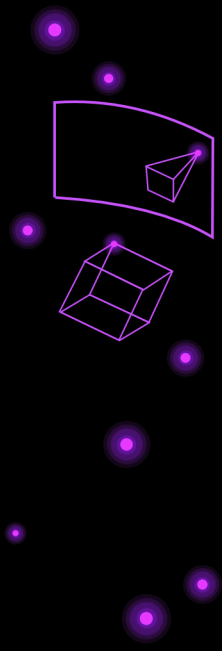
CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS

Introdução ao *Python*

Autoria:
Rafael Divino Ferreira Feitosa



Organizadores:
Renata Dutra Braga
Deborah Silva Alves Fernandes
Taciana Novo Kudo
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG

Introdução ao *Python*

Autoria:
Rafael Divino Ferreira Feitosa

Organizadores:
Renata Dutra Braga
Deborah Silva Alves Fernandes
Taciana Novo Kudo
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG

2024

Revisão Técnica
Igor Gabriel Silva Batista
Cristiane Bastos Rocha Ferreira
Deborah Silva Alves Fernandes

Revisão Editorial
Ana Laura de Sene Amâncio Zara Brisolla

Capa
Iuri Vaz Miranda

Editoração Eletrônica
Luma Wanderley de Oliveira
Layane Grazielle Souza Dias

https://doi.org/10.5216/FEI.int.ebook.978-85-495-0977-2

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Feitosa, Rafael Divino Ferreira
Introdução ao Python [livro eletrônico] / Rafael Divino Ferreira Feitosa ; organizadores Renata Dutra Braga...[et al.]. -- 1. ed. -- Goiânia, GO : Cegraf UFG, 2024.
PDF

Outros organizadores: Deborah Silva Alves Fernandes, Taciana Novo Kudo, Cristiane Bastos Rocha Ferreira, Arlindo Rodrigues Galvão Filho.
Bibliografia.
ISBN 978-85-495-0977-2

1. Ciência da computação 2. Inteligência artificial - Inovações tecnológicas 3. Python (Linguagem de programação para computadores) 4. Tecnologia I. Braga, Renata Dutra. II. Fernandes, Deborah Silva Alves. III. Kudo, Taciana Novo. IV. Ferreira, Cristiane Bastos Rocha. V. Galvão Filho, Arlindo Rodrigues. VI. Título.

24-223323 CDD-005.133

Índices para catálogo sistemático:

1. Python : Linguagem de programação : Computadores : Processamento de dados 005.133

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Introdução ao *Python*

Instituições responsáveis

Universidade Federal de Goiás (UFG)

Centro de Competência Embrapii em Tecnologias Imersivas, denominado AKCIT (Advanced Knowledge Center for Immersive Technologies)

Centro de Excelência em Inteligência Artificial (CEIA)

Instituições financiadoras

Empresa Brasileira de Pesquisa e Inovação Industrial (Embrapii)

Governo do Estado de Goiás

Empresas parceiras do AKCIT

Apoio

Universidade Federal de Goiás (UFG)

Pró-Reitoria de Pesquisa e Inovação (PRPI-UFG)

Instituto de Informática (INF-UFG)

Lista de Abreviaturas e Siglas

API	*Application Programming Interface* - Interface de Programação de Aplicações
Embrapii	Empresa Brasileira de Pesquisa e Inovação Industrial
GPU	*Graphics Processing Unit* - Unidade de Processamento Gráfico
IA	Inteligência Artificial
ID	Identificador
IDEs	*Integrated Development Environments* - Ambientes de Desenvolvimento Integrados
PSF	*Python Software Foundation* - Fundação *Python Software*
TPU	*Tensor Processing Unit* - Unidade de Processamento Tensor
UFG	Universidade Federal de Goiás

Lista de Figuras

Sumário

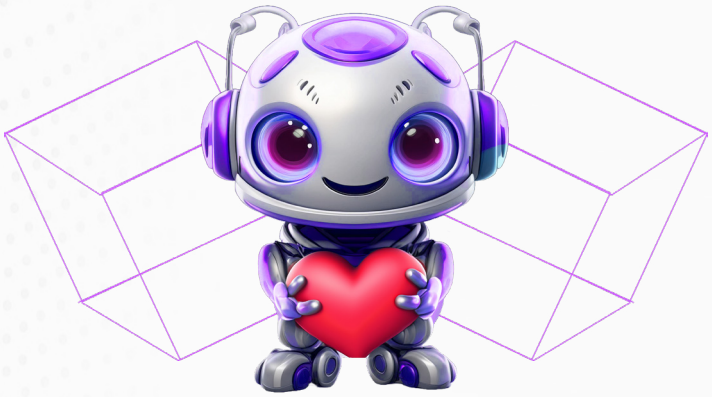
Apresentação

Prezado(a) Participante,

Seja bem-vindo(a) ao Microcurso de **Introdução ao Python**.

Esse Microcurso faz parte da Coleção Formação e Capacitação do Centro de Competências Imersivas, uma parceria entre a Empresa Brasileira de Pesquisa e Inovação Industrial (Embrapii) e a Universidade Federal de Goiás (UFG). A sua oferta foi motivada pela crescente demanda por habilidades em programação e análise de dados, essenciais para diversas áreas do conhecimento.

Neste Microcurso, abordaremos desde os conceitos básicos da linguagem Python, como sua sintaxe e estruturas de dados, até ferramentas e ambientes de desenvolvimento amplamente utilizados. Exploraremos também as estruturas de controle de fluxo, funções e a manipulação de arquivos, proporcionando uma base sólida para a aplicação prática do Python em diversos domínios.



Desejamos um excelente estudo!!!

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Introdução ao *Python*

Unidade I
A linguagem *Python* e os ambientes de desenvolvimento

Unidade I - A Linguagem Python e os Ambientes de Desenvolvimento

1.1 A Linguagem Python

Python é uma das linguagens de programação mais populares e amplamente utilizadas no mundo da ciência de dados e da inteligência artificial (IA). A adoção massiva de Python nesses campos não é por acaso. Há diversas razões pelas quais Python se tornou a escolha preferida para análise de dados e desenvolvimento de modelos de IA. Este texto visa explorar essas razões, abordando desde a facilidade de aprendizado até o ecossistema robusto de bibliotecas e a grande comunidade de suporte.

1.1.1 Facilidade de Aprendizado

Uma das principais razões para a popularidade do Python é sua simplicidade e legibilidade. A sintaxe clara e intuitiva de Python permite que novos programadores aprendam rapidamente a linguagem e comecem a trabalhar em projetos significativos em um curto período de tempo. Para programadores experientes em outras linguagens, a transição para Python é “suave” graças à sua sintaxe simples e expressiva.

1.1.2 Ecossistema Robusto de Bibliotecas

O Python possui um ecossistema robusto de bibliotecas e *frameworks* que facilitam o desenvolvimento de soluções de análise de dados e IA. Algumas das bibliotecas mais populares incluem: NumPy, Pandas, Matplotlib, Seaborn, Scikit-learn, TensorFlow e PyTorch. Na Figura 1, essas bibliotecas mais populares são ilustradas.

Figura 1 - Bibliotecas populares do Python

Fonte: autoria própria.

1.1.3 Grande Comunidade de Suporte

O Python possui uma comunidade ativa e crescente. Isso significa que, ao aprender e trabalhar com Python, você tem acesso a um vasto número de recursos de suporte, como tutoriais, fóruns de discussão e projetos de código aberto. Essa comunidade ativa não apenas facilita a resolução de problemas, mas também impulsiona o desenvolvimento contínuo e a inovação dentro do ecossistema de Python.

1.1.4 Fácil Integração com Outras Linguagens e Ferramentas

O Python é conhecido por sua grande capacidade de integração com outras linguagens de programação e ferramentas. Isso é particularmente útil em projetos de análise de dados e IA, onde é comum precisar interagir com sistemas e *softwares* escritos em outras linguagens, como C, C++, Java e R. Além disso, Python pode ser facilmente integrado com diversas ferramentas de *big data*, como Hadoop e Spark.

1.1.5 Suporte para Computação Científica e Estatística

A capacidade do Python de lidar com computação científica e estatística é um dos principais motivos de sua popularidade na análise de dados. Bibliotecas como SciPy e StatsModels proporcionam ferramentas avançadas para análise estatística, resolução de problemas científicos e engenharia de dados, permitindo que os analistas e cientistas de dados desenvolvam soluções robustas e precisas.

1.1.6 Escalabilidade e Desempenho

Embora Python seja conhecido por sua simplicidade, ele também é altamente escalável. *Frameworks*, como Dask, permitem que você processe grandes volumes de dados de maneira eficiente. Além disso, a integração de Python com bibliotecas escritas em C e C++ permite que ele atinja altos níveis de desempenho, adequando-se a aplicações de larga escala e de alta demanda computacional.

1.1.7 Aplicações em Diversos Domínios

Python é uma linguagem versátil que é aplicada em uma vasta gama de domínios. Na Figura 2, podemos visualizar alguns desses domínios e suas respectivas aplicações.

Figura 2 - Exemplos de áreas e aplicações para Python

Fonte: autoria própria.

1.2 Histórico e Expansão do Python

O Python foi criado no final dos anos 1980 por Guido van Rossum, sendo a primeira versão lançada em 1991. Desde então, tem evoluído para se tornar uma das linguagens de programação mais populares no mundo. Sua adoção na comunidade científica, em particular, tem sido significativa, impulsionada por sua simplicidade, versatilidade e a capacidade de facilitar a colaboração por meio de um ecossistema robusto de bibliotecas.

1.2.1 Surgimento do Python

A primeira versão do Python foi criada como uma sucessora da linguagem ABC, visando oferecer uma linguagem que fosse, ao mesmo tempo, poderosa e fácil de aprender. A versão 1.0 com grandes inovações e recursos expandidos foi lançada em 1994. A filosofia de *design* do Python enfatiza a legibilidade do código e a simplicidade, características

que o tornam ideal tanto para iniciantes quanto para programadores experientes. Desde seu início, Python foi projetado para ser uma linguagem de propósito geral, capaz de ser utilizada em uma ampla variedade de aplicações.

1.2.2 Popularização na Comunidade Científica

A popularização do Python na comunidade científica começou a ganhar força no início dos anos 2000. A comunidade Python é conhecida por seu forte engajamento no desenvolvimento contínuo da linguagem e de suas bibliotecas. Esse engajamento se manifesta de várias maneiras:

- **Contribuições abertas**: Python é uma linguagem de código aberto, e sua comunidade é altamente colaborativa. Desenvolvedores ao redor do mundo contribuem para o desenvolvimento de novas funcionalidades e para a melhoria das existentes. Essa colaboração ocorre principalmente por meio de plataformas, como GitHub, onde projetos de bibliotecas são mantidos.
- **Conferências e eventos**: conferências como PyCon, SciPy e JupyterCon reúnem milhares de desenvolvedores, pesquisadores e entusiastas de Python todos os anos. Esses eventos são fundamentais para a troca de ideias, apresentação de novas bibliotecas e discussão de tendências emergentes na comunidade.
- **Organizações e grupos de usuários**: existem diversas organizações e grupos de usuários dedicados ao desenvolvimento e promoção do Python. A *Python Software Foundation* (PSF), por exemplo, apoia o desenvolvimento da linguagem e organiza eventos que fomentam a colaboração e o aprendizado.
- **Documentação e tutoriais:** a comunidade Python valoriza a criação e a manutenção de uma documentação abrangente e acessível. Muitos desenvolvedores contribuem com tutoriais, exemplos de código e guias detalhados que ajudam novos usuários a aprenderem Python e a utilizarem suas bibliotecas de forma eficaz.

1.3 Ferramentas e Ambientes de Desenvolvimento com Python

O desenvolvimento de projetos de ciência de dados e algoritmos de IA em Python exige um conjunto de ferramentas e ambientes adequados para garantir eficiência, organização e escalabilidade. Desde ambientes interativos a ambientes de desenvolvimento em programação (*Integrated Development Environments* [IDEs]) e ferramentas específicas para análise de dados, o ecossistema Python oferece uma variedade de opções para atender às necessidades dos cientistas de dados e engenheiros de IA.

Os IDEs são ferramentas que facilitam a escrita, o teste e a depuração de código por parte dos desenvolvedores. Eles combinam várias funcionalidades em um único aplicativo para tornar o processo de desenvolvimento mais eficiente e organizado.

Para facilitar a escrita de códigos e otimizar o processo de desenvolvimento, as IDEs precisam ter um conjunto básico de funcionalidades que incluem:

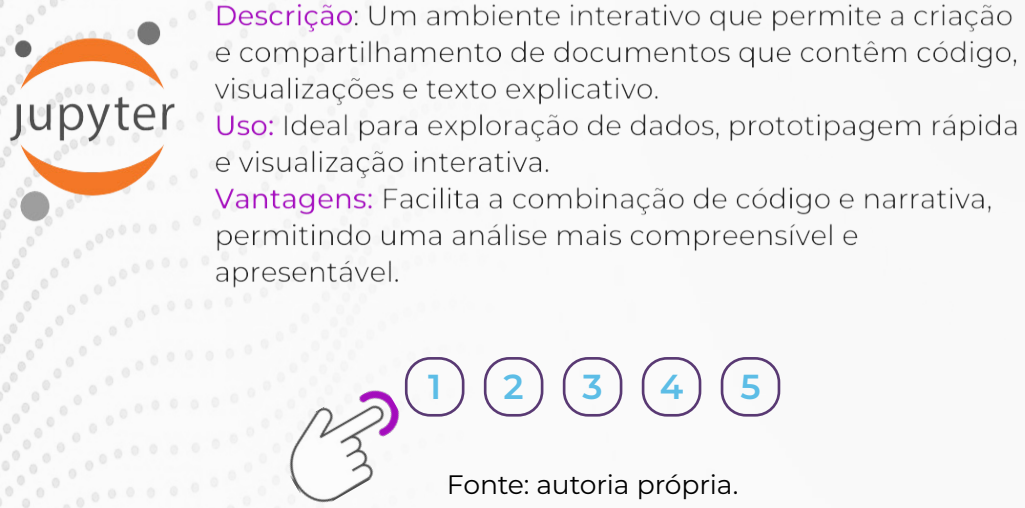
- **Editor de código:** área da IDE onde os desenvolvedores escrevem e editam seu código-fonte.
- **Compilador/interpretador:** ferramentas que traduzem o código-fonte em linguagem de máquina para execução.
- **Depurador:** ferramentas para encontrar e corrigir erros no código.
- **Gerenciamento de projeto:** ferramentas para organizar arquivos e diretórios do projeto.
- **Ferramentas de autocompletar e de integração com a documentação da linguagem:** ajudam a acelerar a escrita de código e a reduzir erros.
- **Controle de versão:** suporte ao gerenciamento de mudanças no código e colaboração com outros desenvolvedores.

Algumas características desejáveis podem ser um diferencial para acelerar a produtividade:

- **Facilidade de uso**: a interface deve ser intuitiva e fácil de navegar, permitindo que os desenvolvedores se concentrem no código e não na ferramenta.
- **Suporte a múltiplas linguagens**: muitos IDEs suportam várias linguagens de programação, o que é útil para desenvolvedores que trabalham com diferentes tecnologias.
- **Personalização**: a capacidade de personalizar o IDE para atender às necessidades específicas do desenvolvedor, como temas, atalhos de teclado e *plugins*.
- **Desempenho**: o IDE deve ser rápido e responsivo, mesmo ao lidar com grandes projetos.
- **Documentação e comunidade de suporte**: documentação abrangente e uma comunidade ativa são importantes para resolver dúvidas e problemas que possam surgir.
- **Ferramentas de teste**: integração com *frameworks* de teste para facilitar o desenvolvimento orientado a testes.
- **Refatoração de código**: ferramentas que ajudam a melhorar e a reestruturar o código, sem alterar seu comportamento externo.
- **Colaboração**: funcionalidades que facilitam a colaboração entre equipes de desenvolvimento, como compartilhamento de sessões de codificação.

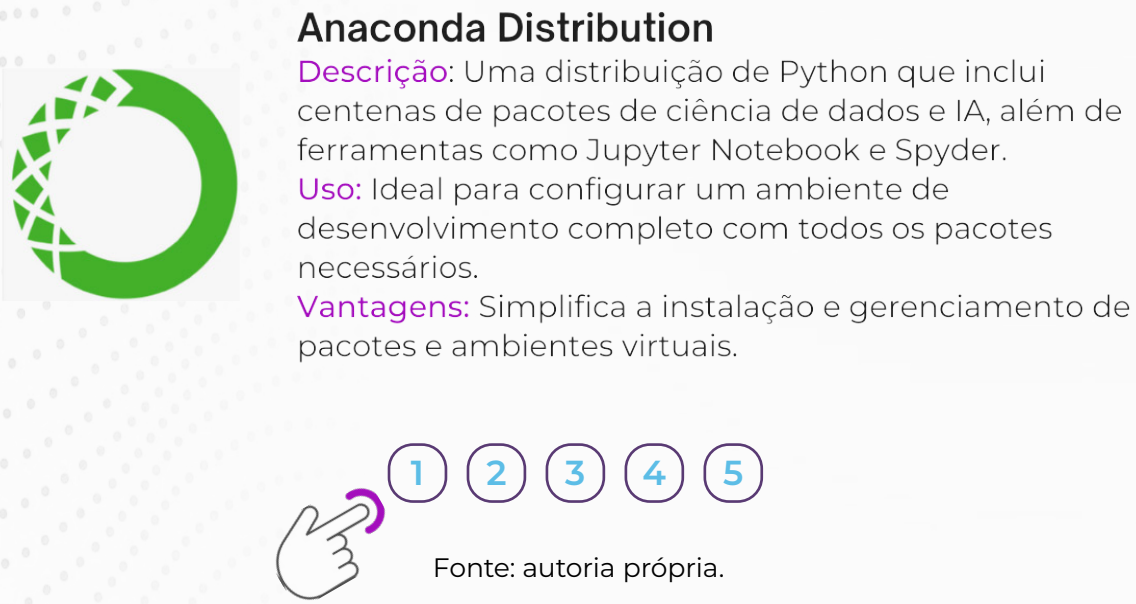
Vamos apresentar alguns ambientes de desenvolvimento mais populares e suas aplicações (Figura 3) e ferramentas específicas para ciência de dados e IA (Figura 4).

Figura 3 - Ambientes de desenvolvimento populares em Python e algumas características



Fonte: autoria própria.

Figura 4 - Ferramentas específicas para ciência de dados e inteligência artificial



Fonte: autoria própria.

1.4 Saiba Mais...

- O *site* oficial da linguagem **Python**, oferece documentação detalhada, tutoriais e recursos para desenvolvedores de todos os níveis.
- O portal oficial do ecossistema **SciPy**, com documentação, tutoriais e recursos sobre computação científica em Python.
- O *site* oficial dos **Jupyter Notebooks**, contendo informações sobre instalação, uso e integração com outras ferramentas de ciência de dados.
- Um guia introdutório para usar o **Google Colab**, uma plataforma baseada em nuvem para *notebooks* Jupyter.
- A página oficial da **IDE PyCharm**, com informações sobre recursos, *download* e tutoriais.
- Documentação oficial para uso de Python no **Visual Studio Code**, incluindo instalação de extensões e configuração do ambiente.
- Página oficial da **Anaconda**, uma distribuição popular que inclui centenas de pacotes para ciência de dados e *machine learning*.
- Guia completo para começar a usar o **TensorFlow**, uma das bibliotecas mais populares para *deep learning*.
- Documentação oficial do **PyTorch**, fornecendo tutoriais, exemplos de código e guias de referência.
- A documentação oficial do **Scikit-learn**, uma biblioteca essencial para *machine learning* em Python.

Para relembrar...

- Aprender Python para análise de dados e desenvolvimento de modelos de IA é uma decisão estratégica para qualquer programador. Com sua sintaxe simples, ecossistema robusto de bibliotecas, grande comunidade de suporte e versatilidade, Python oferece uma plataforma poderosa para abordar desafios complexos e inovar em diversos campos. Se você está buscando expandir suas habilidades e se posicionar na vanguarda da tecnologia, Python é, sem dúvida, uma escolha excelente.
- O surgimento do Python e sua popularização na comunidade científica são resultados de sua simplicidade, versatilidade e do engajamento ativo de sua comunidade. O desenvolvimento contínuo de novas bibliotecas e ferramentas por parte de uma comunidade global de desenvolvedores tem sido fundamental para manter o Python na vanguarda da ciência e da tecnologia. À medida que mais pesquisadores e cientistas adotam Python, a linguagem continuará a evoluir, impulsionada pela colaboração e inovação constantes de sua vibrante comunidade.
- O desenvolvimento de projetos de ciência de dados e algoritmos de IA em Python é facilitado por uma ampla gama de ambientes, IDEs e ferramentas especializadas. Desde ambientes interativos, como Jupyter Notebook e Google Colab, até IDEs completas, como PyCharm e VS Code, as opções disponíveis atendem às diversas necessidades dos desenvolvedores. Ferramentas específicas como TensorFlow, PyTorch e Scikit-learn, combinadas com distribuições como Anaconda, tornam Python uma escolha poderosa e versátil para cientistas de dados e engenheiros de IA. A seleção da ferramenta certa depende das necessidades específicas do projeto, bem como das preferências pessoais do desenvolvedor.

Introdução ao *Python*

Unidade II

Sintaxe Básica do *Python*

Unidade II - Sintaxe Básica do Python

Esta unidade também está disponível no **Google Colab**..

