

# Programação Python



## Boas Vindas!

Seja muito bem-vindo(a) ao curso de Programação Python.

Durante nosso curso, apresentaremos várias ferramentas, recursos e conceitos que serão úteis para modelarmos as aplicações que vamos desenvolver das mais variadas formas. É importante ressaltar que em programação, não existe apenas uma forma de resolver um problema, ou seja, quanto mais recursos conhecermos, maior será a chance de encontrarmos a melhor forma de resolver problemas.

Ao final do mesmo você conseguirá implementar códigos em linguagem Python e utilizar os conceitos necessários para a resolução de problemas comuns à área de programação de computadores e desenvolvimento de softwares para desktop e Web.

## 1. INTRODUÇÃO À LINGUAGEM PYTHON

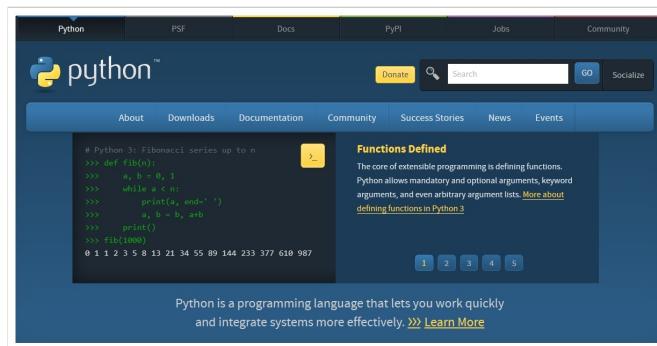
Python é linguagem de alto nível utilizada em vários domínios de aplicação, como por exemplo, para o desenvolvimento Web, arquivos de dados, automação de aplicações, aprendizagem de máquina e etc. Em sua construção, podemos visualizar as características de simplicidade e legibilidade.

### 1.1 CARACTERÍSTICAS E INSTALAÇÃO DO AMBIENTE DE DESENVOLVIMENTO

Nesta seção apresentaremos o ambiente de programação Python, onde podemos baixá-lo e a estrutura básica de programação para a ferramenta selecionada.

#### 1.1.1 Instalação do Python

Para iniciarmos a programação das aplicações e executar os testes com a linguagem, precisamos inicialmente instalar o seu ambiente de desenvolvimento. Para tal, podemos realizar o *download* do ambiente no website oficial da linguagem em <<https://www.python.org>>. Figura 1, podemos visualizar a página inicial do mesmo.



**Figura 1:** Site para o *download* do ambiente de desenvolvimento *Python*.

## 1.1.2 Ambiente de Desenvolvimento

O ambiente de desenvolvimento que será utilizado para os exemplos e exercícios do nosso curso será o *IDLE - Integrated Development and Learning Environment* ou Ambiente Integrado de Desenvolvimento e Aprendizagem, que apesar de razoavelmente simples é bastante útil. O ambiente não faz parte da linguagem, mas vem sendo lançado conjuntamente em seu pacote desde a sua versão 2.3.

É importante ressaltar que para a programação em linguagem *Python* podemos utilizar vários *IDEs - Integrated Development Environments*<sup>1</sup> ou Ambientes de Desenvolvimento Integrados disponíveis no mercado, como por exemplo, o *Visual Studio Code* <<https://code.visualstudio.com/download>>, o *PyCharm* <<https://www.jetbrains.com/pycharm/>>, o *Jupyter* <<https://jupyter.org/>>, o *Google Colab* <<https://colab.research.google.com/notebook>>, o *IDLE*, entre outros. A tela inicial do ambiente de desenvolvimento b do website oficial da linguagem, é ilustrada na Figura 2.



**Figura 2:** Ilustração da tela do *console* inicial (*Shell*) do ambiente de desenvolvimento *IDLE*.

Na Figura 3, podemos visualizar um exemplo de execução da primeira interação com a linguagem *Python*, através da linha de código inserida: `print("Olá Mundo!!!")` e o seu retorno (execução) na Figura 4, onde podemos visualizar a saída da referida linha de código (impressão) na tela do computador.

```
print("Olá Mundo !!!")
```

**Figura 3:** Exemplo de impressão inicial.

```
Olá Mundo !!!
```

**Figura 4:** Visualização da saída do código.

Este é um programa *Python* que imprime a frase "**Olá Mundo!!!**" no *console*. O `print()` é uma função embutida da linguagem que exibe o texto que está entre as aspas.

<sup>1</sup> São softwares que reúnem de forma integrada e geralmente com interface gráfica, ferramentas para a edição de códigos-fonte, automação da interpretação e/ou compilação dos códigos (programas) e ferramental de debugger - debugador e/ou otimizador dos códigos desenvolvidos.

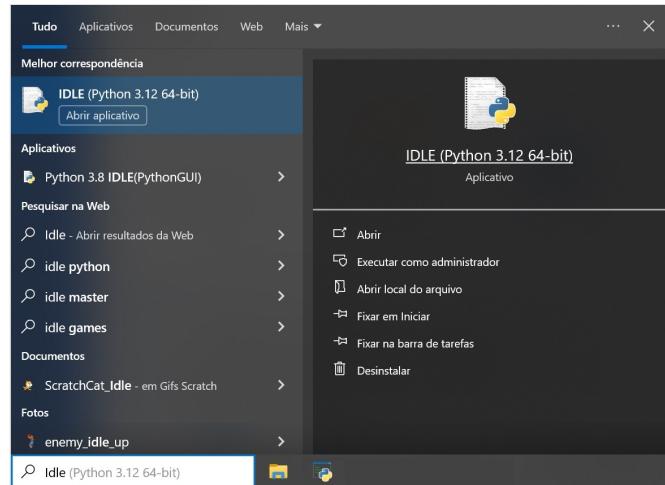
## 2. SINTAXE BÁSICA DE PYTHON

Nesta seção apresentaremos o raciocínio básico que devemos seguir para iniciar em programação Python. As definições de variáveis e de como se declarar corretamente as mesmas também serão exploradas no decorrer do texto.

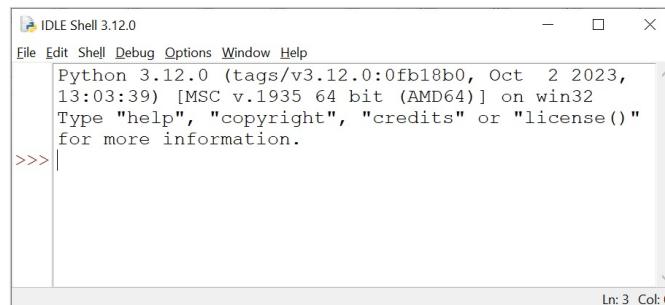
### Atenção

A extensão de arquivos Python é a: **.py** e tanto no **IDLE** como em qualquer outro IDE, você poderá executar e salvar seus programas para posteriormente alterá-los, evitando a necessidade de redigitação e é claro, aumentando a sua produtividade e melhorando a organização dos mesmos em pastas, diretórios e/ou pacotes.

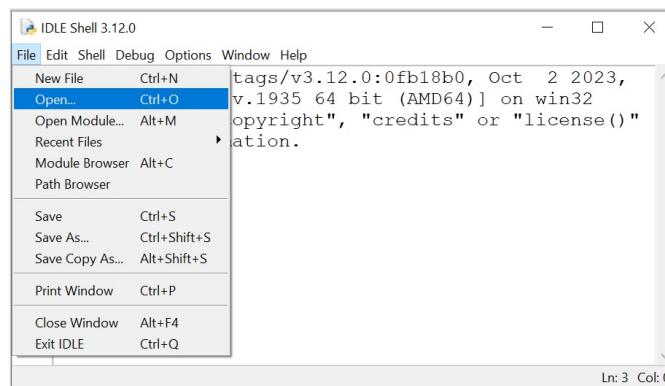
No exemplo inicial que trouxemos, nosso programa Python foi nomeado como **helloworld.py** e o mesmo foi codificado, salvo e depois carregado para execução, como ilustrado nas Figuras I, II, III, IV e V deste quadro.



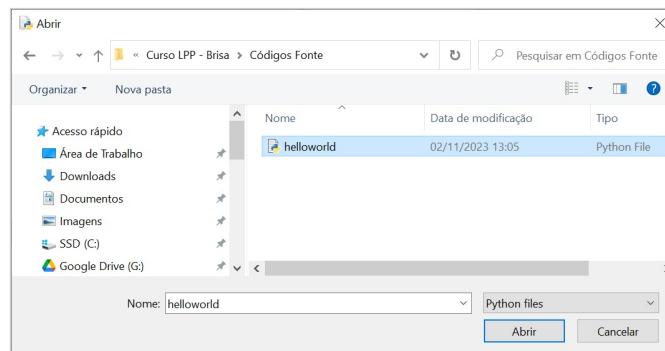
**Figura I:** Chamada do **IDLE** via barra de tarefas do **Windows**.



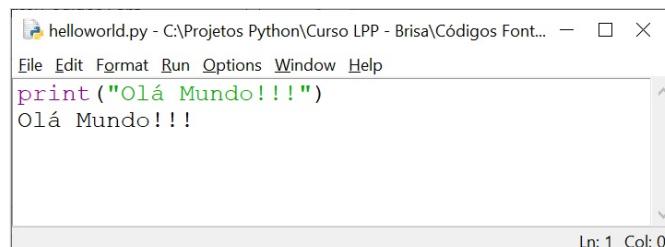
**Figura II:** Tela inicial do **IDLE** em execução.



**Figura III:** Acesso ao menu **File**, opção **Open...** do *IDLE* para busca de arquivos.



**Figura IV:** Arquivo .py encontrado e pronto para abertura.



**Figura V:** Arquivo .py aberto e executado automaticamente.

**Nota:** Com o arquivo aberto, você pode alterá-lo e executá-lo novamente para realizar testes acionando o Menu **Run** opção **R Module** e/ou simplesmente teclando **F5**.

## 2.1 Variáveis

A declaração de variáveis implica na atribuição de valores de forma direta. Em Python, precisamos apenas nomear a variável e atribuído, ou seja, não precisamos da definição do seu tipo de dado.

Neste contexto, na Figura 5 é ilustrado o processo correto de atribuição de valores.



**Figura 5:** Declaração de variável e atribuição de um valor.

### Importante

Para a atribuição de nomes em *Python*, devemos selecionar palavras no infinitivo impessoal, as quais realmente representem que a variável, função, classe ou método representarão em seu código. E para a maior legibilidade dos nomes, é importante que utilizemos letras maiúsculas e minúsculas para diferenciar as palavras.

**Exemplos:** **NumerosFixos**, **JurosSimples**, **\_JurosCompostos**, **MediaAnual**, **Saldo\_CC** e etc.

## 2.2 Tipos de Dados e Operadores

Em *Python* não precisamos de forma explícita, da definição do tipo de dado, ou seja, ao definir o valor que será passado para a variável, automaticamente implicamos o tipo de dado associado àquele nome.

São tipos de dados possíveis na linguagem:

- **Numéricos:** Nesta categoria estão os tipos de dados relacionados a atividades numéricas.
  - Possuímos o tipo inteiro (**int**).
    - Este tipo de dado possibilita armazenarmos dados numéricos inteiros.
  - **Exemplo(s):**
    - **Idade = 18**
    - **AtributoForca = 15**
    - **\_FatorCorrecao = 13**
  - E o tipo real (**float**).
    - Este tipo de dado possibilita armazenarmos dados numéricos reais ou fracionários.
  - **Exemplo(s):**
    - **Altura = 1.70**
    - **\_JurosSimples = 17.50**
    - **JurosCompostos = 19.00**
- **Literais:** Categoria que possibilita atribuirmos valores textuais.
  - Definição do tipo “string” (**str**).
    - É importante ressaltar que para dados do tipo literal devemos utilizar aspas duplas “ ” para sinalizar que o valor é textual.
  - **Exemplo(s):**
    - **Nome= “Eduardo”**
    - **TituloAplicacao=“Exemplos de Software”**
- **Lógicos:** São definidos como verdadeiros e falsos.
  - Na programação *Python*, o nome associado ao tipo de dado lógico é o (**bool**) ou booleano.
  - **Exemplo(s):**
    - **TelaAtiva = True**
    - **\_ValidarDados = False**

Nas Figuras 6 e 7, podemos visualizar a aplicação dos conceitos definidos sobre a definição dos tipos de dados descritos até aqui sua impressão em tela. Como podemos visualizar, não é necessário escrevermos o tipo da variável, pois o mesmo é de forma implícita atribuída quando declaramos o seu conteúdo.

```
# Declaração de variáveis
idade = 30
altura = 1.70
nome = "Eduardo"
ativo_para_acesso = True
restricao = False

# Impressão das variáveis
print("Idade:", idade)
print("Altura:", altura)
print("Nome:", nome)
print("AtivoParaAcesso:", ativo_para_acesso)
print("Restricao:", restricao)
```

**Figura 6:** Atribuição de valores e definição de tipos de dados de forma implícita.

```
Idade: 30
Altura: 1.7
Nome: Eduardo
AtivoParaAcesso: True
Restricao: False
```

**Figura 7:** Execução do programa para visualizar a atribuição de valores atribuídos às variáveis.

É importante ressaltar que em Python podemos também realizar a reatribuição de valores de forma simples e implícita às variáveis acima. A afirmação acima implica na situação onde podemos definir uma variável com um valor e posteriormente, atribuir um valor de outro tipo. O interpretador da linguagem permitirá tal execução.

Nas Figuras 8 e 9, os exemplos de atribuição, reatribuição e a impressão em tela dos valores trabalhados são apresentados. Como o programa é executado linha por linha, quando executamos o comando `print()`, a saída será realizada com as últimas informações inseridas.

```
# Declaração de variáveis
idade = 30
nome = "Eduardo"

# Atribuição de novos valores
idade = 45
nome = "Luis"

# Imprimir variáveis
print("Nome:", nome)
print("Idade:", idade)
```

**Figura 8:** Exemplo de atribuição e reatribuição de valores.

```
Nome: Luis
Idade: 45
```

**Figura 9:** Impressão das operações de reatribuição de valores.

### Importante

Variáveis podem ser globais ou locais. As variáveis globais são visualizadas por todas as instâncias do programa, como se fossem um recurso público. Já as variáveis locais são aninhadas internamente na função principal do programa, ou seja, elas estão internas à função e dessa forma, podemos classificá-las como privadas.

Nas Figuras 10 e 11, podemos visualizar em um exemplo a implementação das variáveis locais e globais. Devemos notar que as globais ficam de fora das funções e as internas, são as locais, uma vez que estão condicionadas à chamada daquela função.

```
# Variável global
nome_global = "Eduardo"

def funcao_local():
    # Variável local
    nome_local = "Luis"
    print("Dentro da função local, " \
          + " " \
          + "o nome é: ", nome_local)

# Chamando a função
funcao_local()

#Acesso à variável global
print("Fora da função, " \
      + " " \
      + "o nome_global é: ", nome_global)
```

**Figura 10:** Exemplo de variáveis locais e globais.

```
Dentro da função local, o nome é:
Luis
Fora da função, o nome_global é:
Eduardo
```

**Figura 11:** Exemplo de impressão das variáveis locais e globais.

Nas Figuras acima, pudemos visualizar que as variáveis globais, podem ser lidas de qualquer local no código, ou seja, dentro ou de qualquer função. Por outro lado, as locais, conforme o próprio nome, podem ser executadas apenas de dentro da função que a declarou.

Com relação aos operadores, em *Python* podemos separá-los em:

- **Operadores aritméticos:**

- Os operadores aritméticos estão associados à matemática.

