isso permite que você se concentre na lógica do problema que está tentando resolver, em vez de se preocupar com detalhes de baixo nível.
3. **Tipagem Dinâmica:** Você não precisa declarar explicitamente o tipo de uma variável (inteiro, texto, etc.) antes de usá-la. O Python infere o tipo automaticamente durante a execução. Isso torna o código mais conciso, mas exige atenção para evitar erros relacionados a tipos inesperados.
4. **Propósito Geral:** Python não se limita a uma única área. É usada em desenvolvimento web, automação de tarefas, inteligência artificial, computação científica e, claro, **análise e ciência de dados**.
5. **Sintaxe Clara e Legível:** A sintaxe do Python foi projetada para ser próxima da linguagem humana, utilizando indentação (espaços no início da linha) para definir blocos de código, o que força a escrita de um código visualmente organizado.

Essas características, combinadas com um vasto ecossistema de bibliotecas (conjuntos de código pré-escrito para tarefas específicas), fazem do Python uma ferramenta excepcional para análise de dados, adotada por gigantes como Google, NASA, Facebook (Meta), Amazon e Spotify.

2.1.2 Configurando o Ambiente: Anaconda e Jupyter Notebook

Para começar nossa jornada, precisamos instalar o Python e as ferramentas necessárias. A maneira mais recomendada para iniciantes em análise de dados é usar a **Distribuição Anaconda**.

O que é Anaconda? É um pacote que inclui:

- O interpretador Python.
- Um gerenciador de pacotes (bibliotecas) chamado `conda`.
- Diversas bibliotecas científicas e de análise de dados pré-instaladas (como NumPy, Pandas, Matplotlib).
- Ferramentas úteis, como o **Jupyter Notebook** e o **JupyterLab**.

O que é Jupyter Notebook? É uma aplicação web interativa que permite criar e compartilhar documentos (chamados *notebooks*) que contêm código executável (como Python), texto formatado (Markdown), equações, visualizações e muito mais. É um ambiente ideal para aprendizado, experimentação e apresentação de análises de dados.

Passos para Instalação:

1. **Download:** Acesse o site oficial do Anaconda: https://www.anaconda.com/download
2. **Escolha seu Sistema Operacional:** Baixe o instalador apropriado para Windows, macOS ou Linux.
3. **Instalação:** Execute o instalador e siga as instruções. Geralmente, as opções padrão são adequadas para iniciantes.
4. **Vídeo de Apoio:** Se precisar de ajuda visual, este tutorial de instalação no YouTube pode ser útil.

> Dúvidas na Instalação?
>
> Se encontrar problemas, não hesite em procurar tutoriais mais específicos para sua versão do sistema operacional ou entrar em contato:
>
> - **Email do Instrutor:** breno.silva@icen.ufpa.br

Após a instalação, você poderá iniciar o Jupyter Notebook (geralmente através do Anaconda Navigator ou pelo terminal/prompt de comando digitando `jupyter notebook`).

2.2 Sintaxe Básica: A Gramática do Python

Toda linguagem tem suas regras. Em Python, a sintaxe define como escrevemos comandos válidos que o interpretador possa entender. Vamos começar com o básico.

2.2.1 Exibindo Informações: A Função `print()`

A primeira função que a maioria dos programadores aprende é a `print()`. Sua finalidade é exibir informações (texto, números, resultados de cálculos) na tela (console ou saída do notebook).

O famoso "Hello, World!":

```
print("Hello, World!")
```

Como funciona?

- `print` é o nome da função.
- Os parênteses `()` são usados para chamar a função e passar informações para ela.
- O que está dentro dos parênteses é chamado de **argumento**. Neste caso, o argumento é o texto (string) `"Hello, World!"`.
- Strings em Python são definidas usando aspas simples (`'...'`) ou duplas (`"..."`).

2.2.1.1 Erros Comuns com `print()`

É normal cometer erros ao aprender. Vejamos alguns deslizes comuns com `print()`:

1. **Nome Incorreto (Case-Sensitive):** Python diferencia maiúsculas de minúsculas.

```
# Exemplo de Código Incorreto
Print("Olá")
```

Erro Gerado: `NameError: name 'Print' is not defined` (O Python não reconhece 'Print' com 'P' maiúsculo).

2. **Faltando Aspas:** Textos (strings) precisam estar entre aspas.

```
# Exemplo de Código Incorreto
print(Olá)
```

Erro Gerado: `SyntaxError: invalid syntax` (O Python não entende 'Olá' como um comando ou variável válida sem aspas).

3. **Aspas Incompletas:** Abrir aspas e não fechar (ou vice-versa).

```
# Exemplo de Código Incorreto
print("Olá)
```

Erro Gerado: `SyntaxError: unterminated string literal` (A string não foi finalizada corretamente).

4. **Misturar Tipos de Aspas:** Começar com simples e terminar com duplas (ou vice-versa).

```
# Exemplo de Código Incorreto
print('Olá")
```

Erro Gerado: `SyntaxError: unterminated string literal`.

> **!** Atenção à Sintaxe!
>
> Erros de sintaxe são como erros de gramática. O interpretador Python precisa que as regras sejam seguidas para entender suas instruções. Prestar atenção aos detalhes (maiúsculas/minúsculas, parênteses, aspas) é fundamental.

Usando Aspas Dentro de Strings:
Se precisar incluir aspas no seu texto, alterne os tipos de aspas:

```
print('Ele disse: "Python é incrível!"')
print("O livro se chama 'O Guia do Mochileiro das Galáxias'.")
```

Quebras de Linha:
Para inserir uma quebra de linha dentro de uma string, use o caractere especial `\n` (barra invertida seguida de 'n'):

```
print("Linha 1\nLinha 2\nLinha 3")
```

2.2.2 Operadores Matemáticos: Calculando com Python

Python pode ser usado como uma calculadora poderosa. Ele suporta os operadores matemáticos básicos e alguns mais avançados. Observe a Tabela 2.1.

Tabela 2.1: Operadores Matemáticos.

Operador	Descrição	Exemplo	Resultado
`+`	Adição	`5 + 3`	`8`
`-`	Subtração	`5 - 3`	`2`
`*`	Multiplicação	`5 * 3`	`15`
`/`	Divisão (`float`)	`10 / 3`	`3.3333`
`//`	Divisão (`int`)	`10 // 3`	`3`
`%`	Módulo (Resto)	`10 % 3`	`1`
`**`	Exponenciação	`2 ** 3`	`8`

Exemplos:

```
# Adição e Subtração
print(10 + 5)
```

```
15
```

```
print(10 - 5.5)
```

```
4.5
```

```
# Multiplicação e Divisão
print(4 * 7)
```

```
28
```

```
print(15 / 4)   # Divisão resulta em float
```

```
3.75
```

```
# Divisão Inteira (descarta a parte decimal)
print(15 // 4)
```

```
3
```

```
# Módulo (resto da divisão inteira)
print(15 % 4)   # 15 dividido por 4 é 3, com resto 3
```

```
3
```

```
# Exponenciação (potência)
print(3 ** 4)   # 3 elevado à 4ª potência
```

```
81
```

Assim como na matemática, tentar dividir por zero em Python gera um erro:

```
10 / 0
```

Erro Gerado: `ZeroDivisionError: division by zero`
Calculando Raiz Quadrada:
Podemos usar exponenciação com expoente fracionário:

```
print(81 ** 0.5) # Raiz quadrada de 81
```

```
9.0
```

No entanto, a forma mais comum e recomendada é usar a função `sqrt()` da biblioteca (módulo) `math`:

```
import math # Importa a biblioteca math

print(math.sqrt(81))
```

```
9.0
```

> **Importando Módulos**
>
> O comando `import math` torna todas as funções e constantes definidas no módulo `math` disponíveis para uso no seu código. Veremos mais sobre módulos e bibliotecas posteriormente.

2.2.3 Expressões Numéricas e Precedência

Podemos combinar múltiplos operadores em uma única expressão:

```
3 + 4 * 2 - 5 / 2 ** 2
```

```
9.75
```

Qual a ordem de execução? Python segue a ordem de precedência padrão da matemática, conhecida como **PEMDAS**:

1. **P**arênteses `()` - Operações dentro de parênteses são executadas primeiro.
2. **E**xponenciação `**`
3. **M**ultiplicação `*`, **D**ivisão `/`, Divisão Inteira `//`, Módulo `%` (executados da esquerda para a direita se tiverem a mesma precedência).
4. **A**dição `+`, **S**ubtração `-` (executados da esquerda para a direita se tiverem a mesma precedência).

No exemplo `3 + 4 * 2 - 5 / 2 ** 2`:

1. `2 ** 2` é `4`

2. `4 * 2` é `8`
3. `5 / 4` é `1.25`
4. `3 + 8` é `11`
5. `11 - 1.25` é `9.75`

Use parênteses para controlar a ordem quando necessário:

```
(3 + 4) * (2 - 5) / (2 ** 2)
```

```
-5.25
```

```
# (7) * (-3) / (4) = -21 / 4 = -5.25
```

2.2.3.1 Uma Nota Sobre Números Decimais (Ponto Flutuante)

Você pode notar resultados ligeiramente inesperados ao trabalhar com números decimais (chamados *floats* em Python):

```
print(0.1 + 0.2)
```

```
0.30000000000000004
```

Por que `0.30000000000000004` e não `0.3`? A maioria dos computadores utiliza a representação binária (base 2) para armazenar números. O problema é que muitas frações decimais (base 10), como `0.1`, não podem ser representadas exatamente em binário. Isso ocorre porque a representação binária tem um número finito de dígitos, assim como ocorre na representação decimal. Por exemplo, ao representar 1/3 em decimal, obtém-se uma repetição infinita: 0.3333 ... Da mesma forma, a conversão de `0.1` para binário gera uma fração periódica infinita.

Os computadores modernos seguem o padrão IEEE 754 para representar números de ponto flutuante. Esse padrão define como os números são armazenados na memória, incluindo a precisão. Em Python, os números de ponto flutuante geralmente utilizam precisão dupla do IEEE 754, que oferece 53 bits de precisão. Quando um número como 0.1 é digitado, o computador encontra a fração binária mais próxima possível dentro desse limite de precisão. O resultado é um valor muito próximo de 0.1, mas não exatamente igual.

> Lidando com Imprecisões de Float
>
> Para a maioria das aplicações em análise de dados, essa pequena imprecisão não é um problema. Ao exibir resultados, você pode arredondar os números usando a função `round()`:
>
> ```
> print(round(0.1 + 0.2, 2)) # Arredonda para 2 casas decimais
> ```
>
> ```
> 0.3
> ```
>
> Para cálculos financeiros ou científicos que exigem alta precisão, Python oferece módulos como `Decimal`.

2.2.4 Comparações e Operadores Lógicos

Frequentemente, precisamos comparar valores em nossos programas. Python oferece operadores de comparação que retornam um valor **booleano**: `True` (Verdadeiro) ou `False` (Falso).

Tabela 2.2: Operadores de Comparação.

Operador	Descrição	Exemplo	Resultado
`<`	Menor que	`5 < 10`	`True`
`>`	Maior que	`5 > 10`	`False`
`<=`	Menor ou igual a	`10 <= 10`	`True`
`>=`	Maior ou igual a	`10 >= 10`	`True`
`==`	Igual a	`5 == 5`	`True`
`!=`	Diferente de	`5 != 10`	`True`

Exemplos:

```
# Idade Mínima igual a 18
# Idade do Usuário igual a 25

print("Usuário é maior de idade?", 25 >= 18)
```

```
Usuário é maior de idade? True
```

```
print("As idades são iguais?", 25 >= 18)
```

```
As idades são iguais? True
```

```
print("As idades são diferentes?", 25 >= 18)
```

```
As idades são diferentes? True
```

Operadores Lógicos:
Podemos combinar múltiplas comparações usando operadores lógicos:

Tabela 2.3: Operadores Lógicos.

