

Sejam Bem-vindos ao curso básico em machine learning

Módulo 1

Programação em Linguagem Python para
Machine Learning

Ementa

- Módulo 1
 - Programação em Linguagem Python para Machine Learning
 - Unidade 1
 - Ambiente de desenvolvimento
 - Visual Studio Code
 - PyCharm
 - Google Colaboratory
 - Unidade 2
 - Variáveis e tipos de dados
 - Ponto Flutuante ou Decimal (float)
 - Complexo (complex)
 - String (str)
 - Boolean (bool)
 - Listas (list)
 - Tuplas (tuple)
 - Dicionários (dict)
 - Matrizes (array)
 - Mudar o tipo de uma variável
 - Erros típicos relacionados ao tipo da variável

Ementa

- Módulo 1
 - Programação em Linguagem Python para Machine Learning
 - Unidade 3
 - Operadores
 - Aritméticos
 - Comparação
 - Atribuição
 - Lógicos
 - Identidade
 - Associação
 - Estruturas de repetição
 - Loop for
 - Loop while
 - Finalizar de loop
 - Função range()
 - Função enumerate()

Ementa

- Módulo 1
 - Programação em Linguagem Python para Machine Learning
 - Unidade 4
 - Estruturas de dados
 - Listas
 - Instrução del
 - Conjuntos
 - Técnicas de iteração

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Visual Studio Code

O Visual Studio Code (também conhecido como VS Code) é um editor de código fonte desenvolvido pela Microsoft. Embora seja frequentemente associado ao desenvolvimento web e ao ambiente Microsoft, o VS Code é uma ferramenta altamente versátil e popular para desenvolvimento em várias linguagens de programação, incluindo Python.

Características e vantagens do Visual Studio Code

Gratuito e de código aberto: O VS Code é distribuído gratuitamente e é um software de código aberto. Isso significa que sua comunidade ativa de desenvolvedores contribui para aprimorar e estender suas funcionalidades por meio de extensões.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Características e vantagens do Visual Studio Code

Multiplataforma: O VS Code é compatível com Windows, macOS e Linux, garantindo uma experiência consistente em diferentes sistemas operacionais.

Extensibilidade: Uma das principais forças do VS Code é sua extensibilidade. Existem milhares de extensões disponíveis na Visual Studio Code Marketplace que podem ser usadas para aprimorar o editor e adicionar suporte para várias linguagens de programação, ferramentas e frameworks.

Suporte a Python: O VS Code é particularmente popular entre os desenvolvedores Python devido ao excelente suporte para essa linguagem. As extensões Python fornecem recursos como realce de sintaxe, sugestões inteligentes de código, formatação automática, depuração integrada e integração com ferramentas populares como o virtual env.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Características e vantagens do Visual Studio Code

Terminal Integrado: O VS Code possui um terminal integrado que permite executar comandos diretamente do editor, facilitando o gerenciamento de projetos e o uso de ferramentas de linha de comando.

Git Integration: A integração Git no VS Code é muito útil para desenvolvedores que trabalham com controle de versão. Ela permite executar operações do Git, como commit, push, pull e merge, diretamente do editor.

Depuração avançada: O VS Code oferece suporte a depuração integrada para várias linguagens, incluindo Python. Isso permite que você defina pontos de interrupção, inspecione variáveis e depure seu código de maneira eficiente.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Características e vantagens do Visual Studio Code

Personalização: O editor é altamente personalizável, permitindo que você ajuste a aparência e os atalhos de teclado para melhor atender às suas preferências e fluxo de trabalho.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Instalação do Visual Studio Code no Linux

Consulte a página <https://code.visualstudio.com/download> para obter uma lista completa das opções de instalação disponíveis. Ao baixar e usar o Visual Studio Code, você concorda com os termos de licença e a declaração de privacidade. A maneira mais fácil de instalar o Visual Studio Code para distribuições baseadas em Debian/Ubuntu é baixar e instalar o pacote .deb (64 bits) , por meio do centro de software gráfico, se disponível, ou por meio da linha de comando com:

```
sudo apt install ./<file>.deb

# If you're on an older Linux distribution, you will need to run this
instead:
# sudo dpkg -i <file>.deb

# sudo apt-get install -f # Install dependencies
```

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Instalação do Visual Studio Code no Windows

1. Baixe o instalador do Visual Studio Code <https://code.visualstudio.com/docs?dv=win> para Windows.
2. Após o download, execute o instalador (VSCodeUserSetup-{version}.exe). Isso levará apenas um minuto.
3. Por padrão, o VS Code é instalado em C:\Users\{Username}\AppData\Local\Programs\Microsoft VS Code.

IMPORTANTE: a instalação vai adicionar o Visual Studio Code ao seu %PATH%, portanto, no console, você poderá digitar 'code .' para abrir o VS Code nessa pasta. Você precisará reiniciar seu console após a instalação para que a alteração na %PATH%variável de ambiente entre em vigor.

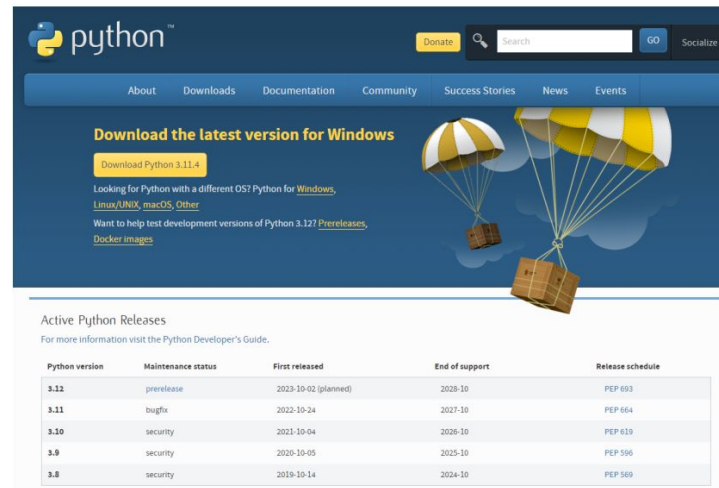
Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Instalando o interpretador Python

A instalação interna do Python 3 no Linux funciona bem, mas para instalar outros pacotes do Python, você deve instalar pip com `get-pip.py`.

Para windows:

Instale o Python em <https://www.python.org/downloads/>. Use o botão Baixar Python que aparece primeiro na página para baixar a versão mais recente



The screenshot shows the Python.org website. At the top, there's a navigation bar with links: About, Downloads, Documentation, Community, Success Stories, News, and Events. Below this, a large banner features the text "Download the latest version for Windows" and a button "Download Python 3.11.4". There are also links for "Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other" and "Want to help test development versions of Python 3.12? Pre-releases, Docker images". The bottom section is titled "Active Python Releases" and contains a table with the following data:

Python version	Maintenance status	First released	End of support	Release schedule
3.12	prerelease	2023-10-02 (planned)	2028-10	PEP 693
3.11	bugfix	2022-10-24	2027-10	PEP 664
3.10	security	2021-10-04	2026-10	PEP 619
3.9	security	2020-10-05	2025-10	PEP 596
3.8	security	2019-10-14	2024-10	PEP 569

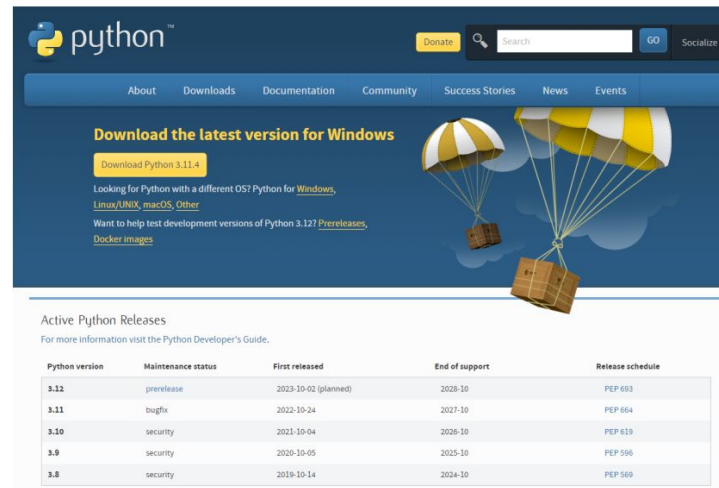
Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Instalando o interpretador Python

A instalação interna do Python 3 no Linux funciona bem, mas para instalar outros pacotes do Python, você deve instalar pip com get-pip.py.