2.1 Notebook Colab: Sintaxe Básica do Python

Este *notebook* aborda a sintaxe básica do Python, incluindo comandos de entrada e saída, comentários, variáveis, tipagem de dados, *casting*, operadores aritméticos, operadores relacionais e lógicos, concatenação de *strings*, *strings* como lista de caracteres, método format, f-strings, comandos Python dentro de *strings*, objeto None e principais funções built-in.

Nas próximas Unidades, discutiremos outros tópicos essenciais para compreender a linguagem Python e os recursos que facilitam a implementação de projetos. Para saber mais, acesse a documentação da versão mais recente da linguagem em https://docs.python.org/.

2.1.1 Comandos Gerais

Codificação guiada

O Python é uma linguagem que, a cada ano, vem ganhando mais adeptos e ampliando suas funcionalidades. Essas funcionalidades, tanto padrão quanto as estendidas por meio de bibliotecas de terceiros, são bem documentadas e nos auxiliam na escrita e entendimento dos recursos oferecidos por cada uma delas.

O Colab possui funcionalidades de autocompletar código e mostrar a documentação da função, similares a outras IDEs.

Para ativar esse recurso:

- Acesse o menu **Ferramentas**, selecione **Configurações**, clique na aba **Editor** e marque a opção **Acionar preenchimento de código automaticamente** se ela estiver desmarcada;
- No canto superior direito da tela principal, clique em **Conectar** ou **Reconectar** para ativar o ambiente de execução.

Auto completar o código

Antes de testar a função de auto completar, certifique-se de clicar em **Conectar** no canto superior direto do Colab. Isso fará com que o ambiente de execução seja iniciado e o *notebook* tenha acesso a todos os recursos.

Utilize **Ctrl + Espaço** para abrir o *menu* de autocompletar enquanto digita. Será exibida uma lista de comandos; continue digitando ou selecione o comando desejado e aperte **Enter** ou clique sobre o comando.

O código abaixo está incompleto intencionalmente para que você explore o recurso de autocompletar. Escreva apenas **prin** e posicione o cursor após a letra i, por exemplo, **pri|**, e aperte **Ctrl + Espaço**. Em seguida escolha a opção **print** (clique sobre ou use a tecla direcional para selecionar e aperte **Enter**) e complete o código com **print('Mensagem')**. Em seguida, clique no ícone ▶ para executar a célula de código

```
pri
```

Mostrar o *help* das funções

A documentação das funções e comandos podem ser acessados de diversas formas:

» *Docstring*: ao abrir os parênteses de uma função, abrirá uma janela *pop-up* com a *docstring* da função, fornecendo informações sobre seus parâmetros e uso.

O código abaixo também está incompleto intencionalmente para que você explore este recurso. Escreva apenas **math.cos** na segunda linha de código e posicione o cursor após a letra s, por exemplo, **math.cos|**, e acrescente um parêntese. O Colab autocompletará o outro parêntese e abrirá um pop-up com a help da função.

```
import math

math.cos
```

» Função ***help()***: Use a função ***help()*** para exibir a documentação completa de uma função ou módulo.

```
help(print)
```

Help on built-in function print in module builtins:

print(...)

print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

» Interrogação: coloque um ponto de interrogação antes ou depois do nome da função para exibir sua documentação. As informações serão exibidas no lado direito da tela em abas separadas.

```
print?
```

```
?print
```

Indentação

O Python é famoso por não ter o sinal de ponto-e-vírgula (;) ao final das linhas de comando como nas demais linguagens. O interpretador Python "entende" a hierarquia do código, bem como o início e fim de blocos de código, baseado na indentação.

É fundamental que os códigos escritos respeitem a indentação, observando, inclusive, que a presença de um único espaço em branco indevido no início da linha irá resultar em um ***IndentationError***, como nos exemplos a seguir.

```
tempo = 'ensolarado'
if (tempo == 'ensolarado'):
  print('Dia perfeito para um passeio!')
# Será exibido 'Dia perfeito para um passeio!'
```

Dia perfeito para um passeio!

O código acima será executado sem erros.

```
tempo = 'ensolarado'
  if (tempo == 'ensolarado'): # Um espaço em branco inserido no
início da linha resulta no erro 'IndentationError: unexpected indent'
  print('Dia perfeito para um passeio!')
```

Dia perfeito para um passeio!

O código acima acusará um ***IndentationError*** em razão de apenas um espaço extra no início da segunda linha.

O ***IndentationError*** ocorre quando a indentação do código está incorreta. Para corrigi-lo:

- Certifique-se de que todos os blocos de código estão devidamente indentados.
- Use um editor de código que destaque a indentação e ajude a mantê-la consistente.

Perceba que o alinhamento dos códigos é realizado com base no espaçamento a partir do lado esquerdo do editor. Isso significa que códigos mais avançados para a direita indicam que esses pertencem (estão dentro) do bloco de código anterior (linha anterior); ao passo que linhas de código que estão no mesmo bloco se iniciam na mesma coluna. No exemplo anterior, **tempo...** e ***if...*** pertencem ao mesmo bloco de código, pois estão no mesmo alinhamento e ***print(...*** pertence ao ***if...*** está avançado em relação ao recuo esquerdo.

Comentários

Comentários são usados para explicar o código e são ignorados pelo interpretador Python. Comentários de uma linha começam com **#** e comentários de múltiplas linhas são delimitados por **‘’’** ou **“””**.

```
# Este é um comentário de uma linha

'''
Este é um comentário
de múltiplas linhas
'''

"""
Este também é um comentário
de múltiplas linhas
"""
```

'\nEste também é um comentário de múltiplas linhas\n'

Comentários de múltiplas linhas são úteis para documentar o código.

2.1.2 Comandos de Entrada e Saída

Em Python, usamos a função ***input()*** para entrada de dados e ***print()*** para saída de dados.

```
# Exemplo de comando de entrada e saída
nome = input("Digite seu nome: ")  # Solicita que o usuário digite seu nome
print(nome)  # Exibe o nome digitado pelo usuário
```

Digite seu nome: Rafael

Rafael

A função ***input()*** em Python é usada para exibir uma mensagem na tela e coletar o valor digitado pelo usuário. Esse valor é retornado como uma *string* e pode ser atribuído a uma variável para uso posterior no programa. Essa função executa, ao mesmo tempo, os comandos de entrada e saída.

Usamos o operador de atribuição **=** para receber o valor informado pelo usuário e armazená-lo na variável **nome**. O comando de atribuição é utilizado para armazenar valores em variáveis. A atribuição usando o operador de igual **=** atribui o valor do lado direito à variável do lado esquerdo.

Variáveis são espaços de memória usados para armazenar valores digitados pelo usuário (por meio do uso do comando de entrada **input()**, juntamente com o comando de atribuição **=**) ou processados pelo algoritmo (por meio do uso do comando de atribuição **=**).

No Google Colab, o **print()** não é obrigatório para mostrar um texto ou o valor de uma variável. Entretanto, ao não utilizá-lo, será mostrada apenas a última saída, caso haja mais de uma.

O comando **print()** aceita tanto aspas (“) quanto apóstrofos (‘) para receber o texto a ser mostrado na tela, embora o apóstrofo seja o mais recomendado.

```
# Exibe 2 mensagens de boas-vindas escritas em duas linhas
print("Olá")
print("Bem-vindo ao curso de Python!")
```

Olá

Bem-vindo ao curso de Python!

```
"Olá"
"Bem-vindo ao curso de Python!" # Exibe apenas a última saída
```

Bem-vindo ao curso de Python!

2.1.3 Variáveis

Em Python, você não precisa declarar o tipo da variável, pois o Python faz isso automaticamente.

```
# Exemplo de variáveis
idade = 25  # Variável inteira
nome = 'João'  # Variável string
altura = 1.75  # Variável float
```

O Python também permite a atribuição simultânea de mais de um valor para mais de uma variável.

Para exibir textos e variáveis utilizando o comando print() são utilizadas vírgulas (,) para separá-los.

```
print(nome, 'tem', idade, 'anos e', altura, 'm de altura.')  # Exibe as informações armazenadas nas variáveis declaradas acima
```

João tem 25 anos e 1.75 m de altura.

Tipagem de dados e *casting*

Python é uma linguagem de programação de alto nível que suporta diversos tipos de dados. Conhecer esses tipos é essencial para manipular e armazenar dados de maneira eficiente. Vamos explorar os principais tipos de dados em Python, incluindo **int**, **float**, **str**, **bool**:

- **int** (inteiro):
  - Números inteiros, positivos ou negativos, sem parte decimal.
- **float** (ponto flutuante):
  - Números reais com parte decimal.
- **str** (*string*):
  - Sequência de caracteres, usada para representar texto.
- **bool** (booleano):
  - Representa valores verdadeiros (*True*) ou falsos (*False*).

```
numero = 10      # Inteiro
decimal = 3.14   # Float
texto = "Olá"    # String
verdadeiro = True  # Booleano

print(type(numero)) # Será exibido <class 'int'>
print(type(decimal)) # Será exibido <class 'float'>
print(type(texto)) # Será exibido <class 'str'>
print(type(verdadeiro)) # Será exibido <class 'bool'>
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

A função ***type()*** serve para mostrar o tipo da variável e será discutida mais adiante.

Podemos converter entre diferentes tipos de dados usando funções de *casting*. Elas consistem em passar a variável a ser convertida entre parênteses para uma função com o mesmo nome do tipo desejado. Operações de *casting* realizam a troca do tipo de dado armazenado na memória. Isso é útil quando precisamos realizar operações entre tipos diferentes de dados, como a concatenação de *strings* que será abordada adiante.

Por exemplo, se temos uma variável **a** que contém um valor real, e desejamos converter esse valor e armazená-lo em uma outra variável **b**, que é do tipo inteiro, usando o *casting* teremos **b = int(a)**. Nesse caso, o valor da variável **a** atribuído à **b** terá a parte fracionária suprimida antes dessa atribuição.

```
# Convertendo string para inteiro
numero_str = "123"
numero_int = int(numero_str)

# Convertendo inteiro para string
numero_str_convertido = str(numero_int)

print(numero_int, type(numero_int)) # Será exibido 123 <class 'int'>
print(numero_str_convertido, type(numero_str_convertido))
# Será exibido 123 <class 'str'>
```

```
123 <class 'int'>
123 <class 'str'>
```

Também podemos alterar o comportamento do comando **print()** entre os parâmetros passados para exibição.

```
nota1, nota2, nota3 = 5, 7.8, 8.1
print(nota1, nota2, nota3, sep='\n')
```

```
5
7.8
8.1
```

No exemplo acima, também apresentamos o parâmetro **sep** para informar ao **print()** o comportamento da saída para cada valor separado por vírgula. Nesse caso, entre o **print** de cada nota é acrescentada uma quebra de linha representada pelo **\n** (o padrão, com já vimos, é um espaço em branco). O **\n** é um caractere de escape. Para conhecer outros caracteres, acesse https://www.w3schools.com/python/gloss_python_escape_characters.asp

Entre dois comandos **print()** o comportamento padrão é a quebra de linha, mas podemos alterar usando o parâmetro **end**.

```
print('Olá', end=', ')
print('seja bem vindo(a)!') # Será mostrado 'Olá, seja bem vindo(a)!', substituindo a nova linha padrão por uma vírgula seguida de um espaço(, )
```

```
Olá, seja bem vindo(a)!
```

2.1.4 Operadores

Operadores aritméticos

Os operadores aritméticos são usados para realizar operações matemáticas:

- Soma: **+**
- Subtração: **-**
- Multiplicação: *****
- Divisão: **/**

- Divisão Inteira: **//**
- Resto: **%**
- Exponenciação: ******

```
# Exemplo de operadores aritméticos
a = 10
b = 3
print('Soma:', a + b)  # Soma resultando em 13
print('Subtração:', a - b)  # Subtração resultando em 7
print('Multiplicação:', a * b)  # Multiplicação resultando em 30
print('Divisão:', a / b)  # Divisão resultando em 3.3333333333333335
print('Divisão Inteira:', a // b)  # Divisão inteira obtém o quociente de uma divisão inteira resultando em 3
print('Resto:', a % b)  # Resto de uma divisão inteira resultando em 1
print('Exponenciação:', a ** b)  # Exponenciação resultando em 1000
```

```
Soma: 13
Subtração: 7
Multiplicação: 30
Divisão: 3.3333333333333335
Divisão Inteira: 3
Resto: 1
Exponenciação: 1000
```

Atenção: Até o **Python 2.8**, a divisão de dois números inteiros sempre resulta em um inteiro, ou seja, caso o resultado seja um número decimal (**float**), esse será truncado e será mostrada apenas a parte inteira. Por exemplo, **5/2** no **Python 3+** resulta **2.5** enquanto no **Python 2.8** resulta **2**. No **Python 2.8** para que se obtenha o resultado correto, pelo menos um dos operandos deve ser do tipo **float**, podendo isso ser realizado por meio de uma operação de *casting*, por exemplo, **float(5)/2**.

Podemos utilizar a forma abreviada para as expressões aritméticas quando o primeiro operando e a variável que receberá o resultado são os mesmos objetos[1]. Ou seja, acontece o que chamamos de operação *in place*, quando uma varíável tem seu valor alterado - subtraído, somado etc. - a partir do seu valor atual.

Por exemplo, **cont = cont + 1** pode ser escrito como **cont += 1**, cont é uma variável contadora que tem seu valor somado a 1 e, portanto, dizemos que *"**cont** recebe ela mesma mais um"*.

Isso é particularmente útil quando temos variáveis contadoras e acumuladoras, comuns em algoritmos, e é aplicável a todos os operadores aritméticos apresentados anteriormente. A seguir, temos alguns exemplos.

[1] Em programação, um objeto é uma entidade que combina dados e comportamentos. Esse conceito é bem amplo e requer mais experiência de programação para compreendê-lo na íntegra. Entretanto, por hora, entenda que objetos em Python e outras linguagens são peças (que armazenam dados) que podem ser manipuladas para montar quebra-cabeças e resolver um problema.

```
num1 = 9
num2 = 4
num1 += 1 # Equivalente a num1 = num1 + 1
print(num1) # Exibe 10
num2 -= 2 # Equivalente a num2 = num2 - 2 e resulta em
print(num2) # Exibe 2
num1 /= num2 # Equivalente a num1 = num1 / num2 e resulta em
print(num1) # Exibe 5.0, lembrando que toda operação de divisão em Python 3+, mesmo que entre 2 números inteiros, resulta em um número decimal ou ponto flutuante (float)
```

```
10
2
5.0
```

Operadores relacionais e lógicos

Operadores relacionais comparam valores e retornam um valor lógico, também conhecido como booleano (*True* ou *False*):

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

Operadores lógicos combinam expressões booleanas:

- ***and***: retorna *True* se ambas as expressões forem verdadeiras
- ***or***: retorna *True* se pelo menos uma das expressões for verdadeira
- ***not***: inverte o valor booleano da expressão

```
# Exemplo de operadores relacionais e lógicos
x = 5
y = 10
print('x é igual a y:', x == y)  # Operador relacional de igualdade com resultado False
print('x é diferente de y:', x != y)  # Operador relacional de diferença com resultado True
print('x é menor que y:', x < y)  # Operador relacional de menor com resultado True
print('x é maior que y:', x > y)  # Operador relacional de maior com resultado False
print('x é menor ou igual a y:', x <= y) # Operador relacional de menor ou igual com resultado True
print('x é maior ou igual a y:', x >= y) # Operador relacional de maior ou igual com resultado False
print('x é menor que 10 E y é maior que 5:', x < 10 and y > 5) # Operador lógico and com resultado True
print('x é menor que 10 OU y é menor que 5:', x < 10 or y < 5) # Operador lógico or com resultado True
print('NÃO x é menor que 10:', not x < 10)  # Operador lógico not com resultado False
```

```
x é igual a y: False
x é diferente de y: True
x é menor que y: True
x é maior que y: False
x é menor ou igual a y: True
x é maior ou igual a y: False
x é menor que 10 E y é maior que 5: True
x é menor que 10 OU y é menor que 5: True
NÃO x é menor que 10: False
```

O operador **not** aguarda a solução da operação relacional para, em seguida, inverter seu valor. Por exemplo, **x < 10** é ***True***, mas após passar pelo ***not***, se torna ***False***.

Não confunda: o operador relacional de comparação de igualdade é composto por dois sinais de igual **==**, enquanto o comando de atribuição de valores a variáveis é apenas um sinal de igual **=**.

Diferença entre os operadores == e is

O operador **==** verifica se os valores de duas variáveis são iguais, enquanto o operador **is** verifica se duas variáveis referem-se ao mesmo objeto.

```
# Exemplo de diferença entre `==` e `is`
a = [1, 2, 3]
b = [1, 2, 3]
c = a
print(a == b)  # True, porque os valores são iguais
print(a is b)  # False, porque são objetos diferentes, ou seja, estão alocados em lugares diferentes da memória
print(a is c) # True, porque c refere-se ao mesmo objeto que a, ou seja, apontam para o mesmo espaço de memória. Na prática, temos dois identificadores para um mesmo valor armazenado na memória.
```

```
True
False
True
```

No exemplo acima, criamos duas listas **a** e **b** (um tipo de vetor que será discutido mais adiante) ambas com valores **[1, 2, 3]** e uma terceira lista **c** recebendo a lista **a**. Apesar de possuírem o mesmo valor, **a** e **b** são objetos diferentes, pois foram criadas de forma independente.

Ao atribuir **c = a** não estamos fazendo uma cópia mas apenas apontando **c** para o mesmo dado apontado por **a**. Para fazer cópias de listas, precisamos realizar uma ***deep copy*** que será discutido adiante.

2.1.5 *Strings*

Concatenação de *strings*

Em Python, você pode concatenar *strings* usando o operador **+**.

```
str1 = 'Hello'
str2 = 'World'
concatenated_str = str1 + ' ' + str2  # Concatena str1 e str2 com um espaço no meio
print(concatenated_str)  # Exibe 'Hello World'
```

```
Hello World
```

Caso precise concatenar uma ***string*** com um valor de outro tipo, deve ser realizado um *casting* para ***string*** utilizando a função **str()**.

```
texto1 = 'Henrique tem '
idade = 1
texto2 = ' ano(s).'
frase = texto1 + str(idade) + texto2  # Concatena as strings com um número convertido para string
print(frase)  # Exibe 'Henrique tem 1 ano(s).'
```

```
Henrique tem 1 ano(s).
```

Strings como lista de caracteres

Em Python, *strings* são tratadas como listas de caracteres, o que significa que você pode acessar caracteres individuais usando índices entre **[** e **]**. Esses índices são os mesmos utilizados em vetores (que serão apresentados mais adiante) e iniciam em **0** (zero) para a primeira posição da lista.

```
s = 'Python'
print(s[0]) #Exibe 'P'. Em Python, o primeiro índice de uma cadeia de caracteres (ou string) e vetores é o 0 (zero).
print(s[1])  # Exibe 'y'
print(s[-1])  # Exibe 'n', o último caractere. Nesse exemplo podemos ver que índices com valores negativos começam a ser contados de forma invertida (de trás para a frente) a partir do final da string.
```

```
P
y
o
```

Método format

O método **format** é usado para formatar *strings*, podendo passar argumentos por posição ou por palavras-chave usando as chaves **{** e **}**.

```
# Usando argumentos por posição
print('Olá, {}! Você tem {} anos.'.format('Maria', 30))  # Exibe 'Olá, Maria. Você tem 30 anos.'

# Usando argumentos por palavras-chave
print('Olá, {nome}! Você tem {idade} anos.'.format(nome='José', idade=25))
#Exibe 'Olá, José. Você tem 25 anos.'
```

Olá, Maria! Você tem 30 anos.

Olá, José! Você tem 25 anos.

No primeiro exemplo acima, os parâmetros (ou argumentos) de **.format()** são **Maria** e **30**. Ao escrever um texto seguido de **.format()**, os pares de chaves **{}** serão substituídos pelos parâmetros **Maria** e **30** na mesma ordem que as chaves aparecem.

Já no segundo exemplo, podemos nomear os parâmetros, no exemplo **nome** e **idade**, e informá-los entre as chaves para identificar onde esses serão substituídos.

Outra forma de fazer esse apontamento é informar os índices dos parâmetros, também iniciando com 0, sem a necessidade de nomeá-los, conforme a seguir.

```
# Usando argumentos por índices
print('Olá, {0}. Você tem {1} anos.'.format('José',25))
# Exibe 'Olá, José. Você tem 25 anos.'
```

No exemplo acima, o método **.format()** possui dois parâmetros: "**José**" e "**25**". "**José**", por ser o primeiro, tem índice **0** e "**25**" tem índice **1**. Portanto, podemos utilizar esses índices dentros das chaves **{}** para indicar ondes eles serão exibidos.

f-strings

f-strings *(*strings* literais formatadas) são usadas para embutir expressões, variáveis e comandos Python dentro de *strings*, precedendo a *string* com **f**.

```
nome = 'Ana'
idade = 22
print(f'Olá, {nome}! Você tem {idade} anos. Nos próximos 10 anos, você completará {idade + 10} anos.') # Exibe 'Olá, Ana. Você tem 22 anos. Nos próximos 10 anos você completará 32 anos.'
```

Olá, Ana! Você tem 22 anos. Nos próximos 10 anos, você completará 32 anos.

No exemplo acima, foi possível efetuar uma operação aritmética **{idade + 10}** dentro do **print()**, pois foi utilizado **f-strings**.

Dentro de **f-strings**, você pode incluir qualquer expressão válida de Python.