Operador	Resultado é `True` se	Exemplo	Resultado
`and`	**Ambas** as condições forem `True`	`(5 < 10) and (10 > 3)`	`True`
`or`	**Pelo menos uma** das condições for `True`	`(5 > 10) or (10 == 10)`	`True`
`not`	A condição seguinte for `False` (inverte o valor lógico)	`not (5 == 10)`	`True`

Exemplos:

```
# Tem Ingresso igual a True
# É maior de idade igual a False

# Pode entrar na festa? (Precisa ter ingresso E ser maior de idade)
print("Pode entrar na festa?", True and False)
```

```
Pode entrar na festa? False
```

```
# Pode receber desconto? (Precisa ter ingresso OU ser maior de idade - exemplo hipotético)
print("Pode receber desconto?", True or False)
```

```
Pode receber desconto? True
```

```
# Não tem ingresso?
print("Não tem ingresso?", not True)
```

```
Não tem ingresso? False
```

```
print((1 and 4) < 3)
```

```
False
```

```
print((1 or 4) < 3)
```

```
True
```

```
print((1 and 2 and 2.99) < 3)
```

```
True
```

```
print((1 or 2 or 2.99) > 3)
```

```
False
```

```
print((5 >= 4.99) and (10 <= 10.01))
```

```
True
```

```
print((5 >= 4.99) and (10 == 10.01))
```

```
False
```

```
print((5 >= 4.99) or (10 <= 10.01))
```

```
True
```

```
print((5 >= 4.99) or (10 == 10.01))
```

```
True
```

```
print(1 == 1)
```

```
True
```

```
print(not 1 == 1)
```

```
False
```

```
print(not not 1 == 1)
```

```
True
```

```
print(not not not 1 == 1)
```

```
False
```

Precedência dos Operadores Lógicos:
Assim como os operadores matemáticos, os lógicos também têm uma ordem de avaliação:

1. `not` é avaliado primeiro.
2. `and` é avaliado em seguida.
3. `or` é avaliado por último.

Exemplo:

```
print(not False and True or False)
```

```
True
```

```
# 1. not False -> True
# 2. True and True -> True
# 3. True or False -> True
```

Use parênteses `()` para garantir a ordem desejada quando a expressão for complexa.

2.2.5 Variáveis: Armazenando Informações

Uma **variável** funciona como um rótulo ou um nome que damos a um local na memória do computador onde um valor (um objeto) está armazenado. Isso nos permite referenciar e reutilizar valores facilmente.

O processo de criar uma variável e associar um valor a ela é chamado de **atribuição**, e usamos o sinal de igual (`=`) para isso.

```
# Atribuição
quantidade_alunos = 19
preco_produto = 34.99
mensagem_boas_vindas = "Olá, estudante!"
curso_ativo = True

# Usando as variáveis
print(quantidade_alunos)
```

```
19
```

```
print(mensagem_boas_vindas)
```

```
Olá, estudante!
```

2.2.5.1 Regras e Convenções para Nomes de Variáveis

Escolher nomes significativos torna o código muito mais fácil de entender.

- **Regras (Obrigatórias):**
  - Nomes devem começar com uma letra (a-z, A-Z) ou underscore (_).
  - O restante do nome pode conter letras, números (0-9) e underscores.
  - Nomes são *case-sensitive* (`idade` é diferente de `Idade`).
  - Não podem ser iguais a **palavras-chave** reservadas do Python (como `if`, `else`, `for`, `while`, `def`, `class`, `import`, `True`, `False`, `None`, etc.).
- **Convenções (Boas Práticas - PEP 8):**
  - Use nomes em minúsculas.
  - Separe palavras com underscores (estilo `snake_case`). Ex: `taxa_juros`, `nome_cliente`.
  - Escolha nomes descritivos que indiquem o propósito da variável.
- **Exemplos Válidos:** `idade`, `nome_completo`, `total_vendas`, `_variavel_privada` (convenção)
- **Exemplos Inválidos:**

```
# Errado: começa com número
1_lugar = "Ouro"

# Errado: contém caractere especial (@)
email@cliente = "teste@exemplo.com"

# Errado: usa palavra-chave (def)
def = "definição"
```

Tentar usar um nome inválido ou uma palavra-chave resultará em `SyntaxError`. Tentar usar uma variável que não foi definida (atribuída) ainda resultará em `NameError`.

2.2.5.2 Atribuição com Expressões e Atualização

Podemos atribuir o resultado de uma expressão a uma variável:

```
preco_unitario = 50
quantidade = 5
desconto = 0.10

valor_bruto = preco_unitario * quantidade
valor_desconto = valor_bruto * desconto
valor_final = valor_bruto - valor_desconto

print("Valor final:", valor_final)
```

```
Valor final: 225.0
```

Para atualizar o valor de uma variável existente, podemos usar a própria variável na expressão à direita do =:

```
contador = 0
print("Contador inicial:", contador)
```

```
Contador inicial: 0
```

```
contador = contador + 1 # Incrementa o valor
print("Contador após incremento:", contador)
```

```
Contador após incremento: 1
```

Python oferece **operadores de atribuição compostos** como atalhos:

- `+=` (Adição): `x += 1` é o mesmo que `x = x + 1`
- `-=` (Subtração): `x -= 5` é o mesmo que `x = x - 5`
- `*=` (Multiplicação): `x *= 2` é o mesmo que `x = x * 2`
- `/=` (Divisão): `x /= 4` é o mesmo que `x = x / 4`
- `//=` , `%=` , `**=` (análogos para divisão inteira, módulo e exponenciação)

```
num = 10
num += 3  # num agora é 13
print(num)
```

```
13
```

```
num *= 2  # num agora é 26
print(num)
```

```
26
```

2.2.5.3 Atribuição Múltipla

Python permite atribuir valores a múltiplas variáveis na mesma linha:

```
x, y, z = 10, 20, "teste"
print(x)
```

```
10
```

```
print(y)
```

```
20
```

```
print(z)
```

```
teste
```

Não parece algo tão interessante, não é? Vamos a um exemplo. Imagine um problema que envolve duas variáveis `a` e `b`. O Python permite a **atribuição múltipla**, o que pode ser útil para trocar valores entre variáveis de forma eficiente.

```
a, b = 1, 200
print(a, b)  # Saída: 1 200
```

```
1 200
```

Agora, pense como poderiamos trocar os valores dessas variáveis. Pensou? Em algum momento deve ter passado pela sua cabeça a seguinte lógica:

```
a = b # Perde-se o valor original de a (1)
print(a)
```

```
200
```

```
b = a # Como perdeu-se a, `b vai continuar com seu valor original (200)
print(b)
```

```
200
```

Em outras linguagens, para trocar valores entre duas variáveis, seria necessário usar uma variável auxiliar:

```
a, b = 1, 200
print(a, b)  # Saída: 1 200
```

```
1 200
```

```
aux = a
a = b
b = aux
print(a, b)  # Agora a = 200 e b = 1
```

```
200 1
```

No entanto, em Python, a troca pode ser feita de forma mais elegante usando atribuição múltipla:

```
a, b = 1, 200
print(a, b)  # Saída: 1 200
```

```
1 200
```

```
a, b = b, a
print(a, b)  # Agora a = 200 e b = 1
```

```
200 1
```

Note que essa abordagem pode ser expandida para múltiplas variáveis.

2.2.6 Tipos de Objetos Fundamentais

Já vimos alguns tipos de dados (valores) que podemos armazenar em variáveis. Cada valor em Python é um **objeto**, e cada objeto pertence a um **tipo** (ou **classe**). O tipo define quais operações podem ser realizadas com aquele objeto.

Podemos verificar o tipo de um objeto usando a função `type()`:

```
numero_inteiro = 100
numero_decimal = 3.14159
texto = "Análise de Dados"
logico = False

print(type(numero_inteiro))
```

```
<class 'int'>
```

```
print(type(numero_decimal))
```

```
<class 'float'>
```

```
print(type(texto))
```

```
<class 'str'>
```

```
print(type(logico))
```

```
<class 'bool'>
```

Principais Tipos Primitivos:

- **`int` (Inteiro):** Números inteiros, positivos ou negativos, sem parte decimal (ex: `-10`, `0`, `42`).
- **`float` (Ponto Flutuante):** Números reais, que possuem uma parte decimal (ex: `-3.14`, `0.0`, `99.99`).
- **`str` (String):** Sequências de caracteres (texto), delimitadas por aspas simples ou duplas (ex: `'Python'`, `"Olá, mundo!"`).
- **`bool` (Booleano):** Representa valores lógicos de Verdadeiro (`True`) ou Falso (`False`).
- **`NoneType` (None):** Um tipo especial que tem apenas um valor: `None`. Usado para representar a ausência de valor.

Além desses, Python possui tipos de dados mais complexos para agrupar informações, que veremos em detalhes mais adiante:

- **`list` (Lista):** Coleção ordenada e *mutável* de itens.
- **`tuple` (Tupla):** Coleção ordenada e *imutável* de itens.
- **`dict` (Dicionário):** Coleção não ordenada de pares chave-valor.
- **`set` (Conjunto):** Coleção não ordenada de itens únicos.

2.2.7 Métodos e Atributos

Em Python, os conceitos de métodos e atributos são fundamentais na programação orientada a objetos (POO). Eles definem as características e comportamentos dos objetos.

2.2.7.1 Métodos

- **Definição:**
  - Métodos são funções definidas dentro de um objeto. Eles definem os comportamentos ou ações que um objeto pode realizar.
  - Pense neles como as "ações" que um objeto pode executar.
- **Exemplo:**
  - Em um objeto `carro`, métodos poderiam ser `ligar()`, `acelerar()` e `frear()`.
- **Acesso:**
  - Chamamos métodos usando a sintaxe `objeto.método()`.

2.2.7.2 Atributos

- **Definição:**
  - Atributos são variáveis que armazenam dados dentro de um objeto. Eles representam as características ou propriedades de um objeto.
  - Pense neles como as "informações" (sobre o objeto por isso características) que um objeto carrega consigo.
- **Exemplo:**
  - Em um objeto `carro`, atributos poderiam ser `cor`, `marca`, `modelo` e `ano`.
- **Acesso:**
  - Acessamos atributos usando a sintaxe `objeto.atributo`.

2.2.7.3 Relação entre Métodos e Atributos

Métodos frequentemente manipulam os atributos de um objeto. Por exemplo, um método `acelerar()` pode modificar o atributo `velocidade` de um objeto `carro`.

Em resumo:

- Atributos são as características que um objeto possui.
- Métodos são as ações que um objeto pode realizar.

Essa distinção permite criar objetos que representam entidades do mundo real com suas próprias características e comportamentos.

2.2.8 Obtendo Ajuda e Explorando

Python oferece ferramentas para ajudar você a aprender e explorar:

- **`help()`:** Fornece documentação sobre funções, módulos ou tipos.

```
# help(print) # Descomente para ver a ajuda da função print
# help(str)   # Descomente para ver a ajuda sobre o tipo string
# help(math)  # Descomente para ver a ajuda sobre o módulo math (precisa importar antes)
```

- **`dir()`:** Lista os nomes (atributos e métodos) definidos por um objeto ou módulo.

```
# print(dir(str)) # Lista métodos e atributos de strings
# print(dir(math)) # Lista funções e constantes do módulo math
```

- **`type()`:** Como já vimos, retorna o tipo de um objeto.

No ambiente Jupyter Notebook, você também pode usar:

- `?` após um nome de função/objeto para ver sua documentação (ex: `print?` ou `frase.upper?`).
- `??` após um nome de função/objeto para tentar ver o código fonte (se disponível).
- Completar com `Tab`: Digite o início de um nome de variável ou método e pressione `Tab` para ver sugestões.

Este capítulo cobriu os fundamentos essenciais para começar a programar em Python. Nos próximos capítulos, construiremos sobre essa base para explorar estruturas de dados mais complexas e as ferramentas específicas para análise de dados.

3 Objetos no Python

3.1 Introdução

Ao longo do curso, já mencionamos alguns tipos de objetos enquanto explorávamos conceitos básicos de Python. Você provavelmente já viu variáveis sendo criadas e usadas, e até notou que diferentes valores podem ter diferentes tipos. Agora, chegou o momento de entender melhor os objetos básicos do Python e como eles funcionam.

Neste capítulo, vamos explorar os principais tipos de objetos que o Python nos oferece, como números, strings, listas e dicionários. Além disso, veremos como identificar o tipo de um objeto, buscar ajuda sobre funções e entender melhor as variáveis disponíveis no código.

Com essa nova organização do curso, o conteúdo foi dividido em capítulos menores para facilitar o aprendizado. Isso evita que tudo seja ensinado de uma vez, tornando a experiência mais dinâmica e leve. Então, vamos começar nossa jornada pelos objetos do Python!

