



**PYTHON**

## **Lógica de Programação e Conceitos Básicos**

Anderson Silva Vanin  
Cíntia Maria de Araújo Pinho

By Tecnologia Única

**única!**

Mauá  
2024

## 1. INTRODUÇÃO

A programação de computadores é uma habilidade essencial no mundo tecnológico contemporâneo, desempenhando um papel crucial no desenvolvimento e funcionamento de uma ampla gama de aplicativos e sistemas. Em sua essência, programar envolve criar conjuntos de instruções lógicas que um computador pode entender e executar, permitindo a automação de tarefas e a resolução de problemas complexos. Esta disciplina está no cerne da revolução digital e é um componente fundamental para profissionais de tecnologia, desde desenvolvedores de software até cientistas de dados.

Ao longo do tempo, várias linguagens de programação foram desenvolvidas para atender às diferentes necessidades e paradigmas de programação. Cada linguagem possui suas próprias características e aplicativos específicos, proporcionando aos programadores uma gama diversificada de ferramentas para expressar suas ideias e solucionar problemas. Com o aumento da complexidade das tecnologias e a demanda por soluções inovadoras, a habilidade de programar tornou-se uma competência altamente valorizada em diversas áreas, ampliando as oportunidades de carreira para aqueles que dominam essa arte.

A programação não se limita apenas à criação de software; ela é também uma ferramenta poderosa para explorar conceitos matemáticos e lógicos, promovendo o desenvolvimento do pensamento analítico e da resolução de problemas. Além disso, a comunidade de programadores é conhecida por sua cultura colaborativa, onde o compartilhamento de conhecimento e a contribuição para projetos de código aberto desempenham um papel crucial no avanço da tecnologia. Em última análise, a programação de computadores é um campo dinâmico que continua a evoluir, moldando o futuro da inovação e desafiando indivíduos a expandir suas habilidades intelectuais.



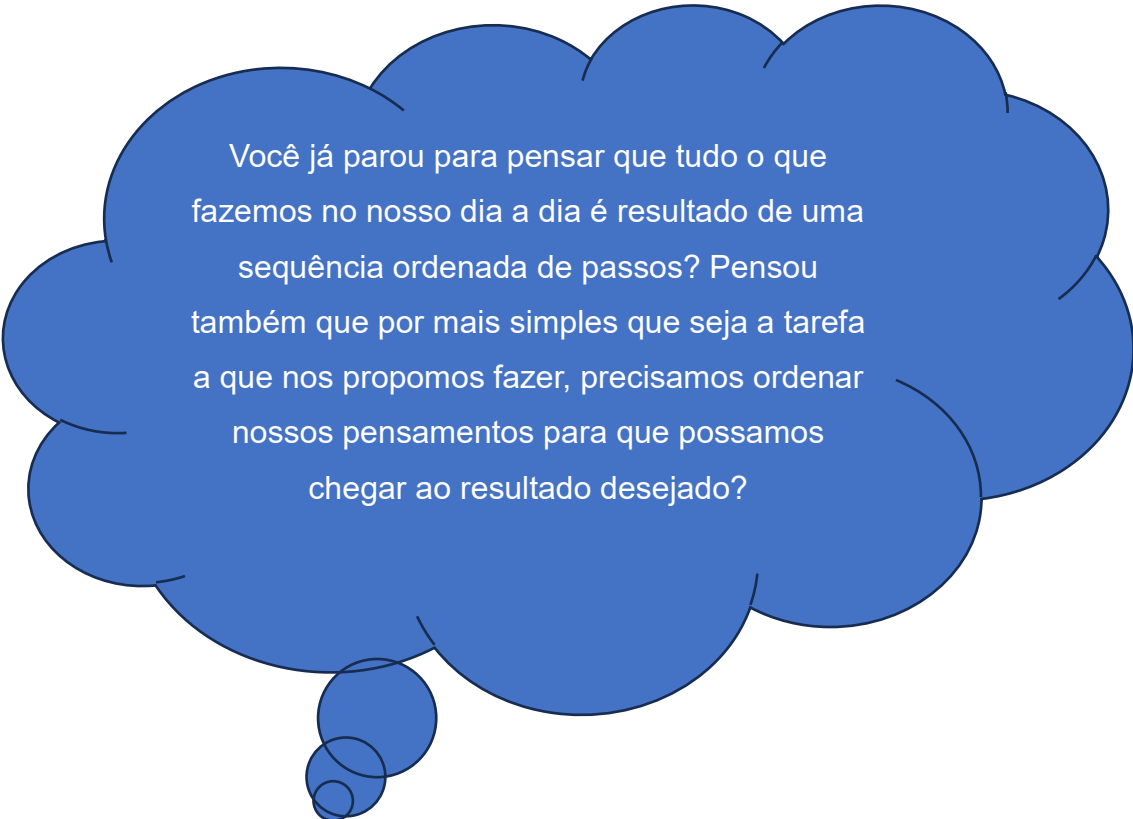
## 1.1. Lógica de Programação, Instruções e Algoritmos

Figura 1 - Ilustração lógica, instruções e algoritmos



A Lógica de Programação é um pilar fundamental no universo da computação, representando a habilidade essencial de criar sequências lógicas e coerentes de instruções que um computador pode compreender e executar. É o alicerce sobre o qual são construídos algoritmos e programas de computador. Em sua essência, a lógica de programação envolve a capacidade de estruturar pensamentos de maneira lógica e sistemática, traduzindo ideias e soluções em um formato que pode ser interpretado pela máquina.