```
import math # Importa a biblioteca math que contém funções matemáticas mais específicas.
raio = 5
print(f'A área de um círculo com raio {raio} é {math.pi * raio ** 2:.2f}')
# Exibe 'v área de um círculo com raio 5 é 78.54'
```

```
A área de um círculo com raio 5 é 78.54
```

A notação **.2f** utilizada no exemplo acima é usada para formatar números de ponto flutuante (*float*) em uma *string*, garantindo que eles sejam exibidos com duas casas decimais, como no exemplo. Essa notação é parte do mecanismo de formatação de *strings*, que pode ser utilizado de várias maneiras:

- **.2**: especifica o número de casas decimais que devem ser exibidas após o ponto decimal. Esse número pode ser alterado conforme sua necessidade.
- **f**: indica que o valor deve ser formatado como um número de ponto flutuante (*float*).

2.1.6 Objeto None

None é um valor especial em Python que representa a ausência de valor ou um valor nulo e é testado utilizando o operador **is**.

Uma curiosidade é que **None** é o único valor do tipo de dado **NoneType**, um tipo especial, e pode ser usado para inicializar variáveis que ainda não possuem valor, evitando assim erros de referência antes que a variável receba um valor válido.

```
x = None
x is None # Exibe True
```

```
True
```

2.1.7 Principais Funções Built-in

Python tem várias funções *built-in* úteis:

- **len()**: retorna o comprimento de um objeto;
- **type()**: retorna o tipo de um objeto;
- **int()**, **float()**, **str()**: convertem para inteiro, *float* e *string*, respectivamente;
- **input()**: lê uma entrada do usuário;
- **print()**: exibe uma saída.

Funções ***built-in*** são funções nativas que estendem os recursos básicos do Python e que já acompanham a instalação mínima da linguagem. Você pode utilizar mais funções importando bibliotecas externas, que abordaremos adiante.

```
# Exemplo de funções built-in
lista = [1, 2, 3, 4]
print('Comprimento da lista:', len(lista))
# Exibe 'Comprimento da lista: 4'
print('Tipo da variável lista:', type(lista))
# Exibe 'Tipo da variável lista: <class 'list'>'
num_str = '123'
num_int = int(num_str)
print('Número inteiro:', num_int)  # Converte string para inteiro e exibe 'Número inteiro: 123'
print('Entrada do usuário:', input('Digite algo: '))  # Solicita entrada do usuário e exibe o valor digitado em uma única linha de código
```

```
Comprimento da lista: 4
Tipo da variável lista: <class 'list'>
Número inteiro: 123
Digite algo: olá
Entrada do usuário: olá
```

Você pode acessar a lista completa dessas funções nativas em https://docs.python.org/3/library/functions.html.

Como o Python é conhecido pelo seu vasto ecossistema de bibliotecas desenvolvidas e mantidas pela comunidade, podemos utilizar o comando **import** para importar esses recursos para ter acesso às suas funcionalidades extendidas, como exemplificado anteriormente com a biblioteca **math**. Essas funcionalidade extendidas cobrem uma ampla gama de aplicações, desde ciência de dados e aprendizado de máquina até desenvolvimento *web* e automação de tarefas.

O comando a seguir lista todas as bibliotecas atualmente suportadas pelo Google Colab. Entretanto, existem formas para se instalar outras bibliotecas que não serão discutidas no escopo deste Microcurso.

```
!pip freeze # Exibe todas as bibliotecas Python disponíveis no Google Colab
```

As bibliotecas estão listadas no **Google Colab**.

2.2 Saiba Mais...

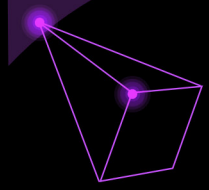
- **Documentação oficial** do Python.
- Lista de **caracteres de escape** em Python.
- Lista completa de **funções nativas** do Python.

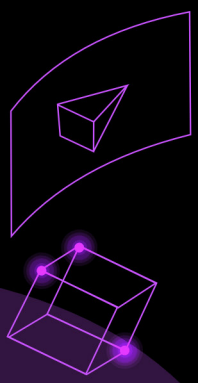
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Introdução ao *Python*

Unidade III

Estruturas de Dados





Unidade III - Estruturas de Dados

3.1 Introdução às Estruturas de Dados

Nesta Unidade, daremos continuidade nos estudos sobre a linguagem Python utilizando *Notebooks Colab*. Entretanto, antes disso, vamos introduzir conceitos básicos da importância e aplicação das estruturas de dados. Com esses fundamentos, você estará apto(a) a colocar em prática o uso dessas estruturas.

3.1.1 O Que São Estruturas de Dados?

Estruturas de dados são formas organizadas de armazenar e gerenciar informações em um programa de computador. Elas fornecem maneiras eficientes de acessar, manipular e usar dados, facilitando a implementação de algoritmos e a resolução de problemas complexos. Em Python, algumas das estruturas de dados mais comuns são: listas, tuplas, dicionários e conjuntos.

3.1.2 Por Que Utilizar Estruturas de Dados?

O uso de estruturas de dados é fundamental na programação por várias razões:

- **Organização de dados:** estruturas de dados ajudam a organizar informações de forma sistemática. Isso torna mais fácil a busca, inserção, atualização e remoção de dados.
- **Eficiência:** cada estrutura de dados é projetada para ser eficiente em certos tipos de operações. Escolher a estrutura correta pode melhorar significativamente o desempenho de um programa.
- **Facilidade de manipulação:** estruturas de dados fornecem operações específicas que simplificam a manipulação dos dados. Por exemplo, listas oferecem métodos para adicionar e remover elementos facilmente.
- **Complexidade reduzida:** ao usar a estrutura de dados apropriada, a complexidade do código pode ser reduzida pois permite que algoritmos sejam implementados de maneira mais clara e concisa.
- **Reusabilidade:** estruturas de dados são reutilizáveis. Uma vez implementadas, as estruturas de dados podem ser usadas em diferentes partes de um programa

ou em diferentes projetos.

- **Melhor gestão de memória:** estruturas de dados ajudam na gestão eficiente da memória, evitando desperdícios e garantindo que os dados sejam armazenados de maneira compacta e acessível.

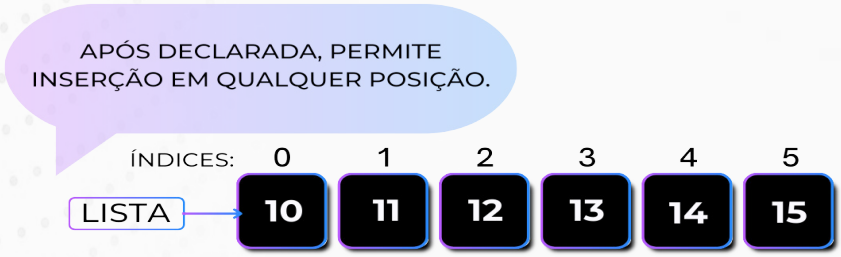
3.1.3 Aplicações das Estruturas de Dados

Cada estrutura de dados possui características específicas que as tornam mais adequadas para certas tarefas. A escolha correta da estrutura de dados pode melhorar significativamente a eficiência e a clareza do código. Vejamos as principais aplicações de cada uma das estruturas de dados mencionadas.

3.1.3.1 Listas

Listas são coleções mutáveis que mantêm seus elementos na mesma ordem que foram adicionados. Elas permitem armazenar qualquer tipo de dado (números, *strings*, objetos, etc.) em qualquer posição, assim como podemos visualizar na imagem a seguir (Figura 5).

Figura 5 - Exemplo de uma lista com seis elementos



Fonte: autoria própria.

Algumas aplicações das listas incluem:

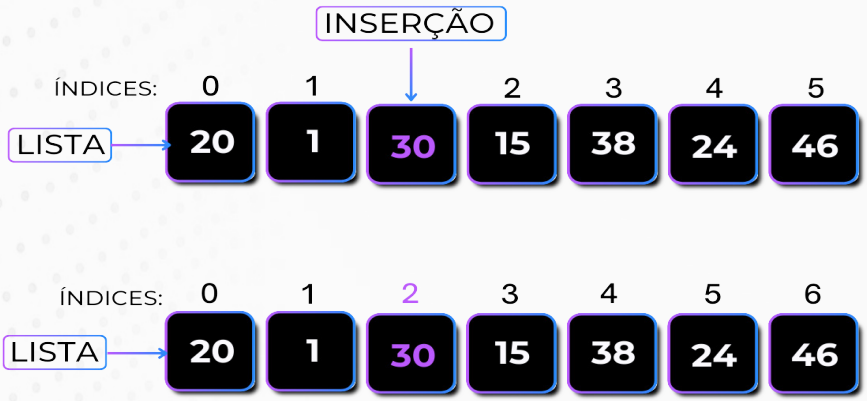
- **Armazenamento dinâmico:** como as listas podem crescer e diminuir de tamanho, elas são ideais para cenários onde a quantidade de dados não é fixa.
- **Iterações e processamento sequencial:** listas são úteis para armazenar sequências de itens que serão processados em ordem.
- **Agrupamento de dados:** quando você precisa agrupar dados heterogêneos (diferentes tipos de dados), as listas são uma boa escolha.

Algumas das principais operações de uma lista incluem:

- **Criação de listas:** inicializar uma lista vazia ou com elementos predefinidos. Este é o ponto de partida para utilizar listas em qualquer aplicação.
- **Acesso a elementos:** acessar elementos individuais da lista usando seu índice.
- **Modificação de elementos:** alteração do valor de um elemento específico da lista, utilizando seu índice. É útil para atualizar dados em uma lista existente.
- **Adição de elementos:** acrescentar um novo elemento ao final da lista ou inserir um novo elemento em uma posição específica dentro da lista, deslocando os elementos subsequentes para a direita.
- **Remoção de elementos:** eliminar a primeira ocorrência de um elemento específico da lista ou em uma posição determinada ou um intervalo de índices.
- **Busca em lista:** encontrar a posição de um elemento específico na lista. Se o elemento estiver presente, retornará seu índice; caso contrário, indicará que o elemento não foi encontrado.

Na Figura 6, podemos visualizar uma lista com elementos iniciais e, logo em seguida, o resultado de uma operação de inserção na posição do índice 2, deslocando em cadeia todos os elementos uma posição à frente.

Figura 6 - Lista inicialmente com seis elementos e a inserção do sétimo elemento na posição do índice 2 (os índices iniciam de zero)



Fonte: autoria própria.

3.1.3.2 Tuplas

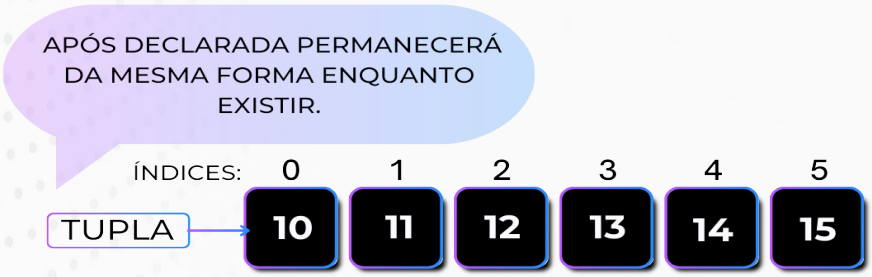
Tuplas são coleções imutáveis que mantêm seus elementos na mesma ordem que foram adicionados. Uma vez criadas, suas entradas não podem ser alteradas. Algumas aplicações das tuplas incluem:

- **Dados constantes:** quando temos um conjunto de dados que não deve ser alterado durante a execução do programa.

- **Chaves de dicionário:** devido à sua imutabilidade, tuplas podem ser usadas como chaves em dicionários.
- **Retorno de múltiplos valores**: funções podem retornar múltiplos valores em uma tupla.
- Uma vez criadas as tuplas, por serem objetos imutáveis, poucas operações são permitidas, sendo as principais:
- **Criação de tuplas:** inicializar uma tupla de elementos predefinidos.
- **Acesso a elementos:** acessar elementos individuais da tupla, usando seu índice.
- **Busca em tupla:** encontrar a posição de um elemento específico na tupla.

Na Figura 7, ao contrário das listas, uma vez criadas, as tuplas não podem ser alteradas. Isso seria possível apenas se criássemos uma nova tupla.

Figura 7 - Exemplo de uma tupla, imutável, criada com seis elementos



Fonte: autoria própria.

3.1.3.3 Dicionários

Dicionários são coleções de pares chave-valor que não mantêm seus elementos ordenados. Algumas aplicações dos dicionários incluem:

- **Mapeamento de dados:** ideal para associar valores a chaves únicas, como um cadastro de usuários onde cada usuário tem um identificador (ID) único.
- **Armazenamento de configurações:** dicionários são usados para armazenar configurações e parâmetros de programas.
- **Contagem e agrupamento:** são úteis para contar ocorrências de itens e agrupar dados de maneira eficiente.

Algumas das principais operações de um dicionário incluem:

- **Criação de dicionários:** inicializar um dicionário vazio ou com elementos predefinidos.
- **Acesso a elementos:** acessar elementos individuais da lista usando seu índice.

- **Modificação de elementos:** alteração do valor de um elemento específico do dicionário, utilizando seu índice.
- **Adição de elementos:** acrescentar um novo elemento ao dicionário.
- **Remoção de elementos:** eliminar a primeira ocorrência de um elemento específico do dicionário ou por meio de uma chave específica.
- **Busca em lista:** encontrar a posição de um valor ou chave específica no dicionário. Se o elemento estiver presente, retornará seu índice; caso contrário, indicará que o elemento não foi encontrado.

A grande vantagem dos dicionários é armazenar valores identificados por chaves únicas que podem ser indexadas. Na Figura 6, apresentamos a estrutura de um dicionário e seus elementos em forma de pares chave-valor.

Figura 8 - Representação de estrutura de um dicionário com pares chave-valor

Fonte: autoria própria.

3.1.3.4 Conjuntos

Conjuntos são coleções que não mantêm seus elementos ordenados e não permitem elementos repetidos. Algumas aplicações dos conjuntos incluem:

- **Eliminação de duplicatas:** quando precisamos garantir que todos os elementos de uma coleção sejam únicos.
- **Operações matemáticas:** conjuntos são ideais para operações como união, interseção e diferença.
- **Verificação de pertinência:** são eficientes para testar a presença de elementos em uma coleção.

Algumas das principais operações entre conjuntos incluem:

- **União:** combina todos os elementos de dois ou mais conjuntos, eliminando duplicatas. Resulta em um novo conjunto contendo todos os elementos únicos dos conjuntos envolvidos.
- **Interseção:** identifica os elementos que são comuns a dois ou mais conjuntos. Resulta em um novo conjunto contendo apenas os elementos presentes em todos os conjuntos envolvidos.
- **Diferença:** subtrai os elementos de um conjunto dos elementos de outro conjunto. Gera um novo conjunto contendo elementos do primeiro conjunto que não estão presentes no segundo conjunto.
- **Diferença simétrica:** combina elementos que estão em um conjunto ou no outro, mas não em ambos. Obtém-se um novo conjunto contendo elementos únicos que estão em apenas um dos conjuntos envolvidos.

A seguir (Figura 9), podemos visualizar o resultado dessas operações entre dois conjuntos.

Figura 9 - Operação de interseção entre dois conjuntos, Grupo A e Grupo B

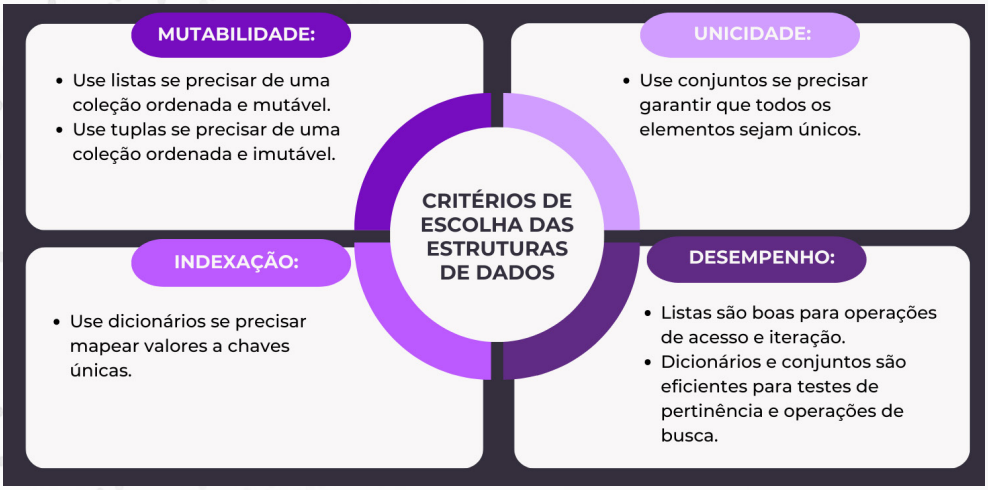
Fonte: autoria própria.

Perceba que os Grupos A e B formam uma interseção ao centro, com os nomes Ana e Antonio fazendo parte de ambos. Por outro lado, uma operação de união entre eles resultaria em um grande grupo com os nomes Carlos, Pedro, Ana, Antonio, Marcos e Paulo, sem repetições. A diferença entre A e B resultaria em Carlos e Pedro, enquanto a diferença entre B e A seria Marcos e Paulo. Já a diferença simétrica entre os grupos A e B resulta em Carlos, Pedro, Marcos e Paulo.

3.1.4 Critérios de Escolhas das Estruturas de Dados

Não existe nenhuma estrutura de dados que resolva todos os problemas e possa ser aplicada de forma ampla e irrestrita. Como vimos anteriormente, cada uma possui suas características e pontos fortes e fracos. A seguir (Figura 8), listamos apenas algumas vantagens para se utilizar cada uma das estruturas discutidas nesta Unidade, divididas em quatro critérios.

Figura 10 - Critérios que podem ser utilizados para a escolha adequada de estruturas de dados



Fonte: autoria própria.

Existem muitas outras estruturas de dados, como pilhas, filas e árvores, mas essas não serão abordadas neste Microcurso.

Compreender e utilizar as estruturas de dados adequadas é um passo essencial para se tornar um desenvolvedor mais eficaz e produzir código mais limpo e eficiente.

3.2 Notebook Colab: Estruturas de Dados

Vamos colocar em prática os conceitos discutidos nesta Unidade no *Notebook Colab*. As Seções 3.2.1 e 3.2.2 estão disponíveis no *Notebook Colab*.

3.2.1 Objetos Mutáveis e Imutáveis

Em Python, os dados são armazenados em objetos. Esses objetos podem ser classificados como mutáveis ou imutáveis, dependendo de sua capacidade de serem modificados após a criação. Compreender a diferença entre esses tipos de objetos é fundamental para a manipulação eficiente de dados e para evitar erros comuns no desenvolvimento de *software*.

Objetos mutáveis

Objetos mutáveis são aqueles cujo estado ou conteúdo pode ser alterado após sua criação. Em outras palavras, é possível modificar, adicionar ou remover elementos desses objetos sem criar um novo objeto.

Listas, dicionários e conjuntos (que serão abordados nesta Unidade) são exemplos de estruturas de dados mutáveis.

Vantagens:

- Eficiência na modificação de grandes volumes de dados sem a necessidade de criar novos objetos.
- Flexibilidade na manipulação de dados.

Desvantagens:

- Pode levar a efeitos colaterais indesejados se não forem manipulados corretamente.
- Maior potencial para erros difíceis de rastrear em programas complexos.

Objetos imutáveis

Objetos imutáveis são aqueles cujo estado ou conteúdo não pode ser alterado após a criação. Qualquer operação que modifique um objeto imutável resultará na criação de um novo objeto, deixando o original inalterado.

Tuplas (que também será abordada nesta Unidade), *strings* e números são exemplos de objetos imutáveis.

Vantagens:

- Simplicidade e segurança, evitando efeitos colaterais indesejados como a alteração de um valor por acidente.
- Melhor desempenho em algumas operações devido à sua natureza imutável.
- Facilita a implementação de estruturas de dados imutáveis, que são inerentemente *thread-safe*.

Antes de apresentarmos o conceito de *thread-safe*, vamos entender que *threads* são unidades básicas de execução dentro de um processo (um programa em execução). Um processo pode conter várias *threads*, que compartilham o mesmo espaço de memória, mas executam de forma independente. Em programação, *threads* são usadas para executar múltiplas tarefas simultaneamente dentro de um único processo, permitindo que um programa faça várias coisas ao mesmo tempo (paralelismo).

O conceito de *thread-safe* em programação refere-se à propriedade de um pedaço de código, uma função, ou um objeto que pode ser utilizado simultaneamente por múltiplas *threads* sem causar comportamentos inesperados ou erros. Em outras palavras, um código é considerado *thread-safe* se ele funciona corretamente e de forma consistente quando acessado por várias *threads* ao mesmo tempo.

Desvantagens:

- Pode ser menos eficiente em termos de memória e processamento ao realizar muitas operações de modificação, pois novos objetos precisam ser criados.

Para saber mais sobre objetos mutáveis e imutáveis, acesse https://www.geeksforgeeks.org/mutable-vs-immutable-objects-in-python/.

3.2.2 Estruturas básicas de dados

Este *notebook* aborda as estruturas de dados básicas do Python: listas, tuplas, dicionários e conjuntos. Cada seção contém uma explicação detalhada, seguida de exemplos de código com comentários.

3.2.2.1 Listas

Conceito

Listas são coleções ordenadas e mutáveis de elementos. Elas são extremamente versáteis e permitem a manipulação de dados de várias formas.

Dizemos que uma coleção de dados é ordenada quando a ordem de inserção dos elementos nessa estrutura não se altera dinamicamente, ou seja, quando a ordem de inserção dos elementos é mantida no decorrer do seu uso.

Aplicações

Listas são usadas quando precisamos de uma coleção de itens ordenada e que pode ser modificada. Elas são úteis para armazenar sequências de dados, como registros de dados, filas de tarefas e muito mais.

Em Python, uma lista é criada usando os colchetes [e] e separando os elementos por vírgula. Importante ressaltar que esses elementos devem respeitar sua tipagem, por exemplo, *strings* devem ser colocadas entre aspas (“) ou apóstrofos (‘).

Por serem uma estrutura de dados, as listas possuem métodos e funções capazes de realizar inúmeras operações que facilitam a manipulação de dados, apresentadas a seguir.

Para saber mais sobre listas, acesse https://www.w3schools.com/python/python_lists.asp.

Criar uma lista

Podemos criar listas vazias apenas atribuindo [] a um identificador ou informando os elementos separados por vírgula entre [e].