Para windows:

Instale o Python em <https://www.python.org/downloads/>. Use o botão Baixar Python que aparece primeiro na página para baixar a versão mais recente



The screenshot shows the Python.org website. At the top, there's a navigation bar with links: About, Downloads, Documentation, Community, Success Stories, News, and Events. Below this, a large banner features the text "Download the latest version for Windows" and a button "Download Python 3.11.4". There are also links for "Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other" and "Want to help test development versions of Python 3.12? Pre-releases, Docker images". The bottom section is titled "Active Python Releases" and contains a table with the following data:

Python version	Maintenance status	First released	End of support	Release schedule
3.12	prerelease	2023-10-02 (planned)	2028-10	PEP 693
3.11	bugfix	2022-10-24	2027-10	PEP 664
3.10	security	2021-10-04	2026-10	PEP 619
3.9	security	2020-10-05	2025-10	PEP 596
3.8	security	2019-10-14	2024-10	PEP 569

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Instalando o interpretador Python

Para ubuntu:

Baixe o script em <https://bootstrap.pypa.io/get-pip.py> .

Abra um terminal/prompt de comando, cd na pasta que contém o get-pip.py arquivo e execute:

```
$ python get-pip.py
```

Observação:

Mais detalhes sobre este script podem ser encontrados no README de <https://github.com/pypa/get-pip>.

Para verificar se você instalou o Python com sucesso em sua máquina, execute um dos seguintes comandos (dependendo do seu sistema operacional):

Linux/macOS: abra uma Janela de Terminal e digite o seguinte comando:

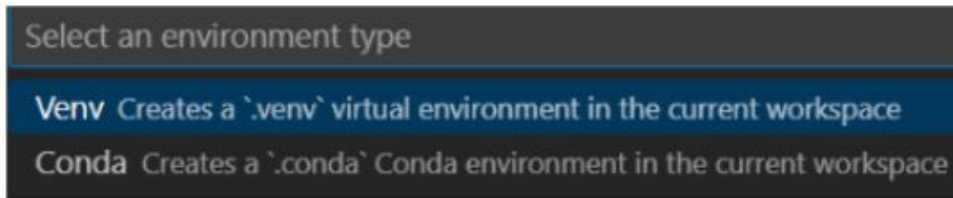
```
$ python3 --version
```

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Crie um ambiente virtual

Uma prática recomendada entre os desenvolvedores Python é usar um arquivo virtual environment. Depois de ativar esse ambiente, todos os pacotes instalados são isolados de outros ambientes, incluindo o ambiente do interpretador global, reduzindo muitas complicações que podem surgir de versões de pacote conflitantes. Você pode criar ambientes não globais no VS Code usando Venv **Python: Create Environment**.

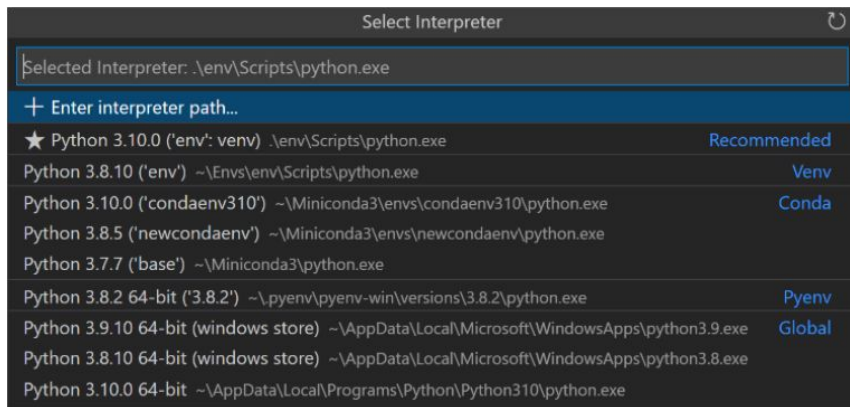
Abra a Paleta de Comandos (Ctrl+Shift+P), comece a digitar o comando Python: Create Environment para pesquisar e, em seguida, selecione o comando. O comando apresenta uma lista de tipos de ambiente, Venv ou Conda. Para este exemplo, selecione Venv.



Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Crie um ambiente virtual

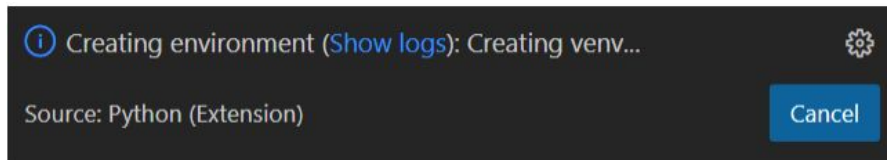
O comando então apresenta uma lista de interpretadores que podem ser usados para o seu projeto. Selecione o interpretador que você instalou no início do tutorial.



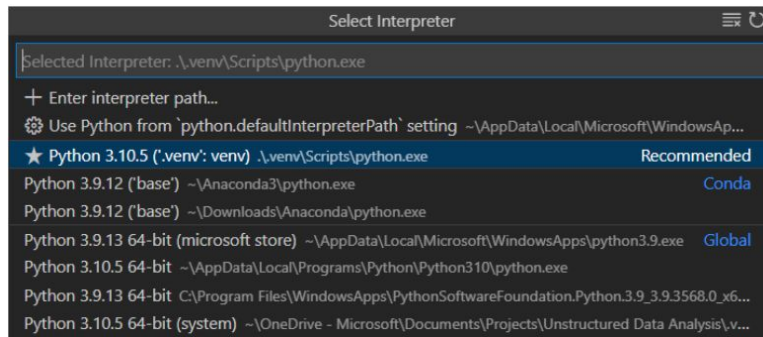
Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Crie um ambiente virtual

Após selecionar o interpretador, uma notificação mostrará o andamento da criação do ambiente e a pasta do ambiente (`/.venv`) aparecerá em sua área de trabalho.



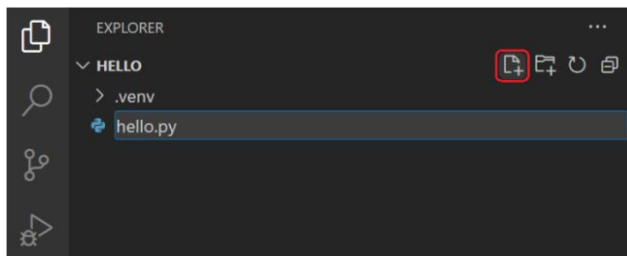
Assegure-se de que seu novo ambiente seja selecionado usando o comando Python: Select Interpreter na Paleta de Comandos.