3.2 Strings

Também chamada de sequência de caracteres, textos ou dados alfanuméricos, uma *string* é um tipo de dado que armazena uma *sequência de caracteres*. Em Python, pode ser definida com aspas simples (`'`), duplas (`"`) ou triplas (`'''` ou `"""`).

```
"Texto com acentos e cedilhas: hoje é dia de caça!"
```

```
'Texto com acentos e cedilhas: hoje é dia de caça!'
```

```
# As strings aceitam aspas simples também
nome = 'Silvio Santos'
nome
```

```
'Silvio Santos'
```

3.2.1 Operações com Strings

Podemos realizar diversas operações matemáticas e manipulações em strings.

```
# Multiplicação repete a string
nome * 3
```

```
'Silvio SantosSilvio SantosSilvio Santos'
```

```
[Input]:  nome * 3.14
[Output]: TypeError: can't multiply sequence by non-int of type 'float'
```

```
# Concatenação de strings
canto1 = 'vem aí, '
canto2 = 'lá '
nome + ' ' + canto1 + canto2 * 6 + '!!'
```

```
'Silvio Santos vem aí, lá lá lá lá lá lá !!'
```

3.2.2 Strings Multilinhas

Para definir strings que ocupam múltiplas linhas, utilize três aspas (''' ou """):

```
str_grande = '''Aqui consigo inserir um textão com várias linhas.
Posso iniciar em uma...
... continuar em outra...
... e seguir quantas precisar.'''

str_grande
```

```
'Aqui consigo inserir um textão com várias linhas.\nPosso iniciar em uma...\n... continuar em out
```

```
print(str_grande)
```

```
Aqui consigo inserir um textão com várias linhas.
Posso iniciar em uma...
... continuar em outra...
... e seguir quantas precisar.
```

Caso seja necessário incluir aspas dentro da string, podemos alternar entre aspas simples e duplas:

```
agua = "Me dá um copo d'água"
agua
```

```
"Me dá um copo d'água"
```

Também podemos usar todas as aspas ao mesmo tempo:

```
todas_as_aspas = """Essa é uma string que tem:
- aspas 'simples'
- aspas "duplas"
- aspas '''triplas'''
Legal, né?"""

print(todas_as_aspas)
```

```
Essa é uma string que tem:
- aspas 'simples'
- aspas "duplas"
- aspas '''triplas'''
Legal, né?
```

3.2.3 Tamanho de uma String

A função embutida `len()` nos permite obter o número de caracteres de uma string, incluindo espaços e pontuação:

```
len('Abracadabra')
```

```
11
```

```
frase = 'Faz um pull request lá'
len(frase)
```

```
22
```

```
palavra = "Python"
len(palavra)
```

```
6
```

3.2.4 Manipulação de Strings

3.2.4.1 Indexação

Cada caractere em uma string possui um índice, começando em `0` para o primeiro elemento e indo até `len(string) - 1` para o último elemento ou `-1`.

> *Índices negativos percorrem de trás para frente*

Para um melhor entendimento inicial, considere a variável criada na subseção anterir:`palavra`. Partindo da definição acima, podemos afirmar que os índices da variável `palavra` segue o formato:

P	y	t	h	o	n
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Vejamos alguns exemplos:

```
# Primeiro caractere
print(palavra[0])
```

```
P
```

```
print(palavra[-6])
```

```
P
```

```
# Primeiro caractere
print(palavra[5])
```

```
n
```

```
print(palavra[-1])
```

```
n
```

3.2.4.2 Fatiamento

Fatiamento (*slincing*) é a capacidade de extrair partes de uma string utilizando `inicio:fim - 1:passo`.

```
# Guardando um objeto do tipo str na variável frase
frase = "A programação em " + palavra + " é diferenciada!"
frase
```

```
'A programação em Python é diferenciada!'
```

```
# Comprimento da variável frase
nStr = len(frase)
nStr
```

```
39
```

```
# Obtendo a frase completa usando fatiamento
print(frase[:nStr - 1])
```

```
A programação em Python é diferenciada
```

```
print(frase[-nStr:])
```

```
A programação em Python é diferenciada!
```

```
# Obtendo a frase de dois em dois caractere
print(frase[:nStr - 1:2])
```

```
Apormçoe yhnédfrnid
```

```
print(frase[-nStr::2])
```

```
Apormçoe yhnédfrnid!
```

```
# Obtendo apenas 'A programação'
print(frase[:14])
```

```
A programação
```

```
print(frase[:-25])
```

```
A programação
```

```
# Obtendo o que há depois de 'A programação'
print(frase[14:])
```

```
em Python é diferenciada!
```

```
print(frase[-25:])
```

```
em Python é diferenciada!
```

```
# Obtendo toda a frase de trás para frente
print(frase[::-1])
```

```
!adaicnerefid é nohtyP me oãçamargorp A
```

> **Nota:** Omitir o primeiro índice (`start`) ou o segundo índice (`stop`) significa, respectivamente, começar desde o começo ou terminar no fim.

Resumindo: para fazer uma fatia de nossa string, precisamos saber de onde começa, até onde vai e o tamanho do passo.

```
fatiável[começo : fim : passo]
```

3.2.4.2.1 Atenção para o uso de indexação e fatiamento

As fatias incluem o índice do primeiro elemento e não incluem o elemento do índice final. Por isso que `frase[0:-1]` perde o último elemento.

Caso o **fim** da fatia seja antes do começo, obtemos um resultado vazio:

```
frase[59:105]
```

```
''
```

O que acontece com uma fatia que está fora da string?

```
frase[123:345]
```

```
''
```

E se o **fim** da fatia for superior ao comprimento da string? Sem problemas, o Python pecorrer a string até o onde der:

```
frase[8:123456789]
```

```
'mação em Python é diferenciada!'
```

Mas um índice fora do intervalo em acesso direto gera erro:

```
[Input]  : frase[123456789]
[Output] : IndexError: string index out of range
```

Quando usamos passos negativos, a fatia começa no **fim** e termina no **começo** e é percorrida ao contrário. Ou seja, invertemos a ordem. Mas tome cuidado:

```
"Python"[2:6]
```

```
'thon'
```

```
"Python"[2:6:-1]
```

```
''
```

```
"Python"[6:2]
```

```
''
```

```
"Python"[6:2:-1]
```

```
'noh'
```

- `"Python"[6:2]`: O índice de início (6) é maior que o índice de fim (2), e o passo é positivo (default). Nesse caso, o resultado é uma string vazia, pois o slicing avança para a direita, mas o fim está à esquerda.
- `"Python"[2:6:-1]`: O índice de início (2) é menor que o índice de fim (6), mas o passo é negativo (-1). O slicing tenta avançar para a esquerda, mas o fim está à direita. Novamente, o resultado é uma string vazia.
- `"Python"[6:2:-1]`: O índice de início (6) é maior que o índice de fim (2), e o passo é negativo (-1). O slicing avança para a esquerda, começando do índice 6 ('n') até o índice 2 ('t'), excluindo-o. Isso resulta na string "noh".

3.2.5 Formatação de Strings

Podemos formatar strings utilizando `f-strings`, `.format()` ou `%`:

```
nome = "Breno"
idade = 21

print(f"Olá, meu nome é {nome} e tenho {idade} anos.")           # f-strings
```

```
Olá, meu nome é Breno e tenho 21 anos.
```

```
print("Olá, meu nome é {} e tenho {} anos.".format(nome, idade)) # format()
```

```
Olá, meu nome é Breno e tenho 21 anos.
```

```
print("Olá, meu nome é %s e tenho %d anos." % (nome, idade))     # Estilo antigo
```

```
Olá, meu nome é Breno e tenho 21 anos.
```

O autor recomenda usar a primeira opção.

3.2.6 Métodos Úteis para Strings

Ao definirmos o objeto `string` em Python, são definidas também algumas ações que este objeto pode executar.

```
# Defini-se um string qualquer
frase = " Python é legal! "

# Visualizar variável
frase
```

```
' Python é legal! '
```

1. Método `str.lower()`: Deixa qualquer caractere em minúsculo.

```
frase.lower()
```

```
' python é legal! '
```

2. Método `str.upper()`: Deixa qualquer caractere em maiúsculo.

```
frase.upper()
```

```
' PYTHON É LEGAL! '
```

3. Método `str.strip()`: Remove os espaços extras.

```
frase.strip()
```

```
'Python é legal!'
```

4. Método `str.title()`: Primeiro caractere de cada palavra em letra maiúscula.

```
frase.strip().title()
```

```
'Python É Legal!'
```

5. Método `str.replace()`: Troca um determinado caractere (1º argumento) por um outro determinado caractere (2º argumento).

```
frase.replace("Python", "Programar")
```

```
' Programar é legal! '
```

6. Método `str.index()`: Retorna o índice de determinado caractere contido na string.

```
frase.index("é")
```

```
8
```

7. Método `str.count()`: Quantifica o número de aparições de determinado caractere na string.

```
frase.count("a")
```

```
1
```

8. Método `str.split()`: Cria uma lista a partir de fatias da string com base em um caractere.

```
# Método split sem argumento (default)
print(frase.split())
```

```
['Python', 'é', 'legal!']
```

```
# Método split com argumento
print(frase.split("é"))
```

```
[' Python ', ' legal! ']
```

3.3 Listas

Listas são uma das estruturas de dados mais usadas em Python. Elas permitem armazenar múltiplos valores em uma única variável e suportam diversos tipos de operações.

3.3.1 Declaração de Listas

Uma lista em Python é definida utilizando colchetes `[]`, e seus elementos são separados por vírgulas:

```
# Lista de números
numeros = [1, 2, 3, 4, 5]

# Lista de strings
frutas = ["maça", "banana", "abacaxi"]

# Lista mista
dados = [25, "João", True, 3.14]
```

Uma lista também pode ser vazia, algo que futaremente veremos que pode ser muito útil, por exemplo:

```
vazia = []
vazia
```

```
[]
```

3.3.2 Indexação e Fatiamento

A ideia de índices e fatias de listas funciona de forma muito parecida com a que foi vista em strings.

```
numeros[0] # Primeiro elemento
```

```
1
```

```
numeros[-1] # Último elemento
```

```
5
```

Assim como na indexação de strings, ao tentar acessar um índice inválido de uma lista é retornado um erro.

Ao ínves de simplesmente acessar um elemneto através de seu índice, podemos obter uma fatia, que pode ser muito mais interessante.

```
numeros[::2] # Do começo ao fim, de 2 em 2 elementos
```

```
[1, 3, 5]
```

```
numeros[::-2] # Do fim ao começo, de 2 em 2 elementos
```

```
[5, 3, 1]
```

```
numeros[:3]   # Três primeiros elementos
```

```
[1, 2, 3]
```

```
numeros[3:]   # Elementos a partir do índice 3
```

```
[4, 5]
```

```
numeros[::-1] # Lista invertida
```

```
[5, 4, 3, 2, 1]
```

3.3.3 Trabalhando com Listas

Imagine que se queira saber se um determinado elemento (objeto) está contido em determinada lista. Poderiamos ficar procurando elemento a elemento, vamos tentar essa abordagem. Verifique se o elemento `0.3146778807984779` está contido na lista abaixo:

```
import random as rd

rd.seed(42)

va = [rd.random() for _ in range(100)]
va
```

```
[0.6394267984578837, 0.025010755222666936, 0.27502931836911926, 0.22321073814882275, 0.7364712141
```

Note que não é viável essa abordagem. Para está finalidade devemos usar o operador lógico `in`. Veja o exemplo:

```
0.3146778807984779 in va # 'elemento' está contido em 'lista'
```

```
True
```

Lembra do operador `not`? Podemos combiná-lo com o `in` para verificar se um elemento 'não está' contido em uma determina lista. Sendo o contrário (negação) da afirmação acima.

```
0.3146778807984779 not in va # 'elemento' não está contido em 'lista'
```

```
False
```

Veja um exemplo de como o `in` funciona em uma outra situação.

```
lista_mista = ['duas palavras', 42, True, ['batman', 'robin'], -0.84, 'hipófise']
42 in lista_mista
```

```
True
```

```
'batman' in lista_mista
```

```
False
```

```
'batman' in lista_mista[3] # Note que o elemento com índice 3 também é uma lista
```

```
True
```

Consegue me dizer quantos elementos têm na lista `va`? Fique tranquilo! Não precisa contar, pode ser usado a função `len` do Python para responder essa pergunta.

```
len(va)
```

```
100
```

```
len(lista_mista[3])
```

```
2
```

3.3.4 Adicionar e Remover elementos de uma Lista

Podemos adicionar elementos de diversas formas:

```
# Adiciona um único elemento ao final da lista
numeros.append(6)
numeros
```

```
[1, 2, 3, 4, 5, 6]
```

```
# Adiciona vários elementos ao final da lista
numeros.extend([7, 8, 9])
numeros
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# Adiciona um elemento em uma posição específica
numeros.insert(2, 15)  # Insere o número 15 na posição 2 (3º elemento)
numeros
```

```
[1, 2, 15, 3, 4, 5, 6, 7, 8, 9]
```