Você já parou para pensar que tudo o que fazemos no nosso dia a dia é resultado de uma sequência ordenada de passos? Pensou também que por mais simples que seja a tarefa a que nos propomos fazer, precisamos ordenar nossos pensamentos para que possamos chegar ao resultado desejado?

As instruções em programação são comandos específicos que direcionam o computador a realizar tarefas específicas. Estas instruções podem variar desde operações simples, como a atribuição de valores a variáveis, até estruturas mais complexas, como loops e condicionais. A clareza e precisão na formulação de instruções são cruciais para garantir a eficácia e a confiabilidade do código, evitando ambiguidades que poderiam levar a erros de execução.



**Figura 2 - Exemplo de instruções para abrir uma lata de refrigerante e beber o conteúdo.**



Observe a figura acima e pense em trocar a ordem de algum dos passos ilustrados. Por exemplo:

- 1 – Tomar conteúdo
- 2 – Abrir a tampa
- 3 – Pegar o recipiente

Será que é possível beber o refrigerante da lata?

Claro que não! É preciso seguir a ordem dos passos ilustrados acima para que consigamos atingir o objetivo final, que neste caso, é tomar o conteúdo.

Os algoritmos, por sua vez, são sequências finitas e ordenadas de passos ou regras que descrevem como resolver um problema ou realizar uma tarefa específica. Eles são a expressão prática da lógica de programação e podem ser implementados em diversas linguagens, sendo uma representação abstrata de uma solução. Um bom algoritmo é eficiente, claro e capaz de resolver o problema proposto independentemente da linguagem de programação escolhida para implementação.

No processo de aprendizado da lógica de programação, os iniciantes frequentemente começam com exercícios simples, como a resolução de problemas matemáticos, para desenvolver a capacidade de pensar de maneira



lógica. Posteriormente, são introduzidas as estruturas de controle, como loops e condicionais, e aprendem a criar algoritmos mais elaborados para resolver problemas do mundo real.

Em resumo, a lógica de programação, as instruções e os algoritmos são conceitos intrinsecamente interligados no desenvolvimento de software. Dominar esses fundamentos é crucial para qualquer programador, pois representa a base sobre a qual são construídas soluções eficientes e elegantes para uma ampla variedade de problemas.

## **1.2. Encadeamento de pensamentos**

Encadear pensamentos é uma técnica cognitiva que envolve a conexão lógica de ideias e informações para chegar a uma solução ou conclusão. Essa abordagem é fundamental para resolver problemas complexos, pois permite a análise sequencial de dados e a formulação de estratégias para atingir um objetivo específico.

No dia a dia, essa técnica pode ser aplicada em diversas situações. Considere, por exemplo, a resolução de um problema de gestão de tempo. O encadeamento de pensamentos pode começar identificando as tarefas prioritárias, estabelecendo uma ordem de execução com base na urgência e importância. Em seguida, podem ser pensadas estratégias para otimizar o tempo gasto em cada atividade, como a eliminação de distrações e a definição de metas de curto prazo.

Outro exemplo prático é a solução de um problema de comunicação. Encadear pensamentos neste contexto envolve a análise de como a mensagem está sendo transmitida, a escolha adequada de palavras, e a consideração do público-alvo. A identificação de possíveis mal-entendidos e a adaptação da comunicação para garantir clareza são passos essenciais nesse processo.

No âmbito da programação, podemos exemplificar o encadeamento de pensamentos ao resolver um problema simples, como o cálculo da média de três números. Inicialmente, é preciso entender a fórmula da média, que consiste na soma dos valores dividida pelo número de elementos. O encadeamento de



pensamentos pode seguir identificando os três números dados, somando-os e, finalmente, dividindo a soma por 3 para obter a média. Este processo é uma representação direta da técnica de encadeamento de pensamentos na resolução de um problema específico.

Portanto, a técnica de encadear pensamentos é uma habilidade cognitiva valiosa que transcende diferentes áreas da vida. Ao aplicar essa abordagem, as pessoas podem abordar problemas com clareza e eficiência, identificando estratégias e implementando soluções de maneira lógica e estruturada.

### 1.3. Exemplo de algoritmos

**Figura 3 - Problema: Trocar uma lâmpada**



Vamos criar um algoritmo simples em português estruturado para resolver o problema de trocar uma lâmpada. Este algoritmo pode servir como uma sequência de passos lógicos para alguém que esteja realizando essa tarefa pela primeira vez. Vamos lá:

#### 1. Início:

- O algoritmo começa.