```
lista _ vazia = [] # Define uma lista sem elementos
frutas = ['maçã', 'banana', 'cereja']  # Define uma lista de frutas
print(frutas)  # Exibe a lista ['maçã', 'banana', 'cereja']
```

['maçã', 'banana', 'cereja']

Acessar elementos da lista

Para acessar um elemento específico da lista, chamamos o identificador da lista seguido do índice desejado entre **[** e **]**. Lembrando que em Python todos os índices de elementos iteráveis começam com **0** (zero).

```
print(frutas[0])  # Exibe o primeiro elemento da lista maçã
print(frutas[-1])  # Exibe o último elemento da lista cereja
```

maçã

cereja

Modificar elementos da lista diretamente

A modificação direta também é permitida quando atribuímos o novo valor à posição desejada (índice).

```
frutas[1] = 'laranja'  # Substitui 'banana' por 'laranja'
print(frutas)  # Exibe a lista modificada
```

['maçã', 'laranja', 'cereja']

Adicionar elementos no final da lista

O método **.append()** adiciona novos elementos ao final da lista.

```
frutas.append('kiwi')  # Adiciona 'kiwi' ao final da lista
print(frutas)  # Exibe a lista após adicionar 'kiwi' ['maçã', 'banana', 'cereja', 'kiwi']
```

['maçã', 'laranja', 'cereja', 'kiwi']

Remover elementos da lista

O método **.remove()** recebe o elemento a ser removido e, se não encontrado, retorna um ValueError. O ValueError ocorre quando uma função recebe um argumento do tipo correto, mas com um valor impróprio, como é o caso de um elemento não encontrado.

```
frutas.remove('maçã')  # Remove 'maçã' da lista. Caso o elemento não exista será retornado um ValueError
print(frutas)  # Exibe a lista após remover 'maçã' ['banana', 'cereja', 'kiwi']
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-00ebf1e89d24> in <cell line: 1>()
----> 1 frutas.remove('maçã')  # Remove 'maçã' da lista. Caso o elemento não exista será retornado um ValueError
      2 print(frutas)  # Exibe a lista após remover 'maçã' ['banana', 'cereja', 'kiwi']

ValueError: list.remove(x): x not in list
```

Ao executar novamente o código anterior, será retornado um erro pois **maçã** não será encontrada na lista.

Adicionar um elemento a uma posição específica

O método **.insert()** insere um novo elemento em uma posição específica da lista. Ele recebe dois parâmetros: primeiro, a posição, e, segundo, o elemento.

```
frutas.insert(1, 'maçã') # Adiciona 'maçã' na segunda posição da lista e desloca 'cereja' e 'kiwi' para as posições 2 e 3, respectivamente.
print(frutas) # Exibe ['banana', 'maçã', 'cereja', 'kiwi']
```

```
['laranja', 'maçã', 'cereja', 'kiwi']
```

Remover e retornar o último elemento da lista

O método **.pop()** remove o último elemento da lista e o retorna para ser armazenado ou exibido.

```
removido = frutas.pop() # Remove 'kiwi' e o armazena em uma variável
print(removido) # Exibe o elemento removido kiwi, armazenado em removido
print(frutas) # Exibe ['banana', 'maçã', 'cereja']
```

```
kiwi
['laranja', 'maçã', 'cereja']
```

Acessar um elemento por um índice inexistente

```
print(frutas[5]) # Retorna um IndexError
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-17-67dfc58dc835> in <cell line: 1>()
----> 1 print(frutas[5]) # Retorna um IndexError

IndexError: list index out of range
```

O **IndexError** ocorre quando você tenta acessar um índice que não existe em uma lista, tupla ou outra sequência.

Remover um elemento usando o índice

O comando **del** remove elementos de uma lista por meio do seu índice.

```
del frutas[2] # Remove 'cereja' que está no índice 3
print(frutas) # Exibe ['laranja', 'maçã']
```

```
['laranja', 'maçã']
```

Obter o índice de um elemento

Também podemos efetuar uma busca por um elemento em uma lista e, se encontrado, mostar a posição em que ele se encontra. Caso contrário, retornará um **ValueError**.

```
frutas.index('maçã') # Busca por 'maçã' na lista e retorna o índice 1 onde ele está localizado
```

```
1
```

Verificar se um elemento está contido em uma lista

O operador **in** é utilizado para testar se um elemento está contido em uma lista e é bem útil quando utilizado em estruturas condicionais (que serão discutidas mais adiante), pois é retornado ***True***, se o elemento for encontrado, ou ***False***, caso contrário.

```
'laranja' in frutas # Retorna True
```

```
True
```

Concatenar listas

Existem duas formas para se concatenar listas:

- Usando o operador **+**: é necessário que o resultado seja armazenado em uma outra lista (ou em uma das listas a serem concatenadas); ou
- Usando o método **.extend()**: é considerada uma operação *in place*[1], pois o resultado é armazenado na lista que invocou a extensão.

[1] Uma operação *in-place* é uma operação que modifica diretamente os dados de uma estrutura, em vez de criar uma nova cópia da estrutura com as modificações. Em outras palavras, a operação *in-place* altera os dados no próprio local (ou memória) onde eles estão armazenados. Esse tipo de operação é eficiente em termos de uso de memória, pois não requer alocação de espaço adicional para armazenar uma cópia modificada dos dados.

```
novas_frutas = ['uva', 'banana', 'mexerica']
frutas += novas_frutas
#frutas = frutas + novas_frutas # Também pode ser escrito na forma abreviada frutas += novas_frutas
print(frutas) # Exibe ['laranja', 'maçã', 'uva', 'banana', 'mexerica']
```

['laranja', 'maçã', 'uva', 'banana', 'mexerica']

```
outras_frutas = ['caqui', 'manga']
outras_frutas.extend(frutas) # Extende outras_frutas adicionando à ela a lista frutas
print(outras_frutas) # Exibe ['caqui', 'manga', 'laranja', 'maçã', 'uva', 'banana', 'mexerica']
```

['caqui', 'manga', 'laranja', 'maçã', 'uva', 'banana', 'mexerica']

Obter o tamanho de uma lista

```
tamanho = len(outras_frutas)
print('A lista contém', tamanho, 'elementos.') # Mostra a quantidade de elementos de uma lista 'A lista contém 7 elementos.'. Não confunda com o último índice da lista que é sempre len-1
```

A lista contém 7 elementos.

Deep copy

As listas são vetores em forma de estruturas de dados, pois, além dos dados, elas possuem métodos próprios que manipulam esses dados. Assim, a cópia de seus dados não pode ser realizada diretamente usando o operador de atribuição **=**. Ao executar o

exemplo a seguir, estamos apenas criando um novo identificador (referência na memória) para a mesma estrutura de dados.

```
a = [1, 2, 3]
b = a
print('a está alocada na posição de memória', hex(id(a)))
print('b está alocada também na posição de memória', hex(id(b)))
```

```
a está alocada na posição de memória 0x791a6bef2180
b está alocada também na posição de memória 0x791a6bef2180
```

A função **id()** retorna o identificador único da variável passada como parâmetro, no exemplo as listas **a** e **b**; e a função **hex()** converte o identificador para hexadecimal, uma base numérica utilizada para referenciar posição na memória RAM. É importante lembrar que a cada execução a variável é alocada em um espaço diferente, liberando a posição anterior para outras aplicações.

Vamos executar um novo experimento. Vamos fazer uma cópia simples (apenas da referência como fizemos anteriormente) e, ao alterar um valor em uma lista, a outra também será alterada, pois, como já discutimos, são a mesma lista com identificadores diferentes.

```
a = [1, 2, 3]
b = a # Copia apenas a referência
b[1] = 4 # Altera o valor usando a referência b mas também altera em a
print(b) # Exibe [1, 4, 3]
print(a) # Exibe também [1, 4, 3]
```

```
[1, 4, 3]
[1, 4, 3]
```

No exemplo a seguir, efetuamos uma *deep copy* (cópia profunda) passando o índice **:** que significa *all* (todos os elementos). Exploraremos melhor esse recurso a seguir.

```
a = [1, 2, 3]
b = a[:]
print('a está alocada na posição de memória', hex(id(a)))
print('b agora é uma outra lista independente e está alocada na posição
de memória', hex(id(b)))
```

```
a está alocada na posição de memória 0x791a68cff940
b agora é uma outra lista independente e está alocada
na posição de memória 0x791a68d9e000
```

Intervalos em listas

A partir de uma lista, podemos seguimentá-la em intervalos para obter um trecho específico. No Python, todo intervalo segue o padrão **[início:fim)**, ou seja, fechado no início e aberto no fim.

Por exemplo, o intervalo **3:7** vai de 3 (inclusive) até 6 (inclusive). Vamos exemplicar as aplicações de intervalos em listas, lembrando que o índice das listas se inicia com **0** (zero).

```
   print(outras _ frutas[:]) # Exibe todas as frutas
['caqui', 'manga', 'banana', 'maçã', 'uva', 'banana',
'mexerica'], equivalente a print(outras _ frutas)
   print(outras _ frutas[1:3]) # Exibe da segunda (índice 1)
até a terceira fruta (índice 2)['manga', 'banana']
   print(outras _ frutas[2:]) # Exibe todas as frutas a partir da
terceira (índice 2) ['banana', 'maçã', 'uva', 'banana', 'mexerica']
   print(outras _ frutas[:3]) # Exibe todas as frutas até
a terceira (índice 2) ['caqui', 'manga', 'banana']
```

['caqui', 'manga', 'laranja', 'maçã', 'uva', 'banana', 'mexerica']

['manga', 'laranja']

['laranja', 'maçã', 'uva', 'banana', 'mexerica']

['caqui', 'manga', 'laranja']

Podemos também informar o passo (*step*) do intervalo desejado na forma **início:fim:passo**, como no exemplo a seguir.

```
   print(outras _ frutas[0:7:2]) # Exibe todas as frutas, alternando de 2
em 2, ou seja, uma sim e outra não ['caqui', 'banana', 'uva', 'mexerica'],
equivalente a print(outras _ frutas[::2])
   print(outras _ frutas[::-1]) # Exibe todas as frutas,
em ordem invertida, do final para o início ['mexerica',
'banana', 'uva', 'maçã', 'banana', 'manga', 'caqui']
```

['caqui', 'laranja', 'uva', 'mexerica']

['mexerica', 'banana', 'uva', 'maçã', 'laranja', 'manga', 'caqui']

Encadeamento de listas

Como já dissemos, uma lista pode armazenar elementos de qualquer tipo, inclusive outras listas, sem limite de profundidade. No caso de uma lista de listas, temos duas dimensões, ou seja, duas listas encadeadas. Para fins de exemplificação, isso seria equivalente a matrizes (com linhas e colunas) mas, na prática, temos bibliotecas específicas para cálculo matricial que disponibilizam vetores bidimensonais. Isso não faz parte do escopo deste Microcurso.

No exemplo a seguir, é mostrada a criação de uma lista de listas e como seus elementos são acessados, informando dois índices: o primeiro índice se refere aos elementos da lista principal e o segundo se refere aos elementos da lista mais interna, por exemplo **[1][2]**. Lembrando que, em Python, todos os índices de elementos iteráveis começam com 0 (zero).

```
   grupos = [['Ana', 'João', 'Clara'], ['Pedro', 'Júlio',
'Luiz'], ['Henrique', 'Carlos', 'Vitória']]
   # Cria duas listas encadeadas (produndidade 2)
   print(grupos) # Exibe [['Ana', 'João', 'Clara'], ['Pedro',
'Júlio', 'Luiz'], ['Henrique', 'Carlos', 'Vitória']]
   print(grupos[1][2]) # Exibe o 3° participante do 2° grupo Luiz
```

```
[['Ana', 'João', 'Clara'], ['Pedro', 'Júlio', 'Luiz'], ['Henrique', 'Carlos', 'Vitória']]
Luiz
```

No caso do encadeamento de mais listas, a quantidade de índices é equivalente à profundidade do encadeamento. Por exemplo, para acessarmos um elemento de três listas encadeadas, devemos informar três índices, por exemplo, **[1][0][2]**.

Quando lidamos com listas de listas encadeadas em múltiplos níveis (ou seja, listas dentro de listas dentro de listas), a quantidade de índices que usamos para acessar um elemento é equivalente à profundidade do encadeamento. Para acessar um elemento específico, você deve fornecer um índice para cada nível de profundidade.

Pode parecer um pouco confuso, mas vamos pensar de outra forma: quando criamos listas encadeadas, estamos basicamente colocando listas dentro de outras listas. A profundidade do encadeamento se refere ao número de níveis de listas aninhadas. Cada nível adicional de listas requer um índice adicional para acessar os elementos dentro da estrutura.

Pense em uma lista como uma matriz bidimensional ou tridimensional. Cada dimensão adicional representa um nível de encadeamento e, para acessar um elemento específico, você precisa especificar a posição em cada dimensão.

3.2.2.2 Tuplas

Conceito

Tuplas são coleções ordenadas e imutáveis de elementos. Elas são semelhantes às listas, mas, uma vez criadas, não podem ser modificadas.

Aplicações

Tuplas são usadas quando queremos garantir que a coleção de itens não será alterada. Elas são úteis para armazenar dados heterogêneos, como coordenadas geográficas,

registros de banco de dados e mais. Por ser imutáveis, são mais eficientes que as listas, ou seja, consomem menos recursos computacionais e executam mais rapidamente, embora também sejam mais restritas em relação aos métodos para manipulação um vez criadas.

Em Python, uma tupla é criada usando os parênteses **(** e **)** e separando os elementos por vírgula. Importante ressaltar que esses elementos devem respeitar sua tipagem, por exemplo, *strings* devem ser colocadas entre aspas (“) ou apóstrofos (‘).

Para saber mais sobre tuplas, acesse https://www.w3schools.com/python/python_tuples.asp.

Criar uma tupla com mais de um elemento

Tuplas são criadas com elementos entre **(** e **)** e são separados por vírgula. Veremos, mais adiante, o caso de tuplas com apenas um elemento.

```
coordenadas = (10.0, 20.0)
print(coordenadas)  # Exibe a tupla (10.0, 20.0)
```

```
(10.0, 20.0)
```

Acessar elementos da tupla

Os elementos de uma tupla são acessados da mesmo forma que os elementos de uma lista.

```
latitude = coordenadas[0]  # Acessa e atribui à variável latitude o primeiro elemento da tupla
longitude = coordenadas[-1]  # Acessa e atribui à variável longitude o segundo elemento da tupla

print(coordenadas[0])  # Exibe 10.0
print(coordenadas[-1])  # Exibe 20.0
```

```
10.0
20.0
```

Criar uma tupla com apenas um elemento

Embora pouco usual, mas possível, para criar uma tupla com apenas um elemento, precisamos acrescentar uma vírgula (,) após o elemento. Caso contrário, o Python converterá o elemento para seu tipo primitivo.

```
print(type((1))) # Será exibido <class 'int'>
print(type((1,))) # Será exibido <class 'tuple'>
print(type(())) # Será exibido <class 'tuple'>
```

<class 'int'>

<class 'tuple'>

<class 'tuple'>

Como você deve ter percebido acima, essa questão não se aplica à criação de tuplas vazias (sem elementos), criadas apenas com **()**, embora ainda menos usual em aplicações práticas.

Modificar elementos de uma tupla

Como discutimos anteriormente, as tuplas são objetos imutáveis e, portanto, não podem ser alteradas após criadas.

```
coordenadas[1] = 15.0 # Resultará em um TypeError
```

TypeError Traceback (most recent call last)
<ipython-input-34-ee344a749195> in <cell line: 1>()
----> 1 coordenadas[1] = 15.0 # Resultará em um TypeError

TypeError: 'tuple' object does not support item assignment

Operações com tuplas

Você também pode aplicar a maioria das operações de listas em tuplas, respeitando a característica imutável da estrutura.

```
tup = () # Cria uma tupla vazia. Use print(hex(id(tup))) para
visualizar o endereço de
                    memória
tup = (1, 2, 3) # Cria uma NOVA tupla, liberando a memória da
anterior. O comando print(hex(id(tup))) resultará em outro endereço
tup = tup + (4, 5, 6)  # Cria uma NOVA tupla a partir
da concatenação de (1, 2, 3) e (4, 5, 6). O comando
print(hex(id(tup))) resultará em outro endereço
print(len(tup)) # Mostra o tamanho da tupla 6
print(tup[:2]) # Intervalo de elementos de uma tupla (1, 2)
print(2 in tup) # Verifica se um elemento está contido na tupla True
```

```
6
(1, 2)
True
```

Também é possível fazer *casting* de tuplas para listas e vice-versa.

```
tup = (1, 2, 3, 4, 5, 6)
lista = list(tup) # Converte uma tupla para uma lista
print(type(lista)) # Exibe <class 'list'>
```

```
<class 'list'>
```

Tuplas podem ser descompactadas diretamente, elemento a elemento, ou de forma estendida.

A descompactação direta ocorre quando informamos uma variável correspondente para cada elemento da tupla. Já a estendida ocorre quando o número de elementos é superior ao número de variáveis que receberão esses elementos. Nesse último caso, usamos o asterisco (*) para indicar qual das variáveis receberá uma lista com os valores excedentes.

```
a, b, c = (1, 2, 3) # Resultará a = 1, b = 2, c = 3
*a, b, c = (1, 2, 3, 4) # Resultará a = [1, 2], b = 3, c = 4
a, *b, c = (1, 2, 3, 4) # Resultará a = 1, b = [2, 3], c = 4
a, b, *c = (1, 2, 3, 4) # Resultará a = 1, b = 2, c = [3, 4]
```

3.2.2.3 Dicionários

Conceito

Dicionários são coleções de pares chave-valor. Eles são desordenados e mutáveis, permitindo acesso rápido aos valores por meio das chaves.

Aplicações

Dicionários são usados quando precisamos de uma associação entre pares chave-valor. Eles são úteis para armazenar dados indexados, como registros de usuários, inventários de produtos e mais.

Em Python, um dicionário é criado usando chaves **{** e **}** e separando os pares **chave:valor** por vírgula. Importante ressaltar que as chaves devem ser sempre elementos imutáveis (*string*, *int*, tuplas etc.) e tanto as chaves quanto os valores devem respeitar sua tipagem, por exemplo, *strings* devem ser colocadas entre aspas (“) ou apóstrofos (‘).

Para saber mais sobre dicionários, acesse https://www.w3schools.com/python/python_dictionaries.asp.

Criar um dicionário vazio

Usamos a função **dict()** para criarmos dicionários vazios, pois chaves vazias **{}** remetem a conjuntos, que serão discutidos a seguir.