Podemos remover/excluir elementos de uma lista das seguintes formas:

```
# Remove a primeira ocorrência de um valor específico
numeros.remove(15)  # Remove o número 15
numeros
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
# Exclui o último elemento da lista
del numeros[-1]
numeros
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

E se, por exemplo, eu precisar começar a lista do zero? Pode-se remover todos os elementos de uma lista, usando o método `clear()`:

```
va.clear()
print("Lista de números aleatórios:")
```

```
Lista de números aleatórios:
```

```
print(va)
```

```
[]
```

```
va = [rd.random() for _ in range(100)]
print("Lista de números aleatórios:")
```

```
Lista de números aleatórios:
```

```
print(va)
```

```
[0.011481021942819636, 0.7207218193601946, 0.6817103690265748, 0.5369703304087952, 0.266825189952
```

3.3.5 Modificando elementos

Como as listas são mutáveis, podemos alterar seus valores diretamente:

```
print(f"Antes da modificação: \n {frutas}")
```

```
Antes da modificação:
 ['maça', 'banana', 'abacaxi']
```

```
# Alterando 'banana' por 'melancia'
frutas[1] = "melancia"

# Visualizando
print(f"Depois da modificação: \n {frutas}")
```

```
Depois da modificação:
 ['maça', 'melancia', 'abacaxi']
```

Outra forma de se fazer a modificação é:

```
# Obtendo a posição (índice) da fruta (string) 'melancia'
id = frutas.index("melancia")

# Alterando 'melancia' por 'banana'
frutas[id] = "banana"

# Visualizando
print(frutas)
```

```
['maça', 'banana', 'abacaxi']
```

3.3.6 Ordenação de Listas

Por vários motivos, pode ser útil ter em mãos uma lista ordenada. Como fazer isso? Veja os exemplos:

```
desordenada = ['b', 'z', 'k', 'a', 'h']
print(f"Lista desordenada: \n {desordenada}")
```

```
Lista desordenada:
 ['b', 'z', 'k', 'a', 'h']
```

```
# Ordenando
desordenada.sort()
print(f"Lista ordenada: \n {desordenada}")
```

```
Lista ordenada:
 ['a', 'b', 'h', 'k', 'z']
```

Voltemos a lista `va`:

```
# Modificando va um pouco
va_modified = [round(va[i] * 100) for i in range(len(va))]
print(va_modified)
```

```
[1, 72, 68, 54, 27, 64, 11, 43, 45, 95, 88, 26, 50, 18, 91, 87, 30, 64, 61, 15, 76, 54, 78, 53, 0
```

```
# Ordenado de forma crescente
va_modified.sort()
print(va_modified)
```

```
[0, 1, 2, 5, 6, 7, 7, 7, 9, 10, 10, 11, 12, 13, 13, 15, 18, 19, 19, 20, 21, 21, 22, 22, 23, 23, 2
```

```
# Ordenado de forma decrescente
va_modified.sort(reverse=True)
print(va_modified)
```

```
[100, 100, 98, 95, 95, 94, 93, 92, 91, 91, 88, 88, 88, 87, 87, 86, 86, 86, 84, 83, 81, 78, 78, 77
```

Além do método `sort`, tem a função nativa do Python. Função `sorted()`:

```
# Ordenado de forma crescente novamente
va_modified = sorted(va_modified)
print(va_modified)
```

```
[0, 1, 2, 5, 6, 7, 7, 7, 9, 10, 10, 11, 12, 13, 13, 15, 18, 19, 19, 20, 21, 21, 22, 22, 23, 23, 2
```

3.3.7 Cópia de Listas

Cópia ou cópias de listas se torna algo de grande valor quando se quer fazer alguma manipulação, porém não se quer alterar as informações originais. Para isso, deve-se usar o método `copy()`:

```
# Criando listas
l1 = [[1, 2, 3], ["x", "y", "z"], [True, False]]
l2 = l1.copy() # l2 é a cópia de l1

# Visualizando
print(l1)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False]]
```

```
print(l2)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False]]
```

```
# Adicioando um elemento novo somente a l2
l2.append([1/4, 1/2, 3/4, 1])

# Visualizando
print(l1)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False]]
```

```
print(l2)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False], [0.25, 0.5, 0.75, 1]]
```

Agora, observe o que acontece se não fizer uso do `copy()`:

```
# Criando listas
l1 = [[1, 2, 3], ["x", "y", "z"], [True, False]]
l2 = l1 # l2 'igual' a l1

# Visualizando
print(l1)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False]]
```

```
print(l2)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False]]
```

```
# Adicioando um elemento novo somente a l2
l2.append([1/4, 1/2, 3/4, 1])

# Visualizando
print(l1)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False], [0.25, 0.5, 0.75, 1]]
```

```
print(l2)
```

```
[[1, 2, 3], ['x', 'y', 'z'], [True, False], [0.25, 0.5, 0.75, 1]]
```

3.3.8 Operando Listas

Fazendo uso ao conhecimento adquirido de strings. Temos os operadores `+` e `*`, que funcionam de forma muito similar e obdecem as mesmas regras.

O operador `+` concatena (semelhante ao método `extend()`) listas:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
c
```

```
[1, 2, 3, 4, 5, 6]
```

O operador `*` repete a lista dado um número de vezes:

```
a * 2
```

```
[1, 2, 3, 1, 2, 3]
```

```
d = c + a + b + 2 * c
d
```

```
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

O Python fornece várias outras operações úteis para listas, calculadas com funções do módulo básico do Python. Entretanto, tais funções só se aplicam a listas numéricas:

```
print(f"Soma da lista 'd': {sum(d)}")
```

```
Soma da lista 'd': 84
```

```
print(f"Maior valor da lista 'd': {max(d)}")
```

```
Maior valor da lista 'd': 6
```

```
print(f"Menor valor da lista 'd': {min(d)}")
```

```
Menor valor da lista 'd': 1
```

Agora, imagine que seja necessário saber quantas vezes um determinado elemento se repete dentro de uma lista. Tal ação pode ser feita pelo método `count()`:

```
# Lembra dessa lista
print(va_modified)
```

```
[0, 1, 2, 5, 6, 7, 7, 7, 9, 10, 10, 11, 12, 13, 13, 15, 18, 19, 19, 20, 21, 21, 22, 22, 23, 23, 2
```

```
# Quanta vezes o número 100 aparece?
print(va_modified.count(100))
```

```
2
```

3.3.9 Uso da Função `range()` em Listas

Em Python, além de funções como print(), len(), sum(), max() e min(), temos a função range(), que também faz parte do módulo básico. Essa função é extremamente útil para criar sequências numéricas, especialmente listas. Imagine que você precise criar uma lista com os números de 1 a 200. Como fazer isso de forma eficiente?

Uma abordagem seria escrever todos os números manualmente:

```
lista_grande = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ..., 200]
```

No entanto, essa não é a forma mais prática. É aí que entra a função `range()`. Com ela, podemos gerar essa lista de forma muito mais simples:

```
list(range(1, 201))  # Note que o limite superior é 201
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 2
```

O `range()` também oferece flexibilidade para gerar sequências com intervalos específicos. Por exemplo, para obter os números de 0 a 29, pulando de 5 em 5:

```
list(range(0, 30, 5))
```

```
[0, 5, 10, 15, 20, 25]
```

Além disso, o `range()` também oferece algumas coisas interessantes. Por exemplo, imprimir os números espaçados de 5 em 5, entre 0 e 30:

```
list(range(0, 30, 5))
```

```
[0, 5, 10, 15, 20, 25]
```

A sintaxe geral do `range()` é: `range(start, stop, step)`, onde:

- `start`: O valor inicial da sequência (inclusivo).
- `stop`: O valor final da sequência (exclusivo).
- `step`: O intervalo entre os valores.

Por que precisamos converter `range()` para `list`?

```
print(range(200))
```

```
range(0, 200)
```

```
print(type(range(200)))
```

```
<class 'range'>
```

Isso acontece porque `range()` retorna um objeto do tipo `range`, que representa uma sequência numérica, mas não é uma lista em si. Para visualizar os números, precisamos convertê-lo explicitamente para uma lista:

```
range_lista = list(range(200))
print(range_lista)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26
```

3.4 Dicionários

Dicionários em Python são estruturas de dados poderosas que armazenam informações no formato chave-valor. Diferentemente das listas, que são indexadas por números, os dicionários usam chaves imutáveis (strings, números, tuplas, etc.) para acessar seus valores.

Um dicionário é uma coleção de pares chave-valor, onde cada chave é única e está associada a um valor: `{chave: valor}`.

- **Chaves Únicas:** Uma característica fundamental dos dicionários é que cada chave deve ser única. Tentar inserir chaves duplicadas resultará na substituição do valor anterior.

3.4.1 Declaração de Dicionários

Em Python, dicionários podem ser criados de diversas maneiras, oferecendo flexibilidade para diferentes situações.

1. **Criação Direta com Chaves e Valores:**

```
# Dicionário com informações de uma receita
receita = {
    "farinha": "2 xícaras",
    "ovos": 3,
    "leite condensado": "1 lata"
}

# Dicionário com números de telefone
telefones = {"ana": 123456, "yudi": 40028922, "julia": 4124492}
```

Neste exemplo, `"ana"` é uma chave que está associada ao valor `123456`. Cada par chave-valor é separado por vírgula.

```
print(receita)
```

```
{'farinha': '2 xícaras', 'ovos': 3, 'leite condensado': '1 lata'}
```

```
print(telefones)
```

```
{'ana': 123456, 'yudi': 40028922, 'julia': 4124492}
```

2. **Criação de um Dicionário Vazio:**

Você pode criar um dicionário vazio usando apenas chaves {}:

```
contatos = {}
contatos
```

```
{}
```

3. **Criação com a Função dict():**

A função `dict()` permite criar dicionários de forma mais explícita, especialmente quando as chaves são strings simples:

```
pessoa = dict(nome="Carlos", idade=30, cidade="São Paulo")
pessoa
```

```
{'nome': 'Carlos', 'idade': 30, 'cidade': 'São Paulo'}
```

4. **Com uma lista de listas:**

```
# Definindo três listas diferentes
l1 = ["brigadeiro", "leite condesado, achocolatado"]
l2 = ["omelete", "ovos, azeite, condimentos a gosto"]
l3 = ["ovo frito", "ovo, óleo, condimentos a gosto"]

# Criando uma lista de listas
lr = [l1, l2, l3]

# Visualizando resultado
lr
```

```
[['brigadeiro', 'leite condesado, achocolatado'], ['omelete', 'ovos, azeite, condimentos a gosto'
```

```
# Transformando lista de listas em um dicionário
receitas = dict(lr)

# Visualizando o resultado
receitas
```

```
{'brigadeiro': 'leite condesado, achocolatado', 'omelete': 'ovos, azeite, condimentos a gosto', '
```

3.4.2 Chaves

Podemos acessar os valores de um dicionário através de suas chaves:

```
# Definindo um dicionário para capitais de estados brasileiros
capitais = {"SP": "São Paulo", "AC": "Rio Branco", "TO": "Palmas",
            "RJ": "Rio de Janeiro", "SE": "Aracaju", "MG": "Belo Horizonte"}

# Acessando o valor correspondente a chave "MG"
capitais["MG"]
```

```
'Belo Horizonte'
```

Caso a chave não exista, podemos evitar erros usando o método `get()`:

```
capitais.get("PA")
```

```
capitais.get("PA", "Não tem!")
```

```
'Não tem!'
```

Note que o método `get()` funciona de forma similar ao código `dicionário[chave]`, entretanto, caso a chave não exista garantimos que o código não gere erro diferente do que aconteceria caso fosse usado `dicionário[chave]`.

Repare, também, que a chave `"PA"` não foi adicionada ao dicionário.