- **Exemplo(s):**

- **Operador de atribuição:**

- Realiza a passagem de um valor para a variável. Símbolo: =

- **Operador de adição:**

- Necessário para somar informações na construção de uma expressão. Símbolo: +

- **Operador de subtração:**

- Utilizado para subtrair informações entre variáveis na construção de uma expressão. Símbolo: -

- **Operador de multiplicação:**

- Necessário para multiplicar informações entre variáveis na construção de uma expressão. Símbolo: \*

- **Operador de divisão:**

- A operação de divisão é utilizada na construção das expressões aplicadas às variáveis de construção do programa (resultante em número real). Símbolo: /

- **Operador de divisão inteira:**

- Operador necessário para implementarmos uma expressão de cálculo (resultante em número inteiro). Símbolo: //

- **Operador de módulo:**

- Operador que possibilita o retorno do resto da divisão. Símbolo: %

- **Operador de exponenciação:**

- Operador de cálculo de exponenciação aplicado na construção de expressões e variáveis. Símbolo: \*\*

A Figura 12 apresenta os conceitos de operações aritméticas aplicados em *Python*. O código implementado possui como objetivo apenas exemplificar a utilização dos operadores, portanto para os mesmos não implementamos os **prints** para exibição de seus resultados (saídas).

```
#Declaração da variável
variavel_a = 10
variavel_b = 3
soma = variavel_a + variavel_b

#Operandos aritméticos
subtracao = variavel_a - variavel_b
multiplicacao = variavel_a * variavel_b
divisao = variavel_a / variavel_b
divisao_inteira = variavel_a // variavel_b
resto = variavel_a % variavel_b
potencia = variavel_a ** variavel_b
```

**Figura 12:** Operadores aritméticos na construção de expressões.

As Figuras 13 e 14, ilustram a aplicação dos operadores de atribuição com a inicialização da **variavel\_a** com o valor inteiro **5**, podemos então visualizar a saída dos resultados produzidos por cada cálculo aplicado. Devemos prestar atenção na estrutura dos cálculos, os quais funcionam da seguinte forma, para cada operador aritmético associado com o símbolo de igualdade, temos na rea a seguinte leitura: A expressão **variável\_a += 2** é equivalente a escrevemos o código-fonte da seguinte forma: **variável\_a = variável\_a + 2**. É importante contextualizar que este padrão se repete com os outros operadores.

```
#Declaração da variável
variavel_a = 5

#Execução dos operandos de atribuição
variavel_a += 2
print("Operando de soma: ", variavel_a)

variavel_a -= 2
print("Operando de subtração: ", variavel_a)

variavel_a *= 2
print("Operando de multiplicação: ", variavel_a)

variavel_a /= 2
print("Operando de divisão: ", variavel_a)

variavel_a //= 2
print("Operando de divisão inteira: ", variavel_a)

variavel_a %= 2
print("Operando de módulo: ", variavel_a)

variavel_a **= 2
print("Operando de exponenciação", variavel_a)
```

**Figura 13:** Exemplos de declaração e execução dos operadores aritméticos.

```

Operando de soma: 7
Operando de subtração: 5
Operando de multiplicação: 10
Operando de divisão: 5.0
Operando de divisão inteira: 2.0
Operando de módulo: 0.0
Operando de exponenciação 0.0

```

**Figura 14:** Saída da execução dos operadores aritméticos.

- **Operadores de comparação:**

- **Igual a:**

- O operador de igualdade analisa se a informação/valor da variável **VariavelA** é igual da informação/valor incutida na **VariavelB**. Símbolo: **==**
    - **VariavelA == VariavelB**

- **Diferente de:**

- O operador de diferença analisa se a informação/valor da **VariavelA** é diferente da informação/valor incutida na **VariavelB**. Símbolo: **!=**
    - **VariavelA != VariavelB**

- **Menor que:**

- Com relação aos operadores, podemos também visualizar se a **VariavelA** é menor que a **VariavelB**. Nesta estrutura, teremos a saída verdadeira se a **VariavelA** for de fato menor que a **VariavelB**. Símbolo: **<**
    - **VariavelA < VariavelB**

- **Maior que:**

- O operador “maior que” implica na verificação se a **VariavelA** é maior que a **VariavelB**. Neste contexto, a saída será verdadeira se a **VariavelA** for de fato maior que a **VariavelB**. Símbolo: **>**
    - **VariavelA > VariavelB**

- **Menor ou igual a:**

- Este operador é definido como complexo, uma vez que, gerará uma saída verdadeira se a **VariavelA** for menor ou igual a **VariavelB**. Símbolo: **<=**
    - **VariavelA <= VariavelB**

- **Maior ou igual a:**

- Este operador também é definido como complexo, uma vez que, gerará uma saída verdadeira se a **VariavelA** for maior ou igual a **VariavelB**. Símbolo: **>=**
    - **VariavelA>= VariavelB**

Nas Figuras 15 e 16, podemos visualizar a execução dos operadores de comparação. Na comparação sempre lemos da esquerda para a direita, então, para verificarmos a igualdade: **variavel\_a == variavel\_b** lemos: A **variavel\_a** é igual a **variavel\_b**. Lembrando para a atribuição, utilizáramos apenas **=** como símbolo de atribuição: **variavel\_a = variavel\_b** e leríamos: A **variavel\_a** recebe o valor da **variavel\_b**, ou seja, à **variavel\_a** é atribuído o valor presente na **variavel\_b**.

```

#Inicialização de variáveis
variavel_a = 10
variavel_b = 5

#Execução dos operadores de comparação
igual = variavel_a == variavel_b
print("Operador de igualdade: ", igual)

diferente = variavel_a != variavel_b
print("Operador de diferença: ", diferente)

maior = variavel_a > variavel_b
print("Operador Maior: ", maior)

menor = variavel_a < variavel_b
print("Operador Menor: ", menor)

maior_igual = variavel_a >= variavel_b
print("Operador Maior e Igual: ", maior_igual)

menor_igual = variavel_a <= variavel_b
print("Operador Menor e Igual: ", menor_igual)

```

**Figura 15:** Inicialização das variáveis e execução dos operadores de igualdade.

```

Operador de igualdade: False
Operador de diferença: True
Operador Maior: True
Operador Menor: False
Operador Maior e Igual: True
Operador Menor e Igual: False

```

**Figura 16:** Saída de execução dos operadores de comparação.

- **Operadores lógicos:**

- Os operadores lógicos podem ser implementados conjuntamente com testes condicionais para avaliar as entradas e gerar saída dependente destas entradas.
- Estes operadores estão associados às portas lógicas. Python possuí três operadores lógicos principais: “**AND**”, “**OR**” e “**NOT**”.
- De forma simplificada, o operador “**and**” possuirá saída verdadeira se ambas as entradas forem verdadeiras. Lemos este operador como “**E**”.
- O operador “**or**” gera uma saída verdadeira se ao menos uma das entradas for verdadeira. Lemos este operador como “**C**”.
- O operador “**not**” é a negação ou inversão da expressão lógica utilizada. Com ele podemos testar se uma operação não é verdadeira: “**not**”**true** ou mesmo inverter uma porta lógica “**and not**” (neste caso a entrada à direita da expressão é falsa conta do inveror).

Nas Figuras 17 e 18 podemos visualizar exemplos relacionados aos operadores lógicos. Na sua composição, foram utilizadas diferentes comparações para testar diferentes valores e suas condições de saída.

```

#Inicialização de variáveis
idade = 25
maior_idade = 18
renda = 30000
renda_base = 25000

#Execução das operações lógicas
elegivel = idade >= maior_idade and renda > renda_base
print("Saída do operador AND: ", elegivel)

#Atualização de valores
renda = 20000

elegivel = idade >= maior_idade or renda > renda_base
print("Saída do operador OR: ", elegivel)

elegivel = idade >= maior_idade and not renda > renda_base
print("Saída do operador NOT: ", elegivel)

#Atualização de valores
renda = 30000

elegivel = idade >= maior_idade and not renda > renda_base
print("Saída do operador NOT: ", elegivel)

```

**Figura 17:** Exemplo de execução de operações lógicas com a atualização de valores nas variáveis.

```

Saída do operador AND: True
Saída do operador OR: True
Saída do operador NOT: True
Saída do operador NOT: False

```

**Figura 18:** Execução das operações lógicas e as suas respectivas saídas.

Nas Figuras acima, verificamos inicialmente se um cliente hipotético possui idade maior que 18 anos e posteriormente a este teste analisamos se ele possui também renda maior do que o parâmetro financeiro de base.

Na segunda operação, atualizamos a renda do cliente para um valor inferior. Agora verificamos se possuímos apenas uma condição verdadeira, como baixamos a renda do cliente para abaixo da **renda\_base**, a saída será verdadeira (**True**).

O terceiro teste possui saída verdadeira também, uma vez que, o cliente possui idade maior que 18 anos, porém, quando usamos o “**and not**”, invertemos a lógica, então o que está à sua direita que a princípio seria falso e geraria uma saída também falsa é inverso que satisfaz a condição da porta lógica “**AND**” em termos de duas entradas verdadeiras para termos uma saída verdadeira.

No teste final, geramos então uma saída falsa, uma vez que, antes colocamos a renda do cliente superior à **renda\_base** e desta forma, como utilizamos o “**and not**”, o que geraria duas entradas verdadeiras e uma saída verdadeira, passamos a possuir então uma entrada verdadeira e outra falsa, e neste caso, a saída será falsa.

- **Operadores de identidade:**

- Os operadores de identidade apontam para objetos. Os mesmos possuem saída lógica e necessitam de ao menos duas variáveis.

- **Operações:**

- **is**
- **is not**

Nas Figuras 19 e 20, podemos visualizar as informações relacionadas a itens presentes em listas.

```

#Criação das listas para aplicação
# dos operandos de identidade
lista_1 = [1, 2, 3]
lista_2 = lista_1
lista_3 = [4, 5, 6]

#Execução dos operadores
mesmo_objeto = lista_1 is lista_2
print("Esta incluido: ", mesmo_objeto)

objeto_diferente = lista_1 is not lista_3
print("Esta incluido: ", objeto_diferente)

```

**Figura 19:** Implementação dos operadores de identidade.

Esta incluido: True  
Esta incluido: True

**Figura 20:** Saída de execução da implementação dos operadores de identidade.

Como pudemos visualizar, através do “**is**”, verificamos se na “**lista\_1**” possuímos itens presentes na “**lista\_2**”. E no “**is not**”, verificamos se os itens presentes na “**lista\_1**” não estão presentes na “**lista\_3**”. No teste do “**mesmo\_objeto**”, as saídas são verdadeiras (**True**), uma vez que, “**lista\_1**” e “**lista\_2**” possuem o mesmo conteúdo e por isso a saída é verdadeira e no “**objeto\_diferente**”, a saída é verdadeira porque o conteúdo da “**lista\_1**” é diferente do conteúdo da “**lista\_3**”.

## 2.3 Expressões e Condicionais

Em Python as expressões são um poderoso recurso de resolução de problemas, para o cálculo de operações com variáveis e também para a manipulação de informações. Em sua construção, possuímos uma ordem de precedência, ou seja, uma ordem de execução dos operadores, a qual segue a ordem de precedência estabelecida na sigla PEMDAS (Parênteses, Exponenciação, Multiplicação, Divisão, Adição e Subtração).

Nas Figuras 21 e 22, podemos visualizar exemplos de construção de algumas expressões matemáticas e as suas respectivas saídas (impressão).

```

#Inicialização de variáveis
nota_a = 10
nota_b = 10
nome = "Eduardo"
sobrenome = "Sei"

#Execução da estrutura para impressão em tela
nome_completo = nome + " " + sobrenome
print("Qual o nome do desenvolvedor: ", nome_completo)

media = nota_a + nota_b / 2
print("Média de notas sem PEMDAS: ", media)

media = (nota_a + nota_b) / 2
print("Média de notas com PEMDAS: ", media)

```

**Figura 21:** Definição das variáveis e a execução de expressões.

Qual o nome do desenvolvedor: Eduardo Sei  
Média de notas sem PEMDAS: 15.0  
Média de notas com PEMDAS: 10.0

**Figura 22:** Saída de execução do programa desenvolvido anteriormente.

Nas Figuras acima, podemos visualizar que, sem o respeito da ordem de execução dos cálculos dentro de uma expressão matemática, o resultado apresentado pode não ser o que esperamos. O entendimento da utilização da sigla PEMDAS é, portanto, essencial para ordenamento da execução dos operadores, e para assim garantirmos que os resultados gerados sejam corretamente calculados.

Com relação às condicionais, em Python possuímos a estrutura “**if**” e “**else**” que nos possibilitam testar a manipulação de dado com base em condições de execução. Por exemplo, podemos construir uma estrutura onde “**SE**” ou “**if**” a pessoa tirou uma nota maior ou igual a 7, estará aprovada e o contrário “**SENÃO**” ou “**else**”, estará reprovada.

Nas Figuras 23 e 24, podemos visualizar os exemplos de implementação desta estrutura em Python.

```
#Inicialização de variáveis
media_escolar = 7
nota_do_aluno = 7

if(nota_do_aluno >= media_escolar):
    print("Aprovado")

else:
    print("Reprovado")
```

**Figura 23:** Exemplo de implementação da estrutura condicional.


Aprovado

**Figura 24:** Exemplo de saída da estrutura condicional.

### 3. ESTRUTURAS DE DADOS EM PYTHON

Nesta seção apresentaremos as estruturas essenciais para a programação em linguagem Python. Definiremos a utilização de listas, tuplas, conjuntos e dicionários. Apresentaremos as suas diferenças e exemplos de aplicação.

#### 3.1 Listas

Listas podem ser entendidas como sequências mutáveis, ou seja, sequências que podemos modificar e que possuímos o controle de ordenamento dos dados.

Em sua estrutura, as listas podem alocar de forma heterogênea, todos os tipos de dados possíveis em sua construção ou utilização. Na sua composição, possuímos o nome da lista e a atribuição de valores entre “[ ]” colchetes, e os dados alocados na lista seguem indexação (índice) de elementos como na construção de uma matriz.

A indexação citada se trata da identificação da posição do elemento em uma lista, como por exemplo, se declararmos: **lista = [“Banana”, “maçã”, “uva”]**, neste exemplo, a “**Banana**” possui índice igual a **0**, a maçã índice **1** e por aí em diante (a indexação vai da esquerda para a direita e deve ser lida do **0** até “**n**” elementos). Exemplos de implementação são apresentados mais adiante nesta seção.

#### 3.2 Tuplas

As tuplas possuem caráter imutável, ou seja, os valores alocados não podem ser alterados durante a sua execução. Na sua composição, adicionamos seu nome e entre “( )” parênteses os valores que ficarão armazenados na tupla.

Tuplas possuem a sua implementação parecida com as listas, porém, reflete a criação de estruturas onde não podemos alterar valores ou posição. Em uma situação real, a utilizaremos para estabelecer valores. Veremos mais a frente um exemplo de sua implementação.

### 3.3 Conjuntos

Em conjuntos, possuímos uma estrutura única, imutável e não ordenada de informações. No caso da estrutura de execução dos conjuntos, possuímos os dados ou valores armazenados em “{ }” chaves. Outra forma de se criar um conjunto é através da função “**set()**”.

#### São conceitos e características de sua implementação:

- A não existência de indexação.
- A utilização dos comandos “in” ou “not in”, de forma semelhante ao que vimos anteriormente com as listas, para verificarmos valor está ou não presente num determinado conjunto.
- Possibilidade de aplicarmos operações de **união**, **interseção** e **diferença**.
  - A **união** implica na criação de um terceiro conjunto com todos os elementos de ambos os conjuntos selecionados. O símbolo utilizado para a união é o (pipe) “|”.
  - A **interseção** é a operação de criação de um conjunto com os elementos que estão presentes em ambos os conjuntos relacionados. O símbolo utilizado para a interseção é o (e comercial) “&”.
  - A operação de **diferença** possibilita que criemos um conjunto onde alocaremos apenas os valores que não estão em ambos conjuntos. O símbolo utilizado para a diferença é o de (subtração) “-”.
- Para as operações entre conjuntos:
  - Podemos utilizar o **add()** para a inserção de elementos nos conjuntos.
  - Possuímos para a remoção de elementos, os comandos **remove()** e **discard()**.
    - **remove()** - na execução de sua operação, podemos gerar um erro.
    - **discard()** - sua execução, mesmo que os elementos não sejam encontrados, nenhum erro de sua aplicação é gerado.

Mais à frente apresentaremos exemplos relacionados a conjuntos em *Python*.

### 3.4 Dicionários

Se trata de uma estrutura de dados complexa para armazenar um conjunto de pares chaves-valor, as chaves são únicas e o valor pode ser duplicado. Os dicionários podem ser definidos como mapeamentos.

Na criação de um dicionário, declaramos uma variável com seu nome e para a atribuição de valores, utilizamos “{ }”. Em sua implementação, possuímos: **exemplo\_dicionario = {“nome” : “Eduardo”}**. Como podemos visualizar, dentro das chaves “{ }”, à esquerda vemos o nome da variável (chave) e à direita, o seu valor.