#### 2. Identificação da Lâmpada Queimada:



- Verificar se a lâmpada está queimada. Pode-se fazer isso observando se a luz não está mais acendendo.

### **3. Desligar a Energia:**

- Certificar-se de que a energia elétrica está desligada para evitar choques elétricos. Se houver um interruptor, desligá-lo. Caso contrário, desligar o disjuntor correspondente.

### **4. Escolha da Nova Lâmpada:**

- Escolher uma nova lâmpada que seja compatível com a base da lâmpada antiga. Verificar as especificações na embalagem da lâmpada.

### **5. Posicionamento de Materiais Necessários:**

- Posicionar uma escada ou cadeira próxima ao local da lâmpada para alcançar facilmente a área.

### **6. Remoção da Lâmpada Antiga:**

- Com cuidado, girar a lâmpada antiga no sentido anti-horário até que ela se solte. Caso seja um modelo de encaixe, pressionar o prendedor e retirar a lâmpada.

### **7. Inserção da Nova Lâmpada:**

- Posicionar a nova lâmpada no soquete e girá-la no sentido horário até que esteja firmemente fixada. Se for um modelo de encaixe, empurrar a nova lâmpada até que o prendedor a mantenha no lugar.

### **8. Ligação da Energia:**

- Restabelecer a energia elétrica. Ligar o interruptor ou ligar o disjuntor novamente.

### **9. Teste da Nova Lâmpada:**





- Acionar o interruptor para verificar se a nova lâmpada está funcionando corretamente.

#### 10. Fim:

- O algoritmo é concluído.

Esse algoritmo simples oferece uma sequência lógica de passos para trocar uma lâmpada com segurança, destacando a importância de desligar a energia antes da substituição. Essa abordagem estruturada pode ser útil para orientar alguém que não está familiarizado com o processo, garantindo uma troca de lâmpada eficiente e segura.

Aqui está um exemplo de algoritmo em português estruturado para calcular a média de três números inteiros:

#### **Algoritmo:** Cálculo da Média de 3 Números Inteiros

##### 1. Início:

- O algoritmo começa.

##### 2. Entrada de Dados:

- Solicitar ao usuário que informe o primeiro número inteiro e armazenar em uma variável, por exemplo, 'num1'.
- Solicitar ao usuário que informe o segundo número inteiro e armazenar em uma variável, por exemplo, 'num2'.
- Solicitar ao usuário que informe o terceiro número inteiro e armazenar em uma variável, por exemplo, 'num3'.

##### 3. Cálculo da Soma:

- Calcular a soma dos três números:  $\text{'soma' = num1 + num2 + num3}$ .

##### 4. Cálculo da Média:

- Calcular a média dividindo a soma pelo número de elementos (3):  $\text{'media' = soma / 3}$ .



**5. Exibição do Resultado:**

- Exibir a média calculada: 'Esse é o resultado da média: [valor da média]'.

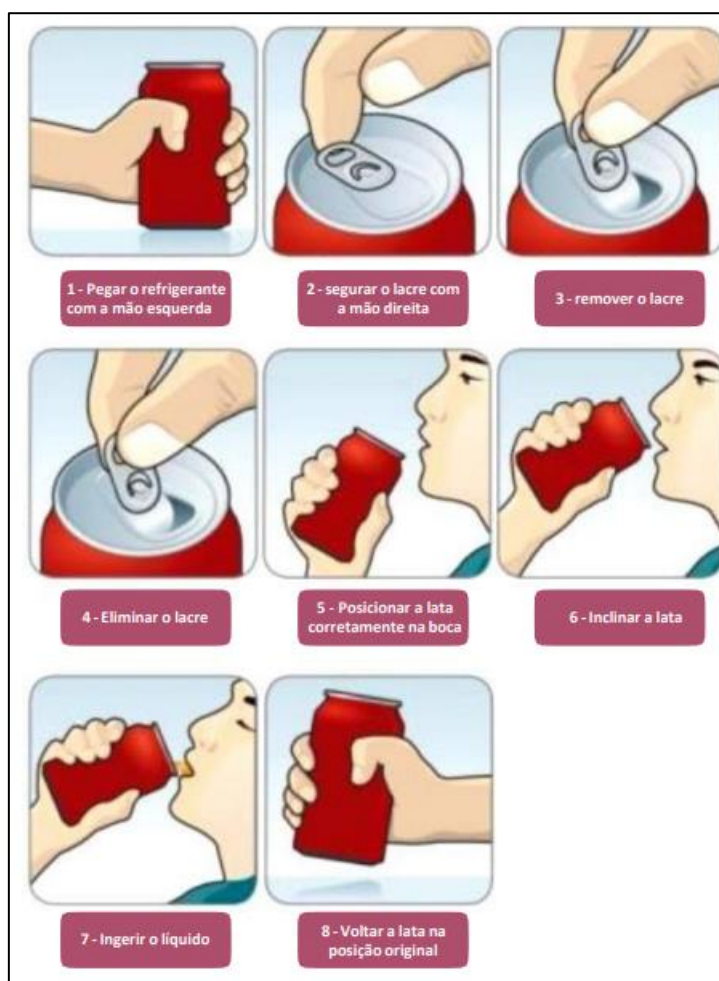
**6. Fim:**

- O algoritmo é concluído.