```
capitais
```

```
{'SP': 'São Paulo', 'AC': 'Rio Branco', 'TO': 'Palmas', 'RJ': 'Rio de Janeiro', 'SE': 'Aracaju',
```

Agora, se o objetivo não for saber o valor associado a determinada chave, mas sim saber se a chave existe, isso pode ser feito usando o método `keys` e o operador lógico `in`:

```
# Chaves do Dicionário
print(capitais.keys())
```

```
dict_keys(['SP', 'AC', 'TO', 'RJ', 'SE', 'MG'])
```

```
# Verificando
print(f'A chave "PA" está no dicionário capitais? {"PA" in capitais.keys()}')
```

```
A chave "PA" está no dicionário capitais? False
```

Note que os valores de um dicionário pode ser qualquer tipo de objeto. No entanto, foi usado, até então, para exemplos, apenas `strings`. Porém, pode-se colocar todo tipo de coisa dentro dos dicionários, incluindo listas e até mesmo outros dicionários:

```
numeros = {"primos": [2, 3, 5], "pares": [0, 2, 4], "ímpares": [1, 3, 5]}
print(numeros)
```

```
{'primos': [2, 3, 5], 'pares': [0, 2, 4], 'ímpares': [1, 3, 5]}
```

```
docente = {
  "Nome": "Prof Dr Vinícius Duarte Lima",
  "Formação": {"Graduação": "Eng Elétrica", "Mestrado": "Eng Elétrica", "Doutorado": "Eng Elétric
  "Idade": 45,
}

print(docente)
```

```
{'Nome': 'Prof Dr Vinícius Duarte Lima', 'Formação': {'Graduação': 'Eng Elétrica', 'Mestrado': 'E
```

Mesmo que os pares `chave: valor` estejam organizados na ordem que foram colocados, não podemos acessá-los por índices como faríamos em listas:

```
[Input]  : numeros[2]
[Output] : KeyError: 2
```

Assim como os valores não precisam ser do tipo string, o mesmo vale para as chaves:

```
numeros_por_extenso = {2: "dois", 1: "um", 3: "três", 0: "zero"}
numeros_por_extenso[0]
```

```
'zero'
```

```
numeros_por_extenso[2]
```

```
'dois'
```

> **Nota:** Listas e outros dicionários não podem ser usados como chaves por serem de tipos mutáveis.

3.4.3 Adicionando, Alterando e Removendo Elementos

Para exemplificar os comandos desta seção, considere o dicionário abaixo.

```
# Informações de Cleiton
pessoa = {"nome": "Cleiton", "idade": 34, "família": {"mãe": "Maria", "pai": "Enzo"}}

# Visualizando
pessoa
```

```
{'nome': 'Cleiton', 'idade': 34, 'família': {'mãe': 'Maria', 'pai': 'Enzo'}}
```

Para adicionar o item "masculino" a chave "genêro" podemos usar os seguintes comandos:

```
# Adicionando elemento
pessoa["genêro"] = "romântico"

# Visualizando
pessoa
```

```
{'nome': 'Cleiton', 'idade': 34, 'família': {'mãe': 'Maria', 'pai': 'Enzo'}, 'genêro': 'romântico
```

De forma equivalente, temos:

```
# Dicionário de meses do ano
meses = {1: "Janeiro", 2: "Fevereiro", 3: "Março"}

# Adicionando o mês de "Abril" na chave 4
meses[4] = "Setembro"

# Visualizando
meses
```

```
{1: 'Janeiro', 2: 'Fevereiro', 3: 'Março', 4: 'Setembro'}
```

As vezes, podem haver inconsistências nos dados, logo, se torna necessário corrigir tais inconsistências. Isso pode ser feito de forma muito simples, veja:

```
# Executando alterações
pessoa["genêro"] = "masculino"
meses[4] = "Abril"

# Visualizando
print(pessoa)
```

```
{'nome': 'Cleiton', 'idade': 34, 'família': {'mãe': 'Maria', 'pai': 'Enzo'}, 'genêro': 'masculino
```

```
print(meses)
```

```
{1: 'Janeiro', 2: 'Fevereiro', 3: 'Março', 4: 'Abril'}
```

Um dúvida que talvez possa surgir é como fazer alteração em listas que estão contidas em dicionários. Na verdade, isso é feito de forma bastante objetiva, veja:

```
# Novo dicionário
lixo = {"plástico": ["garrafa", "copinho", "canudo"],
        "papel": ["folha amassada", "guardanapo"],
        "orgânico": ["batata", "resto do bandeco", "casca de banana"]}

# Visualizando
print(lixo)
```

```
{'plástico': ['garrafa', 'copinho', 'canudo'], 'papel': ['folha amassada', 'guardanapo'], 'orgâni
```

```
# Obtendo lista de interesse para alteração
lista_de_interesse = lixo["plástico"]

# Verificando em qual posição está o elemneto que deve ser alterado
id = lista_de_interesse.index("garrafa")

# Fazendo a alteração diretamente no dicionário
lixo['plástico'][id] = "sacola"

# Visualizando dicionário alterado
lixo
```

```
{'plástico': ['sacola', 'copinho', 'canudo'], 'papel': ['folha amassada', 'guardanapo'], 'orgânic
```

Note que poderiamos ser mais direto, pois as listas em questão têm comprimentos pequenos. Porém, tal código já pode ser implementado para listas de grande comprimento.

Porém, ao lidarmos com dicionários, talvez precisemos excluir algum elemento. Serão apresentadas duas formas de fazer isso.

1. Usando o método `pop()`:

```
pessoa.pop("família")
```

```
{'mãe': 'Maria', 'pai': 'Enzo'}
```

Repare que `dicionário.pop(chave)` excluí o elemento e retorna os itens da chave excluída. Para visualizarmos o resultado basta chamar o dicionário.

```
pessoa
```

```
{'nome': 'Cleiton', 'idade': 34, 'genêro': 'masculino'}
```

2. Usando o comando `del` do Python Básico:

```
del meses[4]
```

Diferente do método `pop()`, o comando `del` não retorna nada. Porém, para visualizar o resultado, precisamos chamar o objeto.

```
meses
```

```
{1: 'Janeiro', 2: 'Fevereiro', 3: 'Março'}
```

Para excluir todos os elementos de um dicionário, temos o método `clear()`:

```
# Apagando todos os elementos do dicionário
lixo.clear()

# Visualizando
lixo
```

```
{}
```

3.4.4 Função `list()` e Função `len()` para Dicionários

A função `list()` converte um dicionário em uma lista contendo apenas suas chaves:

```
institutos_uspsc = {
    "IFSC": "Instituto de Física de São Carlos",
    "ICMC": "Instituto de Ciências Matemáticas e de Computação",
    "EESC": "Escola de Engenharia de São Carlos",
    "IAU": "Instituto de Arquitetura e Urbanismo",
    "IQSC": "Instituto de Química de São Carlos"
}

# Convertendo dicionário em lista de chaves
lista_chaves = list(institutos_uspsc)
print(lista_chaves)
```

```
['IFSC', 'ICMC', 'EESC', 'IAU', 'IQSC']
```

A função `len()` retorna o número de itens em um objeto. Para dicionários, ela conta o número de pares chave-valor:

```
# Contando itens no dicionário
quantidade_institutos = len(institutos_uspsc)
print(quantidade_institutos)
```

```
5
```

```
# Equivalente a contar as chaves convertidas em lista
print(len(list(institutos_uspsc)))
```

```
5
```

3.4.5 Outros comandos para dicionários

Vejamos agora os métodos `items()` e `values()`. Considere o seguinte dicionário:

```
pessoa = {"nome": "Enzo", "RA": 242334, "curso": "fiscomp"}
```

1. `items()` - Retorna uma visão dos pares chave-valor:

```
pares = pessoa.items()
print(list(pares))
```

```
[('nome', 'Enzo'), ('RA', 242334), ('curso', 'fiscomp')]
```

2. `values()` - Retorna uma visão dos valores armazenados:

```
valores = list(pessoa.values())
print(valores)
```

```
['Enzo', 242334, 'fiscomp']
```

Observação importante: A função `list()` aplicada diretamente a um dicionário (`list(pessoa)`) retorna apenas as chaves, equivalente a `list(pessoa.keys())`.

Dicionários em Python (versões 3.7+) mantêm a ordem de inserção, mas a igualdade entre dicionários considera apenas os pares chave-valor, não a ordem:

```
numerinhos = {"um": 1, "dois": 2, "três": 3}
numeritos = {"três": 3, "dois": 2, "um": 1}

print(numerinhos == numeritos)
```

```
True
```

```
print(numerinhos)  # Mostra na ordem de inserção
```

```
{'um': 1, 'dois': 2, 'três': 3}
```

```
print(numeritos)   # Mostra na ordem de inserção diferente
```

```
{'três': 3, 'dois': 2, 'um': 1}
```

Outro comando que pode ser usado para adicionar elementos em um dicionário pode ser o método `update()`.

```
# Lembra do dicionários receitas?
print(receitas)
```

```
{'brigadeiro': 'leite condesado, achocolatado', 'omelete': 'ovos, azeite, condimentos a gosto', '
```

```
# Nova receita
outros_elementos = {"mingau": "massa, leite, açúcar"}

# Adicionando a nova receita
receitas.update(outros_elementos)

# Visualizando
print(receitas)
```

```
{'brigadeiro': 'leite condesado, achocolatado', 'omelete': 'ovos, azeite, condimentos a gosto', '
```

- **Resumão:**

1. Os métodos `items()` e `values()` (e `keys()`) retornam objetos de visualização que refletem automaticamente as alterações no dicionário original;
2. A partir do Python 3.7, a ordem de inserção é preservada como característica da implementação, tornando-se parte da especificação na versão 3.8;
3. Uso do método `update` para integralização de dicionários.

4 Controle de Fluxo & Funções em Python

4.1 Introdução

Em programação, o **controle de fluxo** determina a ordem em que as instruções são executadas em um programa. Em Python, isso permite criar programas que tomam decisões e repetem ações com base em condições específicas, tornando seu código mais dinâmico e adaptável.

Nesta seção, exploraremos as principais estruturas de controle de fluxo:

- **Condicionais - `if`, `elif` e `else`:** Permitem executar blocos de código diferentes dependendo se uma condição é verdadeira ou falsa.
- **Laços - `for` e `while`:** Permitem repetir um bloco de código várias vezes, seja por um número específico de vezes ou enquanto uma condição for verdadeira.
- **Tratamento de exceções - `try`, `except` e `finally`:** Permitem lidar com erros e situações inesperadas sem interromper o programa.
- **Palavras-chave - `break` e `continue`:** Permitem controlar o fluxo de execução dentro de laços.

Por que o controle de fluxo é importante? Imagine um programa que:

- Decide se um usuário tem acesso a um sistema (`if`).
- Repete uma ação até que um download seja concluído (`while`).
- Percorre uma lista de produtos para calcular descontos (`for`).
- Previne erros se um arquivo não for encontrado (`try/except`).

Sem controle de fluxo, nossos programas seriam lineares e limitados. Com ele, ganhamos **flexibilidade** e **poder** para resolver problemas complexos.

Dominar o controle de fluxo é essencial para qualquer pessoa que esteja aprendendo programação, pois ele está presente em praticamente todo código real. Vamos começar a explorar cada conceito com exemplos práticos e exercícios!

4.2 Estruturas de Decisão

As estruturas de decisão em Python são usadas para controlar o **fluxo do programa**, ou seja, para **decidir qual bloco de código será executado** dependendo de uma ou mais condições. Pense como um semáforo: dependendo da cor, uma ação diferente deve ser tomada — o mesmo acontece em um programa.

- Exemplo do mundo real:

```
Se estiver nublado:
    Levarei guarda-chuva
Senão:
    Não levarei
```

Nota: Em Python, a indentação (recuo de quatro espaços ou um *tab*) é obrigatória para definir os blocos de código. Isso significa que o que estiver indentado será considerado parte da condição.

4.2.1 Estrutura `if` Simples

A estrutura mais básica do controle de fluxo é o `if`, que permite executar um bloco de código apenas se uma condição for verdadeira.

```
idade = 20
if idade >= 18:
    print("Você é maior de idade.")
```

```
Você é maior de idade.
```

Neste exemplo, como `idade` é 20, que é maior que 18, a mensagem será exibida.

4.2.2 Estrutura `if/else`

Se quisermos executar um código para o caso em que a condição não é satisfeita, usamos o `else`.

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

```
Você é menor de idade.
```

4.2.3 Estrutura `if/elif/else`

Para avaliar múltiplas condições, usamos `elif` ("else if").

```
nota = 8
if nota >= 9:
    conceito = "A"
elif nota >= 7:
    conceito = "B"
elif nota >= 5:
    conceito = "C"
else:
    conceito = "D"

print("Conceito:", conceito)
```

```
Conceito: B
```

Diferença entre `if/elif` e múltiplos `if` independentes

Há uma diferença ao usar as condições `if/elif` e usar vários `if` de forma consecutivas. É importante entender como essa diferença funciona para que o seu programa não atenda mais de uma condição quando na verdade "apenas uma" é a verdadeira. Vamos há um exemplo claro e objetivo.