Em suas operações possuímos funções específicas para cada ação, como por exemplo:

- A função **get()** possibilita o retorno de informações dentro do dicionário.
- As funções **del()** ou **pop()**, possibilitam apagarmos as informações do dicionário através do apontamento da chave.
- As funções **keys()** e **values()** possibilitam respectivamente que listemos todas as chaves ou valores do dicionário selecionado.
- O método **popitem()** possibilita que removamos e apresentemos o último item do dicionário.
- O método **pop()** necessita que apontemos a chave do item que vamos deletar do dicionário.

### 3.5 Exemplos de Aplicação

Na Figura 25, visualizamos a estrutura de lista e inserção de informações em *Python*. Como podemos evidenciar, as listas trabalham com dados heterogêneos, ou seja, cada declaração de uma lista conta com diferentes tipos de dados. A Figura 26 apresenta a saída da estrutura de lista implementada.

```
#Inicialização de listas
lista_numeros = [1, 2, 3, 4, 5]
lista_strings = ['a', 'b', 'c', 'd']
lista_mista = [10, 'Oi', 3.14, True]
lista_vazia = []

#Implementação e execução das listas
print("Números: ", lista_numeros)
print("Informações: ", lista_strings)
print("Mista: ", lista_mista)
print("Vazia: ", lista_vazia)
```

**Figura 25:** Inicialização de listas e execução das estruturas.

```
Números: [1, 2, 3, 4, 5]
Informações: ['a', 'b', 'c', 'd']
Mista: [10, 'Oi', 3.14, True]
Vazia: []
```

**Figura 26:** Saída e execução de dados construídos para a implementação de listas.

Na Figura 27, um exemplo de tupla é estruturado, em sua implementação podemos visualizar que a declaração é diferente das listas anteriormente apresentadas. Entretanto, algumas semelhanças podem ser vistas, como por exemplo, o caráter heterogêneo. Na Figura 27 podemos visualizar a saída da implementação das tuplas.

```
#Inicialização de tuplas
tupla_1 = (1, 2, 3)
tupla_2 = 'a', 'b', 'c'
tupla_mista = (10, 'hello', 3.14)
tupla_vazia = ()

#Implementação e execução das tuplas
print("Tupla1: ", tupla_1)
print("Tupla2: ", tupla_2)
print("TuplaMista: ", tupla_mista)
print("TuplaVazia: ", tupla_vazia)
```

**Figura 27:** Exemplo de estrutura de implementação das tuplas.

```
Tupla1: (1, 2, 3)
Tupla2: ('a', 'b', 'c')
TuplaMista: (10, 'hello', 3.14)
TuplaVazia: ()
```

**Figura 28:** Saída de execução das implementações de tuplas.

Na Figura 29 podemos visualizar a criação de um conjunto com e sem a função **set()**. Note que a utilização da função **set()** impõe uma declaração: **set([ ])**, ou seja, os dados ou informações são inseridas dentro dos “[ ]”. A Figura 30 apresenta a saída em tela da consola dos conjuntos codificados.

```
#Inicialização de conjuntos
conjunto_1 = {1, 2, 3}
conjunto_2 = set([3, 4, 5])
conjunto_vazio = set()

#Implementação e execução dos conjuntos
print("Conjunto 1: ", conjunto_1)
print("Conjunto 2: ", conjunto_2)
print("Conjunto vazio: ", conjunto_vazio)
```

**Figura 29:** Exemplo de codificação de conjuntos em Python.

```
Conjunto 1: {1, 2, 3}
Conjunto 2: {3, 4, 5}
Conjunto vazio: set()
```

**Figura 30:** Saída da execução do programa de conjuntos.

Para os dicionários, na Figura 31 podemos visualizar a criação de um dicionário **pessoa** com os campos de “**nome**”, “**idade**” e “**cidade**”. A saída da implementação do dicionário pode ser visualizada na Figura 32.

```
#Inicialização do dicionário
pessoa = {
    'Nome': 'Eduardo',
    'Idade': 34,
    'Cidade': 'São Paulo'
}

#Implementação e execução do dicionário
print("Idade da Pessoa: ", pessoa['Idade'])
```

**Figura 31:** Exemplo de execução de um dicionário.

```
Idade da Pessoa: 34
```

**Figura 32:** Saída da execução do programa de implementação de dicionários.

## 4. LOOPS E ITERAÇÕES EM PYTHON

Nesta seção apresentaremos os blocos de repetição, *loops* e suas iterações. No decorrer do texto, exemplos de aplicação e sua aplicação em código Python bem como a saída de cada exemplos serão apresentados.

### 4.1 For, While e Controle de Fluxo

Os *loopings*<sup>1</sup> do tipo “**For**” podem ser utilizados para a iteração de informações e principalmente sobre uma sequência criada, como por exemplo, em uma lista, tupla e etc. Ao se implementá-lo sobre um bloco de código, o *looping* será executado até que sejam finalizadas as iterações baseadas nos itens da estrutura selecionada.

No caso dos *loopings* do tipo “**While**”, assim como a leitura de sua tradução literal “Enquanto”, o *looping* será executado até que a condição seja finalizada.

<sup>1</sup> Loopings ou Laços, são recursos para a execução de determinadas ações até que uma condição seja satisfatória.

## 4.2 Exemplos de aplicação

Na Figura 33 podemos visualizar o *loop for*, o qual possibilita através de um *looping*, ler e imprimir todas as informações da lista chamada “nomes”. Para melhor entender, podemos traduzi-lo de forma livre como “Para”, então, no exemplo a seguir, lemos: Para cada “**nome**” na lista “nomes” imprima os nomes que estão na lista . A Figura 34 apresenta a saída, linha a linha com as informações contidas na lista, via *loop for*.

```
#Inicialização da lista
nomes = ['Eduardo', 'Sei', 'Luis']

#Execução do loop for
for nome in nomes:
    print(nome)
```

Figura 33: Exemplo de implementação do *loop de execução for* em Python.



```
Eduardo
Sei
Luis
```

Figura 34: Saída do programa desenvolvida baseada no programa com *loop for*.

Na Figura 35, podemos visualizar a utilização do *loop for* com a inserção de uma função que gera um intervalo de números a serem armazenados e escritos na tela. A Figura 36 apresenta a saída da leitura do intervalo que a função **range()** proporciona.

```
#Execução do loop for
for numero in range(1, 6):
    print(numero)
```

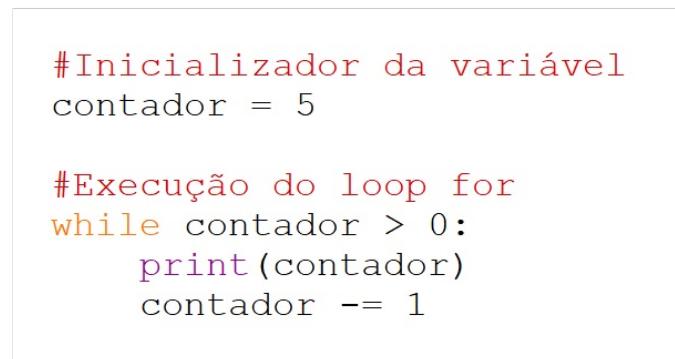
Figura 35: Exemplo de execução do *loop for* e utilização do **range()**.



1  
2  
3  
4  
5

Figura 36: Exemplo de saída do *loop for* com a utilização do `range()`.

Para o *loop while*, podemos visualizar que este bloco utiliza a variável “**contador**” e ela é inicializada em **5**. O **while** pode ser lido como enquanto, então, enquanto o contador for maior que **0**, executamos o bloco com a impressão em `print()` e o decremento. O exemplo pode ser visualizado na Figura 37 e a saída é apresentada na Figura 38.



```
#Inicializador da variável
contador = 5

#Execução do loop for
while contador > 0:
    print(contador)
    contador -= 1
```

Figura 37: Exemplo de execução do *loop while*.



5  
4  
3  
2  
1

Figura 38: Saída da execução do *loop while*.

É importante ressaltar que para percorrer itens através do comando `range()`, podemos designar um intervalo/faixa de índices para ou mesmo imprimir as suas informações. Outras estruturas nas quais podemos executar procedimentos semelhantes são as *strings*, tuplas e os dicionários, sendo que para os casos em que necessitarmos saber o índice e as informações incutidas numa sequência, podemos utilizar a função `enumerate()`.

## 5. FUNÇÕES EM PYTHON

As funções em Python, assim como em outras linguagens de programação são importantes para criarmos blocos que podem ser chamados várias vezes durante a execução dos programas. Nesta seção além de trabalharmos com os conceitos de função, iremos aprofundar também na utilização de variáveis locais e globais, as quais são de suma importância para a otimização dos programas que desenvolveremos.

### 5.1 Definição e Argumentos

Se tratam de blocos de código que podem ser reutilizados para cada ação da qual são desenvolvidos. Pense em uma estrutura r grande para ser consumida, agora, através da utilização de funções, imagine que podemos decompor este problema em etapas mei para facilitar a sua execução.

Em Python, utilizamos uma palavra-chave chamada “**def**” e logo após nomeamos a função e então passamos os seus parâmetr entre parênteses. A Figura 39, ilustras a estrutura básica de estruturação de uma função.

```
def nome_da_funcao(argumento1, argumento2, ...):
    # Corpo da função
    # Código a ser executado
    return valor_de_retorno
```

**Figura 39:** Exemplo de estrutura básica de uma função.

Para explicar a estrutura apresentada na Figura 39, podemos definir:

- **def:** A palavra-chave usada para definir uma função.
- **nome\_da\_funcao:** O nome da função, que deve ser único e seguir as convenções de nomenclatura de variáveis (geralmente é letras maiúsculas e/ou minúsculas, com palavras no infinitivo impessoal separadas por underscores ou *underlines* “\_”, por exemplo: **calcular\_soma**).
- **argumento1, argumento2, ...:** Os argumentos (parâmetros) que a função aceita. Os argumentos são opcionais, e uma função não ter nenhum ou vários argumentos.
- Os dois-pontos “:” marcam o início do corpo da função.
- Corpo da função: O código a ser executado quando a função é chamada.
- **return valor\_de\_retorno:** A instrução **return** (opcional) é usada para retornar um valor da função. Se não houver um **return**, a função retornará **None** (falta de valor) automaticamente.

## 5.2 Escopo e Retorno

Em Python, o escopo se refere à visibilidade e à acessibilidade das variáveis trabalhadas em nosso desenvolvimento. O conceito apresenta onde a variável será executada ou definida e por quem será acessada.

### No conceito de escopo, possuímos ao menos dois tipos

- **Escopo local:** As variáveis definidas dentro de uma função podem ser definidas como locais. Neste contexto, podemos relembrar que possuímos várias classificações de tipos de dados e estes tipos impactam diretamente na capacidade de elementos externos à função e/ou do próprio programa.
  - Quando falamos de escopo local, as variáveis não são chamadas ou visualizadas de fora da função.
- **Escopo global:** Nesta classificação, as variáveis são implementadas fora de qualquer função e desta forma, podem ser acessadas por qualquer função ou instância do programa.

## 5.3 Exemplos de Aplicação

Na Figura 40 podemos visualizar a implementação da função **calcular\_soma()**. Basicamente a função utiliza as variáveis “**a**” e “**b**” para que o usuário digite dois valores e os mesmos sejam calculados na expressão **resultado = a + b**. Ao final, o **resultado** é apresentado ao se chamar a função.

```
def calcular_soma(a, b):
    resultado = a + b
    return resultado
```

**Figura 40:** Exemplo de estrutura básica de uma função.

As variáveis local e global podem ser visualizadas nas Figuras 41 e 42. Em sua estrutura de implementação, podemos diferenciar por uma estar dentro de uma função, então devemos chamar a função para poder alterar o seu valor e a outra estar fora de qualquer função. Neste último caso, qualquer função poderá utilizar a informação.

```
def minha_funcao():
    variavel_local = 10
    print(variavel_local)

minha_funcao()
print(variavel_local)
```

**Figura 41:** Exemplo de execução da função de variáveis locais.

```
variavel_global = 20

def minha_funcao():
    print(variavel_global)

minha_funcao()
print(variavel_global)
```

**Figura 42:** Exemplo da execução da função global.

## 6. MÓDULOS DE PACOTES EM PYTHON

Os módulos de pacotes são uma estrutura fundamental para a organização dos códigos e o desenvolvimento de programas em *Python*. Os módulos podem ser definidos como arquivos de programa que possuem variáveis, funções e/ou classes. Os pacotes podem ser definidos como um agrupamento de módulos ou de forma simplificada, podemos colocar os arquivos programados em um diretório/pasta e este(a) contemplará nosso pacote.

### 6.1 Importação, namespaces e instalação de Pacotes Externos

A importação de bibliotecas, a utilização de **namespaces** e a utilização de pacotes com códigos fontes externos é essencial na construção de aplicações com organização e respeito à estrutura de execução de programas *Python*.

### 6.2 Importação em Python

A importação é um recurso que permite trazermos arquivos de código anteriormente implementados para o código atual. Em termos de organização e projeto, é uma prática essencial para evitarmos redeclararmos recursos e assim duplicar partes do código ou chamar informações anteriormente desenvolvidas.

Podemos realizar a importação de diversas formas, como por exemplo:

- **Importação de um arquivo ou módulo inteiro:** Na Figura 43, realizamos a importação de um arquivo ou módulo em *Python* completo para ser utilizado em nosso programa.

- No exemplo, a palavra “**import**” significa que vamos importar algo, neste caso o módulo “**math**”. Na mesma linha possuímos a palavra “**as**” que possibilita atribuirmos um apelido ou novo nome para chamar os recursos do arquivo ou módulo em questão.

```
import math as matematica
```

**Figura 43:** Exemplo de importação do módulo “math”.

```
import math as matematica
x = matematica.sqrt(4)
print(x)
```

**Figura 44:** Exemplo de implementação da função **sqrt()** proveniente do módulo “math”.

2.0

**Figura 45:** Saída da implementação da função **sqrt()** aplicada ao número 4.

- Nas Figuras 44 e 45, é possível visualizarmos a utilização do módulo para carregarmos a função **sqrt()** a qual possibilita aplicar cálculo de raiz quadrada a um número e posteriormente a sua impressão (saída).
- No exemplo, para aplicar o **sqrt()**, precisamos chamar o módulo **math** agora apelidado de **matematica**, como se segue: **matematica.sqrt()**.
- Importação de partes específicas:** Anteriormente definimos o carregamento de um módulo de funções, porém, sem a definição qual recurso especificamente precisamos. É possível definirmos qual dos recursos vamos utilizar e evitar carregarmos todo o módulo para a implementação. Na Figura 46 visualizamos a forma de carregar apenas a função desejada e/ou necessária, no caso **sqrt()** sem carregarmos todo o módulo **math**.

```
from math import sqrt
```

**Figura 46:** Exemplo de importação apenas a função **sqrt()** pertencente ao módulo **math**.

```
from math import sqrt
raiz = sqrt(4)

print(raiz)
```

**Figura 47:** Exemplo de implementação de cálculo de raiz quadrada com a função **sqrt()**.

2.0

**Figura 48:** Saída da implementação da função `sqrt()`.

- Nas Figuras 47 e 48, visualizamos a implementação do conceito trabalhado. A palavra **from** indica que desejamos carregar algo específico de um módulo, como por exemplo, **from math import sqrt** indica que, a partir do **math**, carregamos apenas o **sqrt()**.
- Como visualizamos, não precisamos mais colocar o nome do módulo, apenas a função que já determinamos em sua declaração.
- **Importação de todos os elementos sem a atribuição de um nome:** É uma terceira opção na importação de módulos, porém importante ressaltar que podemos incorrer em conflito de nomes durante a declaração. Exemplo, se carregarmos a função **exp** módulo **math**, podemos gerar conflito, uma vez que, **exp** não é uma palavra reservada na linguagem de programação **Python** e podemos ter utilizado a mesma palavra anteriormente.
- Nas Figuras 49, 50 e 51, podemos visualizar um exemplo de carregamento do módulo sem a atribuição de um nome. No exemplo utilizamos o asterisco “**\***” para indicar que queremos carregar todas as funções e classes presentes em **math**.

```
from math import *
```

**Figura 49:** Exemplo de carregamento da biblioteca **math** e importação de todas as suas funções.

```
from math import *
raiz = sqrt(4)

print(raiz)
```

**Figura 50:** Exemplo de implementação de código para o cálculo de raiz quadrada.

2.0

**Figura 51:** Saída do Código de cálculo da raiz quadrada de uma raiz quadrada.

Como podemos visualizar, as estruturas apresentadas possibilitam a organização do código e a separação de funções em arquivos de acordo com as suas funções. Desta forma, organizamos nosso código e evitamos duplicação de chamadas, uma vez que, agora podemos chamar módulos externos e implementar suas funções em nosso código atual.