```
aluno = dict() # Cria um dicionário vazio
```

Criar um dicionário com chaves e valores do tipo *string*

```
aluno = {'nome': 'João', 'idade': 20, 'curso': 'Engenharia'}  # Define um dicionário com informações de um aluno
print(aluno)  # Exibe {'nome': 'João','idade': 20, 'curso': 'Engenharia'}
```

{'nome': 'João', 'idade': 20, 'curso': 'Engenharia'}

Acessando valores do dicionário

```
print(aluno['nome'])  # Exibe o valor associado à chave 'nome'
print(aluno.get('cidade'))  # Exibe o valor associado à chave 'cidade' usando o método get se ela existir
```

João

None

No exemplo acima, apresentamos duas formas de acessar diretamente um elemento. Na primeira, caso o índice não seja localizado, é retornado um **KeyError**. Para contornar esse problema, usamos o método **.get()** que retorna **None**, caso o índice passado como parâmetro não exista. Falaremos desse assunto mais adiante.

Modicando valores do dicionário

```
aluno['idade'] = 21  # Atualiza o valor associado à chave 'idade'
print(aluno)  # Exibe o dicionário após a atualização KeyError
```

{'nome': 'João', 'idade': 21, 'curso': 'Engenharia'}

Adicionando novos pares chave-valor

```
aluno['cidade'] = 'Goiânia'  # Adiciona uma nova chave 'cidade' com seu valor
print(aluno)  # Exibe o dicionário após adicionar a nova chave {'nome': 'João', 'idade': 21, 'curso': 'Engenharia', 'cidade': 'Goiânia'}
```

{'nome': 'João', 'idade': 21, 'curso': 'Engenharia', 'cidade': 'Goiânia'}

Removendo pares chave-valor

```
del aluno['curso']  # Remove a chave 'curso' e seu valor associado
print(aluno)  # Exibe o dicionário após a remoção {'nome': 'João', 'idade': 21, 'cidade': 'Goiânia'}
```

{'nome': 'João', 'idade': 21, 'cidade': 'Goiânia'}

Tentando criar um dicionário com chaves mutáveis

```
alunos = {['João', '1° período', 'Lógica de programação']: 6.5, ['Júlia', '1° período', 'Lógica de programação']: 8.9, ['Felipe', '1° período', 'Lógica de programação']: 7.5}
# Resulta em um TypeError
```

TypeError Traceback (most recent call last)
<ipython-input-45-4fa30c37dd4b> in <cell line: 1>()
----> 1 alunos = {['João', '1° período', 'Lógica de programação']: 6.5, ['Júlia', '1° período', 'Lógica de programação']: 8.9, ['Felipe', '1° período', 'Lógica de programação']: 7.5} # Resulta em um TypeError

TypeError: unhashable type: 'list'

O código acima resulta em um **TypeError**, pois tentamos informar nas chaves os dados de alunos em listas (nome, período e disciplina) e nos valores as respectivas notas. Listas são mutáveis e, portanto, não podem ser utilizadas como chaves em dicionários. Entretanto, listas podem ser utilizadas como valores em dicionários. Caso assim desejássemos, poderíamos usar tuplas para manter a forma de armazenar os valores, da seguinte forma:

```
alunos = {['João', '1° período', 'Lógica de programação']: 6.5, ['Júlia', '1° período', 'Lógica de programação']: 8.9, ['Felipe', '1° período', 'Lógica de programação']: 7.5}
# Dicionário válido
print(alunos) # Será exibido {('João', '1° período', 'Lógica de programação'): 6.5, ('Júlia', '1° período', 'Lógica de programação'): 8.9, ('Felipe', '1° período', 'Lógica de programação'): 7.5}
```

{('João', '1° período', 'Lógica de programação'): 6.5, ('Júlia', '1° período', 'Lógica de programação'): 8.9, ('Felipe', '1° período', 'Lógica de programação'): 7.5}

Extrair chaves e valores de um dicionário

A partir de um dicionário, podemos extrair as chaves utilizando o método **.keys()** e os valores utilizando o método **.values()**, separando-os em um objeto iterável. Objeto iterável significa que podemos acessar valores individualmente utilizando o índice entre chaves **[** e **]** ou percorrê-lo, utilizando estruturas de repetição. Entretanto, fica difícil utilizar esse tipo de estrutura, pois ela não possui suporte a todas as funções e métodos de uma **list**. Portanto, a partir da extração das chaves/valores de um dicionário, é usual fazer o *casting* para uma lista.

```
chaves_alunos = list(alunos.keys()) # Extrai as chaves do dicionário usando o método keys() e faz a conversão para uma lista
valores_alunos = list(alunos.values()) # Extrai os valores do dicionário usando o método values() e faz a conversão para uma lista
print(chaves_alunos) # Exibe [('João', '1° período', 'Lógica de programação'), ('Júlia', '1° período', 'Lógica de programação'), ('Felipe', '1° período', 'Lógica de programação')]
print(valores_alunos) # Exibe [6.5, 8.9, 7.5]
```

[('João', '1° período', 'Lógica de programação'), ('Júlia', '1° período', 'Lógica de programação'), ('Felipe', '1° período', 'Lógica de programação')]
[6.5, 8.9, 7.5]

Busca em dicionários

Podemos verificar se uma determinada chave existe em um dicionário utilizando o operador **in**. Entretanto, esse tipo de busca não funciona para os valores, pois os dicionários são estruturas de dados projetadas para indexar valores com suas respectivas chaves que são os objetos de busca em situações reais.

```
print('idade' in aluno) # Retorna True pois existe uma chave 'idade' no dicionário
print(21 in aluno) # Retorna False pois existe um valor 21 no dicionário mas não em uma chave
```

True

False

Embora pouco usual para buscar valores, podemos utilizar o método **.values()** em tempo real.

```
21 in aluno.values()  # Retorna True pois a busca por 21 é realizada em um objeto iterável obtido a partir dos valores do dicionário
```

```
True
```

Ao tentar mostrar um valor a partir de uma chave inexistente é retornado um **KeyError**.

```
aluno['matrícula'] # Retorna um KeyError
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-50-d2b23c8e7347> in <cell line: 1>()
----> 1 aluno['matrícula'] # Retorna um KeyError

KeyError: 'matrícula'
```

Entretanto, podemos utilizar o método **.get()** para evitar o **KeyError** que possui dois tipos de retorno:

- Retorna o valor, se a chave for localizada no dicionário; ou
- **None**, se a chave não existir, evitando erros de execução.

```
print(aluno.get('nome')) # Encontra e retorna João
print(aluno.get('matrícula')) # Não encontra e retorna None
```

```
João
None
```

É possível personalizar o retorno caso a chave não seja localizada informando o retorno desejado no segundo parâmetro do método **.get()**.

```
print(aluno.get('matricula', 'Valor não encontrado.')) # Exibe 'Valor não encontrado.'
```

```
Valor não encontrado.
```

Outras formas de adicionar valores

Vimos anteriormente que para adicionar um valor em um dicionário basta informar o nome do dicionário, seguido pela nova chave e o valor, por exemplo, **aluno['cidade'] = 'Goiânia'**. Entretanto, se a chave já existir no dicionário, o valor será substituído. Caso não desejemos essa substituição, podemos utilizar o método **.setdefault()**, passando a chave e o valor como parâmetros. Assim, se a chave não existir, será adicionada; caso contrário, se existir, o dicionário não será alterado.

```
   aluno.setdefault('matrícula', 2024001001) #
Adiciona a chave e o valor da matrícula
   print(aluno) # Exibe {'nome': 'João', 'idade': 20,
'curso': 'Engenharia', 'matrícula': 2024001001}
   aluno.setdefault('matrícula', 2024001001000) # Tenta
adicionar novamente a chave com outro valor sem sucesso
   print(aluno)  # Exibe o dicionário inalterado {'nome': 'João',
'idade': 20, 'curso': 'Engenharia', 'matrícula': 2024001001}
```

{'nome': 'João', 'idade': 21, 'cidade': 'Goiânia', 'matrícula': 2024001001}

{'nome': 'João', 'idade': 21, 'cidade': 'Goiânia', 'matrícula': 2024001001}

O método **.update()** permite adicionar vários pares de chave-valor a um dicionário existente. Passamos como parâmetro para esse método um dicionário. O efeito é semelhante ao método **.extend()** utilizado em listas, como vimos anteriormente.

```
   aluno.update({'cidade': 'Goiânia', 'estado': 'Goiás'})
# Adiciona 2 novas entradas no dicionário
   print(aluno) # Exibe {'nome': 'João', 'idade': 21, 'curso': 'Engenharia',
'matrícula': 2024001001, 'cidade': 'Goiânia', 'estado': 'Goiás'}
```

{'nome': 'João', 'idade': 21, 'cidade': 'Goiânia', 'matrícula': 2024001001, 'estado': 'Goiás'}

3.2.2.4 Conjuntos

Conceito

Conjuntos são coleções não ordenadas de elementos únicos, ou seja, não permitem elementos duplicados. Apesar de ser uma estrutura de dados mutável, seus elementos devem ser imutáveis.

Aplicações

Conjuntos são usados para armazenar coleções de itens únicos. Eles são úteis para operações de conjunto, como união, interseção e diferença.

Em Python, um conjunto é criado usando chaves **{}** ou a função **set()**. A diferença entre os dicionários e os conjuntos é que os conjuntos não possuem valores, apenas chaves, por isso não permitem que sejam duplicadas. As chaves devem respeitar sua tipagem, por exemplo, strings devem ser colocadas entre aspas (“) ou apóstrofos (‘).

Para saber mais sobre conjuntos acesse w3schools.com/python/python_sets.asp

Criando um conjunto

A função **set()** é utilizada para criar um conjunto vazio. Ao tentarmos criar um conjunto com elementos repetidos, esses serão ignorados e apenas a primeira ocorrência será mantida.

```
frutas = set() # Cria um conjunto vazio
frutas = {'maçã', 'banana', 'cereja', 'banana', 'cereja'}  # Altera
o conjunto com novas chaves, com algumas repetições sem efeito
print(frutas)  # Exibe {'banana', 'maçã', 'cereja'}
```

{'maçã', 'banana', 'cereja'}

Adicionar elementos ao conjunto

O método **.add()** adiciona novos elementos, se não existirem, ao conjunto.

```
frutas.add('laranja')  # Adiciona 'laranja' ao conjunto
print(frutas)  # Exibe {'banana', 'maçã', 'cereja', 'laranja'}
```

{'laranja', 'maçã', 'banana', 'cereja'}

Remover elementos do conjunto

O método **.remove()** remove um elemento existente no conjunto. Caso a chave não seja localizada, será retornado um **KeyError**.

```
frutas.remove('laranja')  # Remove 'laranja' do conjunto
print(frutas)  # Exibe o conjunto após remover 'laranja'
```

{'maçã', 'banana', 'cereja'}

Operações com conjuntos

Podemos efetuar operações equivalentes às operações matemáticas em conjuntos numéricos:

- **União**: obtém a união (junção) de dois conjuntos utilizando o operador **|** ou o método **.union()**.
- **Interseção**: obtém a intersecção (elementos que se repetem) de dois conjuntos utilizando o operador **&** ou o método **.intersection()**.

- **Diferença**: obtém a diferença (elementos encontrados no primeiro conjunto, mas que não estão no segundo conjunto) utilizando o operador **-** ou o método **.difference()**.
- **Diferença simétrica**: obtém a diferença simétrica (elementos que não se repetem entre os conjuntos) utilizando o operador **^** ou o método **.symmetric_difference()**.
- **Subconjunto**: obtém o subconjunto (se todos os elementos do primeiro conjunto estão contidos no segundo conjunto) utilizando o operador **<=** ou o método **.issubset()**.
- **Superconjunto**: obtém o superconjunto (se todos os elementos do segundo conjunto estão contidos no primeiro conjunto) utilizando o operador **>=** ou o método **.issuperset()**.

```
tropicais = {'goiaba', 'banana', 'manga', 'mamão', 'laranja', 'limão'}
citricas = {'laranja', 'limão', 'abacaxi'}
print('União:', tropicais | citricas)   # União equivalente a tropicais.union(citricas)
print('Interseção:', tropicais & citricas) # Interseção equivalente a tropicais.intersection(citricas)
print('Diferença:', tropicais - citricas) # Diferença equivalente a tropicais.difference(citricas)
print('Diferença simétrica:', tropicais ^ citricas)  # Diferença simétrica equivalente a tropicais.symmetric _ difference(citricas)
```

União: {'limão', 'laranja', 'manga', 'goiaba', 'abacaxi', 'mamão', 'banana'}

Interseção: {'laranja', 'limão'}

Diferença: {'manga', 'banana', 'mamão', 'goiaba'}

Diferença simétrica: {'manga', 'abacaxi', 'goiaba', 'mamão', 'banana'}

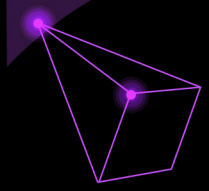
3.3 Saiba Mais...

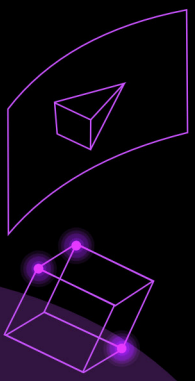
- **Objetos** mutáveis e imutáveis.
- **Listas** em Python.
- **Tuplas** em Python.
- **Dicionários** em Python.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Introdução ao *Python*

Unidade IV
Estruturas de Controle de Fluxo





Unidade IV - Estruturas de Controle de Fluxo

4.1 Introdução às Estruturas de Controle de Fluxo

Estruturas de controle de fluxo são fundamentais na programação, pois permitem que o código “tome decisões” e execute repetidamente determinadas ações com base em condições ou ciclos específicos. Em Python, as principais estruturas de controle de fluxo incluem as instruções condicionais (**if**, **elif**, **else**) e as estruturas de repetição (**for**, **while**). Essas estruturas possibilitam a construção de programas que realizam tarefas complexas, baseadas em lógica condicional e repetitiva.

4.1.1 Instruções Condicionais

As instruções condicionais são usadas para executar determinadas partes do código com base em condições específicas. Em Python, a estrutura básica das instruções condicionais envolve:

- **if:** avalia uma condição e executa um bloco de código, se a condição for verdadeira.
- **elif:** permite adicionar condições adicionais, executando um bloco de código, se a condição anterior for falsa e a nova condição for verdadeira.
- **else:** define um bloco de código que será executado, se nenhuma das condições anteriores for satisfeita.

Essas instruções permitem a implementação de lógica condicional que dirige o fluxo do programa de acordo com diferentes critérios e cenários.

Vamos entender, de forma prática, como podemos utilizar instruções condicionais em nosso dia a dia. Assim também ficará claro como elas poderiam ser utilizadas em programação.

Imagine que você está dirigindo e chega a um cruzamento de estradas. Você tem várias opções sobre qual caminho seguir, dependendo das condições à sua frente:

- **if:** você olha para a placa de trânsito e vê que, se o sinal estiver verde, você deve seguir em frente.
- **elif:** se o sinal não estiver verde, mas você vê uma placa indicando um desvio à direita, você poderá virar à direita.

- **else:** se nenhuma das condições anteriores for verdadeira (por exemplo, o sinal está vermelho e não há desvio), você decide parar e esperar.

Como vimos, as condicionais tradicionais em Python funcionam como essas decisões no cruzamento, permitindo que o programa escolha um caminho diferente dependendo das condições que encontrar.

Ainda podemos utilizar uma estrutura recentemente incluída no Python para simplificar o processo de tomada de decisão em um código: o **match-case**. A estrutura de controle de fluxo **match-case** foi introduzida no Python 3.10 e é usada para combinar uma variável com diferentes valores, executando o bloco de código correspondente. Isso proporciona uma maneira mais elegante e organizada de lidar com múltiplas condições, especialmente quando se está comparando um valor com várias possibilidades específicas.

E como seria o funcionamento do **match-case**?

- **Definição do valor a ser comparado**: o comando **match** define a variável cujo valor será comparado.
- **Definição dos possíveis valores**: cada **case** define um valor específico com o qual o valor da variável será comparado.
- **Execução do bloco de código**: quando um valor é encontrado, o bloco de código correspondente ao **case** é executado. Se nenhum valor correspondente for encontrado, um **case** padrão pode ser definido para lidar com todos os outros casos.

Podemos entender melhor o **match-case** com a seguinte metáfora: imagine que você está em um restaurante de *buffet*, onde há diferentes opções de comidas. Você tem um prato na mão e deseja selecionar a comida que mais gosta, usando o **match-case**:

- **match**: quando você chega diante do *buffet*, você compara o tipo de comida disponível com o que você está procurando.
- **case**: se há o tipo de comida que você quer, você se serve. Cada tipo de comida representa um **case** diferente com uma opção específica.

Assim como no *buffet*, onde você decide qual comida servir com base nas opções disponíveis, o **match-case** em Python permite que você compare uma variável com diferentes valores e execute o bloco de código correspondente ao valor que combina. Isso torna mais fácil e organizado tratar diferentes casos específicos dentro do seu programa.

4.1.2 Estruturas de Repetição

As estruturas de repetição permitem que um bloco de código seja executado repetidamente enquanto uma condição específica for verdadeira. As principais estruturas de repetição em Python são:

- **for:** utilizada para iterar sobre uma sequência (como listas, tuplas, dicionários ou *strings*), executando um bloco de código para cada item na sequência. Essa estrutura é ideal para tarefas que requerem iteração sobre elementos conhecidos.
- **while:** executa um bloco de código enquanto uma condição for verdadeira. É útil quando não se conhece previamente o número de iterações necessárias e a repetição depende de uma condição dinâmica.

Para também entender as estruturas de repetição, vamos usar outra metáfora. Imagine que você está em uma academia de ginástica, realizando uma rotina de exercícios. Cada estrutura de repetição pode ser vista como um tipo diferente de treino:

- **for**: pense em um treino onde você precisa fazer 10 repetições de levantamento de peso. Você sabe exatamente quantas vezes precisa repetir o exercício. A estrutura **for** funciona assim, iterando sobre uma sequência pré-determinada de vezes, como contar de 1 a 10.
- **while**: agora, imagine que você está correndo na esteira e pretende correr até que esteja muito cansado. Você não sabe exatamente quanto tempo ou quantas voltas vai correr, mas continuará enquanto sentir que consegue. A estrutura **while** funciona dessa maneira, repetindo um bloco de código enquanto uma condição específica for verdadeira.

Essas metáforas ajudam a visualizar como as estruturas de repetição controlam a execução contínua de tarefas, seja com um número definido de iterações ou com base em uma condição contínua.

4.1.3 Aplicações das Estruturas de Controle de Fluxo

As estruturas e controle de fluxo são aplicáveis em diversas situações, incluindo:

- **Decisões lógicas:** permitem ao programa “tomar decisões” com base em diferentes condições, como escolher entre alternativas ou validar dados de entrada.
- **Iteração sobre dados:** facilitam a execução repetida de operações sobre elementos de coleções de dados, como realizar cálculos em uma lista de números ou processar elementos de um arquivo.
- **Controle de fluxo de programas complexos:** são essenciais para implementar algoritmos que exigem lógica condicional e repetição, como algoritmos de busca, ordenação e processamento de dados.

As estruturas de controle de fluxo em Python são cruciais para criar programas que podem tomar decisões e executar ações repetitivas com base em condições específicas. Com elas, desenvolvedores podem construir lógica complexa e dinâmica, permitindo a

implementação de algoritmos eficientes e programas interativos. Ao dominar o uso de instruções condicionais e de repetição, você estará apto a construir aplicações robustas e funcionais em Python.

4.2 Notebook Colab: Estruturas de Controle de Fluxo

Vamos colocar em prática os conceitos discutidos nesta Unidade no *Notebook Colab*. A Seção 4.2 está disponível no *Notebook Colab*.

Para aprender mais sobre estruturas de controle de fluxo, acesse https://docs.python.org/pt-br/3/tutorial/controlflow.html.

4.2.1 Estruturas Condicionais

Conceito

Estruturas condicionais permitem que o código execute diferentes blocos de instruções com base em condições específicas. Em Python, usamos **if**, **elif** e **else** para criar essas estruturas.

Aplicações

Estruturas condicionais são usadas para tomar decisões no código, executando certas instruções apenas quando determinadas condições são atendidas.

Condicional if

A estrutura condicional **if** é uma das ferramentas mais fundamentais em programação, permitindo que o código tome decisões com base em condições específicas. Em Python, a estrutura **if** é usada para executar um bloco de código se uma condição for verdadeira.

```
   idade = 18 # Inicializa o valor de idade com 18

   if idade >= 18: # Se idade for maior ou igual a 18, exibe 'Pode votar e obter a CNH'
       print('Pode votar e obter a CNH')  # Essa linha será executada porque idade é 18
```

A estrutura condicional **if-elif** em Python permite que você execute diferentes blocos de código com base em várias condições. O **if** é usado para a primeira condição, **elif** (abreviação de **else if**) para condições adicionais.

```
   idade = 17 # Inicializa o valor de idade com 17

   if idade >= 18: # Se idade for maior ou igual a 18, exibe 'Pode votar e obter a CNH'
        print('Pode votar e obter a CNH')  # Essa linha não será executada porque idade é 17
   elif idade >= 16 and idade < 18: # Se idade for maior ou igual a 16 e menor que 18, exibe 'Pode apenas votar'
        print('Pode apenas votar')  # Essa linha será executada
```

A estrutura condicional **else** em Python é usada em conjunto com **if** (e opcionalmente com **elif**) para executar um bloco de código quando nenhuma das condições anteriores for verdadeira. O **else** fornece um caminho padrão ou alternativo de execução quando todas as outras condições falharem, ou seja, quando todas as condicionais anteriores retornarem **False** para o teste.

```
   idade = 15 # Inicializa o valor de idade com 15

   if idade >= 18: # Se idade for maior ou igual a 18, exibe 'Pode votar e obter a CNH'
        print('Pode votar e obter a CNH')  # Essa linha não será executada
   elif idade >= 16 and idade < 18: # Se idade for maior ou igual a 16 e menor que 18, exibe 'Pode apenas votar'
        print('Pode apenas votar')  # Essa linha não será executada
   else:  # Se idade for menor que 16 (última possibilidade), exibe 'Não pode votar e obter a CNH'
        print('Não pode votar e obter a CNH')  # Essa linha será executada
```

Bloco match-case

A instrução **match** em Python é usada para correspondência de padrões, introduzida recentemente no Python 3.10. Ela é semelhante à instrução **switch** encontrada em outras linguagens de programação. De maneira simplificada, ela pode ser utilizada para testar valores para uma única variável (teste de igualdade **==**) e também permite o uso de operadores lógicos no teste.