Lembre-se! Quanto mais detalhada for a instrução para o computador, mais rápido e fácil ele compreenderá e a executará atingindo o objetivo final.



**Figura 4 - Melhorando as instruções de Abrir e beber o conteúdo de uma lata de refrigerante.**



Este é o primeiro passo para resolver um problema. Para que todos possam compreender o seu algoritmo, é necessário utilizar o Fluxograma.

Mas o que é o Fluxograma? É a representação gráfica da sua sequência lógica (seu algoritmo).

Podemos usar qualquer diagrama ou qualquer desenho? Não. Existem as formas corretas com seus respectivos significados a serem utilizados como veremos na tabela seguir:



Figura 5 - Simbologia para fluxogramas











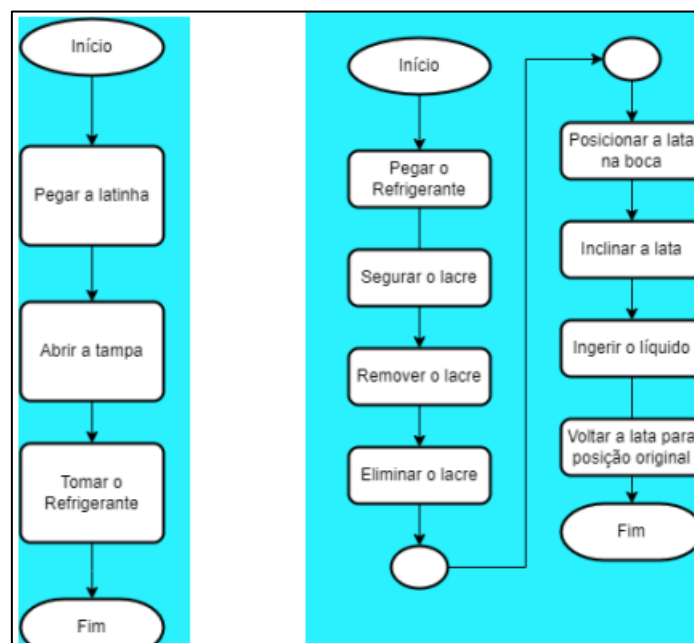
SIMBOLOGIA DO FLUXOGRAMA	
SÍMBOLO	NOME E FUNÇÃO
	NOME = TERMINAL FUNÇÃO = indica INÍCIO ou FIM de um processamento
	NOME = PROCESSAMENTO FUNÇÃO = definição de variáveis ou processamentos em geral (cálculos)
	NOME = ENTRADA MANUAL FUNÇÃO = entrada de dados via teclado, idêntico ao comando LEIA
	NOME = DISPLAY FUNÇÃO = saída de dados, mostra um texto e/ou variável na tela, idêntico ao comando ESCREVA
	NOME = DOCUMENTO FUNÇÃO = saída de dados, envia um texto e/ou variável para a impressora, usado em relatórios. Idêntico ao comando IMPRIMA
	NOME = DECISÃO FUNÇÃO = decisão a ser tomada, retornando verdadeiro ou falso, idêntico ao comando SE
	NOME = CONECTOR FUNÇÃO = desvia o fluxo para uma outra página, sendo interligado pelo conector
	NOME = entrada/saída FUNÇÃO = leitura de gravação de arquivos
	NOME = SETA FUNÇÃO = indica a direção do fluxo
	NOME = LOOP FUNÇÃO = realiza o controle de LOOP



Figura 6 - Fluxograma do algoritmo de beber o conteúdo de uma lata de refrigerante.

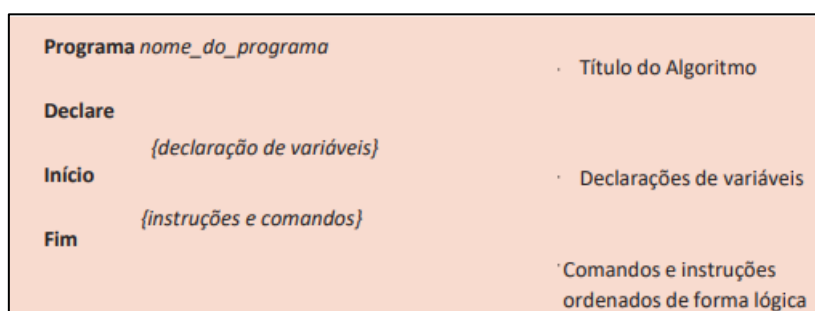


#### 1.4. Pseudocódigo

É escrita mais próxima da Linguagem de Programação, ou seja, não usaremos nenhuma informação técnica da Linguagem, apenas utilizaremos o nosso idioma (português) escrevendo mais próximo das instruções computacionais. Muitos autores chamam o Pseudocódigo de “Portugol” ou “Português Estruturado” devido a estas características.