### 6.3 Namespaces

Anteriormente citamos problemas com nomes e a possibilidade de conflitos. A utilização de **namespaces** nos ajuda a evitar o problema com nomes, uma vez que, os **namespaces** associados às funções implicam em nomes únicos a cada função ou chamada de código.

Para a sua aplicação, podemos separar os **namespaces** em vários tipos:

- **NameSpace Local:** Os nomes atribuídos às funções podem ser entendidos como o seu **namespace** local. No exemplo da Figura 52 é apresentado a criação da função denominada “**minha\_funcao()**”.

```

def minha_funcao():
    # Variável local dentro da função "minha_funcao"
    var_local = "Isso é uma variável local"

    # Impressão da mensagem da "var_local"
    print(var_local)

    # Chamando a função criada
    minha_funcao()

    # Gerará erro, uma vez que esta fora da função
    print(var_local)

```

**Figura 52:** Implementação das funções com o uso de um **namespace** local.

```

Isso é uma variável local
Traceback (most recent call last):
  File "C:\Users\eduar\Desktop\FabricaDeConteudos\Livro 2\Para a entrega\FabricaDeConteúdos\Codigos\Mod04.py", line 12, in <module>
    print(var_local)
NameError: name 'var_local' is not defined

```

**Figura 53:** Exemplo de impressão de mensagem e evidência de problemas gerados por função executada fora do escopo da função local.

- Na Figura 53, apresentamos a saída de execução do código. Note que inicialmente a mensagem é apresentada, porém, a seguir gera um erro por conta chamada do `print()` fora da função.
- Namespace de Módulo:** Cada arquivo programado em Python possui seu **namespace** e, portanto, podemos chama-lo externamente. Através da determinação do nome do arquivo, possuímos o **namespace** do módulo. Este recurso possibilita chamarmos as funções e atributos criados em um arquivo criado anteriormente.
- Um exemplo é a criação de um arquivo Python onde criamos a variável e a definição de uma função para ser chamada posteriormente. Na Figura 54 podemos visualizar a implementação dos recursos necessários.

```

#Namespace modulo_inicial.py

#Variável no namespace de módulo
variavel_de_modulo = "Isso é uma variável de módulo"

#Função no namespace de módulo
def minha_funcao_de_modulo():
    return "Isso é uma função de módulo"

```

**Figura 54:** Exemplo de criação de **namespace** de módulo.

- Em um segundo arquivo, carregaremos o módulo através do nome do arquivo que será chamado por exemplo de: “**modulo\_inicial.py**”.
- Nas Figuras 55 e 56, podemos visualizar a chamada do módulo criado anteriormente em um novo arquivo. O objetivo é explicitar a utilização dos **namespaces** na chamada de módulos e funções.

```

#main.py

# Importar o módulo
import modulo_inicial

# Acesso de variável e função do módulo
print(modulo_inicial.variavel_de_modulo)
print(modulo_inicial.minha_funcao_de_modulo())

```

**Figura 55:** Importação do módulo criado anteriormente e a impressão de informações provenientes do módulo carregado.

```
Isso é uma variável de módulo
Isso é uma função de módulo
```

**Figura 56:** Exemplo de saída proveniente de chamada a um módulo.

- Nos exemplos apresentados, podemos visualizar o **import** para carregar o **modulo\_inicial** e a partir disso, realizamos no **print** função **modulo\_inicial.variavel\_de\_modulo**.
- Inicialmente utilizamos o nome do arquivo importado e após o “.”, possuímos o nome da variável carregada ou função.
- **Namespace Embutido (ou Built-In):** Se refere ao conjunto de nomes que são automaticamente disponíveis em qualquer programa Python. O diferencial deste tipo de **namespace** é a característica de não precisar importar nomes para implementar suas funções.
- **Exemplo:** Na sua implementação, podemos implementar funções do tipo **built-in** para aplicar funções preexistentes nos módulos do Python.
- Na Figura 57, podemos visualizar a criação de funções do tipo **built-in**.

```
#Usando funções built-in
numero = 42
print("Número binário: ", bin(numero))
print("Número absoluto", abs(-5))

#Usando objetos built-in
lista = [1, 2, 3, 4, 5]
print("Comprimento da lista", len(lista))
print("Tipo da variável 'lista': ", type(lista))
```

**Figura 57:** Criação de funções do tipo **built-in**.

- Na Figura 58, podemos visualizar a saída do exemplo de funções e objetos **built-in**. Neste contexto, não declaramos nenhum nome anteriormente.

```
Número binário: 0b101010
Número absoluto 5
Comprimento da lista 5
Tipo da variável 'lista': <class 'list'>
```

**Figura 58:** Saída da utilização da função **built-in**.

A principal função dos **namespaces** é a garantia da integridade dos nomes utilizados na criação do código, ou seja, evitarmos a duplicação das declarações de variáveis. Ao final, o recurso possibilita a otimização e organização de códigos.

## 6.4 Instalação de Pacotes Externos

Em programação Python, podemos instalar os pacotes externos e assim utilizar as variáveis, funções e objetos importadas. Este pacotes são desenvolvidos por terceiros e nos possibilitam carregar e executar funções nas mais variadas aplicações, como por exemplo:

- O pacote denominado **numpy** nos permite o acesso a funções de manipulação de dados e cálculos matemáticos.
- O **matplotlib** e **seaborn** são utilizados para a visualização de dados e plotagem de gráficos.
- O **requests** nos ajuda a fazer solicitações HTTP.

- O **django** e **flask** são utilizados para o desenvolvimento *Web*.

Para a instalação dos pacotes externos, podemos utilizar gerenciadores como o **pip**. Neste caso, ao acessarmos o prompt de comando, podemos digitar o comando, por exemplo, para instalar o pacote **numpy**. Na Figura 59, podemos visualizar um exemplo c instalação do pacote para posteriormente podemos utilizar as suas funções.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.3570]
(c) Microsoft Corporation. All rights reserved.

C:\Users\eduar>pip install numpy
Requirement already satisfied: numpy in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (1.24.4)

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: c:\users\eduar\appdata\local\programs\python\python38\python.exe -m pip install --upgrade pip

C:\Users\eduar>
```

**Figura 59:** Saída de tela da instalação do pacote **numpy** via gerenciador **pip**.

- Como pudemos visualizar, a instalação do pacote necessita apenas da palavra **pip** e o comando **install**. Posteriormente, colocar o nome do pacote que desejamos instalar.
- É importante ressaltar que podemos desinstalar o pacote instalado, basta ao invés de utilizar o **install**, trocamos por **uninstall**. quando encontrarmos um pacote desinstalado, podemos utilizar o comando **pip install --upgrade pip**

#### São comandos importantes também, os apresentados a seguir:

- Instalar uma versão específica de um pacote:
  - **pip install nome\_do\_pacote==versao\_desejada**
- Atualizar um pacote:
  - **pip install --upgrade nome\_do\_pacote**
- Listar pacotes instalados:
  - **pip list**
- Criar um **arquivo** de requisitos:
  - **pip freeze > requirements.txt**
- Instalar **pacotes** a partir de um arquivo de requisitos:
  - **pip install -r requirements.txt**

#### Importante

Em alguns sistemas, podemos utilizar o **pip3** ou mesmo o **pip** para execuções em *Python 3*. O gerenciamento de versões e a instalação de pacotes deve ser realizada com cuidado, uma vez que, podemos gerar conflitos na instalação do ambiente de desenvolvimento.

## 6.5 Exemplos de Aplicação

Podemos realizar a importação de bibliotecas externas para a resolução de problemas específicos, como por exemplo, para tratar solicitações HTTP e assim, com o retorno visualizarmos informações da página *Web*. Para a resolução, primeiramente precisamos importar a biblioteca **requests**.

Na Figura 60, é possível visualizarmos a instalação do pacote **requests** para o gerenciamento de dados e o tratamento de requisições HTTP. Podemos evidenciar um exemplo de implementação na Figura 61 e a visualização da saída na Figura 62.

```
C:\Users\eduar>pip install requests
Collecting requests
  Installing collected packages: urllib3, idna, charset-normalizer, certifi, requests
    Successfully installed certifi-2023.7.22 charset-normalizer-3.3.0
      idna-3.4 requests-2.31.0 urllib3-2.0.6
C:\Users\eduar>
```

**Figura 60:** Exemplo de instalação do pacote “**requests**”.

```
import requests

def main():
    url = 'https://www.python.org'
    resposta = requests.get(url)
    print(resposta)

    if resposta.status_code == 200:
        print(resposta.text)

if __name__ == '__main__':
    main()
```

**Figura 61:** Exemplo de código com a utilização do pacote “**requests**”.

<Response [200]>

Squeezed text (1371 lines).

**Figura 62:** Saída do código de acesso à página <https://www.python.org>.

A saída indica a resposta HTTP que é “**Response 200**” e a frase “**Squeezed text (1375 lines)**” que indica que existem 1375 linhas ocultadas referentes à solicitação HTTP, uma vez que a mesma se trata da visualização do código HTML do website solicitado.

- No código, importamos o pacote **requests**.
- Definimos a função “**main()**”
- Na variável **url**, passamos o endereço que devemos acessar. Neste caso, o website oficial da comunidade Python: “<https://www.python.org>”
- Quando declaramos uma função com o “**\_\_**”, indicamos que ela é especial e que esta estrutura cria uma chamada local ou interna, por exemplo, quando utilizamos o “**\_\_init\_\_**”, o que estiver dentro da função quando o programa for inicializado, será carregado primeiro.
- No exemplo, possuímos o “**\_\_name\_\_ == '\_\_main\_\_'**”, para estes elementos, indicamos que se possuirmos a função “**main**”, o mesmo será executado.

## 7. NOÇÕES DE TRATAMENTO DE EXCEÇÕES EM PYTHON

Na construção de programas em Python, modelamos os programas através de variáveis, funções, blocos de repetição e blocos condicionais. Entretanto, possuímos outras situações que fogem a modelagem comum de opções de saída do programa, como por exemplo, ao criarmos um campo chamado de “**nome\_da\_pessoa**”, esperamos apenas caracteres presentes em nosso alfabeto, contudo, à atribuição de um nome, porém, é possível que o usuário insira números e assim, o cadastro do nome não funcione corretamente.

Para o tratamento destas situações onde teríamos apenas uma mensagem de erro, podemos utilizar o **tratamento de exceções**. Então, a utilização do tratamento de erros nos possibilita tratar erros de atribuição de informações às variáveis, que em uma declaração comum não poderíamos fazer, ou seja, situações que fogem ao funcionamento padrão de execução dos programas.

## 7.1 Try, Except, Finally e Raise

A estrutura de tratamento de exceções pode ser melhor explicada com as informações a seguir:

- **Bloco try-except:** O bloco `try` é utilizado para apontarmos o trecho de código onde podemos gerar uma exceção durante a sua execução. Neste contexto, podemos exemplificar a situação onde possuímos uma variável do tipo texto, porém, com um valor numérico e uma variável numérica com valor inteiro. Esta situação poderá ocorrer quando estivermos manipulando valores entre telas de uma aplicação *Web* ou *mobile*. Esta parte do código *Python* deverá ser implementada dentro do bloco `try`.

- O bloco `except` permitirá que tratemos a entrada com dados adversos e criemos uma saída para a exceção. Nas Figuras 64 podermos visualizar a construção de um exemplo onde criamos a situação de `try-except`.
- No exemplo utilizamos a formatação de uma impressão, onde através dos símbolos “{ }” pudemos apresentar o conteúdo variável trabalhada, neste caso, “e:” do `TypeError()`.

```
try:
    numero_texto = "123"
    numero = 456
    soma = numero_texto + numero
except TypeError as e:
    print(f"Erro: {e}")
```

**Figura 63:** Implementação inicial do bloco `try..except`.

Erro: can only concatenate str  
(not "int") to str

**Figura 64:** Exemplo de erro capturado via tratamento de exceção.

- **Variação do Except (except composto):** Se trata da inserção de vários tratamentos de exceção, neste caso, o trecho de código analisado possui várias possibilidades de exceção. As Figuras 65 e 66 apresentam tal implementação com diferentes possibilidades de tratamento de exceção e a sua saída em tela.

```
try:
    numero_inicial = 5
    resultado = 10 / numero_inicial
except ValueError as value_error:
    print(f"Erro de Valor: {value_error}")
except ZeroDivisionError as zero_division_error:
    print(f"Erro de Divisão por Zero: {zero_division_error}")
except Exception as exception:
    print(f"Erro genérico: {exception}")
else:
    print(f"Resultado: {resultado}")
```

**Figura 65:** Exemplo de implementação de tratamento de exceções compostas.

Resultado: 2.0

**Figura 66:** Saída da implementação de um tratamento de exceções composto.

- Como podemos observar no exemplo anterior, possuímos na variável **numero\_inicial** a atribuição do valor “5”.
- Caso fosse inserido um tipo texto na realização do cálculo, geraríamos uma exceção do tipo “**ValueError**”.
- Se o valor numérico fosse **zero**, a saída seria o erro de divisão por zero ou “**ZeroDivisionError**”.
- Possuímos também o tratamento genérico para exceções ou **exception**, o qual utilizamos quando não possuímos um tipo específico para tratá-lo, como por exemplo, poderíamos capturar um erro qualquer e mandar limpar ou apagar o conteúdo variável.
- **Finally:** O bloco **finally** é inserido opcionalmente ao final dos blocos **try-except**, porém, ao contrário dos anteriores, este é executado independentemente de condições, ou seja, ao finalizarmos o tratamento de exceções, executamos este bloco.
  - O **finally** é essencial quando desejamos finalizar um fluxo com a limpeza do campo de preenchimento ou mesmo fechar um arquivo de texto aberto.
- **Raise:** A função **raise()** nos permite executar uma exceção em estado de execução. Este recurso é necessário quando possuímos comportamento adverso mapeado e queremos lançar uma exceção para resolver tal problema sem a necessidade de parar o funcionamento normal do programa.

### Importante

Possuímos várias exceções do tipo **built-in**, ou seja, que já são padronizadas e estão disponíveis para serem utilizadas na instalação padrão de qualquer ferramenta de programação Python. São exemplos de exceções que podemos utilizar:

**SyntaxError**, **TypeError**, **ValueError** e o **FileNotFoundException**.

## 7.2 Exemplo de Aplicação

No exemplo de implementação a seguir, o exercício é focado na situação onde uma variável “**nome**” que aloca valores textuais, possui saída verdadeira ou correta se passarmos valores válidos conforme o exemplo da Figura 67, o que pode ser confirmado na saída ilustrada na Figura 68.

Já as Figuras 69 e 70 apresentam a situação prática onde inserimos valores numéricos em uma variável “**nome**” que alocaria TypeError. Em sua saída, o erro será gerado uma vez que neste caso, passamos um erro classificado como **TypeError()** ou de tipo incorreto.

```
try:
    nome = "Eduardo Sei"
    for caractere in nome:
        if '0' <= caractere <= '9':
            raise ValueError("O nome não pode conter números.")
    print(f"Nome digitado: {nome}")
except ValueError as e:
    print(f"Erro: {e}")
finally:
    print("Exemplo completo de utilização \
do tratamento de exceções")
```

**Figura 67:** Exemplo de implementação de situação-problema onde analisamos o tratamento de uma variável para a retirada de números.

Nome digitado: Eduardo Sei  
Exemplo completo de utilização do tratamento de exceções

**Figura 68:** Saída de execução do tratamento da variável e retirada de números.

```

try:
    nome = "123"
    for caractere in nome:
        if '0' <= caractere <= '9':
            raise ValueError("O nome não pode conter números.")
    print(f"Nome digitado: {nome}")
except ValueError as e:
    print(f"Erro: {e}")
finally:
    print("Exemplo completo de utilização \
do tratamento de exceções")

```

**Figura 69:** Inserção de números e a verificação do tratamento de exceções.

Erro: O nome não pode conter números.  
Exemplo completo de utilização do tra  
tamento de exceções

**Figura 70:** Exemplo de saída do programa onde inserimos números em variável textual.

## 8. ARQUIVOS DE ENTRADA E SAÍDA EM PYTHON

O gerenciamento de operações de entrada e saída é necessário para a execução várias importantes situações de programação.

A entrada de dados (leitura de informações) pode ser entendida como a execução de operações que podem ser realizadas através de comandos do teclado, movimentação do mouse ou mesmo a partir da entrada de informações de outro programa ou arquivo. Com relação à saída, podemos realizá-la através da gravação (escrita) das informações trabalhadas em um arquivo externo que poderá ser uma planilha .CSV, um banco dados ou apenas numa impressão em tela.

Uma interação de inserção de dados através do teclado pode ser visualizada na Figura 71. No exemplo, podemos inserir um valor dentro da variável “primeiro\_nome”. E a sua saída pode ser visualizada na Figura 72.