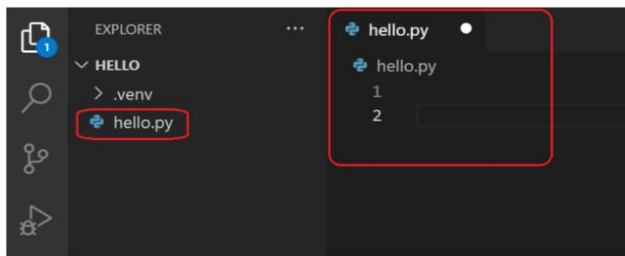
Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Criar um arquivo de código-fonte Python

Na barra de ferramentas do Explorador de arquivos, selecione o botão Novo arquivo na hello pasta:



Nomeie o arquivo hello.py e o VS Code o abrirá automaticamente no editor:



Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Criar um arquivo de código-fonte Python

Ao usar o .py extensão do arquivo, você diz ao VS Code para interpretar este arquivo como um programa Python, para que ele avalie o conteúdo com a extensão Python e o interpretador selecionado. Agora que você tem um arquivo de código em seu espaço de trabalho, insira o seguinte código-fonte em hello.py:

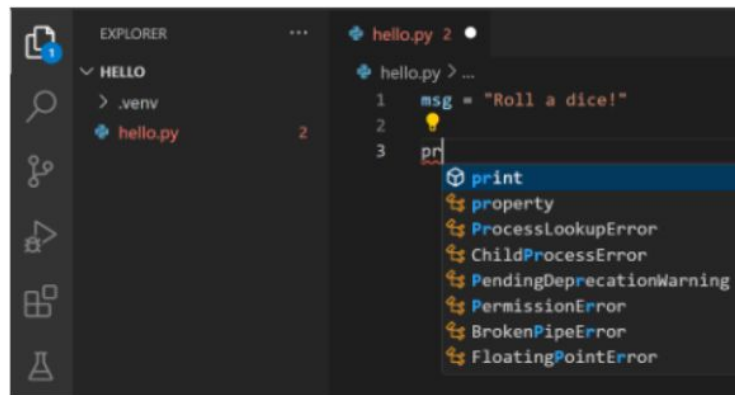
```
msg = 'Roll a dice'  
print(msg)
```

Ao começar a digitar print, observe como o IntelliSense apresenta as opções de preenchimento automático.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Criar um arquivo de código-fonte Python

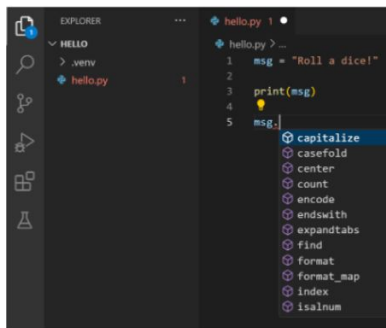
Ao começar a digitar print, observe como o IntelliSense apresenta as opções de preenchimento automático.



Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Criar um arquivo de código-fonte Python

O IntelliSense e os preenchimentos automáticos funcionam para módulos Python padrão, bem como para outros pacotes que você instalou no ambiente do interpretador Python selecionado. Ele também fornece conclusões para métodos disponíveis em tipos de objetos. Por exemplo, como a msg variável contém uma string, o IntelliSense fornece métodos de string quando você digita msg.:

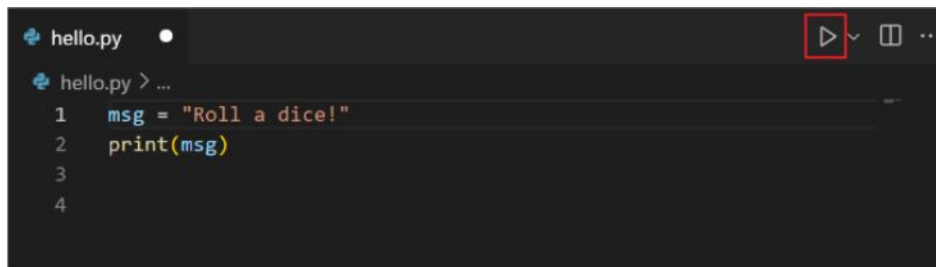


Por fim, salve o arquivo (Ctrl+S). Neste ponto, você está pronto para executar seu primeiro arquivo Python no VS Code.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Executar Hello World

Clique no botão Run Python File in Terminal play no canto superior direito do editor.



```
hello.py •
hello.py > ...
1 msg = "Roll a dice!"
2 print(msg)
3
4
```

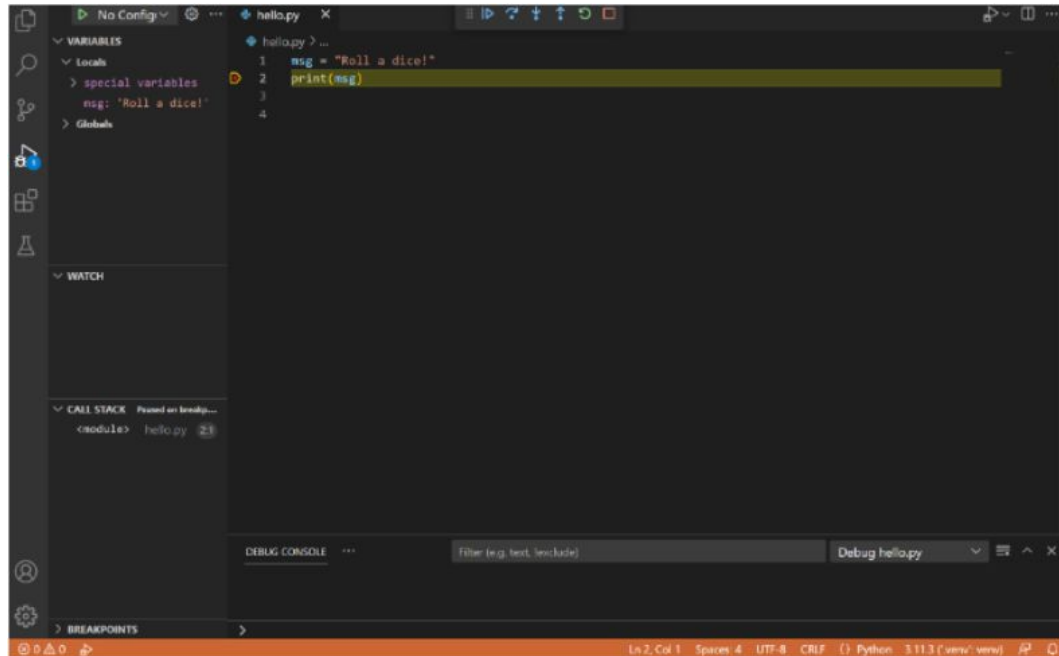
O botão abre um painel de terminal no qual seu interpretador Python é ativado automaticamente e executado `python3 hello.py` (macOS/Linux) ou `python hello.py` (Windows):