A seguir um pequeno modelo, como orientação, para escrever um Pseudocódigo:

Figura 7 - Sintaxe de um pseudocódigo



#### 1.5. Exercícios

Utilizando os conceitos apresentados...



1. Crie um algoritmo e um fluxograma para fritarmos um ovo. Faça este algoritmo com uma sequência de no mínimo 15 passos.

Confira abaixo se você conseguiu resolver os desafios propostos!

### **Respostas:**

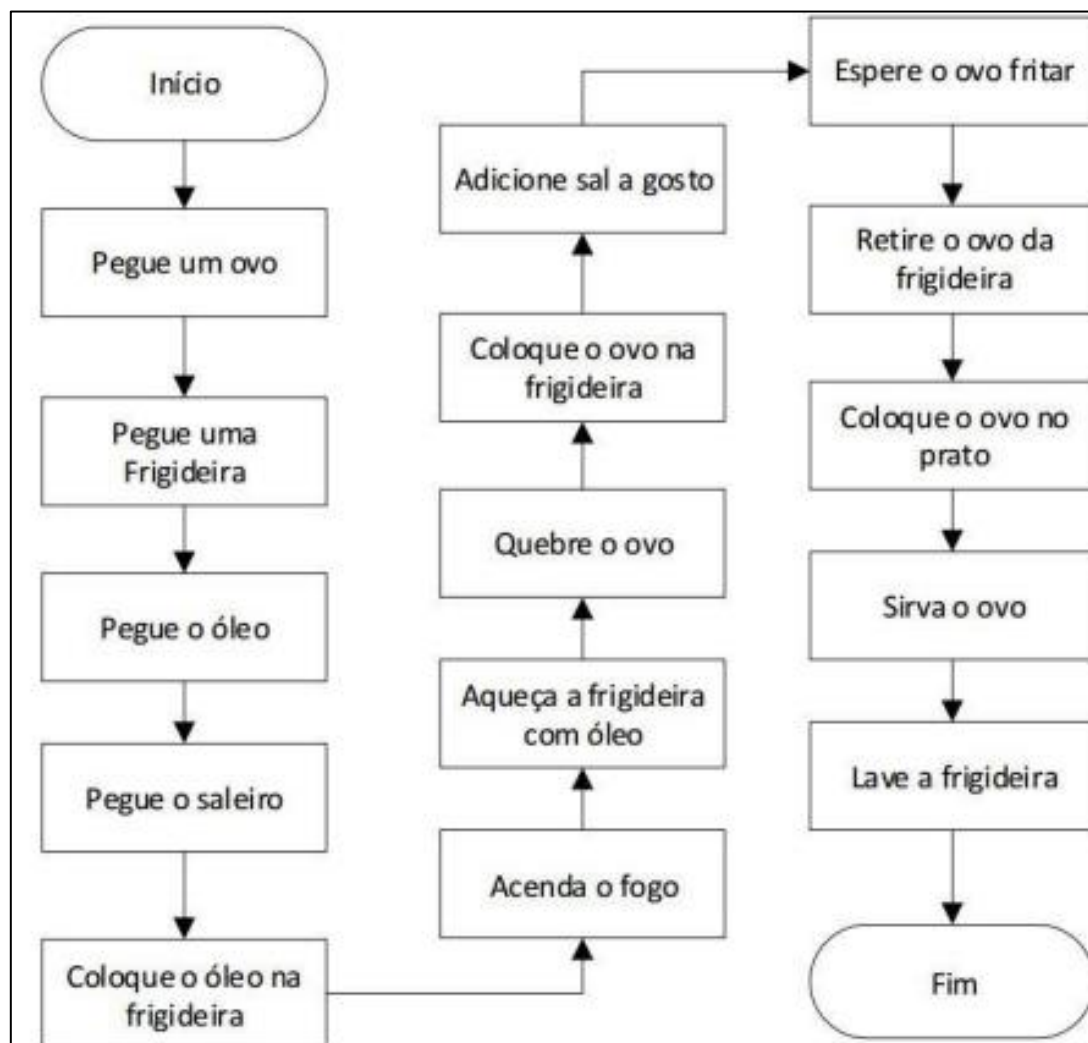
#### **Algoritmo para fritar um ovo**

- 1 - Pegue um ovo;
- 2 - Pegue uma frigideira;
- 3 - Pegue o óleo;
- 4 - Pegue o saleiro;
- 5 - Coloque o óleo na frigideira;
- 6 - Acenda o fogo;
- 7 - Aqueça a frigideira com óleo;
- 8 - Quebre o ovo;
- 9 - Coloque o ovo na frigideira;
- 10 - Adicione sal a gosto;
- 11 - Espere o ovo fritar;
- 12 - Retire o ovo da frigideira;
- 13 - Coloque o ovo no prato;
- 14 - Sirva o ovo;
- 15 - Lave a frigideira.