```

primeiro_nome = input("Digite o primeiro nome: ")
print(f"O meu primeiro nome é: {primeiro_nome}")

```

**Figura 71:** Exemplo de inserção de informações à variável de forma interativa.

Digite o primeiro nome:  
Eduardo  
O meu primeiro nome é:  
Eduardo

**Figura 72:** Exemplo de interação de entrada de dados e saída ou impressão das informações.

No exemplo, é possível visualizarmos que:

- Através da função `input()`, podemos realizar a leitura de informações via teclado.
- A saída será gerada de forma formatada. Sendo que com as chaves “{ }”, o conteúdo da variável será inserido em “`primeiro_nome`”.
- É importante apresentarmos as variações de declaração da função `input()`. No exemplo anterior, o declaramos sem atribuir um específico. Já nas figuras 73, 74 e 75, podemos visualizar um exemplo de definição da leitura de informações do tipo inteiro, un

saída verdadeira ou esperada e ao final, a tentativa de passagem de um texto em uma variável do tipo numérica (**int**).

```
numero_inteiro = int(input("Digite o primeiro número: "))
print(f'O primeiro número digitado é: {numero_inteiro}')
```

**Figura 73:** Exemplo de inserção de informações de forma interativa e com a formatação para receber apenas números inteiros.

```
Digite o primeiro número:
123456
O primeiro número digitad
o é: 123456
```

**Figura 74:** Saída da interação de dados numéricos e sua impressão.

```
Digite o primeiro número: Eduardo
Traceback (most recent call last):
  File "C:\Users\eduar\Desktop\FabricaDeConteudos\Livro 2\Para a entrega\FabricaDeConteudos\Codigos\primeiro_numero.py", line 1, in <module>
    numero_inteiro = int(input("Digite o pri
meiro número: "))
ValueError: invalid literal for int() with b
ase 10: 'Eduardo'
```

**Figura 75:** Exemplo de tentativa de inserir informações textuais.

- Como podemos visualizar, colocamos a função **input()** dentro de uma operação de conversão explícita. Este tipo de operação é em colocarmos a função dentro do tipo de dados inteiro (**int()**).
- Em *Python*, como não precisamos declarar a variável com o seu tipo, podemos reutilizá-la de forma dinâmica e assim, a variável “**primeiro\_nome**” que inicialmente recebeu a informação “**Eduardo**”, na linha seguinte pôde receber o valor “**123456**”. Esta ação pode ser chamada de conversão implícita.

## 8.1 Leitura e Escrita em Arquivos

Como abordado anteriormente, podemos realizar a leitura e escrita de informações ou dados em arquivos externos, como por exemplo, arquivos de texto. A manipulação de arquivos é realizada com as funções **open()** e **close()**. Estas funções são essenciais para a abertura e fechamento de arquivos.

- A função **open()** implica na abertura de um arquivo ou mesmo na criação e abertura de um novo arquivo.
  - A definição da função **open()** possui dois argumentos em sua criação. O primeiro é o nome do arquivo de texto que deverá estar no diretório do código *Python* e em seu segundo, o modo de execução com que o arquivo será tratado.
  - Como por exemplo, executamos o comando: “**open(“arquivo\_de\_exemplo.txt”, “w”)**”.
- A função **close()** realiza o encerramento (finalização) do processo de abertura, escrita e/ou leitura. O encerramento implica que alterações podem ser finalizadas e armazenadas ou salvas na memória (em arquivo).

### São modos de manipulação de um arquivo de texto:

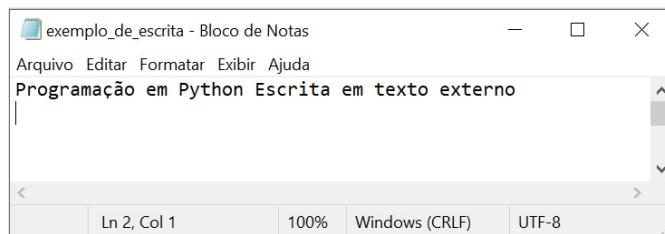
- Modo “w”:** Pode ser entendido como **write** ou comando de escrita. Este modo abre um arquivo existente e apaga todas as informações anteriores. Se o arquivo não existir, o mesmo será criado.
- Modo “a”:** De **append**, este modo abrirá um arquivo para escrita (já existente), porém, não apagará o seu conteúdo. Sendo as inserções adicionadas ao final do arquivo.

- **Modo “r”:** Modo de leitura ou **read**. O arquivo será aberto para a leitura de seu conteúdo. No caso de não possuirmos o arquivo será gerado.
- **Modo “x”:** Cria e abre um arquivo para escrita. Se o arquivo já existir, um erro será gerado. Este modo é utilizado quando deve evitar possíveis duplicações.
- Se utilizarmos o **“b”** em algum modo anterior indicamos que estamos trabalhando com arquivos binários (em linguagem de máquina). Podemos utilizar para leitura **“rb”** ou mesmo **“wb”**. O **“t”** possibilita trabalharmos com informações textuais. O **“rt”** (leitura das informações de texto e o **“wt”** realiza a escrita em arquivos externos.
- O **“a+”** cria um arquivo e anexa informações ao final do mesmo. Por fim, o **“w+”** possibilita a leitura do arquivo e escrita de informações no mesmo.

A seguir podemos visualizar exemplos de escrita e leitura em arquivos externos. As Figuras 76, 77, 78 e 79 apresentam exemplo aplicação dos modos de leitura e escrita em um arquivo de texto.

```
with open('exemplo_de_escrita.txt', 'w') as arquivo_escrita:
    #Escreve informações em arquivo de texto externo
    arquivo_escrita.write("Programação em Python")
    arquivo_escrita.write(" Escrita em texto externo")
```

**Figura 76:** Exemplo de abertura e criação de arquivo.



**Figura 77:** Exemplo de saída com gravação de informações em texto no arquivo.

```
with open('exemplo_de_escrita.txt', 'r') as arquivo_leitura:
    #Leitura de informações em arquivo de texto externo
    mensagem = arquivo_leitura.read()
    print(mensagem)
```

**Figura 78:** Implementação de código para leitura de arquivo de texto.

Programação em Python Escrita  
em texto externo

**Figura 79:** Saída da leitura de informações que já estavam inseridas no arquivo de texto.

- Nas Figuras 76 e 78 possuímos o comando **with**, o qual chama a função **open()**, pela qual passamos os parâmetros de manipulação dos arquivos (Nome e Modo) (**'exemplo\_de\_leitura.txt', 'w'**) e (**'exemplo\_de\_escrita.txt', 'r'**).
- O **“as”** possibilita renomearmos a abertura dos arquivos como **arquivo\_escrita** e **arquivo\_leitura**.
- A função **.write()** realiza a escrita das informações.
- A função **.read()** realiza a leitura do arquivo de texto trabalhado.

## 8.2 Manipulação de Diretórios/Pastas

Em programação, podemos acessar diretórios/pastas e trabalhar as informações que ali estão armazenadas. Neste contexto, de lembrar do conteúdo de módulos e pacotes em programação *Python*.

São operações possíveis para aplicarmos ao gerenciamento de diretórios/pasta, a sua criação, remoção, alteração e exclusão. S bibliotecas necessárias para trabalharmos com diretórios, a “**os**” de Sistema Operacional (OS - *Operational System*) e “**shutil**” de Utilities.

Nas Figuras 80 e 81, podemos visualizar um exemplo de operações de diretórios com a biblioteca “**os**” e a saída de execução da função **mkdir()**.

```
import os

#Criação de diretório
os.mkdir("programacao_python")

#Verificar se o diretório existe
if os.path.exists("programacao_python"):
    print("Diretório existente")
else:
    print("O diretório não existe")

#Listar as informações de um diretório
arquivos_diretorios = os.listdir("programacao_python")
for arquivo_diretorio in arquivos_diretorios:
    print(arquivo_diretorio)

#Remover o diretório
os.rmdir("programacao_python")
```

**Figura 80:** Saída da leitura de informações que já estavam inseridas no arquivo de texto.

FabricaDeConteudos > Livro 2 > Para a entrega > FabricaDeConteudos > Códigos		
Name	Date modified	Type
__pycache__	10/22/2023 2:20 AM	File folder
python_programacao	10/11/2023 2:34 AM	File folder
003	10/22/2023 1:00 AM	Python File
04	10/22/2023 1:00 AM	Python File
07	10/22/2023 1:11 AM	Python File

**Figura 81:** Pasta **programacao\_python** criada através do código implementado.

- A função **mkdir()** trata da criação de diretórios/pastas. O nome do diretório é passado dentro dos parênteses “( )”.
- A função **path.exists()** verifica se o diretório já existe.
- A função **listdir()** realiza a listagem de arquivos dentro do diretório verificado.
- A função **rmdir()** realiza a remoção do diretório passado dentro “( )”.

A biblioteca **shutil()** possibilita elevarmos o nível de controle dos diretórios. São ações possíveis, a possibilidade de copiar, mover ou excluir diretórios. A biblioteca também possibilita manipular o conteúdo dos diretórios, porém, apenas com as operações citadas neste parágrafo. Nas Figuras 82 e 83, podemos visualizar a implementação e a saída do tratamento de diretórios com o módulo **shutil**.

```
import shutil

#Cópia de diretórios
shutil.copytree("programacao_python", "python_programacao")

#Movimentação de diretórios
shutil.move("programacao_python", "python_programacao")

#Exclusão de diretórios
shutil.rmtree("programacao_python")
```

**Figura 82:** Manipulação dos diretórios através do módulo **shutil**.

Name	Date modified	Type
__pycache__	10/22/2023 2:20 AM	File folder
python_programacao	10/11/2023 2:34 AM	File folder
003	10/22/2023 1:00 AM	Python File
04	10/22/2023 1:09 AM	Python File
07	10/22/2023 1:11 AM	Python File

Figura 83: Exemplo de alteração do nome da pasta através do módulo **shutil**.

Na Figura 82, são apresentadas as funções:

- A função **copytree()** que realiza a cópia de uma árvore de diretórios/pastas para outra.
- A função **move()** realiza a movimentação do conteúdo de uma pasta para a outra.
- A função **rmtree()** que realiza a remoção de um diretório e toda a sua árvore de diretórios/pastas.

### 8.3 Trabalhando com CSV e JSON

Até este ponto, trabalhamos com interações na própria saída de códigos do *Python* e em interações com arquivos de texto. Mas existem também outras possibilidades mais complexas para a manipulação de arquivos com formatação e organização.

São exemplos de soluções que nos oferecem melhores condições de manipulação de arquivos e nos fornecem entradas e saídas complexas para trocas de informações (dados) entre sistemas, as que ocorrem através de arquivos CSV (*Comma-Separated Values*) JSON (*JavaScript Object Notation*).

No caso dos dados CSV, podemos produzir dados organizados através da separação por vírgula ou ponto e vírgula. Em sua formatação, possuímos a separação dos dados por coluna e linha, ou seja, possuímos os rótulos das informações e seus dados armazenados,

Na Figura 84, o exemplo de implementação com biblioteca “**csv**” nos possibilita executar a leitura dos dados em um arquivo denominado “**arquivo\_dados.csv**” e modo de leitura “**r**”.

Como podemos observar, a biblioteca “**csv**” funciona de forma semelhante à biblioteca “**os**” anteriormente vista. A função **reader()** realiza a leitura das informações em arquivos de tabelas. Na Figura 85, podemos visualizar sua saída.

Os símbolos “\” ao longo do texto realizam a organização do texto no código.

```
import csv

# Abrir um arquivo CSV para leitura
with open("arquivo_dados.csv", "r", \newline = '') as arquivo_csv:
    leitor_csv = csv.reader(arquivo_csv)

    # Iterar pelas linhas do arquivo
    for linha in leitor_csv:
        print(f'Nome: {linha[0]}, \\\nIdade: {linha[1]}, \\\nCidade: {linha[2]}')
```

Figura 84: Implementação de manipulação de arquivo .CSV.

Nome: Andre, Idade: 18,  
Cidade: Itu

Figura 85: Saída da implementação e leitura de informações em arquivo .CSV.

```

import csv

# Dados a serem escritos no arquivo CSV
dados = [
    ['Eduardo', 34, 'São Paulo'],
    ['Sei', 34, 'Suzano'],
    ['Luis', 45, 'São Paulo']
]

# Abrir um arquivo CSV para escrita
with open("dados_saida.csv", "w", \newline = '') as arquivo_csv:

    escritor_csv = csv.writer(arquivo_csv)

    # Escrever os dados no arquivo CSV
    for linha in dados:
        escritor_csv.writerow(linha)

```

**Figura 86:** Implementação de escrita de informações em arquivo .CSV.

The screenshot shows a Microsoft Excel spreadsheet titled 'dados\_saida - Excel'. The data is organized into columns A, B, and C. Row 1 contains 'Eduardo', '34', and 'São Paulo'. Row 2 contains 'Sei', '34', and 'Suzano'. Row 3 contains 'Luis', '45', and 'São Paulo'. The formula bar at the top shows 'C10' and the formula 'dados\_saida'. The status bar at the bottom right indicates 'Pronto' and '100%'. The ribbon menu is visible at the top.

	A	B	C	D	E	F	G
1	Eduardo	34	São Paulo				
2	Sei	34	Suzano				
3	Luis	45	São Paulo				
4							

**Figura 87:** Evidência da saída das informações escritas no arquivo .CSV.

- Nas Figuras 86 e 87, podemos visualizar a criação de uma lista heterogênea denominada como “dados”.
- A função `.writer()` realiza a escrita de informações no arquivo `dados_saida.csv`.
- A função `.writerow()` realiza a escrita linha a linha.

O JSON é um formato de manipulação de dados versátil e que nos permite ler e escrever informações sobre a relação chave-valor. Na Figura 88 é possível destacarmos os processos de leitura e na Figura 89 apresentamos a escrita de dados do arquivo.

```

import json

agenda = {
    "Nome": "Eduardo",
    "Idade": 30,
    "Cidade": "São Paulo"
}

with open('JSON_dados.json', 'w') as escrita_JSON:
    json.dump(agenda, escrita_JSON)

print("Dados JSON escritos com sucesso")

```

**Figura 88:** Implementação de informações para a escrita em arquivo JSON.

Dados JSON escritos com sucesso

**Figura 89:** Saída da escrita de informações em arquivo JSON.

```
import json
with open('JSON_dados.json', 'r') as leitura_JSON:
    info = json.load(leitura_JSON)
print(info)
```

**Figura 90:** Leitura de informações em arquivo JSON.

```
{'Nome': 'Eduardo', 'Idade': 30, 'Cidade': 'São Paulo'}
```

**Figura 91:** Impressão das informações de saída do arquivo JSON.

As Figuras 88, 89, 90 e 91, apresentam a manipulação de informações que estão presentes no arquivo JSON nomeado: **JSON\_dados.json**. Nos exemplos podemos visualizar:

- O método **dump()** o qual possibilita inserirmos informações num arquivo JSON.
- O método **load()** que nos permite carregar informações de um arquivo JSON.

## 9. PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON

A Programação Orientada a Objetos (POO), é um paradigma fundamental para o desenvolvimento de programas modernos, uma que, propicia gerarmos uma solução ao problema de forma modular, com a possibilidade de reutilizar partes de código e a sua manutenção de forma facilitada.

No quesito **modular**, separamos o problema de forma a representar as funções principais de solução do problema, como por exemplo em um **software** para o “**Gerenciamento de um Supermercado**”, onde precisamos entre outros de:

- **Login**
- **Cadastro de usuários**
- **Cadastro de produtos**
- **Gerenciamento de recursos**
- **Arquivo de execução**

Os elementos citados representam cada arquivo ou módulo de um sistema. Para cada uma das estruturas, podemos desenvolver de forma separada e desta forma, podemos **reutilizá-las** em outros projetos/sistemas. Na modelagem do projeto, vemos o arquivo de execução e este item é por onde vamos referenciar todos os módulos anteriores, ou seja, apenas no arquivo que será executado é que chamaremos as outras funções que definem as variáveis necessárias para o funcionamento do programa conforme o esperado.

Um exemplo é que um módulo de “**cadastro de usuários**” é replicável entre vários projetos como:

- **Cadastro de clientes de uma pizzaria**
- **Padaria**
- **Empresa de logística**
- **E etc.**

Nos subitens a seguir, definiremos os conceitos de classe, atributos, métodos e objetos. Estes conceitos serão úteis para alcançar os efeitos de se utilizar o paradigma da orientação a objetos.

## 9.1 Criação de Classes e Objetos

As **classes** podem ser definidas como as estruturas que definem um módulo do projeto ou programa. Em sua definição, as nome no infinitivo do impessoal, uma vez que, não devemos especificar alguém ou alguma função da empresa.

São exemplos de **classe**, categorias denominadas como “**pessoa**”, “**veiculo**” ou mesmo “**animais**”. Perceba que não estamos definindo que tipo de pessoa, veículo ou animal, neste caso, a pessoa pode ser um coordenador da empresa ou diretor.

Nas Figuras 92 e 93, podemos visualizar a declaração de cada uma destas classes.