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Python + - □ □ ... ^ ×
PS C:\hello> & c:/hello/.venv/Scripts/python.exe c:/hello/hello.py
Roll a dice!
PS C:\hello>
```

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Exemplo breakpoint



Modulo 1: Unidade 1 – Ambiente de desenvolvimento

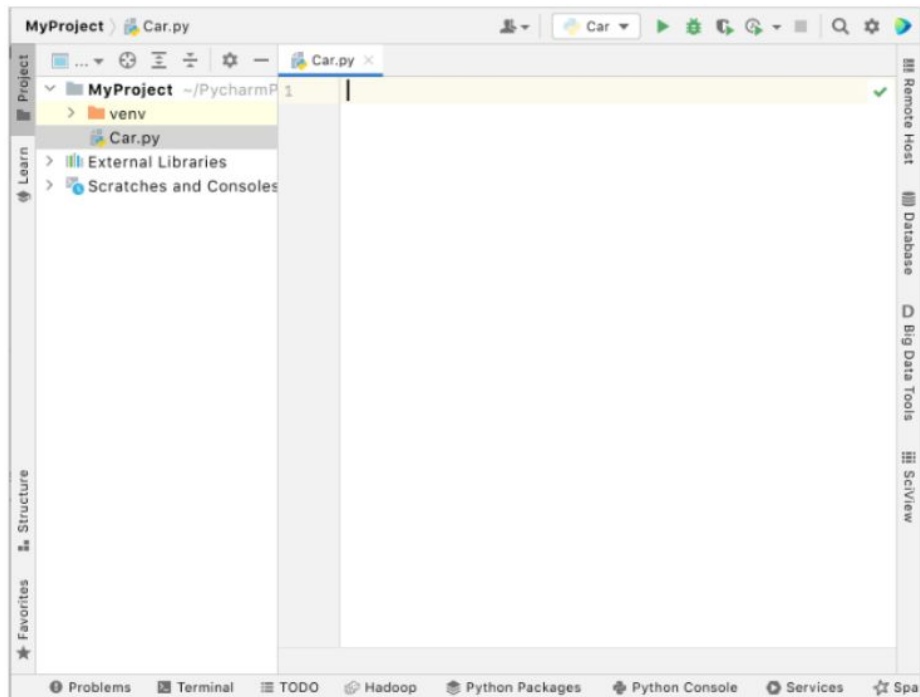
PyCharm

O PyCharm é uma IDE de plataforma que fornece experiência consistente nos sistemas operacionais Windows, macOS e Linux.

O PyCharm está disponível em duas edições: Professional e Community. A edição Community é um projeto de código aberto e é gratuito, mas tem menos recursos. A edição Professional é comercial e oferece um excelente conjunto de ferramentas e recursos.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

PyCharm



Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Google Colaboratory

O Google Colaboratory, também conhecido como Google Colab, é uma plataforma gratuita oferecida pelo Google que permite executar e colaborar em notebooks Jupyter online. Google Colab é baseado no ambiente de nuvem do Google e utiliza recursos da Google Cloud Platform para fornecer poder computacional a usuários de ciência de dados, aprendizado de máquina, inteligência artificial e outras áreas relacionadas.

Principais características do Google Colab

Notebooks interativos: Os notebooks do Colab são interativos e permitem a execução de código em células individuais, o que facilita a exploração de dados e a criação de modelos de aprendizado de máquina de forma iterativa.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Google Colaboratory

Principais características do Google Colab

Ambiente baseado em Jupyter: O Colab é baseado no projeto Jupyter, o que significa que você pode usar códigos em Python (também suporta R e outros) junto com texto formatado, gráficos, equações e outras mídias em um único documento.

Acesso a GPUs e TPUs: O Colab oferece acesso gratuito a GPUs (unidades de processamento gráfico) e TPUs (unidades de processamento tensorial) para acelerar o treinamento de modelos de aprendizado de máquina, tornando-o especialmente útil para tarefas intensivas em computação.

Armazenamento na nuvem: Os notebooks do Colab são salvos no Google Drive, o que facilita o armazenamento e o compartilhamento de projetos com outras pessoas.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Google Colaboratory

Principais características do Google Colab

Bibliotecas pré-instaladas: O Colab já possui várias bibliotecas populares pré-instaladas, como TensorFlow, PyTorch, Pandas, NumPy, entre outras, para facilitar o trabalho com ciência de dados e aprendizado de máquina.

Colaboração em tempo real: Assim como o Google Docs, o Google Colab permite que várias pessoas colaborem em um mesmo notebook em tempo real, facilitando o trabalho em equipe e a aprendizagem conjunta.

Integração com outras ferramentas do Google: O Colab tem integração com outras ferramentas do Google, como Google Sheets e BigQuery, o que permite a importação de dados diretamente dessas fontes.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Google Colaboratory

Principais características do Google Colab

Bibliotecas pré-instaladas: O Colab já possui várias bibliotecas populares pré-instaladas, como TensorFlow, PyTorch, Pandas, NumPy, entre outras, para facilitar o trabalho com ciência de dados e aprendizado de máquina.

Colaboração em tempo real: Assim como o Google Docs, o Google Colab permite que várias pessoas colaborem em um mesmo notebook em tempo real, facilitando o trabalho em equipe e a aprendizagem conjunta.

Integração com outras ferramentas do Google: O Colab tem integração com outras ferramentas do Google, como Google Sheets e BigQuery, o que permite a importação de dados diretamente dessas fontes.

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Google Colaboratory

Importa a biblioteca do colab e montando o drive

```
# digite este trecho para montar o drive  
from google.colab import drive  
drive.mount('/content/drive')
```

Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Google Colaboratory

Célula de código

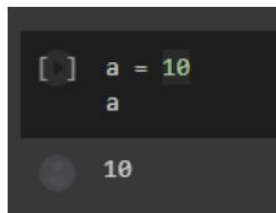
Abaixo está uma célula de código. Assim que o botão da barra de ferramentas indicar CONNECTED, clique na célula para selecioná-la e execute o conteúdo das seguintes formas:

Clique no ícone Reproduzir na calha esquerda da célula;

Digite Cmd/Ctrl+Enter para executar a célula no local;

Digite Shift+Enter para executar a célula e mover o foco para a próxima célula (adicionando uma se não existir); ou

Digite Alt+Enter para executar a célula e insira uma nova célula de código imediatamente abaixo dela.



Modulo 1: Unidade 1 – Ambiente de desenvolvimento

Google Colaboratory

Célula de texto

Esta é uma célula de texto . Você pode clicar duas vezes para editar esta célula. Células de texto usam sintaxe de remarcação. Para saber mais, consulte nosso guia de descontos.

Você também pode adicionar matemática às células de texto usando o LaTeX para ser renderizado pelo MathJax . Basta colocar a instrução dentro de um par de sinais \$.

Tempo de execução

Na versão do Colab sem custo financeiro, os notebooks são executados por no máximo 12 horas, dependendo da disponibilidade e dos padrões de uso. O Colab Pro, Pro+ e o pagamento por utilização oferecem maior disponibilidade de computação com base no seu saldo de unidades.

Em geral, os notebooks são executados por no máximo 12 horas, dependendo da disponibilidade e dos padrões de uso.

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Ponto Flutuante ou Decimal (float)

O tipo "float" é utilizado para representar números de ponto flutuante, ou seja, números com casas decimais. Por exemplo, 3.14, -2.71, 0.5 são todos exemplos de números em ponto flutuante.

Os números de ponto flutuante são representados seguindo o padrão IEEE 754, que define a forma como os computadores armazenam e realizam operações com números em ponto flutuante.

Em Python, os números de ponto flutuante são criados automaticamente quando utiliza-se casas decimais em um número, ou pode ser criado explicitamente utilizando o formato "float(valor)".

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Ponto Flutuante ou Decimal (float)

Criando números de ponto flutuante

Alguns exemplos de como criar números de ponto flutuante em Python:

```
# Exemplos de números de ponto flutuante
```

```
numero1 = 3.14
```

```
numero2 = -2.71
```

```
numero3 = 0.5
```

Operações com números de ponto flutuante

Podemos realizar todas as operações matemáticas básicas com números de ponto flutuante.