```
if 1 == 1:
    print("Caiu no 1º if")
elif 1 >= 1:
    print("Caiu no elif")
else:
    print("Caiu no else")
```

```
Caiu no 1º if
```

Perceba que duas condições em teste são verdadeiras. Porém, ao usarmos a estrutura `if/elif`, apenas *o bloco com a primeira condição verdadeira é executado*. No exemplo, apenas o blobo seguinte a condição `if a == 1` foi executado.

Agora, veja o exemplo adaptado a estrutura de `if` consecutivos:

```
if 1 == 1:
    print("Caiu no 1º if")
```

```
Caiu no 1º if
```

```
if 1 >= 1:
    print("Caiu no 2º if")
else:
    print("Caiu no else")
```

```
Caiu no 2º if
```

No bloco de código acima não há uma regra pré definida pelo Python. Ambos os blocos `if` serão avaliados independentemente. Isso pode ser útil em algumas situações, mas exige atenção!

4.2.4 Outros Exemplos de Condicionais

- **1º Exemplo - Condicionais Aninhadas:** Útil quando queremos verificar uma condição dada que outra já foi satisfeita. Ou seja, uma estrutura condicional pode estar dentro de outra.

```
# Definindo uma variável numérica
value = 2.35

# 1ª condição (externa)
if value <= 1:
  print("O valor é menor ou igual a 1")

  # 1ª condição (interna) dada que 1ª condição (externa) foi satisfeita
```

```
  if value < 0.5:
    print("E é menor que 0,5")

  # 2ª condição (interna) dada que 1ª condição (externa) foi satisfeita
  elif value == 0.5:
    print("O valor é igual a 0,5")

  # 3ª condição (interna) dada que 1ª condição (externa) foi satisfeita
  else:
    print("O valor é maior que 0,5")

# 2ª condição
elif value <= 2:
  print("O valor é menor ou igual a 2")

  # 1ª condição (interna) dada que 2ª condição (externa) foi satisfeita
  if value < 1.5:
    print("E é menor que 1,5")

  # 2ª condição (interna) dada que 2ª condição (externa) foi satisfeita
  elif value == 1.5:
    print("O valor é igual a 1,5")

  # 3ª condição (interna) dada que 2ª condição (externa) foi satisfeita
  else:
    print("O valor é maior que 1,5")

# 3ª condição
else:
  print("Sabe-se apenas que o valor é maior que 2")

  # 1ª condição (interna) dada que 3ª condição (externa) foi satisfeita
  if value <= 2.5:
    print("Podendo variar entre (2; 2,5]")

  # 2ª condição (interna) dada que 3ª condição (externa) foi satisfeita
  elif value <= 3:
    print("Podendo variar entre (2,5; 3]")

  # 3ª condição (interna) dada que 3ª condição (externa) foi satisfeita
  else:
    print("O valor é maior que 3")
```

```
Sabe-se apenas que o valor é maior que 2
Podendo variar entre (2; 2,5]
```

- **2º Exemplo - Classificação de Valores:** Já pararam para pensar como o conceito é dado pelo sistema da Universidade Federal do Pará? A classificação ocorre da seguinte maneira:

```python
import random as rd

rd.seed(123456789)
review = [10 * rd.random() for _ in range(5)]
mean = sum(review) / len(review)
print(f"Média das Avaliações: {round(mean, 2)}")
```

```
Média das Avaliações: 7.66
```

```python
# Em qual conceito está média estaria?
if (mean >= 0) and (mean < 5):
  print("O conceito do aluno foi INSUFICIENTE")
elif (mean >= 5) and (mean < 7):
  print("O conceito do aluno foi REGULAR")
elif (mean >= 7) and (mean < 9):
  print("O conceito do aluno foi BOM")
elif (mean >= 9) and (mean <= 10):
  print("O conceito do aluno foi EXCELENTE")
else:
  print("Média de Avaliações Inválida")
```

```
O conceito do aluno foi BOM
```

- **2º Exemplo - Classificação de Valores:** Já pararam para pensar como o conceito é dado pelo sistema da Universidade Federal do Pará? Mas também pode ser feita de forma mais direta:

```python
print(f"Média das Avaliações: {round(mean, 2)}")
```

```
Média das Avaliações: 7.66
```

```python
# Em qual conceito está média estaria?
if 0 <= mean < 5:
  print("O conceito do aluno foi INSUFICIENTE")
elif 5 <= mean < 7:
  print("O conceito do aluno foi REGULAR")
elif 7 <= mean < 9:
  print("O conceito do aluno foi BOM")
elif 9 <= mean < 10:
  print("O conceito do aluno foi EXCELENTE")
else:
  print("Média de Avaliações Inválida")
```

```
O conceito do aluno foi BOM
```

- **3º Exemplo - Condição com Expressão Ternária:** Forma reduzida do `if/else`, muito últil para atribuições simples. Exemplo com `string`:

```python
# Dados do usuário
age = 19
CNH = True

# Verificação com Expressão Ternária
result = "Está apto a dirigir!" if age >= 18 and CNH != False else "Não está apto a dirigir!"

# Visualizar
print(result)
```

```
Está apto a dirigir!
```

- **3º Exemplo - Condição com Expressão Ternária:** Forma reduzida do `if/else`, muito últil para atribuições simples. Exemplo com `number`:

```python
# Definindo números
x, y = 4, 5

# Verificação com Expressão Ternária
maior_valor = x if x > y else y

# Visualizar
print(maior_valor)
```

```
5
```

4.2.5 Considerações e Boas Práticas

- Use `elif` quando apenas uma entre várias condições pode ser verdadeira;
- Evite aninhamentos profundos de `if`;
- Use operações compostas como `7 <= mean < 9` para mais clareza;
- Sempre comente blocos de código complexos;
- Lembre das **?@tbl-operLogic** que apresentam os *operadores lógicos* e **?@tbl-conecLogic** que mostra os *conectores lógicos*.

4.3 Estruturas de Repetição

As estruturas de repetição são usadas para executar um bloco de código várias vezes, de forma automática, até que uma condição seja satisfeita ou uma sequência de elementos seja percorrida.

Em Python, temos duas principais formas de repetição:

- `for`: Ideal quando sabemos o número de repetições ou estamos percorrendo uma estrutura (lista, string, dicionário, etc);
- `while`: Ideal quando não sabemos o número exato de repetições e precisamos que o código continue enquanto uma condição for verdadeira.

4.3.1 Estrutura `for`

A estrutura `for` é uma das formas mais comuns de repetição em Python. Ela é ideal quando:

- Sabemos de antemão quantas vezes o código deve ser repetido; ou
- Queremos percorrer os elementos de uma sequência (como listas, strings, dicionários, etc.).

Vamos começar com o básico: o objeto `range()`.

```
# Lembra do objeto range()?
range(3)
```

```
range(0, 3)
```

O comando acima não imprime os números, ele apenas cria um objeto que representa uma sequência de 0 até 2 (três elementos, começando do zero). Para visualizar os elementos, podemos convertê-lo em uma lista:

```
list(range(3))
```

```
[0, 1, 2]
```

Agora vamos usar o `range()` dentro de um laço `for`:

```
for i in range(3):
    print(i)
```

```
0
1
2
```

> Aqui, o loop `for` vai executar o bloco de código três vezes, e a variável `i` assume os valores 0, 1 e 2 a cada repetição.

Você também pode fazer a mesma coisa passando uma lista explicitamente:

```
print(f"Lista exemplo para iteração: {list(range(3))}")
```

```
Lista exemplo para iteração: [0, 1, 2]
```

```
print("Iniciando contagem:")
```

```
Iniciando contagem:
```

```
for contador in list(range(3)):
  print(f"Passo {contador + 1}: Valor atual = {contador}")
```

```
Passo 1: Valor atual = 0
Passo 2: Valor atual = 1
Passo 3: Valor atual = 2
```

```
print("Contagem concluída!")
```

```
Contagem concluída!
```

O resultado será o mesmo. Nesse caso, você está iterando diretamente sobre os elementos de uma lista.

4.3.1.1 Iterando em Strings

Strings são sequências de caracteres, e podemos percorrê-las com o `for`.

```
texto = "Curso de Python para Análise de Dados."
for caractere in texto:
    print(caractere)
```

```
C
u
r
s
o

d
e

P
y
t
h
o
n

p
a
r
a

A
n
á
l
i
s
e

d
e

D
```

```
a
d
o
s
.
```

A cada repetição, a variável `letra` recebe um caractere da string. Esse recurso é útil para manipulações de texto.

4.3.1.2 Interando em Listas

Suponha que você tenha uma lista com nomes de frutas, e deseja imprimir cada fruta em letras maiúsculas.

```
# Lista de frutas
frutas = ["maçã", "banana", "laranja", "abacaxi", "uva"]

# Imprimindo cada fruta em caixa alta
for fruta in frutas:
    print(fruta.upper())
```

```
MAÇÃ
BANANA
LARANJA
ABACAXI
UVA
```

O método `upper()` transforma a string para letras maiúsculas. A variável fruta assume o valor de cada elemento da lista a cada iteração.

Vamos ver outro exemplo: elevar números ao quadrado e ao cubo:

```
# Criando a lista de números de 0 a 5
numbers = list(range(6))

# Iterando sobre a lista
for number in numbers:
    print(f"Número: {number}")
    print(f"Número ao quadrado: {number ** 2}")
    print(f"Número ao cubo: {number ** 3}\n")
```

```
Número: 0
Número ao quadrado: 0
Número ao cubo: 0

Número: 1
Número ao quadrado: 1
Número ao cubo: 1
```

```
Número: 2
Número ao quadrado: 4
Número ao cubo: 8

Número: 3
Número ao quadrado: 9
Número ao cubo: 27

Número: 4
Número ao quadrado: 16
Número ao cubo: 64

Número: 5
Número ao quadrado: 25
Número ao cubo: 125
```

`number` assume os valores de 0 a 5 e cada iteração é mensurado o seu quadrado e o seu cubo.

4.3.1.3 Iterando em Dicionários

Dicionários são estruturas de dados compostas por pares `chave:valor`. Podemos iterar por eles usando o método `items()`, que retorna as chaves e valores simultaneamente:

```python
# Dicionário com traduções da palavra "gato"
translations = {"Português": "gato", "Inglês": "cat", "Francês": "chat"}

# Percorrendo as chaves e os valores
for idioma, palavra in translations.items():
    print(f"{idioma} -> {palavra}")
```

```
Português -> gato
Inglês -> cat
Francês -> chat
```

A cada iteração, `idioma` recebe a chave e `palavra` recebe o valor correspondente.

Você também pode usar `zip()` para fazer a iteração:

```python
# Outra forma de iterar: combinando as chaves e os valores com zip()
for idioma, palavra in zip(translations.keys(), translations.values()):
    print(f"{idioma} -> {palavra}")
```

```
Português -> gato
Inglês -> cat
Francês -> chat
```

O `zip()` combina os elementos das duas listas (chaves e valores) em pares.

Iterando com `enumerate()`: Às vezes, além de acessar o valor de uma lista ou dicionário, também queremos saber a posição (índice) do elemento. Para isso, usamos `enumerate()`.

Vamos aplicar isso a um dicionário um pouco mais complexo:

```
# Dicionário com informações de pessoas
dados = {
    "Nome": ["Igor", "Allan", "Victória", "Izabella", "Fernando"],
    "Idade": [39, 17, 98, 45, 27],
    "Animal de Estimação": ["Gato", "Tigre", "Arara", "Javali", "Ratatouille"]
}

# Percorrendo os dados
for i, key in enumerate(dados.keys()):
    if i == 0:
        print("Informações Coletadas:")
    for id in dados[key]:
        if id == dados[key][0]:
            print(f"   {i + 1}. {key}:")
            print(f"      {id}")
        else:
            print(f"      {id}")
```

```
Informações Coletadas:
   1. Nome:
      Igor
      Allan
      Victória
      Izabella
      Fernando
   2. Idade:
      39
      17
      98
      45
      27
   3. Animal de Estimação:
      Gato
      Tigre
      Arara
      Javali
      Ratatouille
```

Aqui temos dois níveis de repetição. O primeiro laço percorre cada "coluna" do dicionário, enquanto o segundo imprime os dados. O `enumerate()` é usado para acessar o índice `i` da chave atual, que nos ajuda a numerar facilitando a visualização de chave a chave.

Esse tipo de estrutura é útil para imprimir dados de maneira organizada, como se fosse uma tabela ou relatório.

4.3.1.4 Compreensão de Listas (List Comprehension)

Uma forma compacta e elegante de construir listas com base em laços `for` é a compreensão de listas. A sintaxe básica é:

```
[expressao for item in lista]
```

Isso pode ser lido como: *aplique a* ***expressao*** *a cada* ***item*** *da* ***lista***.
Vejamos um exemplo simples.