```
dia = 'sexta'

match dia: # Variável a ser testada
    case "segunda": # Possível valor
        print('Início da semana')
    case "sexta": # Possível valor
       print('Quase fim de semana') # Será exibido 'Quase fim de semana'
    case "sábado" | "domingo": # Possível valor sábado ou domingo
        print('Fim de semana')
   case _ : # Valor padrão caso nenhum dos valores acima seja verdadeiro
        print('Dia comum')
```

4.2.2 Estruturas de Repetição

Conceito

Estruturas de repetição, também conhecidas como loop ou laço de repetição, permitem que um bloco de código seja executado várias vezes. Em Python, usamos **for** e **while** para criar essas estruturas.

Aplicações

Estruturas de repetição são usadas para iterar sobre coleções de dados ou executar um bloco de código, enquanto uma condição é verdadeira.

Repetição for

Para o uso básico do **for** em Python utilizamos a função **range()**. Ela gera um objeto iterável com um intervalo de números inteiros de **0** até o limite superior definido pelo parâmetro informado. A atualização da variável controladora i (variável que conta o número de repetições) é realizada por meio do operador **in**. Lembrando que em Python toda função ou indexação que utiliza intervalos possui o limite superior aberto, ou seja, itera até o **limite-1**. Veja o exemplo a seguir.

```
for i in range(5):  # Repete o bloco de código 5 vezes
    print(i)  # Exibe o valor de i a cada iteração, de 0 a 4
```

```
0
1
```

```
2
3
4
```

Também podemos definir o limite inferior para a função **range()**, ou seja, de qual número inteiro o intervalo será iniciado. Em Python, o padrão é **0** e esse limite inferior é fechado, ou seja, é considerado no início da contagem.

```
   for i in range(2,5):    # Repete o bloco de código 3 vezes, iniciando no 2 até o 4
       print(i)   # Exibe o valor de i a cada iteração, de 2 a 4
```

```
2
3
4
```

Adicionalmente podemos especificar o passo do intervalo gerado pela função **range()**. Para isso, especificamos um terceiro parâmetro indicando o "salto" da sequência.

```
   for i in range(1,10,2):     # Repete o bloco de código 5 vezes, iniciando no 1 até o 9, de 2 em 2
       print(i)   # Exibe 1 3 5 7 9
```

```
1
3
5
7
9
```

Também podemos utilizar o operador **in** para iterar em uma lista.

```
   animais = ['cachorro', 'gato', 'coelho', 'peixe'] # Criar uma lista de animais
   for animal in animais: # Itera na lista
     print(animal, end=', ') # Exibe cachorro, gato, coelho, peixe,
```

```
cachorro, gato, coelho, peixe,
```

Caso seja necessário enumerar os itens da lista que será iterada, podemos passá-la como parâmetro para a função **enumerate()**. Nesse caso, a função retorna um objeto

iterável de tuplas com dois valores que devem ser recebidos por, também, duas variáveis controladoras: uma do tipo **int** para receber a contagem e outra para o elemento da lista.

```
   animais = ['cachorro', 'gato', 'coelho', 'peixe'] # Criar uma lista de animais
   for i, animal in enumerate(animais): # Itera na lista
      print('O ' + str(i+1) + '° animal é o ' + animal + '.') # Exibe O 1° animal é o cachorro... O 2° animal é o gato... e assim por diante
```

O 1° animal é o cachorro.

O 2° animal é o gato.

O 3° animal é o coelho.

O 4° animal é o peixe.

No exemplo acima, somamos **i+1** (convertido para string para ser possível concatenar com outras strings) para dar o efeito de sequência ordinal sem alterar o valor da variável **i**, apenas com efeito de saída utilizando o **print()**.

Comprehensions

Compreensões (*comprehensions*) em Python são uma maneira concisa e eficiente de criar listas, conjuntos e dicionários. Elas permitem a construção de novas coleções a partir de iteráveis existentes de forma mais legível e compacta. Construímos essas *comprehensions* utilizando a estrutura de repetição for.

List comprehension

List comprehension é uma forma concisa de criar listas. A sintaxe básica é:

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

onde **expression** é a expressão que produz os elementos da lista, **item** é a variável que toma cada valor do iterable e **if condition** é opcional, usada para filtrar os elementos.

No exemplo a seguir, utilizaremos *list comprehension* para criar uma lista de quadrados de 0 a 9.

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

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Acima, **x**2** é a expressão que calcula o quadrado do item **x**, gerado a partir de um intervalo de números pelo iterável range**(10)**.

No exemplo a seguir, também geramos uma lista de **0** a **9** filtrando apenas os números pares por meio do **if**.

```
pares = [x for x in range(10) if x % 2 == 0] # O if filtra os valores de x cujo o resto da divisão (%) destes números por 2 resulta em 0
print(pares) # Exibe [0, 2, 4, 6, 8]
```

```
[0, 2, 4, 6, 8]
```

As variáveis utilizadas em uma operação de *list comprehension* (e outras de *dict comprehension* e *set comprehension* que veremos mais adiante) possuem validade (ou escopo, como também veremos mais adiante) apenas dentro da *comprehension*, não podendo ser acessadas fora desse contexto.

Dict comprehension

Dict comprehension é semelhante a *list comprehension*, mas cria um dicionário. A sintaxe básica é:

```
{key: value for item in iterable if condition}
```

Onde **key** é a chave e **value** é o valor no dicionário.

```
quadrados_dict = {x: x**2 for x in range(10)}
print(quadrados_dict) # Exibe {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

No exemplo acima, geramos um dicionário em que os valores de cada par são obtidos a partir dos valores das respectivas chaves elevados ao quadrado **x**2**.

```
pares_dict = {x: x**2 for x in range(10) if x % 2 == 0}
print(pares_dict) # Exibe {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

Nesse exemplo, adicionamos um filtro **if x % 2 == 0** para que fossem geradas apenas chaves com números pares (divisível naturalmente por 2).

Set comprehension

Set comprehension é semelhante a *dict comprehension*, mas cria um conjunto. A sintaxe básica é:

```
{expression for item in iterable if condition}
```

Veja, a seguir, os mesmos exemplos utilizados em *list comprehension*.

```
# Set comprehension para criar um conjunto de quadrados de 0 a 9
quadrados_set = {x**2 for x in range(10)}
print(quadrados_set)
```

```
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

```
# Set comprehension para criar um conjunto de números pares de 0 a 9
pares_set = {x for x in range(10) if x % 2 == 0}
print(pares_set)
```

```
{0, 2, 4, 6, 8}
```

Observe que como os conjuntos são objetos que não mantém a ordem dos elementos, os valores podem ser mostrados de maneira desordenada.

Repetição while

A estrutura **while** executa um teste condicional no início de cada repetição. As repetições são realizadas enquanto a condição retornar **True**. Isso quer dizer que o **while** permite executar repetidamente um bloco de código enquanto uma condição específica for verdadeira. É uma maneira eficiente de criar loops que continuam até que a condição definida seja falsa e, consequentemente, a repetição seja interrompida.

```
cont = 0 # Inicializa uma variável contadora por meio da atribuição do valor 0
while cont < 5: # Continua executando enquanto cont for menor que 5 (enquanto verdadeiro)
    print(cont)  # Exibe o valor de cont a cada iteração
    cont += 1  # Incrementa cont em 1 a cada iteração. Equivalente a cont = cont + 1
```

```
0
1
2
```

```
3
4
```

No exemplo acima, enquanto a condição **cont < 5** for verdadeira, o bloco de comandos do **while** a seguir será repetido. Em outras palavras, a repetição acontece enquanto a variável **cont** tiver seu valor menor que **5**. Quando o valor de **cont** atingir 5, o comando de repetição **while** será encerrado.

4.2.3 Instruções de Salto

Instrução break

A instrução **break** é usada para interromper a execução de um *loop* prematuramente. Quando o **break** é executado, o *loop* termina imediatamente. O comando break é util como recurso para garantir que uma estrutura de repetição não continue sendo executada, caso ocorra alguma situação. Normalmente, é utilizado em conjunto com uma condicional **if** no teste de situações para efetuar a tomada de decisão para uma parada repentina. Algumas dessas situações incluem:

- **Encerrar um loop ao encontrar um valor**: se você estiver procurando por um valor específico em uma lista, por exemplo, pode usar o **break** para sair do *loop* assim que encontrar o valor desejado, evitando iterações desnecessárias.
- **Sair de um loop infinito**: *loops* infinitos são úteis em certas situações, como servidores que aguardam conexões. O **break** pode ser usado para sair do *loop* quando uma condição de parada for atendida.

```
for i in range(10):  # Repetiria de 0 a 9 se não houvesse o break
    if i == 5:  # Condição para interromper o loop
        break  # Interrompe o loop quando i é 5
    print(i)  # Exibe os valores de 0 a 4
```

Instrução continue

A instrução **continue** é usada para pular a iteração atual de um *loop* e continuar com a próxima iteração. Enquanto o **break** interrompe todas as próximas iterações após sua ocorrência, o **continue** passa para a próxima iteração, ignorando o código subsequente da iteração atual após a sua ocorrência.

```python
for i in range(10):  # Repete de 0 a 9
    if i % 2 == 0:  # Condição para pular a iteração
        continue # Pula a iteração quando i é par não executando os códigos abaixo dessa linha
    print(i)  # Exibe os valores ímpares de 1, 3, 5, 7 e 9
```

```
1
3
5
7
9
```

4.2.4 Outras Instruções de Controle

Além do **break** e do **continue**, existem outras instruções que podem ser utilizadas para dar fluidez e maior controle na implementação de códigos, mas que não são aplicadas exclusivamente em estruturas de repetição, podendo ser usadas em outros contextos.

Instrução pass

A instrução **pass** é um *placeholder* (espaço reservado). É usada quando uma instrução é necessária sintaticamente, mas não há necessidade de nenhuma ação ou código. Seu uso não é comum, mas necessário em algumas situações, principalmente quando precisamos estruturar um bloco de código, mas não temos os códigos que serão inseridos nesse bloco; nesse caso, **pass** evita um **SyntaxError** de um bloco vazio e permite uma implementação futura à medida que o código amadurece.

```python
for i in range(10):  # Repete de 0 a 9
    if i % 2 == 0:
        pass # Futura implementação
    else:
        pass # Futura implementação
```

4.2.5 Estrutura try-except

A estrutura **try-except** é usada para capturar e tratar exceções em Python. Isso permite que o programa lide com erros de maneira controlada. O uso dessa estrutura é util quando o programador consegue prever possíveis erros de execução e deseja manter a execução do algoritmo. A maioria desses erros é proveniente de entradas realizadas por usuários ou outros sistemas que não podem ser controladas. Portanto, é uma estrutura que também pode ser utilizada para validação de dados de entrada.

No exemplo a seguir, recebemos um número digitado pelo usuário e usamos esse valor como divisor em uma operação de divisão. Caso o valor informado pelo usuário seja **0**, precisamos controlar o erro gerado por uma divisão por zero; caso contrário, mostramos o resultado. No **except** não informamos o tipo de erro que estamos monitorando, portanto, na ocorrência de qualquer erro, o **expect** será executado.

num = int(input('Informe um número inteiro diferente de 0 para efetuar a operação de divisão:')) # Digite 0 para simular o erro

```
num = int(input('Informe um número inteiro diferente de 0 para efetuar a operação de divisão:')) # Digite 0 para simular o erro

try:
    result = 10 / num  # Essa linha causará um erro se o usuário informar 0 para num
    print(result)
except:  # Em caso de QUALQUER TIPO DE ERRO
    print('Erro: divisão por zero não é permitida.') # Exibe 'Erro: divisão por zero não é permitida.' se usuário informar 0
```

```
Informe um número inteiro diferente de 0 para efetuar a operação de divisão:a
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-ace9b6e7fd2e> in <cell line: 1>()
----> 1 num = int(input('Informe um número inteiro diferente de 0 para efetuar a operação de divisão:')) # Digite 0 para simular o erro
      2
      3 try:
      4     result = 10 / num  # Esta linha causará um erro se o usuário informar 0 para num
      5     print(result)
ValueError: invalid literal for int() with base 10: 'a'
```

Além de podermos tratar erros genéricos, como no exemplo acima, podemos especificar o que deve ser feito na ocorrência de um erro específico para cada caso. No exemplo a seguir, o **except** somente será executado se o **try** reportar uma classe de erro **ZeroDivisionError**. Qualquer outra classe de erro será ignorada, gerando uma exceção.

Você pode consultar todas as classes de erros em https://docs.python.org/3/library/exceptions.html.

```
   num = int(input('Informe um número inteiro diferente de 0 para efetuar
a operação de divisão:')) # Digite 0 para simular o erro

   try:
      result = 10 / num  # Essa linha causará uma exceção ZeroDivisionError
se o usuário informar 0 para num
       print(result)
   except ZeroDivisionError:  # Captura a exceção ZeroDivisionError
        print('Erro: divisão por zero não é permitida.') # Exibe 'Erro:
divisão por zero não é permitida.' se usuário informar 0
```

```
Informe um número inteiro diferente de 0 para efetuar a operação de divisão:a
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-afca73497f5f> in <cell line: 1>()
----> 1 num = int(input('Informe um número inteiro diferente de 0 para efetuar a
operação de divisão:')) # Digite 0 para simular o erro
      2
      3 try:
      4     result = 10 / num  # Esta linha causará uma exceção ZeroDivisionError se o
usuário informar 0 para num
      5     print(result)
ValueError: invalid literal for int() with base 10: 'a'
```

Observe que no exemplo acima fizemos o casting **int(input(...)**, pois todos os valores digitados pelo usuário são recebidos pelo comando **input()** como uma string. Sempre ao receber um valor numérico, deve ser realizada a conversão antes de realizar operações.

Em outro exemplo, a seguir, tentamos abrir um arquivo inexistente, gerando um erro da classe **FileNotFoundError**.

```
   try:
       with open('arquivo_inexistente.txt', 'r') as file:  # Tentativa de
abrir um arquivo inexistente
           content = file.read()
   except FileNotFoundError:  # Captura a exceção FileNotFoundError
        print('Erro: arquivo não encontrado.') # Exibe 'Erro: divisão por
zero não é permitida.'
```

Mesmo não especificando a classe de erro no **except**, podemos capturá-la por meio do **Exception as e**, armazendo o texto do erro no alias **e** para exibí-lo, caso seja necessário.

```
   num = int(input('Informe um número inteiro diferente de 0 para efetuar a operação de divisão:')) # Digite 0 para simular o erro


   try:
       result = 10 / num  # Essa linha causará um erro se o usuário informar 0 para num
        print(result)
   except Exception as e:  # Captura e armazena o texto do erro no Exception e utilizamos o alias e para exibir no print abaixo
         print('O script retornou o seguinte erro:', e) # Exibe 'O script retornou o seguinte erro: division by zero'
```

```
Informe um número inteiro diferente de 0 para efetuar a operação de divisão:0
O script retornou o seguinte erro: division by zero
```

Dentro de um **try-except** podemos encadear várias classes de erros e definir ações para cada uma delas, especificando um **except** para cada erro.

```
   try:
     num = int(input('Informe um número inteiro diferente de 0 para efetuar a operação de divisão:')) # Digite 0, ou um float ou um texto para simular o erro
      result = 10 / num   # Essa linha causará um ZeroDivisionError se o usuário informar 0 para num ou ValueError se a entrada não for um número inteiro
     print(result)
   except ValueError: # Caso a entrada não seja um número inteiro resultará na exceção ValueError
     print("Erro: Informe um número inteiro.")
   except ZeroDivisionError:  # Caso ocorra erro no cálculo por divisão por 0 resultará na exceção ZeroDivisionError
     print('Erro: Informe um número inteiro maior que 0.')
```

Para finalizar, o **try-catch** permite o uso da claúsulas **else** e **finally**.

O comando **else** é executado quando não ocorre nenhum erro, enquanto o **finally** é executado independentemente do resultado do **try-catch**, ou seja, sempre.

```
   try:
     num = int(input('Informe um número inteiro diferente de 0 para efetuar a operação de divisão:')) # Digite 0, ou um float ou um texto para simular o erro
      result = 10 / num   # Essa linha causará um ZeroDivisionError se o usuário informar 0 para num ou ValueError se a entrada não for um número inteiro
   except ValueError: # Caso a entrada não seja um número inteiro resultará na exceção ValueError
     print("Erro: Informe um número inteiro.")
   except ZeroDivisionError:  # Caso ocorra erro no cálculo por divisão por 0 resultará na exceção ZeroDivisionError
     print('Erro: Informe um número inteiro diferente de 0.')
   else: # Caso não ocorra nenhum erro
     print('O resultado é ', result) # Colocamos o print do resultado aqui visualizarmos o else executando
   finally:
     print('Fim!')
   #print('Fim!') poderia ser escrito aqui com o mesmo efeito. Observem que pela indentação, esse comando está fora do try-catch
```

Como o **finally** é executado de qualquer forma, sintaticamente, ele é opcional e serve apenas para manter a coerência de contexto dos códigos, ou seja, apenas para receber códigos referentes ao contexto do **try-catch** que devem ser executados ao final do bloco.

4.3 Saiba Mais...

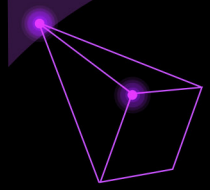
» Tutorial sobre **estruturas de controle de fluxo**.

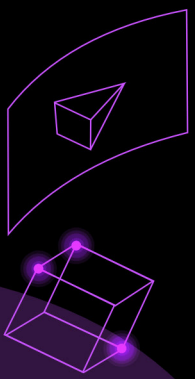
» **Classes de erros** para tratamento de exceções em Python.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Introdução ao *Python*

Unidade V
Funções





Unidade V - Funções

5.1 Notebook Colab: Funções

A Seção 5.1 também está disponível no *Notebook Colab*.

5.1.1 Conceitos Básicos

Funções são blocos de código que executam uma tarefa específica e podem ser reutilizadas em diferentes partes do programa. Elas ajudam a tornar o código mais modular e fácil de entender.

A própria linguagem Python já traz consigo um grande número de funções *built-in* em sua instalação mínima, como vimos nas Unidades anteriores. Também é possível ampliar a disponibilidade dessas funções instalando no ambiente e importando outras bibliotecas.

Entretanto, à medida que avançamos nos estudos de uma linguagem, torna-se necessário implementarmos nossas próprias funções com o objetivo de reaproveitarmos nossos códigos em outros projetos e acelerar o desenvolvimento de *scripts*.

Quanto ao tipo, podemos criar funções nomeadas e anônimas. Uma vez criadas, as funções nomeadas podem ser utilizadas em qualquer lugar, inclusive outros projetos: basta importá-las e chamá-las. Já as funções anônimas, também conhecidas no Python como funções *lambda*, não possuem nomes (identificadores) e são criadas e executadas apenas no código em que elas foram definidas. Apesar de parecer um pouco estranho criar funções anônimas que não podem ser utilizadas em outro lugar, no Python, tanto é considerada uma boa prática, quanto útil em algumas situações que veremos adiante, como para filtragem e mapeamento.

Um exemplo do uso de funções é apresentado a seguir, no qual usamos a função **max()** para obter o valor máximo de uma lista de alturas. Essa função é incluída na instalação mínima do Python (*built-in*), recebe como parâmetro uma lista e retorna o maior valor dessa lista.

```
alturas = [1.5, 1.67, 1.47, 1.79, 1.69, 1.87]
alt_max = max(alturas)
print('A maior altura é', alt_max) # Exibe 'A maior altura é 1.87'
```

A maior altura é 1.87

Vamos aprender agora a criar e utilizar nossas próprias funções.

5.1.2 Funções Nomeadas

As funções nomeadas tanto podem ou não receber parâmetros quanto podem ou não retornar valores. Tudo irá depender das necessidades do seu projeto. Entretanto, funções com passagem de parâmetros e retorno de valores são as mais comuns e úteis para serem reaproveitadas.

Usamos o comando **def** para declarar uma função, seguido do identificador e um par de parênteses que receberão os parâmetros.

Funções sem parâmetros e sem retorno

São funções que não recebem nenhum parâmetro e não retornam nenhum valor; apenas executam uma série de instruções.

```
def saudacao(): # Declara uma função nomeada saudacao
  print("Olá, seja bem-vindo!")
```

```
# Chama a função
saudacao() # Exibe 'Olá, seja bem-vindo!'
```

Olá, seja bem-vindo!

Executar o código acima sem executar a célula em que declaramos a função resultará em um **NameError**, pois o Python não conhece essa função. Caso a função seja alterada, é necessário executar novamente sua célula para atualizá-la. Entretanto, uma vez declarada, você pode chamá-la em qualquer parte do código, conforme demonstrado a seguir.

```
# Chama novamente a função
saudacao() # Exibe 'Olá, seja bem-vindo!'
```

Olá, seja bem-vindo!

Observe que, mesmo não recebendo nenhum parâmetro, devemos manter o par de parênteses vazio, tanto na declaração quanto na chamada. Mais adiante aprenderemos como criar funções recebendo parâmetros. No caso do exemplo acima, a função não possui um retorno mas o comando **print** é usado como saída da função. Também veremos, mais adiante, como retornar valores a partir de uma função.

Outro recurso importante quando criamos nossas próprias funções é documentá-las para futuras consultas, tanto nossas quanto de outros programadores. Na documentação,

incluímos informações básicas de finalidade da função, parâmetros de entrada e retorno. Utilizamos o recurso de comentários de mais de uma linha, colocando as informações entre ''' e ''' ou """ e """.

```
def saudacao():
  '''
  Exibe uma mensagem de boas vindas.

  Parâmetros:
  Nenhum

  Retorno:
  Nenhum
  '''
  print("Olá, seja bem-vindo!") # Exibe 'Olá, seja bem-vindo!'
```

Executando a célula acima, "atualizamos" o código da função incluindo a documentação dela, ou seja, o **docstring**[1]. Agora, ao chamarmos a função sem os parênteses e, consequentemente, sem a passagem de parâmetros é exibido o **docstring**.