```
# Exemplos de operações com números de ponto flutuante
```

```
soma = numero1 + numero2
```

```
subtracao = numero1 - numero3
```

```
multiplicacao = numero2 * numero3
```

```
divisao = numero1 / numero3
```

```
potencia = numero1 ** 2
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Ponto Flutuante ou Decimal (float)

Precisão e arredondamento

Uma característica importante dos números de ponto flutuante é que eles têm uma precisão limitada. Isso significa que nem sempre podemos representar todos os números reais com exatidão. Por exemplo:

```
resultado = 0.1 + 0.2  
print(resultado) # O resultado será 0.30000000000000004
```

Para contornar problemas de precisão, é comum utilizar funções de arredondamento, como `round()`, para obter um resultado mais adequado às necessidades do cálculo.

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Ponto Flutuante ou Decimal (float)

Conversão entre tipos de dados

Em alguns casos, pode ser necessário converter um valor "float" para outro tipo de dado, como "int" ou "str". Para fazer isso, podemos usar as funções `int()` e `str()`.

```
# Conversão para int
numero_inteiro = int(numero1)

# Conversão para str
numero_str = str(numero2)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Ponto Flutuante ou Decimal (float)

Funções matemáticas embutidas

Python oferece diversas funções matemáticas embutidas que podem ser usadas com números de ponto flutuante, tais como `abs()`, `round()`, `max()`, `min()`, `sum()`, entre outras.

```
numeros = [1.5, 2.3, 0.8, -2.0]
valor_absoluto = abs(numero2)
maior_numero = max(numeros)
soma_total = sum(numeros)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Complexo (complex)

O tipo complex é utilizado para representar números complexos, que são compostos por uma parte real e uma parte imaginária. Um número complexo é escrito na forma "a + bj", onde "a" é a parte real, "b" é a parte imaginária e "j" é a unidade imaginária ($\sqrt{-1}$).

Os números complexos são utilizados para lidar com situações matemáticas que envolvem raízes negativas e problemas em várias áreas, como engenharia, física e ciência da computação.

Em Python, os números complexos são criados utilizando o formato "complex(a, b)".

Criando números complexos

Vamos ver alguns exemplos de como criar números complexos em Python:

```
# Exemplos de números complexos
complexo1 = complex(3, 4) # Representa o número 3 + 4j
complexo2 = complex(-2, 1) # Representa o número -2 + 1j
complexo3 = 1 + 2j # Podemos também criar usando a notação direta
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Complexo (complex)

Números complexos – Operações

Podemos realizar várias operações matemáticas com números complexos, incluindo adição, subtração, multiplicação, divisão e potenciação.

```
# Exemplos de operações com números complexos
```

```
soma = complexo1 + complexo2
```

```
subtracao = complexo1 - complexo3
```

```
multiplicacao = complexo1 * complexo2
```

```
divisao = complexo2 / complexo3
```

```
potencia = complexo3 ** 2
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Complexo (complex)

Números complexos – Acessando a parte real e imaginária

Para acessar a parte real e imaginária de um número complexo, podemos usar as propriedades "real" e "imag".

```
# Exemplos de acesso à parte real e imaginária
```

```
parte_real = complexo1.real
```

```
parte_imaginaria = complexo1.imag
```


Modulo 1: Unidade 2 – Variáveis e tipos de dados

Complexo (complex)

Números complexos – Funções matemáticas e complexas

Python oferece algumas funções matemáticas e complexas específicas para trabalhar com números complexos, como `abs()`, `conjugate()`, `phase()`.

Exemplos de funções matemáticas e complexas

```
modulo = abs(complexo2)
```

```
conjugado = complexo1.conjugate()
```

```
fase = phase(complexo3) # Retorna o ângulo em radianos
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Complexo (complex)

Números complexos – Conversão entre tipos de dados

Em alguns casos, pode ser necessário converter um valor "complex" para outro tipo de dado, como "int", "float" ou "str". Para fazer isso, podemos usar as funções "int()", "float()" e "str()".

```
# Conversão para int e float
parte_real_inteira = int(complexo1.real)
parte_imaginaria_float = float(complexo1.imag)

# Conversão para str
complexo_str = str(complexo2)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

String (str)

Números complexos – Conversão entre tipos de dados

O tipo "string" é utilizado para representar sequências de caracteres, como texto. As strings são um dos tipos de dados mais fundamentais e amplamente utilizados em Python.

Strings são coleções imutáveis de caracteres que podem incluir letras, números, símbolos e espaços. Em Python, as strings são criadas delimitando o texto com aspas simples (' '), aspas duplas (" ") ou aspas triplas ("" " ou "" "" "" """). O tipo "str" é utilizado internamente pelo interpretador Python para representar e manipular sequências de caracteres.

.

Modulo 1: Unidade 2 – Variáveis e tipos de dados

String (str)

Criando strings

Vamos ver alguns exemplos de como criar strings em Python:

```
# Exemplos de strings
string1 = 'Olá, mundo!'
string2 = "Python é incrível!"
string3 = '''Strings em Python são flexíveis e poderosas.'''
```

Acessando caracteres em uma string

Podemos acessar caracteres individuais de uma string utilizando índices. Os índices em Python começam em 0.

```
# Exemplo de acesso a caracteres em uma string
mensagem = "Python"
primeiro_caracter = mensagem[0] # 'P'
segundo_caracter = mensagem[1] # 'y'
ultimo_caracter = mensagem[-1] # 'n' (último caracter da string)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

String (str)

Operações com strings

Python oferece várias operações que podem ser realizadas com strings, como concatenação, repetição, fatiamento (slicing), entre outras.

```
# Exemplos de operações com strings
nome = "João"
sobrenome = "Silva"
nome_completo = nome + " " + sobrenome # Concatenação
mensagem_repetida = "Olá! " * 3 # Repetição
parte_do_nome = nome[0:2] # Fatiamento (slicing)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

String (str)

Funções e métodos úteis para strings

Existem várias funções e métodos incorporados ao Python que facilitam a manipulação e processamento de strings.

```
# Exemplos de funções e métodos para strings
```

```
texto = "Olá, mundo!"
```

```
tamanho = len(texto) # Retorna o tamanho da string
```

```
maiusculo = texto.upper() # Converte para letras maiúsculas
```

```
minusculo = texto.lower() # Converte para letras minúsculas
```

```
contagem = texto.count('o') # Conta a ocorrência de um caracter ou substring
```

```
substituida = texto.replace('mundo', 'Python') # Substitui parte da string por outra
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

String (str)

Formatação de strings

Python oferece várias maneiras de formatar strings, permitindo a inserção de valores dinâmicos em locais específicos.

```
# Exemplo de formatação de strings
nome = "Alice"
idade = 25
mensagem_formatada = "Olá, meu nome é {} e tenho {}
anos.".format(nome, idade)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Boolean (bool)

O tipo "boolean" é utilizado para representar valores lógicos, ou seja, valores que podem ser verdadeiros (True) ou falsos (False). Esses valores são fundamentais em muitos aspectos da programação, especialmente em estruturas de controle de fluxo e tomada de decisões.