```
class pessoa:
    #Classe pessoa
    print("Classe pessoa")

class veiculo:
    #Classe veiculo
    print("Classe veiculo")

class animal:
    #Classe animal
    print("Classe animal")
```

**Figura 92:** Criação de classes.

```
Classe pessoa
Classe veiculo
Classe animal
```

**Figura 93:** Saída da criação de classes em Python.

- O comando **class** possibilita criarmos uma classe do programa.
- A palavra posterior a palavra de criação da classe é o nome da mesma e posteriormente os dois pontos “**:**”.
  - Abaixo desta palavra possuímos as funções, atribuições ou mesmo comentários relacionados a esta classe em específico

Os atributos podem ser definidos como as características das classes. No exemplo da classe “**pessoa**” é possível visualizar os atributos comuns a qualquer pessoa como:

- **Nome**
- **Idade**
- ...

A Figura 94 apresenta a criação dos atributos para as classes **pessoa**, **veiculo** e **animal**.

```

class pessoa:
    #Criação de atributos para pessoa
    Nome = ""
    Idade = 00

class veiculo:
    #Criação de atributos para veículo
    Marca = ""
    Modelo = ""

class animal:
    #Criação de atributos para animal
    Nome = ""
    Tipo = ""

```

**Figura 94:** Implementação de atributos nas classes **pessoa**, **veiculo** e **animal**.

- Os métodos podem ser definidos como os comportamentos da classe. No exemplo da classe **pessoa** podemos visualizar o método **falar\_pessoa()**. Na Figura 95, os métodos relacionados a cada classe são implementados.

```

class pessoa:
    #Criação de atributos para pessoa
    nome_pessoa = ""
    idade_pessoa = 00
    def falar_pessoa(self, falar_nome, falar_idade):
        self.falar_nome = nome_pessoa
        self.falar_idade = idade_pessoa

class veiculo:
    #Criação de atributos para veículo
    marca_veiculo = ""
    modelo_veiculo = ""
    def falar_veiculo(self, falar_marca, falar_modelo):
        self.falar_marca = marca_veiculo
        self.falar_modelo = modelo_veiculo

class animal:
    #Criação de atributos para animal
    nome_animal = ""
    tipo_animal = ""
    def falar_animal(self, falar_nome, falar_tipo):
        self.falar_nome = nome_animal
        self.falar_tipo = idade_animal

```

**Figura 95:** Exemplo de implementação de métodos às classes criadas.

Em sua criação, passamos três argumentos dentro dos parênteses “( )”:

- O **self** que realiza um chamado aos recursos internos do método. Cria um objeto com o atributo declarado.
- O “**falar\_nome**” e o “**falar\_idade**” que são utilizados como passagem de valores para posteriormente serem impressos.

Um exemplo é onde ao executar o **self**, passamos como parâmetro o “**falar\_nome**” que recebe o que atribuirmos em “**nome\_pessoa**”, ou seja, quando formos escrever o nome da pessoa, devemos usar o “**nome\_pessoa**”, uma vez que é através deste atributo que atribuímos o nome à pessoa e o relacionamos com o método.

Os objetos podem ser definidos como a classe em execução ou a classe em estado de execução. Quando falamos em objetos, a classe em execução está na memória RAM - *Randomic Access Memory* e pode ser recuperada ou ter seu(s) valor(es) sobreposto(s).

Nas Figuras 96 e 97, podemos visualizar a criação de objetos, a partir das classes “**pessoa**”, “**veiculo**” e “**animal**”.

```

#Criação dos objetos
pessoa_1 = pessoa()
pessoa_1.nome_pessoa = "Eduardo"

veiculo_1 = veiculo()
veiculo_1.marca_veiculo = "BMW"

animal_1 = animal()
animal_1.tipo_animal = "Aquático"

#Impressão dos objetos na tela
print(pessoa_1.nome_pessoa)
print(veiculo_1.marca_veiculo)
print(animal_1.tipo_animal)

```

**Figura 96:** Exemplo de criação de objetos a partir de classes.

Eduardo  
BMW  
Aquático

**Figura 97:** Saída da criação de objetos a partir de classes.

- No exemplo anterior, atribuímos à classe **pessoa()** um atributo. No exemplo, o atributo “**pessoa\_1**”.
- Posteriormente a sua atribuição, podemos apontar os atributos em que desejamos inserir algum conteúdo, como em “**pessoa\_1.nome\_pessoa = “Eduardo”**”.
- Ao final, imprimimos o valor incutido em cada atributo.

## 9.2 Encapsulamento, Herança e Polimorfismo

O encapsulamento pode ser definido como um ato de proteção, neste contexto, podemos proteger as informações de diversas maneiras, como por exemplo, através da criação de métodos e/ou atributos públicos, protegidos, privados e abstratos.

Os atributos ou métodos **públicos** são aqueles em que podemos reutilizar seu conteúdo em qualquer chamada, mesmo as exteriores que o(s) definiu ou no mesmo módulo. Os métodos públicos nos permitem compartilhar informações com outras partes do programa. As Figuras 98 e 99 apresentam um exemplo de implementação e posteriormente, a sua saída no *console* (tela).

```

class pessoa:
    def __init__(self):
        self.idade_publico = 34

    def falar_publico(self):
        return f"Exemplo de método público"

pessoa_1 = pessoa()

print(pessoa_1.idade_publico)
print(pessoa_1.falar_publico())

```

**Figura 98:** Exemplo de declaração de classe, métodos e atributos (declaração de atributos e métodos públicos).

34  
Exemplo de método público

**Figura 99:** Saída da declaração de atributos e métodos públicos.

Os atributos ou métodos **protegidos** aplicam um bloqueio para acessos externos aos seus recursos, porém, é possível recuperar informações desde que estejamos no mesmo diretório ou pasta. Este nível de proteção é aplicado para todos que estiverem fora da sendo que apenas chamadas de arquivo do mesmo local são consideradas confiáveis.

Nas Figuras 100 e 101, apresentamos um exemplo de atributo e método protegido.

```
class conta:
    def __init__(self):
        self._saldo_conta = 0

    def _consulta_protegida(self):
        return f"Seu saldo é: {self._saldo_conta}"

#Criação dos objetos
consulta_banco = conta()

#Impressão dos objetos na tela
print(consulta_banco._saldo_conta)
print(consulta_banco._consulta_protegida)
```

**Figura 100:** Declaração de atributos e métodos protegidos.

```
0
<bound method conta._consulta_protegida of <__main__.conta object at 0x0000026BCB6FC260>>
```

**Figura 101:** Saída da declaração de atributos e métodos protegidos.

Como pode ser visualizado, possuímos alguns pontos específicos para a classificação de um atributo ou método protegido em Python:

- As atributos e métodos classificados como protegidos devem possuir à frente de sua definição, o “\_”.

Os atributos ou métodos classificados como **privados** aplicam uma proteção ainda maior, os atributos e métodos para esta classificação não podem ser acessadas externamente mesmo que estejam no mesmo diretório. Já os atributos ou métodos **abstratos** podem ser instanciados (criados na memória) ou não geram objetos.

Nas Figuras 102 e 103, apresentamos um exemplo de atributo de método privado e ao final, a saída de erro que geramos ao tentar acessar as informações externamente.

```

class conta:
    def __init__(self):
        self.__saldo_conta = 0

    def __consulta_privada(self):
        return f"Seu saldo é: {self.__saldo_conta}"

#Criação do objeto
saldo = conta()

#Impressão do objeto na tela
print(saldo.__consulta_privada())

```

**Figura 102:** Declaração de atributos e métodos privados.

```

Traceback (most recent call last):
  File "C:\Users\eduar\Desktop\FabricaDe
Conteudos\Livro 2\Para a entrega\Fabrica
DeConteúdos\Codigos\privado.py", line 12
, in <module>
    print(saldo.__consulta_privada())
AttributeError: 'conta' object has no at
tribute '__consulta_privada'. Did you me
an: '__conta__consulta_privada'?

```

**Figura 103:** Saída (mensagem de erro) da tentativa de chamar informações privadas.

Utilizamos o encapsulamento **abstrato** quando definimos as classes mais abstratas que possuímos, como por exemplo, as classes “**pessoa**”, “**veiculo**” e “**animal**”. Na Figura 104, podemos visualizar a estrutura de um encapsulamento abstrato.

```

from abc import ABC, abstractmethod

class pessoa(ABC):
    @abstractmethod
    def falar(self):
        pass

```

**Figura 104:** Exemplo de declaração de método abstrato.

Como podemos visualizar, o carregamento de dois métodos específicos do módulo **abc** o **ABC** e o **abstractmethod**. Posteriormente ao carregamento do módulo e seus métodos podemos definir:

- Primeiramente, **abc** quer dizer **Abstract Base Classes**.
- O **@abstractmethod** implica na definição de que o método descrito a seguir, é abstrato.
- O comando **pass** faz com que o código continue e que a execução passe para as linhas subsequentes.

### Importante

Como pudemos visualizar, os atributos e métodos protegidos e privados não são acessíveis diretamente. Neste sentido, quando devemos alterar um valor interno e com este nível de proteção, devemos utilizar métodos de acesso como os **getters** e os **setters**.

Os **getters** realizam a recuperação de valores de forma indireta e os **setters** armazenam ou modificam os valores nos atributos protegidos. Podemos comparar o seu funcionamento a um banco, não sacamos ou depositamos diretamente no cofre, interagimos com o caixa que valida se podemos depositar ou retirar o dinheiro apontado.

O conceito de **herança** possibilita realizarmos a reutilização dos atributos e métodos criados na classe **Pai** ou **Superclasse**. As subclasses poderão acessar as informações da sua superclasse. Em um exemplo rápido, na superclasse “**pessoa**”, definimos os atributos comuns a qualquer pessoa. Mas sabemos que dentro da classificação de “**pessoa**” podemos separar mais ainda, como por exemplo, uma pessoa pode ser um **coordenador** de sistemas ou um **gerente**.

As categorias de pessoas, **coordenador** e **gerente** são subclasses de **pessoa** e neste caso, podem **receber** ou **herdar** atributos e métodos de “**pessoa**”. Nas Figuras 105 e 106 podemos visualizar um exemplo de superclasse e subclasse, assim como a saída de algumas informações em tela.

```

class pessoa:
    def __init__(self, nome):
        self.nome = nome

    def falar(self):
        pass

class coordenador(pessoa):
    def falar(self):
        return f"Eu, {self.nome} sou o coordenador"

class gerente(pessoa):
    def falar(self):
        return f"Eu, {self.nome} sou o gerente"

#Declaração dos objetos
pessoa_1 = coordenador("Eduardo")
pessoa_2 = gerente("Luis")

#Impressão dos objetos
print(pessoa_1.nome)
print(pessoa_1.falar())

print(pessoa_2.nome)
print(pessoa_2.falar())

```

**Figura 105:** Implementação da herança entre classes.

```

Eduardo
Eu, Eduardo sou
o coordenador
Luis
Eu, Luis sou o
gerente

```

**Figura 106:** Saída ou impressão dos objetos posteriormente à relação de herança.

Em POO possuímos ainda o **polimorfismo**. Este recurso possibilita que realizemos o acesso a um recurso de diferentes formas por exemplo, podemos definir o método **falar()** na superclasse e em suas subclasses apenas sobreescriver o valor, neste sentido, declaramos apenas um requisito.

Podemos visualizar na Figura 107 um exemplo de aplicação do polimorfismo. No exemplo apresentado, o método **falar()** é sobre sem a necessidade de se alterar o seu nome ou mesmo alterar a chamada interna dos parênteses “( )”, o que neste caso é o **self**.

### 9.3 Exemplo de Aplicação

Para exemplificarmos todo o conhecimento adquirido nesta seção, podemos construir um exemplo onde criamos uma superclasse chamada **veiculo** e uma subclasse **carro** e tratamos o encapsulamento e o polimorfismo para a criação de objetos, a partir desta estrutura, como por exemplo, um **Mitsubishi Lancer** ou mesmo um **Subaru WRX**.

Na Figura 107 podemos visualizar a implementação do exemplo e na Figura 108 a saída com os objetos criados na execução do código.

```
#Declaração da superclasse
class veiculo:
    def __init__(self, marca, modelo):
        self.__marca = marca #Privado
        self.__modelo = modelo #Privado

    def get_info(self):
        return f"Marca: {self.__marca}, Modelo: {self.__modelo}"

#Subclasse e reutilização(Polimorfismo) do método __init__
class carro(veiculo):
    def __init__(self, marca, modelo, ano):
        super().__init__(marca, modelo)
        self.__ano = ano #Privado

    def get_info(self):
        return f"Tipo: Carro, {super().get_info()}, Ano: {self.__ano}"

#Polimorfismo aplicado ao método
def exibir_info_veiculo(veiculo):
    print(veiculo.get_info())

#Objeto criado.
carro_1 = carro("Mitsubishi", "Lancer", 2023)
carro_2 = carro("Subaru", "WRX", 2023)

exibir_info_veiculo(carro_1) #Saída da função com polimorfismo
exibir_info_veiculo(carro_2) #Saída da função com polimorfismo
```

**Figura 107:** Implementação de polimorfismo.