#### **Fluxograma Fritar um Ovo**



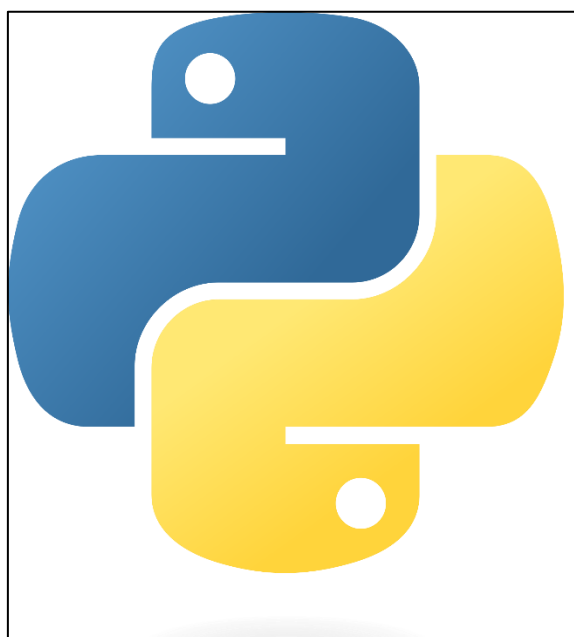
Figura 8 - Fluxograma Fritar um Ovo



## 2. PYTHON

Python é uma linguagem de programação de alto nível, interpretada e de propósito geral, conhecida por sua sintaxe clara e legibilidade. Criada por Guido van Rossum e lançada pela primeira vez em 1991, Python rapidamente ganhou popularidade devido à sua simplicidade e versatilidade. Sua filosofia de "leitura de código é mais importante do que a escrita" e o princípio do "zen do Python" destacam a importância da clareza e da facilidade de uso.

**Figura 9 - Logotipo Python**



Atualmente, Python é amplamente utilizado em diversas áreas da tecnologia. Sua biblioteca padrão abrangente e a grande variedade de pacotes de terceiros facilitam o desenvolvimento em áreas como desenvolvimento web, automação, análise de dados, ciência de dados, aprendizado de máquina e inteligência artificial (IA). A facilidade de aprendizado e a comunidade ativa contribuem para sua adoção em todos os níveis, desde iniciantes até profissionais experientes.

No contexto da Inteligência Artificial, Python emergiu como uma linguagem líder. A comunidade de IA adotou Python devido à sua flexibilidade e ao suporte robusto para bibliotecas específicas, como TensorFlow, PyTorch e





Scikit-Learn. Essas bibliotecas oferecem ferramentas poderosas para construção e treinamento de modelos de machine learning, redes neurais e algoritmos de IA. A clareza sintática do Python também facilita a implementação de algoritmos complexos, acelerando o desenvolvimento de soluções inovadoras em IA.

A aplicação de Python em projetos de IA estende-se a várias áreas, incluindo reconhecimento de imagem, processamento de linguagem natural, visão computacional e automação de tarefas inteligentes. Sua integração eficiente com hardware especializado, como GPUs, impulsiona o desempenho de algoritmos de treinamento em larga escala. A popularidade do Python no campo da IA é evidenciada por sua presença em conferências acadêmicas, projetos de código aberto e ambientes de pesquisa avançada.

Em resumo, Python é uma linguagem de programação multifuncional que desempenha um papel proeminente no cenário tecnológico contemporâneo. Sua aplicação na Inteligência Artificial é particularmente notável, proporcionando uma base sólida para a criação, implementação e desenvolvimento de soluções inovadoras que impulsionam o avanço da IA nos dias de hoje.

## **2.1. Como aprender Python**

Para iniciar o aprendizado da linguagem Python, é recomendável começar com uma abordagem estruturada e passos graduais. Em primeiro lugar, familiarize-se com a sintaxe básica da linguagem, que é notável por sua simplicidade e legibilidade. Recursos online, como tutoriais interativos, cursos online e documentação oficial, são excelentes fontes para adquirir os conceitos fundamentais.

Um passo crucial é a prática ativa. Comece resolvendo pequenos problemas e desafios de programação para aplicar os conceitos aprendidos. Plataformas como HackerRank, Codecademy e exercícios disponíveis em livros de programação Python são ótimos para desenvolver as habilidades práticas. Escrever código regularmente é essencial para ganhar confiança e fluência na linguagem.



Além disso, participar de comunidades online, como fóruns, grupos no Reddit ou redes sociais dedicadas a Python, permite interação com outros aprendizes e profissionais. Essa troca de conhecimentos e experiências pode oferecer insights valiosos, dicas práticas e soluções para desafios específicos.