O tipo "boolean" em Python possui apenas dois valores possíveis: True e False. Estes valores são muito importantes para determinar a lógica de execução de um programa e para a avaliação de expressões condicionais. Boolean (bool) - Criando valores

Boolean (bool) - Criando valores

Veja alguns exemplos de como criar valores booleanos:

```
# Exemplos de valores booleanos
verdadeiro = True
falso = False
```


Modulo 1: Unidade 2 – Variáveis e tipos de dados

Boolean (bool)

Boolean (bool) – Operações lógicas

Python oferece operadores lógicos que permitem combinar valores booleanos para obter novos resultados lógicos.

Os principais operadores lógicos são:

and: retorna True apenas se ambos os operandos forem True;

or: retorna True se pelo menos um dos operandos for True;

not: inverte o valor booleano, transformando True em False e False em True;

```
valor1 = True
valor2 = False
resultado_and = valor1 and valor2
resultado_or = valor1 or valor2
resultado_not = not valor1
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Boolean (bool)

Boolean (bool) – Expressões condicionais

Os valores booleanos são amplamente usados em expressões condicionais, como declarações "if", "elif" e "else". Essas estruturas permitem que o programa tome decisões com base nas condições estabelecidas.

```
# Exemplo de expressão condicional
idade = 18
if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Boolean (bool)

Boolean (bool) – Retornos booleanos de funções

As funções em Python também podem retornar valores booleanos. Isso é útil quando queremos que a função retorne uma resposta lógica a alguma pergunta ou condição.

```
# Exemplo de função com retorno booleano
```

```
def e_par(numero):  
    return numero % 2 == 0
```

```
resultado = e_par(5)
```

```
print(resultado) # Output: False
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Boolean (bool)

Boolean (bool) – Comparações

Em Python, também podemos realizar comparações entre valores, resultando em um valor booleano (True ou False). As principais operações de comparação são:

`==`: igual a

`!=`: diferente de

`<`: menor que

`>`: maior que

`<=`: menor ou igual a

`>=`: maior ou igual a

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Listas (list)

Boolean (bool) – Comparações

O tipo "list" é utilizado para representar listas, que são coleções ordenadas e mutáveis de elementos. As listas são extremamente versáteis e amplamente utilizadas em Python para armazenar conjuntos de dados relacionados.

As listas são conjuntos de elementos que podem ser de diferentes tipos de dados, como números, strings ou até outras listas. Em Python, as listas são criadas utilizando colchetes [] e os elementos são separados por vírgulas.

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Listas (list)

Criando listas

```
# Exemplos de listas
lista_numeros = [1, 2, 3, 4, 5]
lista_frutas = ['maçã', 'banana',
               'laranja']
lista_mista = [10, 'python', True]
```

Listas - Acessando elementos

Podemos acessar elementos individuais de uma lista utilizando índices. Os índices em Python começam em 0.

```
# Exemplo de acesso a elementos em uma lista
numeros = [10, 20, 30, 40, 50]
primeiro_elemento = numeros[0] # 10
segundo_elemento = numeros[1] # 20
ultimo_elemento = numeros[-1] # 50 (último elemento da lista)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Listas (list)

Listas - Operações

```
# Exemplos de operações com listas
```

```
lista_a = [1, 2, 3]
```

```
lista_b = [4, 5, 6]
```

```
# Adição de elementos
```

```
lista_a.append(4) # [1, 2, 3, 4]
```

```
lista_b.insert(0, 0) # [0, 4, 5, 6]
```

```
# Remoção de elementos
```

```
lista_a.remove(2) # [1, 3]
```

```
elemento_removido = lista_b.pop(2) # elemento_removido = 5, lista_b = [0,4, 6]
```

```
# Concatenação de listas
```

```
lista_concatenada = lista_a + lista_b # [1, 3, 0, 4, 6]
```

```
# Fatiamento (slicing)
```

```
sublista = lista_concatenada[1:4] # [3, 0, 4]
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Listas (list)

Funções úteis para listas

Python possui algumas funções incorporadas que facilitam a manipulação de listas.

```
# Exemplos de funções úteis para listas
```

```
numeros = [5, 2, 8, 1, 9]
```

```
tamanho = len(numeros) # Retorna o tamanho da lista (5)
```

```
maior = max(numeros) # Retorna o maior valor (9)
```

```
menor = min(numeros) # Retorna o menor valor (1)
```

```
soma_total = sum(numeros) # Retorna a soma dos elementos (25)
```


Modulo 1: Unidade 2 – Variáveis e tipos de dados

Tuplas (tuple)

O tipo "tupla" é utilizado para representar coleções ordenadas e imutáveis de elementos. As tuplas são semelhantes às listas, mas a principal diferença é que as tuplas não podem ser modificadas após a sua criação, tornando uma escolha adequada para dados que não devem ser alterados.

As tuplas são semelhantes às listas, mas são criadas utilizando parênteses () em vez de colchetes []. A principal característica das tuplas é que elas são imutáveis, o que significa que os elementos não podem ser adicionados, removidos ou alterados após a criação.

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Tuplas (tuple)

Tuplas- Criação

```
# Exemplos de tuplas
tupla_numeros = (1, 2, 3, 4, 5)
tupla_frutas = ('maçã', 'banana', 'laranja')
tupla_mista = (10, 'python', True)
```

Tuplas - Acessando elementos

Assim como nas listas, podemos acessar elementos individuais de uma tupla utilizando índices.

```
# Exemplo de acesso a elementos em uma tupla
numeros = (10, 20, 30, 40, 50)
primeiro_elemento = numeros[0] # 10
segundo_elemento = numeros[1] # 20
ultimo_elemento = numeros[-1] # 50 (último elemento da tupla)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Tuplas (tuple)

Tuplas – Operações

Como as tuplas são imutáveis, algumas operações que são válidas para listas não são permitidas para tuplas. No entanto, podemos realizar operações como concatenação e repetição.

```
# Exemplos de operações com tuplas
```

```
tupla_a = (1, 2, 3)
```

```
tupla_b = (4, 5, 6)
```

```
# Concatenação de tuplas
```

```
tupla_concatenada = tupla_a + tupla_b # (1, 2, 3, 4, 5, 6)
```

```
# Repetição de tupla
```

```
tupla_repetida = tupla_a * 3 # (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Dicionários (dict)

O tipo "dicionário" é utilizado para representar coleções de elementos que são armazenados em pares chave-valor. Os dicionários são estruturas de dados muito poderosas e versáteis, permitindo a organização e recuperação eficiente de informações.

Os dicionários em Python são coleções não ordenadas de elementos. Cada elemento é representado por um par chave-valor, onde a "chave" é um identificador único e o "valor" é a informação associada àquela chave. Os dicionários são criados utilizando chaves {} e os pares chave-valor são separados por dois pontos (:).

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Dicionários (dict)

Dicionários - Criação

Exemplos de dicionários

```
dicionario_pessoas = {  
    'nome': 'João',  
    'idade': 30,  
    'cidade': 'São Paulo'  
}
```

```
dicionario_notas = {  
    'Matemática': 9.5,  
    'Ciências': 8.0,  
    'História': 7.5  
}
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Dicionários (dict)