```
Tipo: Carro, Marca: Mitsubishi,
Modelo: Lancer, Ano: 2023
Tipo: Carro, Marca: Subaru,
Modelo: WRX, Ano: 2023
```

**Figura 108:** Saída de implementação dos objetos posteriormente à aplicação de polimorfismo.

- No exemplo, o **super()** realiza a chamada do método apontado na classe original ou aquela que primeiro chamou o método associado.
  - Neste caso, **super().\_\_init\_\_(marca, modelo)** chamam o método e seus atributos da superclasse **veiculo**.

#### Dica

Os conceitos de POO são bastante poderosos e embora simples, necessitam ser muito bem compreendidos para sua fixação implementação correta. Neste sentido sugerimos que você assista com bastante atenção aos dois vídeos a seguir, os quais estão disponíveis no YouTube.

[Introdução à Programação Orientada a Objetos \(POO\): uma explicação fácil](#)

e

## 10. DESENVOLVIMENTO WEB COMPLETO

O desenvolvimento *Web* implica na existência de mais de uma camada na arquitetura de um sistema programado. Um programa desenvolvemos localmente e o acessamos de outro computador gera uma relação nomeada cliente e servidor. O servidor com a base de dados onde as informações serão armazenadas, manipuladas e o cliente que acessará e realizará operações.

A palavra *Web* neste contexto acrescenta mais uma camada, no caso, entre cliente e servidor, uma vez que, a comunicação agora será realizada via recursos de Internet e páginas de navegação. Aplicações de três camadas possibilitam ao usuário acessar informações sem a necessidade de estar na mesma rede ou mesmo no local onde a aplicação está hospedada. No decorrer desta seção visualizaremos as definições dos conceitos relacionados ao desenvolvimento *Web*, um novo *framework* e a comunicação com bancos de dados (bases de informações).

### 10.1 Conceitos e Arquitetura de Informação

Como dito anteriormente, trabalharemos com uma arquitetura de sistemas em 3 camadas e assim interagiremos com outros como o de arquitetura da informação. Através destes são definidas boas práticas de desenvolvimento de *software* e a sua utilização para alcançarmos aplicações mais intuitivas, otimizadas, organizadas, que atendam a requisitos de usabilidade, acessibilidade e ao final, supram as necessidades dos usuários.

São requisitos para o correto desenvolvimento nesta arquitetura:

- **A organização:** O site ou aplicação *Web* deverá prover uma navegação lógica e fluída aos usuários. As telas e suas transições devem fazer sentido para quem as acessar, ou seja, no caso das páginas de um mercado, a compra de materiais para reparos domésticos rápidos não deverá estar juntas de cosméticos para bebês.
- **O mapa do site:** É um recurso importante para os desenvolvedores, uma vez que contextualiza a navegação dentro do site. Até os mesmos podem ser visualizadas todas as relações entre pontos-chave da aplicação.
  - Existem várias técnicas de prototipagem e desenvolvimento de aplicações.
  - Do ponto de vista mais interativo ou visual, possuímos o *wireframe* (esqueleto de um projeto, o qual auxilia no entendimento do usuário irá interagir com o site ou aplicação). É uma técnica de prototipagem em baixa fidelidade para validação de conceitos de usabilidade (funcionamento/execução).
- **Foco no usuário:** É importante ressaltarmos que o desenvolvimento deverá ter como foco final o usuário, porém, a palavra final quer dizer que apenas na entrega devemos colocar a aplicação em contato com o usuário. É extremamente importante que o usuário participe dos processos de criação, desenvolvimento e validação.
  - Deste modo criaremos aplicações que farão sentido ao usuário, que refletirão seu modo de contextualizar as informações de fato representando o seu comportamento e expectativas.

### 10.2 Introdução aos Frameworks Flask e Django

São exemplos de frameworks (módulos) de desenvolvimento *Web*, o *Flask* <<https://flask.palletsprojects.com/>> e o *Django* <<https://www.djangoproject.com/>>. Ambos os são utilizados em *Python* e apesar da similaridade possuem aplicações e abordagens de desenvolvimento diferentes.

O *Flask* possui como características, sua facilidade de aprendizagem, capacidade de trabalhar com diferentes extensões (flexibilidade), propriedades que nos permitem manipular informações de forma mais livre através desta solução.

A instalação do *Flask* segue o mesmo fluxo que utilizamos na instalação do *numpy*. Neste caso usamos o *pip*, que é o gerenciador de pacotes do *Python*. Para a instalação de pacotes é uma boa prática de desenvolvimento a utilização de ambientes virtuais (áreas), pois tais recursos nos ajudam a criar um encapsulamento para a proteção de códigos em nosso ambiente.

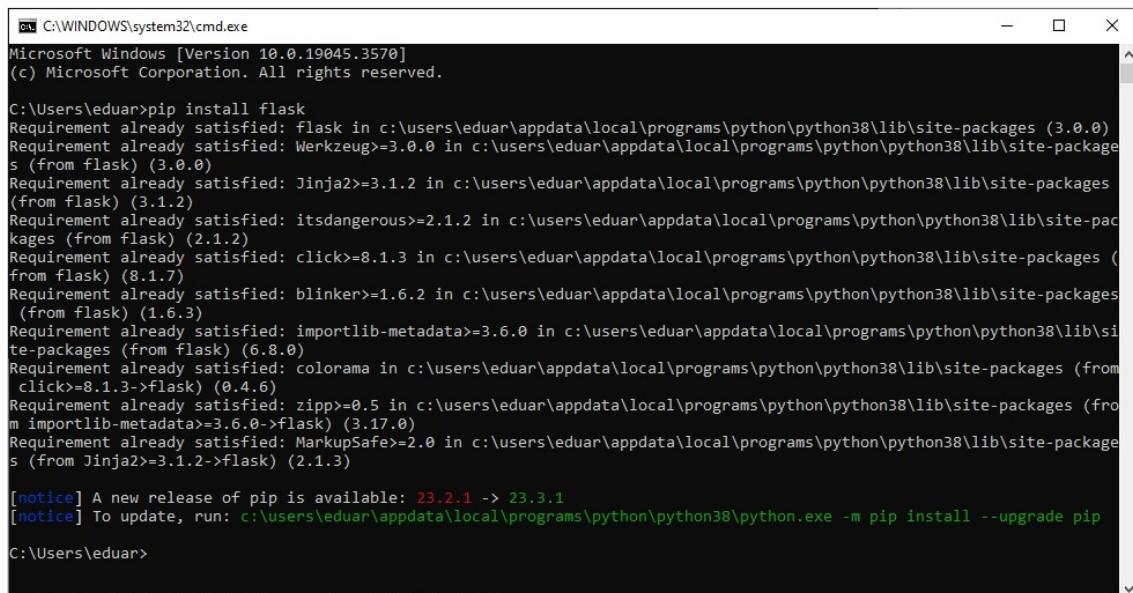
Para a criação de um ambiente virtual particular nosso, podemos executar o comando: `python -m venv nome_do_ambiente`, onde o nome do ambiente é de sua escolha.

Algumas vantagens da criação de um ambiente virtual específico:

- Não alterar a instalação *Python* padrão.
- Não poluir o SO com bibliotecas desnecessárias.

- Projetos diferentes podem ser configurados de forma customizada, cada um com o seu *framework* específico e apenas com o necessário de cada biblioteca.
- Facilidade de replicação do ambiente de desenvolvimento em outras máquinas/instâncias (servidores) através do arquivo **requirements.txt**<sup>1</sup>

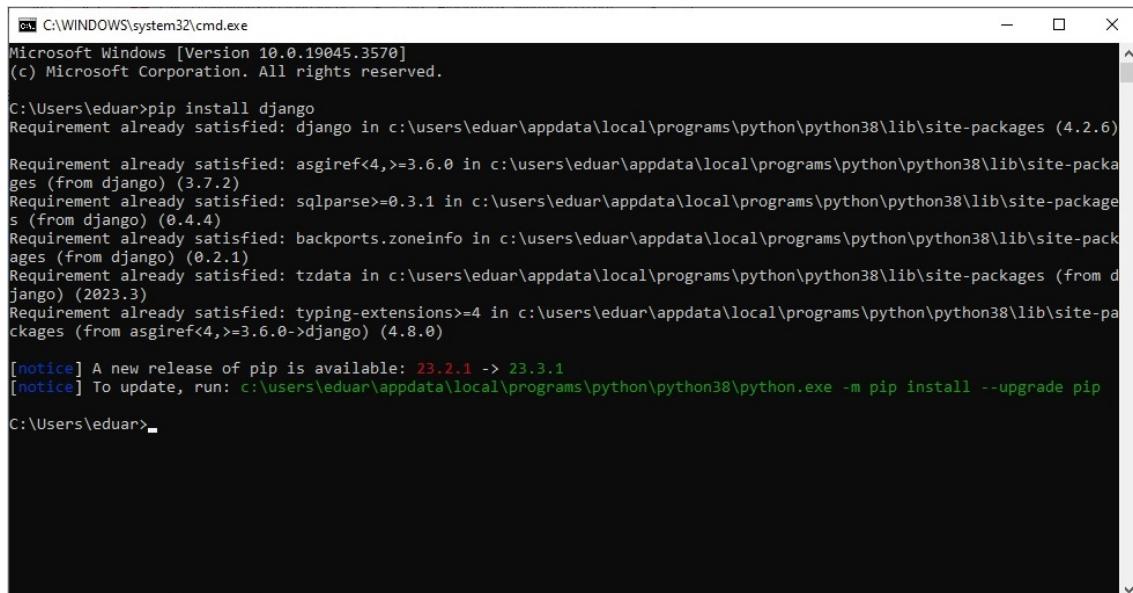
Com relação a instalação dos módulos, podemos executar os comandos **pip install flask** e **pip install django**, conforme pode ser visualizado nas Figuras 109 e 110 a seguir.



```
C:\Users\eduar>pip install flask
Requirement already satisfied: flask in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (3.0.0)
Requirement already satisfied: Werkzeug>=3.0.0 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from flask) (3.0.0)
Requirement already satisfied: Jinja2>=3.1.2 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from flask) (3.1.2)
Requirement already satisfied: itsdangerous>=2.1.2 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from flask) (2.1.2)
Requirement already satisfied: click>=8.1.3 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from flask) (8.1.7)
Requirement already satisfied: blinker>=1.6.2 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from flask) (1.6.3)
Requirement already satisfied: importlib-metadata>=3.6.0 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from flask) (6.8.0)
Requirement already satisfied: colorama in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from click>=8.1.3->flask) (0.4.6)
Requirement already satisfied: zipp>=0.5 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from importlib-metadata>=3.6.0->flask) (3.17.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from Jinja2>=3.1.2->flask) (2.1.3)

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: c:\users\eduar\appdata\local\programs\python\python38\python.exe -m pip install --upgrade pip
C:\Users\eduar>
```

Figura 109: Instalação do *Flask* através do gerenciador de pacotes **pip**.

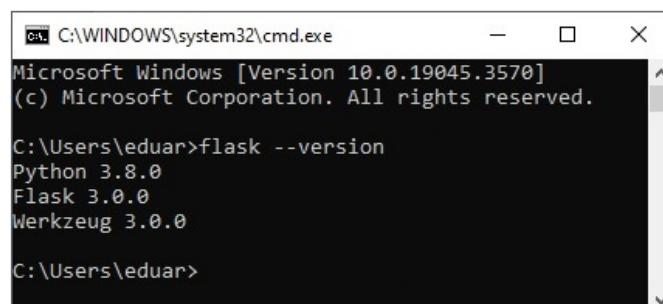


```
C:\Users\eduar>pip install django
Requirement already satisfied: django in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (4.2.6)

Requirement already satisfied: asgiref<4,>=3.6.0 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from django) (3.7.2)
Requirement already satisfied: sqlparse>=0.3.1 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from django) (0.4.4)
Requirement already satisfied: backports.zoneinfo in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from django) (0.2.1)
Requirement already satisfied: tzdata in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from django) (2023.3)
Requirement already satisfied: typing-extensions>=4 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from asgiref<4,>=3.6.0->django) (4.8.0)

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: c:\users\eduar\appdata\local\programs\python\python38\python.exe -m pip install --upgrade pip
C:\Users\eduar>
```

Figura 110: Instalação do *Django* através do gerenciador de pacotes **pip**.



```
C:\Users\eduar>flask --version
Python 3.8.0
Flask 3.0.0
Werkzeug 3.0.0
C:\Users\eduar>
```

**Figura 111:** Verificação de instalação do pacote *Flask*.

Com as instalações concluídas, podemos desenvolver uma aplicação *Web* com o *framework Flask*. Na Figura 112 visualizaremos o primeiro teste de implementação de página *Web* com a linguagem *Python*.

```
from flask import Flask

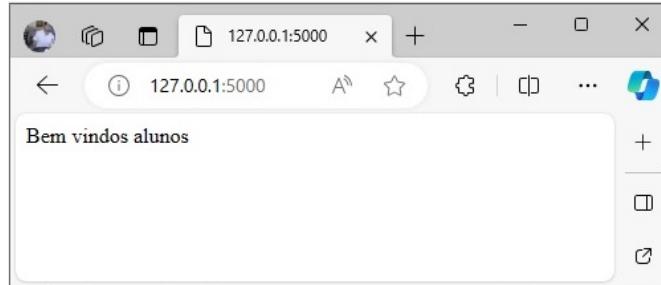
app = Flask(__name__)

@app.route('/')
def boas_vindas():
    return 'Bem vindos alunos'

if __name__ == '__main__':
    app.run()
```

**Figura 112:** Implementação do módulo *Flask* e execução da abertura de página *Web* localmente.

```
C:\Windows\System32\cmd.exe - flask --app app.py run
C:\Users\eduar\Desktop\FabricaDeConteudos\Livro 2\Codigos>flask
--app app.py run
* Serving Flask app 'app.py'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

**Figura 113:** Execução do servidor de página com a utilização do *Flask*.**Figura 114:** Exemplo de abertura da página em um *WebBrowser* (navegador) para a visualização do conteúdo criado.

- No código apresentado na Figura 112, o atributo **app** recebe o módulo **Flask** com o nome da localização atual do arquivo.
- O **@app.route('/')** realiza o apontamento da rota de acesso à raiz da página, neste caso será o **boas\_vindas()**.
- No último bloco de código verificamos se o aplicativo está sendo executado diretamente e então executamos a aplicação.
- Na Figura 113, iniciamos o servidor *Web*. Note que a mensagem - “Warning: This is a development server. Do not use it in a production deployment.” em tradução livre - “Aviso: Este é um servidor de desenvolvimento. Não o use em uma implantação de produção.”, é emitida pela característica da máquina utilizada (no caso um computador *desktop* local), porém não se preocupe pois o programa funcionará.
- Na Figura 114, as informações programadas são apresentadas na tela através do endereço IP padrão (*localhost*): **127.0.0.1** e porta **:5000**

O *framework Django* é considerado mais robusto que o *Flask*, possuindo várias funções pré-produzidas ou prontas para serem implementadas. Na Figura 111, já ilustramos como o *Django* pode ser instalado em nossa máquina local, entretanto, o seu acesso e administração se dá de forma diferente do *Flask*.

Para a configuração do *Django*, primeiramente devemos acessar o *prompt* de comandos digitando **cmd** ou **command** na barra c tarefas, e então executar a seguinte linha de comando:

```
django-admin startproject programacao_python
```

- O nome “**programacao\_python**” pode ser substituído por um de sua preferência.

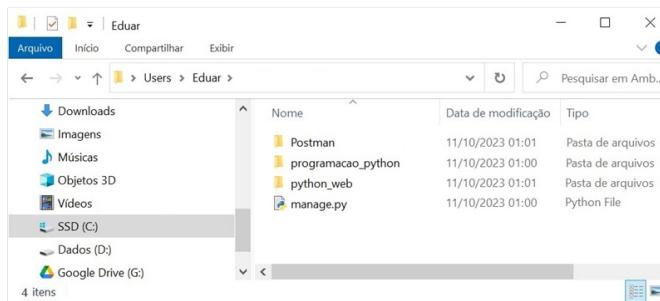
Nas Figuras 115, 116 e 117, podemos visualizar o processo de preparação do *Django* para desenvolvermos aplicações *Web*.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.3570]
(c) Microsoft Corporation. All rights reserved.

C:\Users\eduar>django-admin startproject programacao_python
```

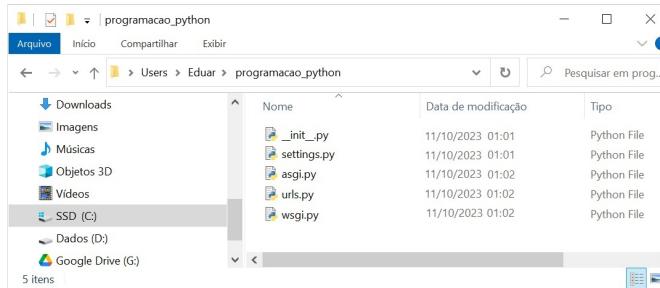
**Figura 115:** Criação de diretório para alocação do projeto/ambiente em *Django*.

- Na estrutura apresentada temos:



**Figura 116:** Diretórios e arquivo do ambiente.

- **programacao\_python:** É o diretório do projeto criado em *Django*.
- O **manage.py:** Se trata de um utilitário de linha de comando, responsável pelo fornecimento de alguns controles sobre a execução do ambiente *Web*.



**Figura 117:** Arquivos de configuração na pasta do ambiente (projeto) criado.

- O **\_\_init\_\_.py:** É um arquivo vazio, porém, está relacionado a um pacote de desenvolvimento necessário no projeto.
- O **settings.py:** É o arquivo de configurações gerais do projeto em *Django*.
- O **asgi.py:** Parte do código necessário para integrar chamadas ou servidores *Web* desta ordem com a aplicação.
- O **urls.py:** É um índice de páginas ou endereços na construção de um projeto com *Django*.
- O **wsgi.py:** Parte do código necessário para integrar chamadas ou servidores *Web* desta ordem com a aplicação.

Com estes conceitos definidos e o ambiente de projeto finalizado (criado), podemos executar o comando: **py manage.py runserver** no *prompt* de comando, de dentro do diretório do projeto para gerarmos uma página *Web* (url) e então abrirmos nosso pr

Nas Figuras 118 e 119, podemos visualizar um exemplo de execução do servidor e posteriormente a página correspondente aberta no navegador.

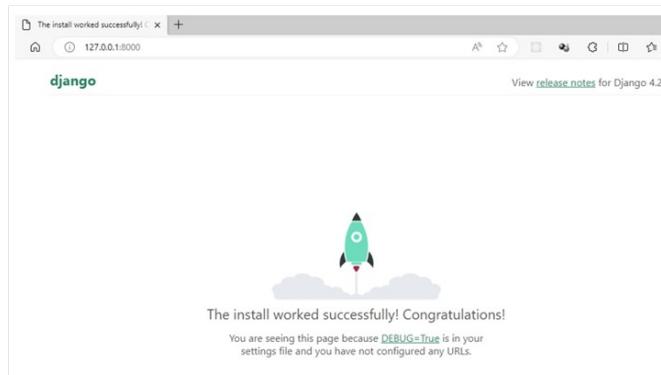
```
C:\Windows\System32\cmd.exe - py manage.py runserver
C:\Users\eduar\python_web>py manage.py run
Unknown command: 'run'.
Type 'manage.py help' for usage.