À medida que avança, considere a exploração de projetos pequenos e práticos. Criar scripts simples, desenvolver programas básicos e contribuir para projetos de código aberto são maneiras eficazes de consolidar o aprendizado. À medida que a confiança aumenta, é possível explorar tópicos mais avançados, como desenvolvimento web, ciência de dados ou IA, com base nos interesses pessoais.

Em resumo, começar a aprender Python envolve uma combinação de estudo teórico, prática constante e participação em comunidades online. Essa abordagem progressiva permitirá que os iniciantes construam uma base sólida e desenvolvam as habilidades necessárias para explorar as inúmeras aplicações da linguagem Python no vasto mundo da programação e tecnologia.

## 2.2. Como usar o Python

O **Google Colab**, um serviço gratuito baseado em nuvem, oferece um ambiente de desenvolvimento interativo para Python, especialmente voltado para aprendizado de máquina e análise de dados. Ao iniciar no Colab, é crucial entender alguns comandos essenciais para se familiarizar com a plataforma.

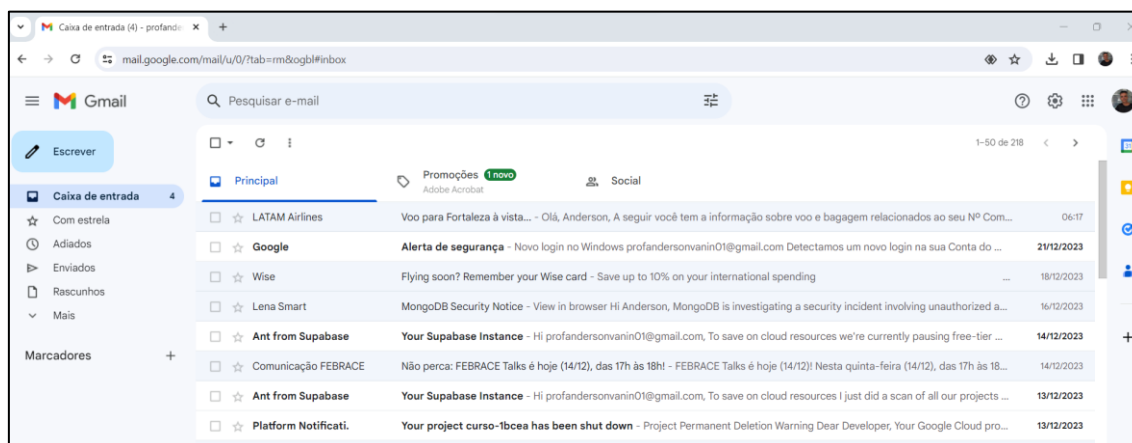
Figura 10 - Logotipo Google Colab



Antes de iniciar, certifique-se de ter uma conta no google que tenha espaço de armazenamento disponível. Caso não tenha uma conta, crie-a para que você possa acompanhar os próximos tópicos e ter um registro em arquivos de todo o seu aprendizado.

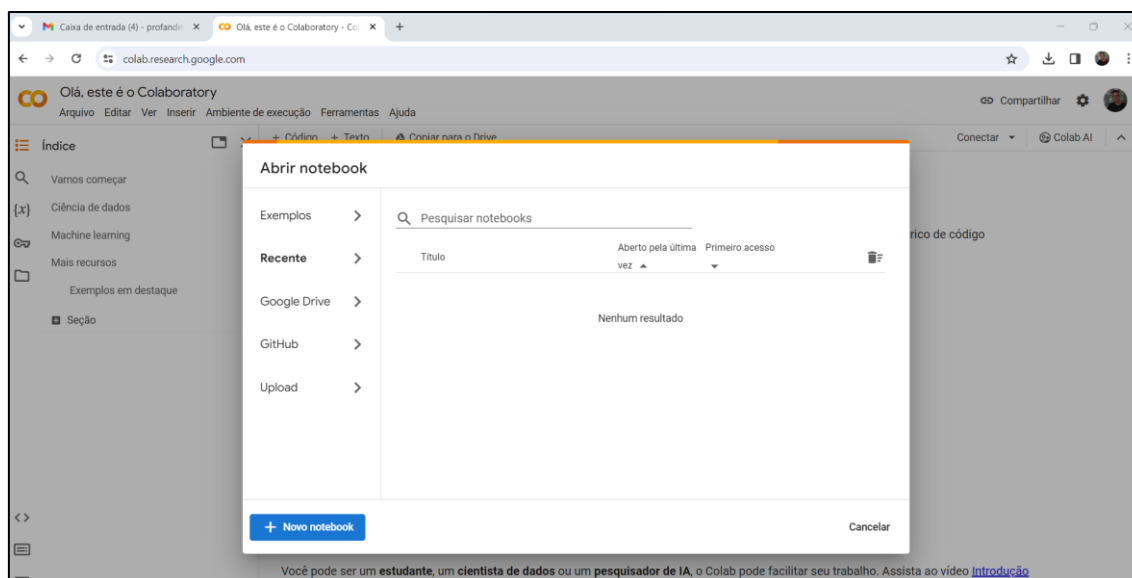
Inicie fazendo login em sua conta de e-mail do google, por exemplo.