Dicionários – Acessando elementos

Podemos acessar os valores associados às chaves de um dicionário utilizando as chaves como índices.

```
# Exemplo de acesso a elementos em um dicionário
```

```
pessoa = {  
    'nome': 'Maria',  
    'idade': 25,  
    'cidade': 'Rio de Janeiro'  
}
```

```
nome = pessoa['nome'] # 'Maria'
```

```
idade = pessoa['idade'] # 25
```

```
cidade = pessoa['cidade'] # 'Rio de Janeiro'
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Dicionários (dict)

Dicionários - Operações

Exemplo de operações com dicionários

```
pessoa = {  
    'nome': 'João',  
    'idade': 30,  
    'cidade': 'São Paulo'  
}  
  
#Adicionar novo elemento  
pessoa['profissao'] = 'Engenheiro'  
  
# Modificar valor associado a uma chave  
pessoa['idade'] = 31  
  
# Remover elemento  
del pessoa['cidade']
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Dicionários (dict)

Dicionários – Funções úteis

Python oferece algumas funções incorporadas para manipulação de dicionários, como `len()`, `keys()`, `values()`, `items()`.

```
# Exemplos de funções úteis para dicionários
```

```
pessoa = {  
    'nome': 'João',  
    'idade': 30,  
    'cidade': 'São Paulo'  
}
```

```
tamanho = len(pessoa) # Retorna o número de pares chave-valor (3)
```

```
chaves = pessoa.keys() # Retorna as chaves do dicionário ('nome', 'idade', 'cidade')
```

```
valores = pessoa.values() # Retorna os valores do dicionário ('João', 30, 'São Paulo')
```

```
pares = pessoa.items() # Retorna uma lista de tuplas com os pares chave-valor
```


Modulo 1: Unidade 2 – Variáveis e tipos de dados

Matrizes (array)

Dicionários – Funções úteis

Embora Python não tenha um tipo de dados nativo específico para matrizes, é possível representá-las usando listas de listas ou usando bibliotecas especializadas, como NumPy, que oferecem suporte avançado para manipulação de matrizes e cálculos matriciais.

Matrizes – Representando matrizes com listas de listas

Em Python, podemos representar matrizes usando listas de listas. Cada lista interna representa uma linha da matriz.

```
# Exemplo de representação de matriz com listas de listas
```

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

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Matrizes (array)

Matrizes - Acessando elementos

Podemos acessar elementos individuais de uma matriz utilizando índices para as linhas e colunas.

```
# Acessando elementos em uma matriz
```

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

```
elemento = matriz[1][2] # Acessando o elemento na segunda linha e terceira coluna (6)
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Matrizes (array)

Matrizes - Trabalhando com NumPy

```
# Exemplo de uso do NumPy para trabalhar com matrizes
```

```
import numpy as np
```

```
# Criando matrizes usando NumPy
```

```
matriz_a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
matriz_b = np.array([[9, 8, 7], [6, 5, 4], [3, 2, 1]])
```

```
# Adição de matrizes
```

```
soma = matriz_a + matriz_b
```

```
# Multiplicação por um escalar
```

```
escalar = 2
```

```
resultado = escalar * matriz_a
```

```
# Multiplicação de matrizes
```

```
produto = np.dot(matriz_a, matriz_b)
```

```
# Transposição de matriz
```

```
transposta = matriz_a.T
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Mudar o tipo de uma variável

Conversão para inteiro (int)

Exemplo 1: Conversão de um valor numérico para inteiro

```
valor_float = 3.14
```

```
inteiro = int(valor_float) # Resultado: 3
```

Exemplo 2: Conversão de uma string para inteiro

```
valor_string = "42"
```

```
inteiro = int(valor_string) # Resultado: 42
```

Conversão para lista (list)

Exemplo 1: Conversão de uma string para lista de caracteres

```
texto = "Python"
```

```
lista_caracteres = list(texto) # Resultado: ['P', 'y', 't', 'h', 'o', 'n']
```

Exemplo 2: Conversão de uma tupla para lista

```
tupla = (1, 2, 3)
```

```
lista = list(tupla) # Resultado: [1, 2, 3]
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Erros típicos relacionados ao tipo da variável

TypeError – Tentativa de soma entre tipos incompatíveis

Este erro ocorre quando tentamos realizar uma operação de soma entre tipos que não são compatíveis.

```
# Exemplo de TypeError
numero = 5
texto = "Python"
resultado = numero + texto # TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

ValueError – Conversão inválida de string para inteiro

Este erro ocorre quando tentamos converter uma string que não representa um número inteiro válido.

```
# Exemplo de ValueError
texto = "Python"
numero = int(texto) # ValueError: invalid literal for int() with base 10: 'Python'
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Erros típicos relacionados ao tipo da variável

KeyError – Acesso a uma chave inexistente em um dicionário

```
# Este erro ocorre quando tentamos acessar uma chave que não existe em um dicionário.  
# Exemplo de KeyError  
dicionario = {'a': 1, 'b': 2, 'c': 3}  
valor = dicionario['d'] # KeyError: 'd'
```

IndexError – Acesso a um índice inválido em uma lista ou tupla

```
# Este erro ocorre quando tentamos acessar um índice que não existe em uma lista ou tupla.  
lista = [1, 2, 3]  
elemento = lista[3] # IndexError: list index out of range
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Erros típicos relacionados ao tipo da variável

AttributeError – Tentativa de chamar um método inexistente em um objeto

Este erro ocorre quando tentamos chamar um método que não existe para o tipo de objeto em questão

```
texto = "Python"  
tamanho = texto.length() # AttributeError: 'str' object has no attribute 'length'
```

TypeError – Concatenação de sequências de tipos diferentes

Este erro ocorre quando tentamos concatenar sequências (como listas, tuplas ou strings) de tipos diferentes.

```
lista = [1, 2, 3]  
tupla = (4, 5, 6)  
resultado = lista + tupla # TypeError: can only concatenate list (not "tuple") to list
```

Modulo 1: Unidade 2 – Variáveis e tipos de dados

Erros típicos relacionados ao tipo da variável

TypeError – Comparação de tipos não comparáveis

Exemplo de TypeError

```
numero = 42
```

```
texto = "42"
```

```
resultado = numero < texto # TypeError: '<' not supported between instances of 'int' and 'str'
```

IMPORTANTE: uma boa prática adicionar tratamento de exceções (usando try-except) para capturar e lidar com erros relacionados a tipos de variáveis de forma adequada.

```
try:
```

```
    # code to run
```

```
except:
```

```
    # handle error
```


Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Operadores

Aritméticos

Os operadores aritméticos são utilizados para realizar operações matemáticas básicas em Python, como adição, subtração, multiplicação, divisão, entre outras.