```
# Forma tradicional
quadrados = []
for num in range(1, 11):
    quadrados.append(num ** 2)

# Visualizar
print(quadrados)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
# Forma com list comprehension
quadrados = [num ** 2 for num in range(1, 11)]

# Visualizar
print(quadrados)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A cada iteração é calculado o quadrado do `num` que assume valores de 1 a 10.

4.3.1.4.1 Compreensão de Listas com Condição `if`

É possível aplicar condicionais em compreensão de listas. De forma geral, a seguinte sintaxe é seguida:

```
[expression for item in list if condition]
```

Logo, a linha de código acima diz *aplique a* ***expression*** *para cada* ***item*** *da* ***list*** *dado que* ***condition*** *é verdadeira.*
Vejamos alguns exemplos:

```
# Forma tradiconal
pares = []
for num in range(1, 21):
  if num % 2 == 0:
    pares.append(num)

# Visualizar
print(pares)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
# Forma com list comprehension
pares = [num for num in range(1, 21) if num % 2 == 0]

# Visualizar
print(pares)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Apenas os números pares serão incluídos na lista.

Também é possível combinar várias condições:

```
# Forma tradiconal
multiplos = []
for num in range(100):
  if num % 2 == 0 and num % 5 == 0:
    multiplos.append(num)

# Visualizar
print(multiplos)
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

```
# Forma com list comprehension
multiplos = [num for num in range(100) if num % 2 == 0 and num % 5 == 0]

# Visualizar
print(multiplos)
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

A lista conterá números múltiplos de 2 **e** de 5.

```
# Forma tradiconal
multiplos = []
for num in range(100):
  if num % 2 == 0 and num % 5 == 0 and num % 6 == 0:
    multiplos.append(num)

# Visualizar
print(multiplos)
```

```
[0, 30, 60, 90]
```

```
# Forma com list comprehension
multiplos = [num for num in range(100) if num % 2 == 0 and num % 5 == 0 and num % 6 == 0]

# Visualizar
print(multiplos)
```

```
[0, 30, 60, 90]
```

A lista conterá números múltiplos de 2, 5 **e** 6.

```
# Forma tradiconal
multiplos = []
for num in range(100):
  if num % 2 == 0 and num % 5 == 0 or num % 6 == 0:
    multiplos.append(num)

# Visualizar
print(multiplos)
```

```
[0, 6, 10, 12, 18, 20, 24, 30, 36, 40, 42, 48, 50, 54, 60, 66, 70, 72, 78, 80, 84, 90, 96]
```

```
# Forma com list comprehension
multiplos = [num for num in range(100) if num % 2 == 0 and num % 5 == 0 or num % 6 == 0]

# Visualizar
print(multiplos)
```

```
[0, 6, 10, 12, 18, 20, 24, 30, 36, 40, 42, 48, 50, 54, 60, 66, 70, 72, 78, 80, 84, 90, 96]
```

A lista conterá números múltiplos de 2 **e** de 5 **ou** de 6.

4.3.1.4.2 Compreensão de Listas com `if` e `else`

A estrutura muda levemente:

```
[expression_if if condition else expression_else for item in list]
```

Em outras palavras: *execute* ***`expression_if`*** *caso* ***`condition`*** *seja verdadeira e* ***`expression_else`*** *caso contrário para cada* ***`item`*** *da* ***`list`***.

Vamos a um exemplo.

```
# Forma tradiconal
sucess_number_div_5 = []
for number in range(26):
  if number % 5 == 0:
    sucess_number_div_5.append("sucess")
  else:
    sucess_number_div_5.append("failure")

# Visualizar
print(sucess_number_div_5)
```

```
['sucess', 'failure', 'failure', 'failure', 'failure', 'sucess', 'failure', 'failure', 'failure',
```

```
# Forma com list comprehension
sucess_number_div_5 = ["sucess" if number % 5 == 0 else "failure" for number in range(26)]

# Visualizar
print(sucess_number_div_5)
```

```
['sucess', 'failure', 'failure', 'failure', 'failure', 'sucess', 'failure', 'failure', 'failure',
```

A expressão acima retorna uma lista com as palavras `"sucess"` ou `"failure"` dependendo do valor de cada número.

4.3.1.4.3 Múltiplas Compreensão de Listas

Aqui é exigida um pouco mais de atenção. Considere a matriz

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

e

$$\mathbf{x}^\top = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}.$$

Como fazer isso usando compreensão de listas?

```
matrix = [
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12]
]

print(matrix)
```

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Para facilitar o entendimento e melhorar a aprendizagem, vamos fazer usando estrutura `for` de forma simples.

```
# Forma tradiconal
transposta = []
for column in range(len(matrix[0])):
  row = []

  for element in matrix:
    row.append(element[column])

  transposta.append(row)

# Visualizar
print(transposta)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Entretanto, é possível otimizar o código usando compreensão de listas.

```
# Forma com list comprehension
transposta = [[element[column] for element in matrix] for column in range(len(matrix[0]))]

# Visualizar
print(transposta)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

4.3.2 Estrutura `while`

A estrutura de repetição `while` é usada quando **não sabemos com antecedência quantas vezes o bloco de código deve ser repetido**. A repetição acontece **enquanto uma condição lógica for verdadeira**. Assim que essa condição se torna falsa, o laço é encerrado automaticamente.

A estrutura geral é:

```
while condição:
    # bloco de código
```

Isso significa que o código dentro do `while` será executado repetidamente **enquanto** a condição especificada for verdadeira. Por isso, é **muito importante** garantir que a condição eventualmente se torne falsa — do contrário, o programa pode entrar em um **loop infinito**.

4.3.2.1 Exemplos de Uso

4.3.2.1.1 Contador Simples

```
contador = 1

while contador <= 5:
    print(f"Contador: {contador}")
    contador += 1
```

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
```

Esse é o uso mais básico de um `while`. A variável de controle é atualizada em cada repetição até que a condição se torne falsa.

4.3.2.1.2 Apenas Números Pares

```python
n = 1

while n <= 10:
    if n % 2 == 0:
        print(f"{n} é par")
    n += 1
```

```
2 é par
4 é par
6 é par
8 é par
10 é par
```

Neste exemplo, o `if` dentro do `while` adiciona uma **condição extra**, filtrando apenas os números pares entre 1 e 10.

4.3.2.1.3 Exemplo com `input()`

```python
number = int(input("Digite um número positivo: "))

while number <= 0:
    print("O número fornecido não é positivo. Tente novamente!")
    number = int(input("Digite um número positivo: "))

print(f"Parabêns. O número fornecido é positivo!")
```

Esse exemplo usa `input()` para verificar se o valor digitado é positivo ou não.

4.3.2.1.4 Exemplo com string: invertendo uma palavra

```python
palavra = "Python"
reverso = ""
i = len(palavra) - 1

while i >= 0:
    reverso += palavra[i]
    i -= 1

print(f"A palavra '{palavra}' invertida é '{reverso}'.")
```

```
A palavra 'Python' invertida é 'nohtyP'.
```

Aqui usamos `while` para percorrer uma string de trás para frente e gerar sua versão invertida.

4.3.2.1.5 Exemplo com lista: somando números até esvaziar a lista

```python
numeros = [10, 20, 30, 40, 50]
soma = 0

while numeros:
    valor = numeros.pop()  # Remove o último elemento
    soma += valor

print(f"Soma dos valores: {soma}")
```

```
Soma dos valores: 150
```

Esse exemplo mostra como o `while` pode operar **enquanto uma lista tiver elementos** — uma abordagem comum para **estrutura de pilha**.

4.3.2.1.6 Exemplo com múltiplas condições

```python
dados, i = [12, -3, 5, 0, -8, 19, -1, 0, 7], 0

positivos, negativos, zeros = 0, 0, 0

while i < len(dados):
    if dados[i] > 0:
        positivos += 1
    elif dados[i] < 0:
        negativos += 1
    else:
        zeros += 1
    i += 1

print(f"Positivos: {positivos}, Negativos: {negativos}, Zeros: {zeros}")
```

```
Positivos: 4, Negativos: 3, Zeros: 2
```

Esse exemplo simula uma **classificação de dados** usando `while` com múltiplas condições `if-elif-else`.

4.4 Comandos-Chave

Em desenvolvimento Python, à medida que a complexidade do código aumenta, torna-se fundamental o domínio de mecanismos para o tratamento robusto de erros e o controle preciso da execução de estruturas de repetição. Esta seção explora dois grupos essenciais de comandos que possibilitam um fluxo de execução mais refinado e resiliente.

- **Tratamento de Exceções:** Python oferece um conjunto de construções para o tratamento de exceções, permitindo o desenvolvimento de código capaz de responder de forma controlada a eventos inesperados durante a execução. Ao utilizar blocos `try`, `except` e, opcionalmente, `finally`, é possível interceptar e manipular erros, evitando a interrupção abrupta do programa e implementando estratégias de recuperação ou finalização controlada de recursos.

- **Controle de Fluxo em Loops:** Adicionalmente às estruturas condicionais básicas, Python disponibiliza palavras-chave específicas para o controle do fluxo de execução dentro de laços de repetição (`for` e `while`). Os comandos `break` e `continue` fornecem mecanismos para alterar o comportamento padrão dos loops:
  - `break`: Interrompe a execução do loop corrente e transfere o controle para a próxima instrução após o loop.
  - `continue`: Interrompe a iteração corrente e passa para a próxima iteração do loop.

O domínio destas ferramentas de tratamento de exceções e controle de fluxo em loops é crucial para a escrita de código Python robusto, eficiente e capaz de lidar com cenários complexos de execução. A correta aplicação destes conceitos contribui significativamente para a qualidade e a manutenibilidade de projetos de software.

4.4.1 Tratamento de Exceções

Em Python, erros em tempo de execução são inevitáveis (e.g., entrada inválida, divisão por zero, arquivo ausente). O **tratamento de exceções**, via blocos `try/except`, permite que o programa **continue executando** ao invés de interromper. Ao prever e capturar erros específicos, o código reage de forma controlada, garantindo maior robustez e estabilidade da aplicação. Essencial para software confiável.

Um exemplo da estrutura básica é apresentado no bloco de código abaixo:

```
try:
    # Código que pode gerar erros
    numero = int(input("Digite um número: "))
    resultado = 10 / numero
    print("Resultado:", resultado)
except ZeroDivisionError:
    # Executado se ocorrer o erro específico
    print("Erro: Divisão por zero!")
except Exception as e:
    # Captura qualquer outro erro
    print(f"Erro inesperado: {e}")
else:
    # Executado se NENHUM erro ocorrer
    print("Operação bem-sucedida!")
finally:
    # Sempre executado (com ou sem erros)
    print("Fim do bloco try-except")
```

```
try:
    # Código que pode gerar erros
    numero = 0
    resultado = 10 / numero
    print("Resultado:", resultado)
except ZeroDivisionError:
    # Executado se ocorrer o erro específico
    print("Erro: Divisão por zero!")
except Exception as e:
```

```
    # Captura qualquer outro erro
    print(f"Erro inesperado: {e}")
else:
    # Executado se NENHUM erro ocorrer
    print("Operação bem-sucedida!")
finally:
    # Sempre executado (com ou sem erros)
    print("Fim do bloco try-except")
```

```
Erro: Divisão por zero!
Fim do bloco try-except
```

- O bloco `try` encapsula o código passível de gerar erros em tempo de execução.
- O bloco `except` especifica o tratamento para tipos particulares de erros (exceções) que possam ocorrer dentro do bloco `try`. Múltiplos blocos `except` podem ser definidos para lidar com diferentes tipos de exceções. A captura genérica de qualquer exceção pode ser realizada com `except Exception`.
- O bloco `finally` é executado invariavelmente, independentemente da ocorrência ou não de uma exceção dentro do bloco `try`. Sua principal aplicação reside na execução de rotinas de finalização, como o fechamento de arquivos ou a liberação de conexões.

4.4.2 Palavras-Chave

Dentro de laços de repetição `for` e `while` em Python, as palavras-chave `break` e `continue` oferecem mecanismos essenciais para o controle preciso do fluxo de execução. `break` interrompe imediatamente a execução do loop, transferindo o controle para a instrução seguinte ao bloco do loop. `continue`, por sua vez, encerra a iteração corrente e passa para a próxima iteração do loop. A utilização estratégica dessas palavras-chave contribui para a organização e a eficiência do código em cenários onde a execução padrão do loop necessita ser alterada condicionalmente.