[1] *Docstrings* são *strings* de documentação utilizadas em Python para documentar módulos, funções, classes e métodos. Elas fornecem uma maneira de descrever o quê o código faz, seus parâmetros, valores de retorno e outras informações relevantes. *Docstrings* são importantes para criar código legível e bem documentado.

```
saudacao # Exibe apenas o docstring
```

Funções sem parâmetros e com retorno

Funções que retornam um valor ao final de sua execução por meio do comando **return**. Esse valor pode ser armazenado em uma variável e/ou usado em outras partes do programa.

Exemplo:

```
def soma(): # Função que retorna a soma de dois números
  result = 10 + 6
  return result
```

```
    # Chama a função e armazena o resultado
    resultado = soma() # Recebe o resultado
    print('O resultado da soma é:', resultado) # Exibe 'O resultado da soma
é: 16'
```

O resultado da soma é: 16

Observe que, como a função acima possui um retorno, precisamos armazenar esse retorno ou em uma variável, no caso **result**, ou exibir na tela, passar como parâmetro de outras funções, etc. O importante é lembrar que todo retorno precisa ser esperado.

Funções com parâmetros e com retorno

Funções que recebem valores (parâmetros) para realizar suas operações e retornam um ou mais resultados. Estes parâmetros são definidos na declaração da função.

Exemplo:

```
    def saudacao _ personalizada(nome): # Declara uma função que recebe um
parâmetro nome
        return f"Olá, {nome}, seja bem-vinda!" # Retorna uma f-string com o
parâmetro recebido
```

```
    mensagem = saudacao _ personalizada("Maria") # Chama a função com um
parâmetro e armazena o retorno
    print(mensagem) # Será exibido 'Olá, Maria, seja bem-vinda!'
```

Olá, Maria, seja bem-vinda!

Em Python, as funções podem retornar mais de um valor. Para isso, é obrigatório que para cada retorno exista uma variável para recebê-lo.

A função a seguir recebe dois números e retorna o resultado das quatro operações básicas.

```
    def operacoes(n1, n2):
        '''
        Calcula a soma, subtração, multiplicação e divisão de 2 números

        Parâmetros:
        n1, n2 (num): 2 números

        Retorno:
        soma, sub, div, mult (num): resultado das 4 operações básicas
        '''
```

```
    soma = n1 + n2
    sub = n1 - n2
    div = n1 / n2
    mult = n1 * n2

    return soma, sub, div, mult
```

```
num1 = float(input('Digite o 1° número:'))
num2 = float(input('Digite o 2° número:'))
resul1, resul2, resul3, resul4 = operacoes(num1, num2) # As variáveis resul1, resul2, resul3 e resul4 recebem os 4 retornos da função, nessa ordem.
print('Soma:', resul1)
print('Subtração:', resul2)
print('Divisão:', resul3)
print('Multiplicação:', resul4)
```

```
Digite o 1° número:10
Digite o 2° número:5
Soma: 15.0
Subtração: 5.0
Divisão: 2.0
Multiplicação: 50.0
```

Observe, no exemplo acima, que, para as operações de subtração e divisão, a ordem dos operandos altera o resultado. Isso acontece, pois a função espera que os parâmetros sejam passados na mesma ordem que foram definidos na declaração, ou seja, primeiro **n1** e depois **n2**.

Podemos flexibilizar essa ordem nomeando os parâmetros assim que realizamos a chamada na função. Para isso precisamos acessar o **docstring** da função para conhecer os nomes dos parâmetros esperados e, assim, passar o par **parâmetro = valor** para cada entrada esperada.

```
operacoes # Exibe o docstring da função
```

```
num1 = float(input('Digite o 1° número:'))
num2 = float(input('Digite o 2° número:'))
resul1, resul2, resul3, resul4 = operacoes(n2 = num2, n1 =num1) # Chama a função com
```

```
    os parâmetros nomeados passando primeiro n2 e depois n1
    print('Soma:', resul1)
    print('Subtração:', resul2)
    print('Divisão:', resul3)
    print('Multiplicação:', resul4)
```

Também podemos declarar funções sem um número definido de parâmetros, passando quantos valores forem necessários por posição ou nomeados.

Parâmetros passados por posição são recebidos na função como tuplas. Enquanto os parâmetros passados nomeados são tratados como dicionários.

Ainda podemos usar uma abordagem híbrida, declarando funções que recebem parâmetros fixos e variáveis.

A seguir, temos uma função que recebe uma quantidade ilimitada de números e retorna o somatório desses. Para isso, declaramos a função com um parâmetro **args** precedido de um asterisco *****. Esse asterisco indica que será recebida uma tupla de valores que podem ser iterados.

```
    def somatorio(*args): # É uma convenção usar o args para receber parâmetros posicionais mas você pode escolher qualquer outro identificador.
        soma = 0
        for num in args:
            soma += num

        return soma
```

```
    print(somatorio(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)) # Exibe 45
```

```
45
```

Agora criamos uma função bem simples, sem retorno (embora, a depender do contexto, você pode criar funções desse tipo com retorno) que recebe um conjunto de parâmetros nomeados em forma de dicionário. Para isso, declaramos a função com um parâmetro **kwargs** (também uma convenção que pode ser alterada) precedido de dois asteriscos ******. Esses asteriscos indicam que será recebido um conjunto chave-valor de parâmetros nomeados.

```
    def somatorio(**kwargs):
        soma = 0
        for var, valor in kwargs.items(): # Por ser um dicionário utilizamos o método items() para acessar o par chave-valor a cada iteração e retornar para o par var-valor respectivamente
```

```
        soma += float(valor)
        print('A variável', var, 'tem valor', valor, 'e a soma parcial é',
soma)
```

```
    somatorio(n1=1, n2=3, n3=5, n4=7, n5=9, n6=11, n7=13, n8=15)
```

```
A variável n1 tem valor 1 e a soma parcial é 1.0
A variável n2 tem valor 3 e a soma parcial é 4.0
A variável n3 tem valor 5 e a soma parcial é 9.0
A variável n4 tem valor 7 e a soma parcial é 16.0
A variável n5 tem valor 9 e a soma parcial é 25.0
A variável n6 tem valor 11 e a soma parcial é 36.0
A variável n7 tem valor 13 e a soma parcial é 49.0
A variável n8 tem valor 15 e a soma parcial é 64.0
```

Por fim, um exemplo de abordagem híbrida utilizando parâmetros fixos e posicionais (também poderiam ser nomeados).

```
    def soma(n, *numeros): # Recebe o 1° parâmetro em n e os demais (tupla)
em numeros
      print(n)
      print(numeros)
      for num in numeros:
        print(n,' + ', num, ' = ', n + num)
```

```
    soma(5, 1, 2, 3, 4, 5)
```

```
5
(1, 2, 3, 4, 5)
5 + 1 = 6
5 + 2 = 7
5 + 3 = 8
5 + 4 = 9
5 + 5 = 10
```

5.1.3 Funções *lambda*

Funções *lambda* são pequenas funções anônimas definidas usando a palavra-chave **lambda**. Elas podem ter múltiplos argumentos, mas apenas uma expressão. As funções *lambda* são úteis quando você precisa de uma função simples por um curto período de tempo e que não será reutilizada em outro lugar que não seja o código atual.

A sintaxe de uma função *lambda* é:

```
lambda argumentos: expressão
```

Em que **argumentos** são os parâmetros de entrada da função, posicionados antes dos dois pontos **:** na sintaxe, e **expressão** é o processamento realizado com os parâmetros, retornando um resultado, podendo ser um cálculo ou um teste lógico, por exemplo.

Para aprender mais sobre funções lambda, acesse https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/.

Vamos criar uma função *lambda* que realize a soma de dois números.

```
soma = lambda x, y: x + y # Define uma função lambda que soma dois números
```

```
resultado = soma(10, 5) # Chama a função lambda e armazena o resultado
print(f"O resultado da soma é: {resultado}") # Exibe 'O resultado da soma é: 15'
```

O resultado da soma é: 15

Funções lambda com map()

A função **map()** aplica uma função (anônima ou nomeada) a todos os itens de um iterável (como uma lista) e retorna um iterador (que pode ser convertido em uma lista, por exemplo).

Exemplo:

```
numeros = [1, 2, 3, 4, 5] # Lista de números
dobrados = map(lambda x: x * 2, numeros) # Usa map() para dobrar cada número na lista
print(list(dobrados)) # Converte o mapa em uma lista e exibe [2, 4, 6, 8, 10]
```

[2, 4, 6, 8, 10]

Funções *lambda* com filter()

A função **filter()** cria uma lista de elementos para os quais uma função retorna verdadeiro.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # Lista de números
pares = filter(lambda x: x % 2 == 0, numeros) # Usa filter() para obter números pares. A expressão x % 2 == 0 é usada para avaliar cada valor e filtrar apenas para True
print(list(pares)) # Converte o filtro em uma lista e exibe [2, 4, 6, 8, 10]
```

[2, 4, 6, 8, 10]

5.1.4 Escopo de Variáveis

Agora que aprendemos a criar nossas próprias funções e sabendo que essas funções podem ter suas próprias variáveis, é importante compreender o escopo de criação das variáveis.

O escopo de uma variável refere-se à região do código onde a variável é reconhecível. Variáveis em Python podem ter diferentes níveis de escopo, o que determina sua acessibilidade dentro do programa. Existem principalmente dois tipos de escopo:

- Escopo global
- Escopo local

Escopo global

Uma variável inicializada fora de qualquer função ou bloco tem escopo global e pode ser acessada de qualquer parte do código. Em linguagens de programação que exigem a declaração de variáveis informando seu tipo, a definição do escopo é definida no momento da declaração.

O código, a seguir, exemplifica a criação de uma variável **x = 10** no escopo global e sua exibição dentro e fora de uma função.

```
x = 10

def imprimir_valor():
  print(f"Valor de x dentro da função: {x}") # Exibe 'Valor de x dentro da função: 10'

imprimir_valor()
print(f"Valor de x fora da função: {x}") # Exibe 'Valor de x fora da função: 10'
```

Valor de x dentro da função: 10

Valor de x fora da função: 10

Observe que **x** tem o mesmo valor nos diferentes escopos.

Escopo local

Uma variável declarada dentro de uma função tem escopo local e só pode ser acessada dentro dessa função.

No exemplo a seguir, **y = 10** dentro da função **imprimir_valor()**. Ao tentarmos mostrar seu valor fora do escopo local, resultará em um **NameError**. Utilizamos o exemplo anterior alterando apenas o escopo de inicialização da variável para local (dentro da função) e o nome de variável para **y**. A alteração do nome da variável para **y** foi necessária, pois, no contexto de execução do Google Colab, as variáveis inicializadas em outras células são acessíveis em todo o *notebook* durante a mesma sessão de execução.

```
  def imprimir _ valor():
    y = 10
    print(f"Valor de y dentro da função: {y}") # Exibe 'Valor de y dentro
da função: 10'

  imprimir _ valor()
  print(f"Valor de y fora da função: {y}") # Exibe NameError: name 'y' is
not defined
```

```
Valor de y dentro da função: 10
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-7a39256ae0c9> in <cell line: 6>()
      4
      5 imprimir_valor()
----> 6 print(f"Valor de y fora da função: {y}") # Exibe
NameError: name 'y' is not defined
```

A seguir, declaramos uma variável **z = 1** no escopo global e **z = 5** no escopo local. Por se tratar de escopos diferentes, **z** fora e dentro da função são objetos diferentes e independentes. Caso queira demonstrar, descomente as linha que mostram o endereço delas na memória.

```
  z = 1
  #print(hex(id(z))) # Exibe o endereço de memória ocupado por z global

  def imprimir _ valor():
    z = 5
    #print(hex(id(z))) # Exibe o endereço de memória ocupado por z local
    print(f"Valor de z dentro da função: {z}") # Exibe 'Valor de z dentro
da função: 5'
```

```
    imprimir_valor()
    print(f"Valor de z fora da função: {z}") # Exibe 'Valor de z fora da
função: 1'
```

Valor de z dentro da função: 5

Valor de z fora da função: 1

Para sinalizarmos ao Python que variáveis com o mesmo identificador se tratam do mesmo valor em escopos diferentes, usamos o comando **global** antecedendo a variável no escopo local. No código a seguir, alteramos o escopo de **z** de local para global e demonstramos por meio do **print** dos valores. Observem que ainda assim as duas variáveis são objetos diferentes (ocupam endereços de memória diferentes) mas estão lincadas nos escopos local e global.

```
    z = 1
    print(hex(id(z))) # Exibe o endereço de memória ocupado por z global

    def imprimir_valor():
        global z # Informa ao Python que qualquer referência à variável
z dentro da função se refere à uma outra variável no contexto global
(diferente de z declarada no início)
        z = 5
        print(hex(id(z))) # Exibe o endereço de memória ocupado por z dentro
da função mas que está vinculado ao valor de z global
        print(f"Valor de z dentro da função: {z}") # Exibe 'Valor de z dentro
da função: 5'

    imprimir_valor()
    print(f"Valor de z fora da função: {z}") # Exibe 'Valor de z fora da
função: 5'
```

0x7ddce430c0f0

0x7ddce430c170

Valor de z dentro da função: 5

Valor de z fora da função: 5

Perceba que mesmo alterando o escopo de **z** declarada dentro da função para global, ela ainda é um objeto diferente de **z** declarada antes da função. A princípio, poderíamos imaginar que o comando **global** indica que são as mesmas variáveis. Entretanto, o que ocorre, é apenas uma ligação entre elas para duplicar os valores. Na prática, isso significa que temos duas posições diferentes de memória com o mesmo identificador **z** mas que estão lincadas de forma que, ao alterar o valor de uma, automaticamente o valor da outra será alterado.

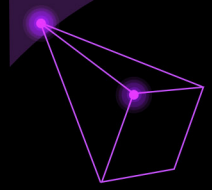
5.2 Saiba mais...

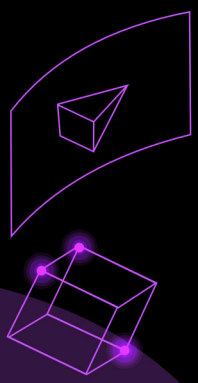
» **Funções anônimas** em Python.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Introdução ao *Python*

Unidade VI
Manipulação de Arquivos





Unidade VI - Manipulação de Arquivos

6.1 Notebook Colab: Manipulação de Arquivos

A Seção 6.1 está disponível no *Notebook Colab*.

6.1.1 Conceitos Básicos

Manipulação de arquivos é uma parte fundamental da programação, permitindo que os programas leiam e escrevam dados em arquivos armazenados no disco. Isso permite a persistência das informações após o processamento dos dados.

Python oferece várias funções e métodos para abrir, ler, escrever e fechar arquivos. Vamos explorar esses conceitos básicos e algumas operações comuns em exemplos práticos.

No Google Colab, todos os arquivos salvos ficam em um disco temporário que é montado sempre que um ambiente de execução é iniciado e ficam acessíveis enquanto esse estiver conectado. Entretanto, após ser encerrado, os arquivos são excluídos.

Antes de iniciarmos os exemplos, vamos aprender a montar o nosso Google Drive para manter os arquivos salvos mesmo após o encerramento do ambiente de execução.

```
from google.colab import drive

# Montar o Google Drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

O código acima utiliza a classe **drive** da biblioteca **google.colab** para montar o Google Drive da conta logada no caminho /content/drive.

Por ser um sistema *linux-based*, as barras para a direita **/** são utilizadas para separar os diretórios.

Esse código requer autorização para que o Google Colab acesse todos os seus arquivos do Google Drive. Cuidado ao dar essa permissão para *notebooks* criados por terceiros e que você não conhece os códigos.

Após a montagem do Google Drive, podemos acessar **Arquivos** no lado esquerdo e abrir **MyDrive**. Agora, qualquer arquivo que seja salvo ou acessado no Google Drive estará disponível no caminho /content/drive/MyDrive/<caminho do arquivo>.

Neste Microcurso, não abordaremos a manipulação de arquivos binários, mas, sim, apenas de arquivos texto (.txt).

6.1.2 Operações com Arquivos

Mostrar o conteúdo de um arquivo

O uso da palavra-chave **with** em Python é uma forma de gerenciar contexto que simplifica o trabalho com recursos que precisam ser adquiridos e, eventualmente, liberados. Em operações de arquivo, o **with** garante que o arquivo será devidamente fechado após a conclusão das operações, mesmo que uma exceção seja levantada durante a execução do bloco.

Existem outras formas de se abrir um arquivo para manipulação mas, entendendo que o **with** traz a segurança e a estabilidade desejadas para a execução dessa tarefa, essas outras formas não serão abordadas no presente Microcurso.

```
def mostrar_conteudo(arquivo):
    with open(arquivo, 'r') as file:
        conteudo = file.read()
        print(conteudo)
```

No código acima:

- **with** inicia um bloco de contexto onde o arquivo aberto será gerenciado.
- **open(arquivo, 'r')** abre o arquivo especificado em modo de leitura ('r').
- **as file** atribui o objeto de arquivo aberto à variável **file** (pode ser escolhido outro nome ou identificador).
- **file.read()** lê todo o conteúdo do arquivo e o armazena na variável **conteudo**.

Antes de executar o código a seguir, clique aqui para baixar o arquivo **words.txt** e salve-o na pasta raiz (fora que qualquer diretório) do Google Drive vinculado à conta que você está utilizando para acessar esse *notebook*.

```
mostrar_conteudo('/content/drive/MyDrive/words.txt')
```

Quando o bloco de código dentro do **with** é concluído, o arquivo é automaticamente fechado. Isso acontece mesmo se uma exceção for levantada dentro do bloco. O método

__exit__ do objeto de arquivo é chamado, garantindo que o recurso seja liberado corretamente.

No exemplo sem **with**, precisamos garantir manualmente que **file.close()** seja chamado, mesmo que uma exceção ocorra. Usar **with** torna o código mais conciso e seguro, automatizando a gestão do recurso. Veja, a seguir, como seria o mesmo código sem o **with**.

```
def mostrar_conteudo(arquivo):
    file = open(arquivo, 'r')
    conteudo = file.read()
    file.close() # Necessário fechar o arquivo após ler o conteúdo
    print(conteudo)
```

Além do modo de abertura de leitura **r** utilizado no exemplo anterior, ainda existem:

- **'w' (write)**: abre o arquivo para escrita. Se o arquivo já existir, ele será truncado (todo o conteúdo será apagado). Se o arquivo não existir, ele será criado.
- **'a' (append)**: abre o arquivo para escrita em modo de adição. Os dados são escritos no final do arquivo sem truncá-lo. Se o arquivo não existir, ele será criado.

Escrever em um arquivo

No exemplo a seguir, **file.write(conteudo)** escreve a *string* **conteudo** no arquivo aberto. A função **write** aceita uma *string* como argumento e escreve essa *string* no arquivo. O cursor de escrita é movido para o final do texto, pronto para a próxima operação de escrita, se houver.

```
def criar_e_escrever(arquivo, conteudo):
    with open(arquivo, 'w') as file:
        file.write(conteudo)
```

```
criar_e_escrever('/content/drive/MyDrive/hello.txt', 'Olá, mundo!')
```

```
mostrar_conteudo('/content/drive/MyDrive/hello.txt')
# Função já declarada anteriormente. Exibe 'Olá, mundo!'
```

Olá, mundo!

Escrever uma lista em um arquivo

Vamos criar uma lista de *strings* e escrever cada item em um arquivo como uma linha separada. Usamos o **for** para iterar sobre a lista e, para cada item, o escrevemos no arquivo concatenado com uma quebra de linha **\n**.

```
def lista_para_arquivo(arquivo, lista):
    with open(arquivo, 'w') as file:
        for item in lista:
            file.write(item + '\n')
```

```
frutas = ['caqui', 'manga', 'banana', 'maçã', 'uva', 'banana', 'mexerica']
lista_para_arquivo('/content/drive/MyDrive/frutas.txt', frutas)
```

```
mostrar_conteudo('/content/drive/MyDrive/frutas.txt')
# Exibe caqui manga banana maçã uva banana mexerica
```

caqui
manga
banana
maçã
uva
banana
mexerica

Armazenar as linhas de um arquivo em uma lista

A função **readlines** em Python é usada para ler todas as linhas de um arquivo e armazená-las em uma lista, onde cada elemento da lista é uma linha do arquivo.

```
def linhas_para_lista(arquivo):
    with open(arquivo, 'r') as file:
        linhas = file.readlines()
    return linhas # Exibe ['caqui\n', 'manga\n', 'banana\n', 'maçã\n', 'uva\n', 'banana\n', 'mexerica\n']
```

```
linhas = linhas_para_lista('/content/drive/MyDrive/frutas.txt')
print(linhas)
```

['caqui\n', 'manga\n', 'banana\n', 'maçã\n', 'uva\n', 'banana\n', 'mexerica\n']

Verificar se um arquivo está em uso

Verificar se um arquivo está em uso antes de manipulá-lo é uma prática importante para evitar conflitos, corrupção de dados e erros inesperados. A seguir, são listadas algumas razões sobre a importância dessa boa prática:

1. **Prevenção de conflitos**: quando múltiplos processos ou *threads* tentam acessar o mesmo arquivo simultaneamente, podem ocorrer conflitos. Por exemplo, um processo pode tentar ler um arquivo enquanto outro está escrevendo nele, levando a inconsistências e a resultados imprevisíveis.
2. **Evitar corrupção de dados**: se dois processos tentarem escrever no mesmo arquivo ao mesmo tempo, os dados podem ser corrompidos. Isso ocorre porque as operações de escrita não são atômicas e os dados de um processo podem sobrescrever ou misturar-se com os dados de outro.
3. **Garantir consistência dos dados**: garante que os dados sejam lidos ou escritos de maneira consistente. Isso é especialmente importante em aplicativos que dependem de dados precisos e atualizados.
4. **Evitar erros e exceções**: tentativas de acessar um arquivo que já está sendo usado por outro processo podem resultar em exceções como **FileNotFoundError**, **PermissionError** ou **OSError**. Verificar se o arquivo está em uso ajuda a evitar esses erros e permite que o programa lide com a situação de maneira previsível.
5. **Segurança dos dados**: Em ambientes multiusuário, a manipulação simultânea de arquivos pode levar a problemas de segurança, como vazamento de dados ou acesso não autorizado.