<pre># adição a = 5 b = 3 resultado = a + b # Resultado: 8</pre>	<pre># subtração a = 10 b = 7 resultado = a - b # Resultado: 3</pre>
<pre># multiplicação a = 3 b = 4 resultado = a * b # Resultado: 12</pre>	<pre># divisão a = 10 b = 3 resultado = a / b # Resultado: 3.3333333333333335</pre>

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Operadores

Aritméticos

```
# divisão inteira
```

```
a = 10
```

```
b = 3
```

```
resultado = a // b # Resultado: 3
```

```
# resto da divisão
```

```
a = 10
```

```
b = 3
```

```
resto = a % b # Resultado: 1
```

```
# potência
```

```
a = 2
```

```
b = 3
```

```
resultado = a ** b # Resultado: 8
```

```
# precedência dos operadores
```

```
resultado = 2 + 3 * 4 # Resultado: 14 (multiplicação antes da adição)
```

```
resultado = (2 + 3) * 4 # Resultado: 20 (parênteses para alterar a ordem de avaliação)
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Operadores

Comparação

<pre># Igual (==) a = 5 b = 5 resultado = a == b # Resultado: True</pre>	<pre>Diferente (!=) a = 10 b = 7 resultado = a != b # Resultado: True</pre>
<pre># maior que (>) a = 10 b = 7 resultado = a > b # Resultado: True</pre>	<pre># menor que (<) a = 5 b = 8 resultado = a < b # Resultado: True</pre>
<pre># maior ou igual (>=) a = 5 b = 5 resultado = a >= b # Resultado: True</pre>	<pre># menor ou igual (<=) a = 10 b = 15 resultado = a <= b # Resultado: True</pre>

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop for

Loop for – Sintaxe

```
for elemento in sequencia:  
    # Código a ser executado para cada elemento
```

Loop for – Iteração com listas

```
# Exemplo de loop "for" com lista  
frutas = ['maçã', 'banana', 'laranja']  
for fruta in frutas:  
    print(fruta)
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop for

Loop for – Iteração com strings

```
# Exemplo de loop "for" com string
texto = 'Python'
for letra in texto:
    print(letra)
```

Loop for – Iteração com intervalo numérico

```
# Exemplo de loop "for" com
intervalo numérico
for numero in range(1, 6):
    print(numero)
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop for

Loop for – Função "range()" com passo

```
# Exemplo de loop "for" com intervalo numérico e passo
for numero in range(1, 11, 2):
    print(numero)
```

Loop for – Lista de tuplas

```
# Exemplo de loop "for" com lista de tuplas
pessoas = [('João', 25), ('Maria', 30), ('Pedro', 22)]
for nome, idade in pessoas:
    print(f"{nome} tem {idade} anos.")
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop for

Loop for – Dicionário

```
# Exemplo de loop "for" com dicionário
idades = {'João': 25, 'Maria': 30, 'Pedro': 22}

# Iteração sobre as chaves
for nome in idades:
    print(nome)

# Iteração sobre os valores
for idade in idades.values():
    print(idade)
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop for

Loop for – Dicionário e Enumerando elementos com "enumerate()"

```
# Exemplo de loop "for" com dicionário
```

```
idades = {'João': 25, 'Maria': 30, 'Pedro': 22}
```

```
# Iteração sobre as chaves
```

```
for nome in idades:
```

```
    print(nome)
```

```
# Iteração sobre os valores
```

```
for idade in idades.values():
```

```
    print(idade)
```

```
# Exemplo de uso de "enumerate()"
```

```
frutas = ['maçã', 'banana', 'laranja']
```

```
for indice, fruta in enumerate(frutas):
```

```
    print(f"Índice: {indice}, Fruta: {fruta}")
```


Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop while

Loop while – Sintaxe

```
while condição:  
    # Código a ser executado enquanto a condição for verdadeira
```

Loop while – Execução

O loop "while" continuará a ser executado enquanto a condição especificada for avaliada como "True". É importante garantir que a condição seja eventualmente "False", caso contrário, o loop continuará indefinidamente, resultando em um loop infinito.

```
# Exemplo de loop "while"  
contador = 0  
while contador < 5:  
    print(contador)  
    contador += 1
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop while

Loop while – Loop infinito e o comando "break"

```
# Exemplo de loop "while" com "break"
```

```
contador = 0
```

```
while True:
```

```
    print(contador)
```

```
    contador += 1
```

```
    if contador >= 5:
```

```
        break
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop while

Loop while – Comando "continue"

Exemplo de loop "while" com "continue"

```
contador = 0
while contador < 5:
    contador += 1
    if contador == 3:
        continue # Pula a iteração quando contador == 3
    print(contador)
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Loop while

Loop while – Iterando sobre listas

Exemplo de loop "while" para iterar sobre lista

```
frutas = ['maçã', 'banana', 'laranja']
```

```
indice = 0
```

```
while indice < len(frutas):
```

```
    print(frutas[indice])
```

```
    indice += 1
```

Modulo 1: Unidade 3 – Operadores e Estruturas de Repetição

Estruturas de repetição

Função range()

Sintaxe

```
range(início, fim, passo)
```

Utilizando "range()" para criar listas

```
# Exemplo: Criando uma lista de números pares
```

```
numeros_pares = list(range(2, 11, 2))
```

```
print(numeros_pares)
```

```
# Saída: [2, 4, 6, 8, 10]
```

Modulo 1: Unidade 4 – Estruturas de dados

As estruturas de dados em Python são ferramentas fundamentais para organizar, armazenar e manipular dados de maneira eficiente.

Listas

Operações com listas

```
numeros = [1, 2, 3]
```

Adicionar elemento ao final da lista

```
numeros.append(4) # números agora é [1, 2, 3, 4]
```

Remover elemento específico da lista

```
numeros.remove(2) # números agora é [1, 3, 4]
```

Verificar o tamanho da lista

```
tamanho = len(numeros) # tamanho é 3
```

Concatenar duas listas

```
outra_lista = [5, 6, 7]
```

```
concatenada = numeros + outra_lista # concatenada é [1, 3, 4, 5, 6, 7]
```

Modulo 1: Unidade 4 – Estruturas de dados

Listas

Removendo elementos de uma lista

```
frutas = ['maçã', 'banana', 'laranja']

del frutas[1] # Remove o elemento na posição 1 (banana)
print(frutas) # Saída: ['maçã', 'laranja']
```

Removendo elementos de um dicionário

```
peessoas = {'João': 25, 'Maria': 30, 'Pedro': 22}

del pessoas['Maria']
print(pessoas) # Saída: {'João': 25, 'Pedro': 22}
```

Modulo 1: Unidade 4 – Estruturas de dados

Conjuntos

Para criar um conjunto em Python, utilizamos chaves {} ou a função set(). Vamos ver alguns exemplos:

```
# Exemplo 1: Conjunto com chaves {}
```

```
frutas = {'maçã', 'banana', 'laranja'}
```

```
# Exemplo 2: Conjunto com a função set()
```

```
numeros = set([1, 2, 3, 4, 5])
```


Modulo 1: Unidade 4 – Estruturas de dados

Conjuntos

Operações

```
conj1 = {1, 2, 3}
conj2 = {3, 4, 5}

# Adicionar elemento
conj1.add(4) # conj1 agora é {1, 2, 3, 4}

# Remover elemento
conj1.remove(2) # conj1 agora é {1, 3, 4}

# Verificar a existência de um elemento
if 3 in conj1:
    print("O número 3 está no conjunto conj1.")

# União de conjuntos
uniao = conj1.union(conj2) # uniao é {1, 3, 4, 5}

# Interseção de conjuntos
intersecao = conj1.intersection(conj2) # intersecao é {3}

# Diferença entre conjuntos
diferenca = conj1.difference(conj2) # diferenca é {1, 4}
```

Modulo 1: Unidade 4 – Estruturas de dados

Técnicas de iteração

Ao iterar sobre dicionários, a chave e o valor correspondente podem ser obtidos utilizando o método “items()”.

```
knight = {'gallahad': 'the pure', 'robin': 'the brave'}  
for k, v in knight.items():  
    print(k, v)
```

Resultado

```
gallahad the pure  
robin the brave
```



 (85) 99115-1117

 www.instagram.com/avanti.ia/

 www.linkedin.com/company/avantiatlantico

www.atlanticoavanti.com.br