C:\Users\eduar\python_web>py manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work pr
operly until you apply the migrations for app(s): admin, auth, c
ontenttypes, sessions.
Run 'python manage.py migrate' to apply them.
October 11, 2023 - 17:22:32
Django version 4.2.6, using settings 'python_web.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

**Figura 118:** Execução do projeto criado em *Django*.



**Figura 119:** Exemplo de execução de página HTML em *Django*.

<sup>1</sup> Arquivo que lista todas as dependências necessárias para a execução de um projeto Python.

## 10.3 Acesso a Bancos de Dados

*Python* é uma linguagem de programação altamente versátil, ela possui conexões com outras linguagens, como por exemplo a linguagem SQL - *Structured Query Language*. A qual é o padrão para criação e manipulação de bancos de dados, e que pode ser acessada através de diferentes gerenciadores, como é o caso do SQLite, MySQL e SQLAlchemy.

Com a finalidade de desenvolvemos aplicações cada vez mais seguras e acessíveis, devemos utilizar recursos que nos forneça opções de vasto armazenamento e alta disponibilidade, ou seja, que comportem uma grande massa de dados e que estas informaçõe estejam disponíveis quando necessário.

Para trabalharmos com bancos de dados, precisamos utilizar as bibliotecas referentes ao gerenciador escolhido. A instalação de dessas bibliotecas, podem ser visualizadas nas Figuras 120 e 121 a seguir.

Para as instalarmos devemos utilizar as seguintes linhas de comando:

- Para o SQLite:

**pip install pysqLite3 ou pip install sqlite3**

- Para o MySQL:

**pip install mysql-connector-python**

- Para o SQLAlchemy:

**pip install sqlalchemy**

Nas Figuras 120 e 121 podemos visualizar exemplos de instalação dos pacotes necessários para realizarmos operações de armazenamento em diferentes bancos de dados.

```
C:\Windows\System32\cmd.exe
ogramacao_python>pip install sqlalchemy
Requirement already satisfied: sqlalchemy in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (2.0.21)
Requirement already satisfied: typing-extensions>=4.2.0 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from sqlalchemy) (4.8.0)
Requirement already satisfied: greenlet<0.4.17 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from sqlalchemy) (3.0.0)

[note] A new release of pip is available: 23.2.1 -> 23.3.1
[note] To update, run: c:\users\eduar\appdata\local\programs\python\python38\python.exe -m pip install --upgrade pip
C:\Users\eduar\Desktop\FabricaDeConteudos\Livro 2\Para a entrega\FabricaDeConteudos\programacao_python>
```

**Figura 120:** Instalação do **sqlalchemy** através do **pip**.

```
C:\Windows\System32\cmd.exe
ogramacao_python>pip install mysql-connector-python
Requirement already satisfied: mysql-connector-python in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (8.1.0)
Requirement already satisfied: protobuf<=4.21.12,>=4.21.1 in c:\users\eduar\appdata\local\programs\python\python38\lib\site-packages (from mysql-connector-python) (4.21.12)

[note] A new release of pip is available: 23.2.1 -> 23.3.1
[note] To update, run: c:\users\eduar\appdata\local\programs\python\python38\python.exe -m pip install --upgrade pip
C:\Users\eduar\Desktop\FabricaDeConteudos\Livro 2\Para a entrega\FabricaDeConteudos\programacao_python>
```

**Figura 121:** Instalação do **mysql-connector-python** através do **pip**.

Com a instalação do módulo escolhido, podemos então realizar as operações de escrita e leitura em nossos bancos de dados. C exemplo que vamos utilizar aqui serão manipulados com o SQLiteStudio.

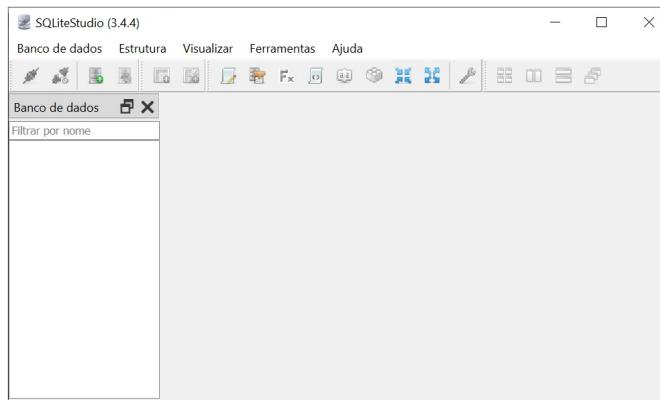
O processo de manipulação de dados armazenados em bancos de dados, segue basicamente as atividades listadas:

- Conexão com o banco de dados e recuperação de registros.
- Criação de tabelas e campos nas mesmas.
- Inserção e remoção de registros (conteúdo das tabelas).

### 10.3.1 Conexão com Banco de Dados e Recuperação de Registros

Para executarmos operações com gerenciadores de bancos de dados, primeiramente precisamos baixar e instalar o gerenciador nossa escolha, como por exemplo, em nosso caso, o SQLiteStudio <<https://sqlitestudio.pl/>>.

Na Figura 122, visualizamos a ferramenta em questão sendo executada pela primeira vez em nosso equipamento de testes.



**Figura 122:** Tela principal do gerenciador de banco de dados **SQLiteStudio**.

A Figura 123 apresenta a conexão com o banco de dados, neste caso a gerenciaremos via SQLiteStudio.

```
import sqlite3
conexao_banco = sqlite3.connect('programacao_BD.db')
cursor = conexao_banco.cursor()
```

**Figura 123:** Criação do banco de dados.

- Na primeira linha carregamos o pacote **sqlite3**.
- Na segunda linha, criamos a conexão com o banco de dados. A variável **conexao\_banco** recebe **sqlite3.connect()** e dentro de parênteses “( )” colocamos o nome do banco de dados, em nosso exemplo **programacao\_BD.db**
- Na terceira linha possuímos a criação do recurso **cursor** que utilizaremos para inserir, pesquisar e deletar informações do banco de dados.

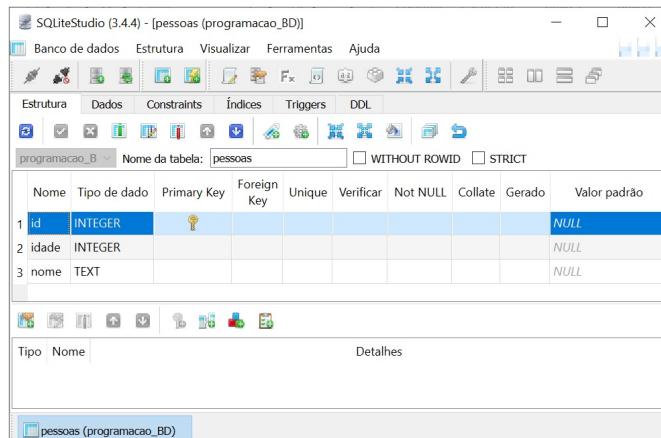
### 10.3.2 Criar uma Tabela e os Seus Campos

Posteriormente à conexão com o banco de dados, podemos realizar as operações com a sua estrutura. Na Figura 124, podemos visualizar a criação da tabela **pessoas** se a mesma não existir. Através do **cursor**, podemos executar o código em linguagem SQL. Na Figura 125 as colunas criadas durante a execução do banco podem ser abertas no SQLiteStudio.

- No **execute()** possuímos um código em SQL para a criação da tabela se ela não existir e os seus campos **id** (chave primária), **nome** e **idade**.
- O método **commit()** possibilita o envio de aviso ao banco de dados para que as alterações tenham efeito.
- O **close()** realiza a finalização da conexão.

```
import sqlite3
conexao_banco = sqlite3.connect('programacao_BD.db')
cursor = conexao_banco.cursor()
cursor.execute('''CREATE TABLE IF NOT EXISTS pessoas (
    id INTEGER PRIMARY KEY,
    name TEXT,
    idade INTEGER) ''')
conexao_banco.commit()
conexao_banco.close()
```

**Figura 124:** Criação da tabela **pessoas** no banco de dados **programacao\_BD**.



**Figura 125:** Visualização da estrutura da tabela **pessoas** no SQLiteStudio.

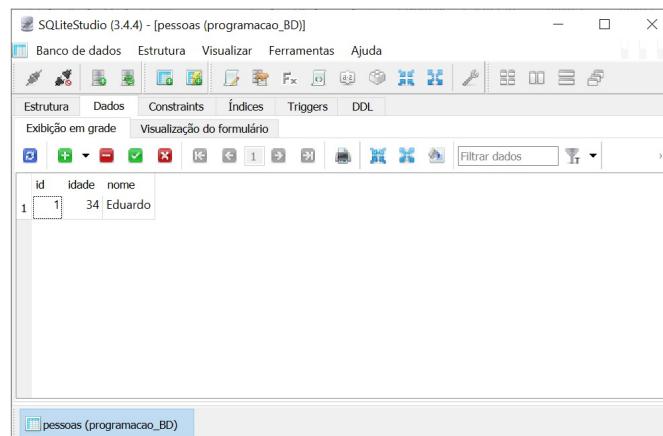
### 10.3.3 Inserir os Registros (conteúdo da tabela)

Na Figura 126 é possível visualizarmos no código para inserirmos um registro no banco de dados. Na Figura 127 podemos visualizar o registro adicionado no banco de dados e visualizado no SQLiteStudio.

- Na execução do código SQL, os símbolos “?, ?” (interrogação), significam que os valores são inseridos dinamicamente pelo usuário.

```
import sqlite3
conexao_banco = sqlite3.connect('programacao_BD.db')
cursor = conexao_banco.cursor()
inserir_dados = ("Eduardo", 34)
cursor.execute("INSERT INTO pessoas (nome, idade) VALUES (?, ?)", inserir_dados)
conexao_banco.commit()
conexao_banco.close()
```

**Figura 126:** Inserção de dados na tabela **pessoas** do banco de dados **programacao\_BD**.



**Figura 127:** Saída da execução do código de criação do primeiro registro no banco de dados.

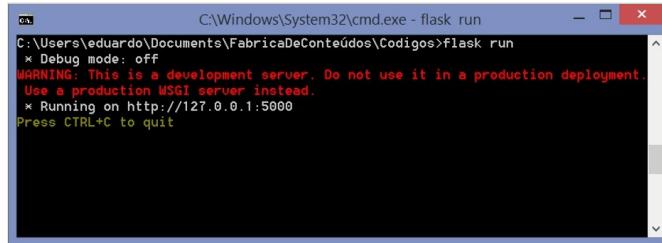
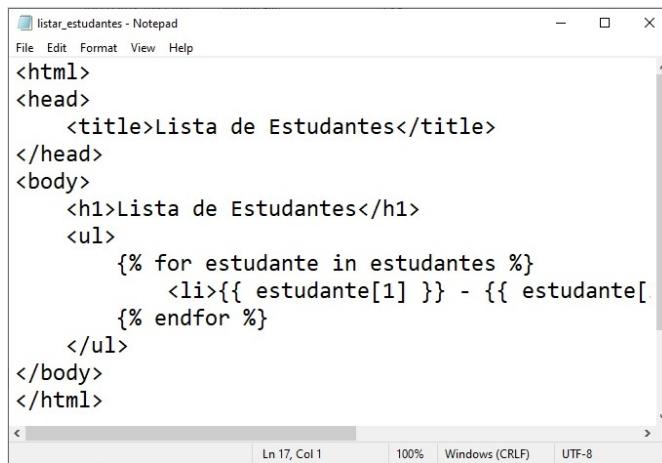
#### Importante

Para o gerenciamento de bancos de dados, precisamos estudar os códigos de armazenamento, consulta, modificação e exclusão de dados, os quais em grande parte das vezes são realizados por meio de comandos SQL nativos.

### 10.4 Exemplos de Aplicação

No exemplo de aplicação, podemos visualizar os conceitos trabalhados até o momento de forma prática. Nas Figuras 128 e 129 apresentamos um exemplo de implementação de manipulação de dados com a biblioteca *Flask*.

A Figura 128 é apenas uma apresentação do recurso necessário para podermos executar o código. Na Figura 129, apresentamos a página HTML para a visualização das informações e na Figura 130, apresentamos o código com que podemos integrar os exemplos anteriores realizando sua chamada.

Figura 128: Criação da conexão com o *Flask*.Figura 129: Arquivo HTML *listar\_estudantes*.Figura 130: Exemplo de implementação do código utilizando o *Flask* e a página Web em HTML.

- Em nosso último exemplo, podemos visualizar:

- O `render_template()` realiza o carregamento da página HTML com as informações geradas via *Flask* e o banco de dados.

### Desafio

Você foi contratado para desenvolver uma aplicação *Web* para uma pequena livraria que opera por meio de uma página *online* na qual vende livros físicos e eBooks. O cliente deseja que a aplicação permita aos usuários visualizar e comprar seus livros, adicionar livros ao carrinho de compras e realizar *checkout*. Além disso, o cliente quer que a aplicação tenha um painel de administração onde eles possam gerenciar o estoque de livros e as vendas. Como você abordaria o desenvolvimento dessa aplicação usando o framework *Flask*?

## BREVE DESPEDIDA...

Chegamos ao final de nosso curso, esperamos que você agora seja capaz de aplicar os conceitos, utilizar os módulos/bibliotecas/ferramentas vistas, nos mais variados problemas de desenvolvimento de software e que tenhamos deixado claro que um problema de programação pode ser solucionado de diferentes maneiras.

O elemento mais importante sobre qualquer conteúdo é a certeza de que separamos de estudar e nos atualizar, ficaremos para portanto, não se limite a este curso, avance para novos conhecimentos, pesquise sobre mais conceitos de programação e se aprofunde nos elementos apresentados aqui e o mais importante, treine com as ferramentas vistas, teste outras e aplique de forma prática tudo que puder.

Com estas sugestões, deixamos aqui os nossos sinceros parabéns por sua conclusão e desejamos profundamente que dê continuidade em novas pesquisas sobre estes e outros assuntos.

## Tags do conteúdo

Python, Operadores, Funções, Métodos, Atributos

Variáveis Locais, Variáveis Globais

Classes, Bibliotecas, Listas, Tuplas

Conjuntos, Dicionários, Escopo, Retorno

Classes, Loops, Loopings, Blocos, Controles de Fluxo

Herança, Encapsulamento, Polimorfismo, Exceções, Try, Except, Finally, Raise, Tratamento de Erros

Banco de Dados, Gerenciador de Banco de Dados

SQL, Structured Query Language, SQLite, SQLiteStudio, MySQL, SQLAlchemy

HTML, Django, Flask, Cliente-Servidor

## Referências (sugeridas)

BEIGHLEY, Lynn. **Use a cabeça! SQL**. 2 ed. Alta Books, 2008.

GRINBERG, Miguel. **Desenvolvimento Web com Flask: Desenvolvendo aplicações web com Python**. Novatec, 2019.

MACIEL, Francisco Marcelo de Barros. **Python e Django: Desenvolvimento web moderno e ágil**. Alta Books, 2020.

MATTHES, Eric. **Curso Intensivo de Python: Uma introdução prática e baseada em projetos à programação**. 3 ed. Novatec, 20

MCFEDRIES, Paul. **HTML, CSS, & JavaScript All-in-One For Dummies**. For Dummies, 2023.

NOBLE, Jeff. **HTML, XHTML e CSS para Leigos**. Alta Books, 2015.

QUINTÃO, Patrícia. **Banco de Dados Relacionais - Parte I**. Livro Eletrônico. [S.l.: s.n.], 2020.

RAMALHO, Luciano. **Python Fluente**. Novatec, 2015.

TETILA, E. C. **Banco de Dados Relacional**. Curitiba: Appris Editora, 2022.

### Nota:

Todas as marcas citadas e/ou exibidas neste material, pertencem aos seus respectivos fabricantes e/ou desenvolvedores.