Exemplos de cenários aplicáveis

- Sistemas de backup: durante a execução de um backup, é fundamental garantir que os arquivos não estejam sendo modificados para evitar a cópia de dados corrompidos.
- Aplicações bancárias: garantir que transações financeiras sejam registradas corretamente sem interferência de outros processos.
- Editores de texto: ao salvar alterações em um documento, é importante garantir que o arquivo não esteja sendo acessado por outro editor ou processo.

O código a seguir verifica se um arquivo está em uso tentando renomear o arquivo para ele mesmo. A lógica por trás dessa verificação é baseada no comportamento do sistema operacional ao manipular arquivos que estão sendo acessados por outros processos.

```python
import os

def arquivo_em_uso(arquivo):
    try:
        os.rename(arquivo, arquivo)
        return False
    except OSError:
        return True
```

O comando **os.rename(arquivo, arquivo)** da biblioteca **os** tenta renomear o arquivo para ele mesmo. Essa operação é uma forma indireta de verificar se o arquivo está em uso, pois a maioria dos sistemas operacionais não permite renomear um arquivo que está sendo acessado por outro processo.

Se o arquivo estiver em uso por outro processo, a tentativa de renomeação falhará e retornará um **OSError**. O **OSError** captura uma ampla gama de erros relacionados ao sistema operacional, incluindo erros de arquivos e diretórios.

Se a tentativa de renomear for bem sucedida (ou seja, o arquivo não está em uso), a função retorna **False**. Se a tentativa falhar e resultar em um **OSError**, a função capturará a exceção e retornará **True**, indicando que o arquivo está em uso.

```python
print(arquivo_em_uso('/content/drive/MyDrive/frutas.txt'))
```

```
False
```

Tratamento de exceções

Além do **OSError** exemplificado acima, podemos tratar exceções durante a manipulação de arquivos em Python utilizando outras classes de erros. Isso é importante, pois lidar com exceções é crucial para garantir que seu programa possa tratar erros inesperados de maneira robusta e informativa. A seguir, listamos algumas exceções comuns que podem ocorrer durante a manipulação de arquivos em Python:

- **FileNotFoundError**: quando o arquivo especificado não é encontrado.
- **PermissionError**: quando o programa não tem permissão para acessar o arquivo.
- **IsADirectoryError**: quando o caminho especificado é um diretório e não um arquivo.
- **NotADirectoryError**: quando uma parte do caminho que deveria ser um diretório não é.
- **IOError**: ocorre em várias situações de erro de E/S (entrada/saída), incluindo as já mencionadas. **IOError** é um *alias* para **OSError** em Python 3.

Contar o número de linhas

Contar quantas linhas existem em um arquivo.

```
def contar _ linhas(arquivo):
    with open(arquivo, 'r') as file:
        return len(file.readlines())
```

```
print(contar_linhas('/content/drive/MyDrive/frutas.txt')) # Exibe 7
```

```
7
```

Contar o total de palavras

No código a seguir, os métodos **.read()** e **.split()** são usados para ler o conteúdo de um arquivo e contar o número de palavras nesse arquivo:

- **conteudo.split()** divide a *string* **conteudo** em uma lista de palavras.
- O método **split()** divide a *string* em espaços em branco por padrão (também podemos delimitar outros caracteres de escape como espaços, quebras de linha, tabulações etc.) e retorna uma lista das palavras resultantes.
- Se **conteudo** for 'Olá, mundo!', então **conteudo.split()** retornará ['Olá,', 'mundo!']
- **len(palavras)** retorna o número de elementos na lista **palavras**, ou seja, o número de palavras no arquivo.

```
def contar _ palavras(arquivo):
    with open(arquivo, 'r') as file:
        conteudo = file.read()
        palavras = conteudo.split()
    return len(palavras)
```

```
599
```

Contar a quantidade de ocorrências de uma palavra

Vamos contar o número de ocorrências de uma palavra específica em um arquivo usando o método **.count()**.

O método **count()** em Python é usado para contar o número de ocorrências de um valor específico dentro de uma lista. No exemplo a seguir, **count()** é utilizado para contar quantas vezes uma palavra específica aparece no conteúdo de um arquivo de texto.

```
def contar_palavra_especifica(arquivo, palavra):
    with open(arquivo, 'r') as file:
        conteudo = file.read()
        palavras = conteudo.split()
    return palavras.count(palavra)
```

16

Mostrar as primeiras n linhas

O exemplo a seguir define uma função chamada **mostrar_primeiras_linhas**, que tem o objetivo de exibir as primeiras **n** linhas de um arquivo de texto:

- **for i in range(n)**: este laço for itera n vezes, onde n é o número de linhas que queremos ler do arquivo.
- **linha = file.readline()**: em cada iteração, o método **readline** lê uma linha do arquivo e a armazena na variável **linha**. O método **readline** lê a linha até encontrar um caractere de nova linha (**\n**) ou o final do arquivo.
- **if linha == '':** esse **if** verifica se a linha lida é uma **string* *vazia, o que indica que o final do arquivo foi alcançado antes de ler **n** linhas. Se for o caso, o laço **for** é interrompido usando o comando **break**.
- **print(linha.strip())**: se a linha não for vazia, ela será exibida. O método **strip** é usado para remover quaisquer espaços em branco ou caracteres de nova linha do início e do final da linha.

```
def mostrar_primeiras_linhas(arquivo, n):
    with open(arquivo, 'r') as file:
        for i in range(n):
            linha = file.readline()
            if linha == '':
                break
            print(linha.strip())
```

```
mostrar_primeiras_linhas('/content/drive/MyDrive/words.txt', 3)
```

Tamanho do arquivo em bytes

Para mostrar o tamanho de um arquivo, usamos a função **os.path.getsize().** Essa função é utilizada para obter o tamanho de um arquivo em *bytes*. Ela faz parte do módulo **os.path**, acessível por meio do **import os**, que fornece ferramentas para manipular nomes de arquivos e diretórios de forma independente do sistema operacional.

```
import os

def tamanho_arquivo(arquivo):
    return os.path.getsize(arquivo)
```

```
print(tamanho_arquivo('/content/drive/MyDrive/frutas.txt')) # Exibe 46
```

Criar diretório e salvar arquivos

O código a seguir cria um novo diretório e salva um arquivo dentro dele. A função **criar_diretorio_e_arquivo()** recebe três parâmetros:

- **diretorio**: o nome do diretório onde o arquivo será criado.
- **arquivo**: o nome do arquivo a ser criado.
- **conteudo**: o conteúdo que será escrito no arquivo.

Usamos algumas funções úteis da biblioteca **os**:

- **os.path.exists(diretorio)**: essa linha verifica se o diretório especificado já existe.
- **not os.path.exists(diretorio)**: a condição **not** inverte o resultado da verificação, ou seja, se o diretório não existir, o bloco de código dentro do **if** será executado.
- **os.makedirs(diretorio)**: Essa linha cria o diretório especificado, incluindo todos os diretórios intermediários necessários. Se o diretório já existir, essa linha será ignorada.

```
def criar_diretorio_e_arquivo(diretorio, arquivo, conteudo):
    if not os.path.exists(diretorio):
        os.makedirs(diretorio)
    caminho_completo = os.path.join(diretorio, arquivo)
    with open(caminho_completo, 'w') as file:
        file.write(conteudo)
```

```
criar_diretorio_e_arquivo('/content/drive/MyDrive/novo_diretorio', '/content/drive/MyDrive/novo_diretorio/teste.txt', 'Isso é uma linha de teste\nEssa é outra linha')
```

```
mostrar_conteudo('/content/drive/MyDrive/novo_diretorio/teste.txt')  # Exibe Isso é uma linha de teste Essa é outra linha
```

Isso é uma linha de teste

Essa é outra linha

6.1.3 Boas Práticas na Manipulação de Arquivos

Além do uso do **with**, que garante que o arquivo será devidamente fechado após a conclusão das operações, mesmo que uma exceção seja levantada durante a execução do bloco, existem outras boas práticas que podem ser aplicadas para tornar a manipulação de arquivos mais segura.

- **Modo de abertura**: escolha o modo de abertura do arquivo com cuidado ('r' para leitura, 'w' para escrita, 'a' para anexar etc.). Cada modo tem um propósito específico e usar o modo correto evita perda de dados e erros de acesso.
- **Manipulação de erros**: implemente tratamento de exceções ao trabalhar com arquivos para capturar e lidar com erros como arquivos inexistentes ou problemas de permissão. Isso ajuda a tornar seu código mais robusto e confiável.
- **Fechamento de arquivos**: assegure-se de fechar os arquivos após a manipulação para liberar recursos do sistema. Embora o uso do bloco **with** cuide disso automaticamente, fechar arquivos manualmente é necessário se **with** não for utilizado.
- **Leitura e escrita eficiente**: para grandes arquivos, prefira ler e escrever em pedaços em vez de carregar o arquivo inteiro na memória. Isso melhora a eficiência do uso da memória e pode aumentar a velocidade de processamento.
- **Verificação da existência de arquivos**: antes de tentar abrir um arquivo para leitura ou escrita, verifique se o arquivo existe e se você tem as permissões necessárias. Isso ajuda a evitar erros de acesso e manipulação.
- **Uso de caminhos absolutos**: use caminhos absolutos em vez de relativos para evitar problemas de localização de arquivos. Caminhos absolutos garantem que seu código funcione corretamente independentemente do diretório de execução.
- **Gerenciamento de recursos**: minimize o tempo em que os arquivos ficam abertos. Abra arquivos, execute as operações necessárias e feche-os o mais rápido possível para liberar os recursos.

- **Nomeação de arquivos e diretórios**: utilize nomes de arquivos e diretórios descritivos e consistentes para facilitar a manutenção e compreensão do código. Evite caracteres especiais e espaços em branco que possam causar problemas de compatibilidade.

6.2 Saiba Mais...

- Tutorial sobre **manipulação de arquivos** em Python.

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Introdução ao *Python*

Unidade VII
Encerramento

Unidade VII - Encerramento

Nesta Unidade de encerramento, revisamos alguns dos conceitos apresentados ao longo deste *ebook*. O objetivo é consolidar o conhecimento adquirido e refletir sobre pontos como as características importantes e os benefícios da linguagem Python, a história do Python, ambientes e ferramentas de desenvolvimento, sintaxe básica, estruturas de dados, estruturas de controle de fluxo, funções e manipulação de arquivos.

7.1 Revisão dos Conceitos Fundamentais

7.1.1 Linguagem Python

- **Popularidade e versatilidade**: Python é conhecido por sua sintaxe simples e intuitiva, o que facilita a leitura e escrita de código. Essa facilidade de uso atrai tanto iniciantes quanto desenvolvedores experientes. Além disso, Python é uma linguagem de propósito geral, utilizada em *web development*, automação, análise de dados e desenvolvimento de aplicativos.
- **Comunidade ativa**: a grande comunidade de desenvolvedores contribui constantemente para o avanço da linguagem e de suas bibliotecas. Isso garante que Python esteja sempre atualizado com as últimas tendências e práticas em ciência de dados e IA.
- **Uso em IA e ciência de dados**: Python possui uma vasta gama de bibliotecas especializadas, como Pandas para manipulação de dados, NumPy para cálculos numéricos, SciPy para funções científicas, Matplotlib e Seaborn para visualização de dados, Scikit-Learn para aprendizado de máquina e TensorFlow e PyTorch para deep learning. Essas bibliotecas simplificam o processo de desenvolvimento e experimentação.

7.1.2 História do Python

- **Histórico**: desenvolvido por Guido van Rossum e lançado em 1991, Python se destacou pela sua simplicidade e legibilidade. Desde então, tem evoluído constantemente, incorporando novas funcionalidades e mantendo-se relevante no mundo da programação.
- **Comunidade científica**: Python começou a ganhar popularidade entre cientistas e pesquisadores devido à sua capacidade de integração com outras

linguagens e ferramentas. Sua simplicidade permitiu que cientistas focassem mais na análise de dados e menos na sintaxe da linguagem.

- **Engajamento**: a comunidade de desenvolvedores é extremamente ativa, contribuindo com novas bibliotecas, atualizações e melhorias. Essa colaboração contínua tem sido fundamental para o crescimento e a popularização do Python em diversas áreas, especialmente em ciência de dados e IA.

7.1.3 Ambientes e Ferramentas de Desenvolvimento

- **IDEs e editores**: ferramentas como PyCharm, Jupyter Notebook, VS Code, e Spyder são populares para desenvolvimento em Python. Cada uma oferece recursos específicos, como depuração, integração com Git, e execução de código interativo.
- **Plataformas de colaboração**: o Google Colab é uma ferramenta poderosa que permite escrever e executar código Python em *notebooks* Jupyter na nuvem. Isso facilita a colaboração em tempo real e o acesso a recursos computacionais como *Graphics Processing Units* (GPUs) e *Tensor Processing Units* (TPUs).
- **Gerenciamento de pacotes**: Pip e Conda são os principais gerenciadores de pacotes em Python. Pip é o gerenciador padrão de pacotes do Python, enquanto Conda é utilizado em ambientes Anaconda, facilitando a instalação e gerenciamento de pacotes e ambientes virtuais.

7.1.4 Sintaxe Básica

- **Comandos de entrada e saída**: **input()** para capturar dados do usuário e **print()** para exibir informações na tela.
- **Comentários e variáveis**: comentários são importantes para documentar o código. Em Python, são iniciados com **#** ou entre **‘’’** ou **“””**. Variáveis armazenam dados e podem ter diferentes tipos, como inteiros, *floats*, *strings* e booleanas.
- **Operadores**:
  - **Aritméticos**: soma (+), subtração (-), multiplicação (*), divisão (/), divisão inteira (//), resto (%), exponenciação (**).
  - **Relacionais**: comparam valores (==, !=, >, <, >=, <=).
  - **Lógicos**: operam sobre valores booleanos (**and**, **or**, **not**).
- **Manipulação de *strings***: *strings* são sequências de caracteres. Métodos como **format**, **f-strings** (**f”texto {variavel}”**) e manipulação de *strings,* como listas de caracteres, são essenciais.

7.1.5 Estruturas de Dados

- **Listas, Tuplas, Dicionários e Conjuntos**:
  - **Listas**: coleções ordenadas e mutáveis. Úteis para armazenar itens em sequência.
  - **Tuplas**: coleções ordenadas e imutáveis. Utilizadas para armazenar dados que não devem ser alterados.
  - **Dicionários**: coleções de pares chave-valor. Permitem acesso rápido a valores por suas chaves.
  - **Conjuntos**: coleções de itens únicos e não ordenados. Úteis para operações de conjunto como união, interseção e diferença.
- **Mutabilidade**: em Python, objetos mutáveis podem ser modificados após sua criação, enquanto objetos imutáveis não podem ser alterados. Isso afeta a performance e o comportamento dos programas, especialmente em operações de manipulação de dados.

7.1.6 Estruturas Condicionais e de Repetição

- **Condicionais**: permitem executar diferentes blocos de código com base em condições. Utilizam **if**, **elif** e **else** e **match-case**.
- **Repetição**: laços **for** e **while** são usados para iterar sobre sequências e repetir operações. Instruções **break**, **continue**, e **pass** controlam o fluxo dentro dos laços.
- **Tratamento de exceções**: **try**, **except**, **else** e **finally** são usados para capturar e tratar exceções, garantindo que o programa continue funcionando mesmo em caso de erros.

7.1.7 Funções

- **Conceitos básicos**: funções são blocos de código reutilizáveis. São definidas com **def** e podem receber parâmetros e retornar valores.
- **Tipos de funções**:
  - **Funções nomeadas**: funções que podem ser chamadas passando valores e recebendo retorno.
  - **Funções anônimas (*lambda*)**: funções pequenas sem nome.
- **Boas práticas**: documentação usando *docstrings*, nomeação clara de funções e variáveis, e modularização do código para facilitar a manutenção e leitura.

7.1.8 Manipulação de Arquivos

- **Abertura e fechamento**: uso do bloco **with** para abrir arquivos, garantindo que sejam fechados corretamente após a operação. Modos como '**r**', '**w**', '**a**', '**b**', '**t**', '**+**' são usados conforme a necessidade.
- **Leitura e escrita**: métodos como **read**, **readline**, **readlines**, **write** são essenciais para manipulação de dados em arquivos. Operações incluem leitura de todo o arquivo, linha a linha e escrita de dados.

7.2 Considerações Finais

Vimos, neste Microcurso, os principais tópicos necessários para iniciar a programação em Python, oferecendo uma base sólida para continuar explorando a linguagem em aplicações de ciência de dados, desenvolvimento de IA e outras áreas de tecnologia. Continue praticando e aplicando o conhecimento adquirido para aprofundar suas habilidades e desenvolver projetos mais avançados.

Referências

ANACONDA. **Anaconda** [Internet]. Disponível em: https://www.anaconda.com/. Acesso em: 31 jul. 2024.

GEEKSFORGEEKS. **Mutable vs immutable objects in Python** [Internet]. Disponível em: https://www.geeksforgeeks.org/mutable-vs-immutable-objects-in-python/. Acesso em: 31 jul. 2024.

GOOGLE. **Google Colab** [Internet]. Disponível em: https://colab.research.google.com/. Acesso em: 31 jul. 2024

JETBRAINS. **PyCharm** [Internet]. Disponível em: https://www.jetbrains.com/pycharm/. Acesso em: 31 jul. 2024.

JUPYTER. **Jupyter Project** [Internet]. Disponível em: https://jupyter.org/. Acesso em: 31 jul. 2024.

PYTHON SOFTWARE FOUNDATION. **Controle de fluxo** [Internet]. Disponível em: https://docs.python.org/pt-br/3/tutorial/controlflow.html. Acesso em: 30 jul. 2024.

PYTHON SOFTWARE FOUNDATION. **Exceptions: Python Standard Library** [Internet]. Disponível em: https://docs.python.org/3/library/exceptions.html. Acesso em: 30 jul. 2024.

PYTHON SOFTWARE FOUNDATION. **File handling in Python** [Internet]. Disponível em: https://www.geeksforgeeks.org/file-handling-python/. Acesso em: 30 jul. 2024.

PYTHON SOFTWARE FOUNDATION. **Python documentation** [Internet]. Disponível em: https://docs.python.org. Acesso em: 30 jul. 2024.

PYTHON SOFTWARE FOUNDATION. **Python Standard Library: built-in functions** [Internet]. Disponível em: https://docs.python.org/3/library/functions.html. Acesso em: 30 jul. 2024.

PYTHON SOFTWARE FOUNDATION. **Python** [Internet]. Disponível em: https://www.python.org/. Acesso em: 29 jul. 2024.

PYTORCH. **PyTorch documentation** [Internet]. Disponível em: https://pytorch.org/docs/stable/index.html. Acesso em: 31 jul. 2024.

SCIKIT-LEARN. **Scikit-learn documentation** [Internet]. Disponível em: https://scikit-learn.org/0.21/documentation.html. Acesso em: 31 jul. 2024.

SCIPY. SciPy: **Fundamental algorithms for scientific computing in Python** [Internet]. Disponível em: https://www.scipy.org/. Acesso em: 30 jul. 2024.

TENSORFLOW. **TensorFlow** [Internet]. Disponível em: https://www.tensorflow.org/guide?hl=pt-br. Acesso em: 31 jul. 2024.

VISUAL STUDIO CODE. **Python in Visual Studio Code** [Internet]. Disponível em: https://code.visualstudio.com/docs/languages/python. Acesso em: 31 jul. 2024.

W3SCHOOLS. **Python dictionaries** [Internet]. Disponível em: https://www.w3schools.com/python/python_dictionaries.asp. Acesso em: 31 jul. 2024.

W3SCHOOLS. **Python escape characters** [Internet]. Disponível em: https://www.w3schools.com/python/gloss_python_escape_characters.asp. Acesso em: 31 jul. 2024.

W3SCHOOLS. **Python lists** [Internet]. Disponível em: https://www.w3schools.com/python/python_lists.asp. Acesso em: 30 jul. 2024.

W3SCHOOLS. **Python tuples** [Internet]. Disponível em: https://www.w3schools.com/python/python_tuples.asp. Acesso em: 31 jul. 2024.

AKCIT

CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS

CEIA
CENTRO DE EXCELÊNCIA EM INTELIGÊNCIA ARTIFICIAL

GOVERNO DE GOIÁS
O ESTADO QUE DÁ CERTO

EMBRAPII

INF
INSTITUTO DE INFORMÁTICA

PRPI
PRÓ-REITORIA DE PESQUISA E INOVAÇÃO

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

SOBRE O E-BOOK

Tipografia: Montserrat

Publicação: Cegraf UFG
Câmpus Samambaia, Goiânia - Goiás. Brasil. CEP 74690-900
Fone: (62) 3521-1358
https://cegraf.ufg.br