---

break

```
# Encontra o primeiro número divisível por 7
for num in range(1, 10):
    if num % 7 == 0:
        print(f"Encontrado: {num}")
        break
    print(num)
```

```
1
2
3
4
5
6
Encontrado: 7
```

```python
while True:
    senha = input("Digite a senha (1234) para sair: ")
    if senha == "1234":
        print("Senha correta! Saindo...")
        break
    print("Senha incorreta!")
```

---

continue

```python
# Não irá imprimir o número 7
for num in range(1, 10):
    if num == 7:
        continue
    print(num)
```

```
1
2
3
4
5
6
8
9
```

```python
produtos = ["camiseta", "caneca", None, "poster", "", "adesivo"]
for item in produtos:
    if not item:  # None ou string vazia
        continue
    print(f"Processando: {item.upper()}")
```

```
Processando: CAMISETA
Processando: CANECA
Processando: POSTER
Processando: ADESIVO
```

4.5 Funções

Vamos relembrar o conceito de *função* dada por Guidorizzi (2001):

> "*Uma função f é uma relação entre dois conjuntos A e B, representada pela tripla $(A, B, a \mapsto b)$, onde A é o conjunto de partida (domínio), B é o conjunto de chegada (contradomínio) e $a \mapsto b$ é a regra que associa a cada elemento $a \in A$ um único elemento $b \in B$.*"

Para fins técnicos, as funções em Python seguem esse mesmo raciocínio. Uma função é um bloco de código que executa uma tarefa específica e pode ser reutilizado várias vezes ao longo do programa. Ao invés de repetir o mesmo conjunto de comandos, colocamos esses comandos dentro de uma função e chamamos essa função sempre que precisarmos dela. Pense nela como uma "minifábrica" que recebe certas entradas (opcionalmente), processa-as e produz uma saída (opcionalmente).

No universo da programação, a capacidade de organizar o código de forma organizad, modular e reutilizável é um pilar para a construção de sistemas eficientes e de fácil manutenção. Em Python, esse pilar é sustentado pelas funções. Nesta seção, exploraremos desde os conceitos básicos até aspectos mais avançados, com aplicações práticas.

Você pode pensar em uma função como uma "máquina" que recebe certos dados (chamados parâmetros ou argumentos), processa esses dados e, em muitos casos, devolve um resultado (valor de retorno).

A importância das funções é multifacetada, apresentando várias vantagens de fazer seu uso. Elas são fundamentais para:

- **Modularidade:** Funções permitem quebrar programas grandes e complexos em partes menores e mais gerenciáveis. Isso facilita o desenvolvimento, a depuração e a compreensão do código.
- **Reusabilidade de Código (DRY - Don't Repeat Yourself):** Uma vez definida, uma função pode ser chamada múltiplas vezes de diferentes partes do programa, eliminando a necessidade de reescrever o mesmo código repetidamente.
- **Abstração:** Funções escondem os detalhes de implementação de uma tarefa complexa. Você precisa saber o que a função faz (sua finalidade), mas não necessariamente como ela faz.
- **Manutenção:** Se uma alteração for necessária em uma lógica específica, basta modificar a função correspondente. Isso é muito mais eficiente do que procurar e alterar várias ocorrências do mesmo código espalhado pelo programa.
- **Legibilidade:** O uso de funções com nomes descritivos torna o código mais fácil de ler e entender, pois cada função representa uma etapa lógica clara no fluxo do programa.

4.5.1 Sintaxe Básica

Em Python, para definir uma função é utilizado a palavra chave `def`, seguida do nome da função e parênteses para os possíveis parâmetros e `:`. Ao pressionar *enter* o bloco será automaticamnete identado (*quatro espações ou um tab*) pelo editor. Todo código identado faz parte da função.

```
def name_function():
  # seu código será disposto aqui
  # e o Python irá compreende-lo como parte da função
```

- **Exemplos:**

```
def greeting1():
  print("Olá. Sejá bem-vindo ao Curso de Python para Análise de Dados!")


def greeting2(nome):
  print(f"Olá, {nome}. Sejá bem-vindo ao Curso de Python para Análise de Dados!")


greeting1()
```

```
Olá. Seja bem-vindo ao Curso de Python para Análise de Dados!
```

```
greeting2("Breno")
```

```
Olá, Breno. Seja bem-vindo ao Curso de Python para Análise de Dados!
```

4.5.2 Parâmetros e Argumentos

Funções podem receber dados de entrada para realizar suas tarefas. Esses dados são passados através de **parâmetros** na definição da função e **argumentos** na chamada da função.

4.5.3 Tipos de Parâmetros/Argumentos:

1. **Argumentos Posicionais:** São os argumentos passados na ordem em que os parâmetros são definidos.

```
def saudar(nome, sobrenome):
  print(f"Olá, {nome} {sobrenome}!")

saudar("Maria", "Silva") # Saída: Olá, Maria Silva!
```

```
Olá, Maria Silva!
```

```
saudar("Silva", "Maria") # Saída: Olá, Silva Maria!
```

```
Olá, Silva Maria!
```

2. **Argumentos de Palavra-chave (Keyword Arguments):** São passados explicitamente associando o nome do parâmetro ao seu valor. A ordem não importa.

```
def email(nome, dominio="icen.ufpa.br"):
    return f"{nome.lower()}@{dominio}"

print(email(nome="joao"))                              # Saída: joao@icen.ufpa.br
```

```
joao@icen.ufpa.br
```

```
print(email(dominio="ufpa.br", nome="ANA")) # Saída: ana@ufpa.br
```

```
ana@ufpa.br
```

3. **Parâmetros com Valores Padrão (Default Parameters):** Permitem definir um valor padrão para um parâmetro. Se o argumento não for fornecido na chamada da função, o valor padrão é utilizado.

```
def calcular_potencia(base, expoente=2): # expoente tem valor padrão 2
    return base ** expoente

print(calcular_potencia(3))    # Saída: 9 (3^2)
```

```
9
```

```
print(calcular_potencia(2, 4)) # Saída: 16 (2^4)
```

```
16
```

Cuidado: Parâmetros com valores padrão devem ser definidos após os parâmetros sem valores padrão.

4. **Argumentos Arbitrários (`*args` e `**kwargs`):** Permitem que uma função aceite um número variável de argumentos.

- `*args` (non-keyword arguments): Coleta um número variável de argumentos posicionais em uma tupla.

```
def soma_tudo(*numeros):
    total = 0
    for num in numeros:
        total += num
    return total

print(soma_tudo(1, 2, 3))        # Saída: 6
```

```
6
```

```
print(soma_tudo(10, 20, 30, 40)) # Saída: 100
```

```
100
```

- `**kwargs` (keyword arguments): Coleta um número variável de argumentos de palavra-chave em um dicionário.

```
def exibir_perfil(**info):
  for chave, valor in info.items():
      print(f"{chave.replace("_", " ").title()}: {valor}")

exibir_perfil(nome="Carlos", idade=30, cidade="São Paulo")
```

```
Nome: Carlos
Idade: 30
Cidade: São Paulo
```

```
# Saída:
# Nome: Carlos
# Idade: 30
# Cidade: São Paulo

exibir_perfil(nome_cliente="Carlos", idade_cliente=30, cidade_cliente="São Paulo")
```

```
Nome Cliente: Carlos
Idade Cliente: 30
Cidade Cliente: São Paulo
```

```
# Saída:
# Nome Cliente: Carlos
# Idade Cliente: 30
# Cidade Cliente: São Paulo
```

4.5.4 Retorno de Valores

Vimos que é possível imprimir valores na tela através de funções. Entretanto e se precisarmos dos valores que são calculados por determinada função? De acordo com o que apreendemos basta atribuir a saída do programa a uma variável.

```
# Atribuição
grr = greeting2("Emilly Rose")
```

```
Olá, Emilly Rose. Sejá bem-vindo ao Curso de Python para Análise de Dados!
```

Também de acordo com o que apreendemos, basta chamar a variável `grr` em qualquer parte do código que ela mostrará a frase: `Olá, Emilly Rose. Sejá bem-vindo ao Curso de Python para Análise de Dados!`.

```
# Chamando variável
grr
```

Note que a saída do código acima, não resultou em nada. Ou ainda, pode ter resultado, porém, o resutado foi vazio (`None`).

```
# Verificação
grr == None
```

```
True
```

Note que, por consequência, valores definidos dentro de uma função não são exportados para fora dela. Isto é, algo definido no `local` de um função não será definido no ambiente `global`.

```
def minha_funcao():
      arr = 10  # Variável local
      print(arr)

minha_funcao()
```

```
10
```

```
try:
  print(arr)  # Erro! 'arr' só existe dentro da função.

except NameError:
  print("Variável 'arr' não definida!")
```

```
Variável 'arr' não definida!
```

Isso ocorre porque a estrutura que está sendo utilizada pela função `greeting2` não foi feita para retornar valores, mas sim, para apenas os imprimir na tela.

- **Boas Práticas de Escopo:**
    1. *Priorize o Escopo Local:* É uma boa prática projetar funções para serem o mais independentes possível, evitando depender de variáveis globais. Isso melhora a modularidade e reduz efeitos colaterais indesejados.
    2. *Evite o Uso de `global`:* A palavra-chave `global` permite modificar uma variável global dentro de uma função. Seu uso deve ser evitado ao máximo, pois pode levar a um código confuso e difícil de depurar. Se uma função precisa alterar dados globais, é preferível que ela retorne os novos valores para que o código chamador possa atualizar a variável global explicitamente.
    3. *Passagem de Argumentos:* Sempre que uma função precisar de dados externos, passe-os como argumentos, em vez de depender de variáveis globais.

Para que a função realmente retorne o valor que foi obtido/calculado por ela, basta utilizar um comando interno do Python chamado `return`. Vamos a exemplos:

1. Soma de dois termos:

```
def my_function_sum1(a, b):
  summ = a + b

  return summ

def my_function_sum2(a, b):
  return a + b


a = my_function_sum1(1/2, 1/2)
b = my_function_sum2(1/2, 1/2)

print(a == b)
```

```
True
```

2. Soma de n termos:

```
def my_function_sum(*numbers):
  summ = 0
  for num in numbers:
    if type(num) == list:
      for n in num:
        if (type(n) != int) or (type(n) != float):
          continue

        summ += n
    else:
        summ += num
```

```
  return summ

my_function_sum([1, 2, "a", 3], 5, 10, 20, 32)
```

67

Note que, se eu quiser o resultados desta soma para usá-lo em outra hora, basta atribuir isso a uma variável.

```
# Guardando soma
summ = my_function_sum([1, 2, "a", 3], 5, 10, 20, 32)

# Visualizando
print(summ)
```

67

```
# Verificando
print(summ == None) # A saída deve ser: False
```

```
False
```

4.5.5 Funções Anônimas - `lambda functions`

Funções lambda são pequenas funções anônimas (sem nome) que são definidas usando a palavra-chave `lambda`. Elas são restritas a uma única expressão e são frequentemente usadas para tarefas curtas e simples onde uma função completa seria excessiva.

- **Sintaxe:** `lambda argumentos: expressão`

```
# Definindo função anônima
square = lambda x: x ** 2

# Fazendo uso da função
print(square(2))
```

4

```
# Definindo função anônima
linear = lambda x, y: 2 * x + y

# Fazendo uso da função
print(linear(2, 3))
```

7

```
# Definindo função anônima
tt = lambda x: x * 2 + x

# Fazendo uso da função
print(tt([1, 2, 3]))
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Essa maniera de se definir uma função é comumente usada em conjunto das funções `sorted()`, `filter()` e `map()`.

```
list_of_numbers = [2.5, 1, 5, 9, 0.1, 2, 8, 3, 12]

# Exemplo de uso com sorted()
print(sorted(list_of_numbers, key = lambda x: 1 / x ** 2))
```

```
[12, 9, 8, 5, 3, 2.5, 2, 1, 0.1]
```

```
# Exemplo de uso com filter()
print(list(filter(lambda x: x % 2 == 0, list_of_numbers)))
```

```
[2, 8, 12]
```

```
# Exemplo de um com map()
print(list(map(lambda x: x * 2, list_of_numbers)))
```

```
[5.0, 2, 10, 18, 0.2, 4, 16, 6, 24]
```

Referêcias

Guidorizzi, Hamilton Luiz. 2001. *Um Curso de Cálculo, Volume 1.* 5ª ed. Rio de Janeiro: LTC.