**Figura 11 - Entrando em sua conta do Google**



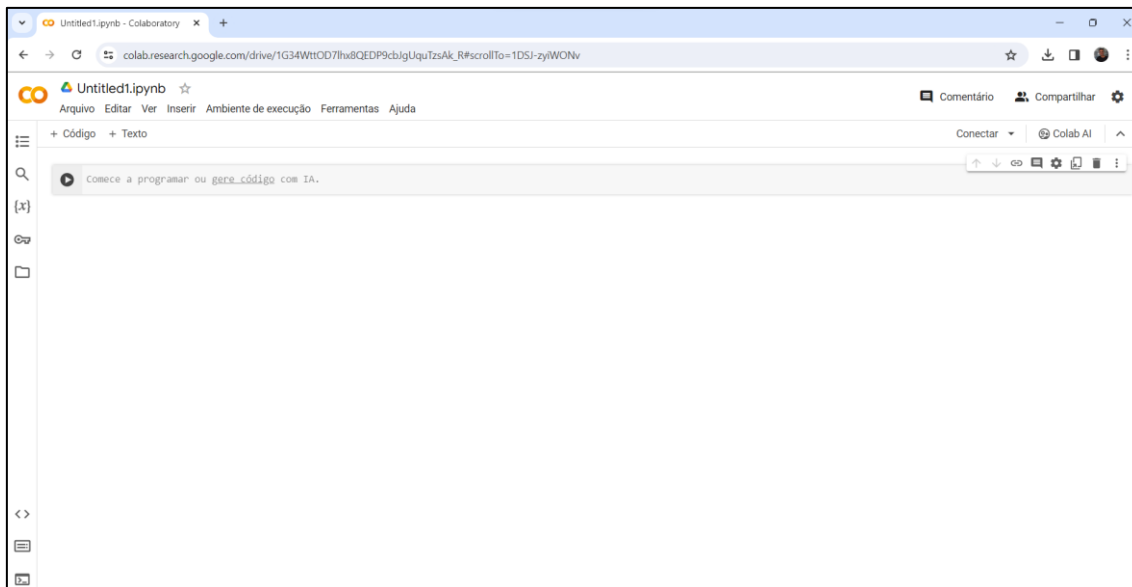
Abra uma nova guia em seu navegador e digite <https://colab.research.google.com/>.

**Figura 12 - Acessando o Google Colab**



Agora o primeiro passo é abrir um novo **notebook** no Colab. Para isso, acesse o Google Colab através do navegador e clique em "Novo Notebook". Isso criará um ambiente de programação Python.

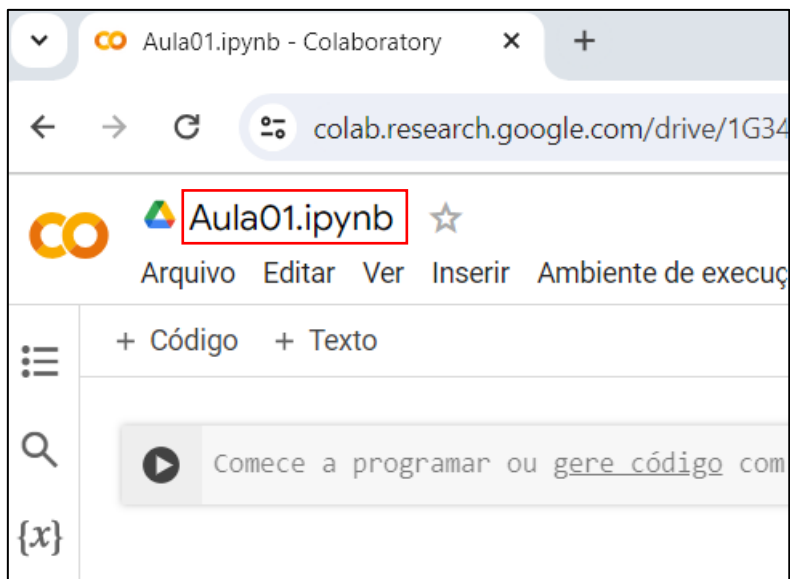
**Figura 13 - Ambiente Colab**



Inicialmente, vamos dar um nome para o nosso arquivo de modo a organizarmos melhor nossos exercícios daqui para adiante. Clique sobre o nome que aparece *Untitled1.ipynb*, onde **Untitled1** é o nome do arquivo que iremos alterar e **ipynb** é a extensão do arquivo que é executada no Google Colab. Altere o arquivo para **Aula01.ipynb**.

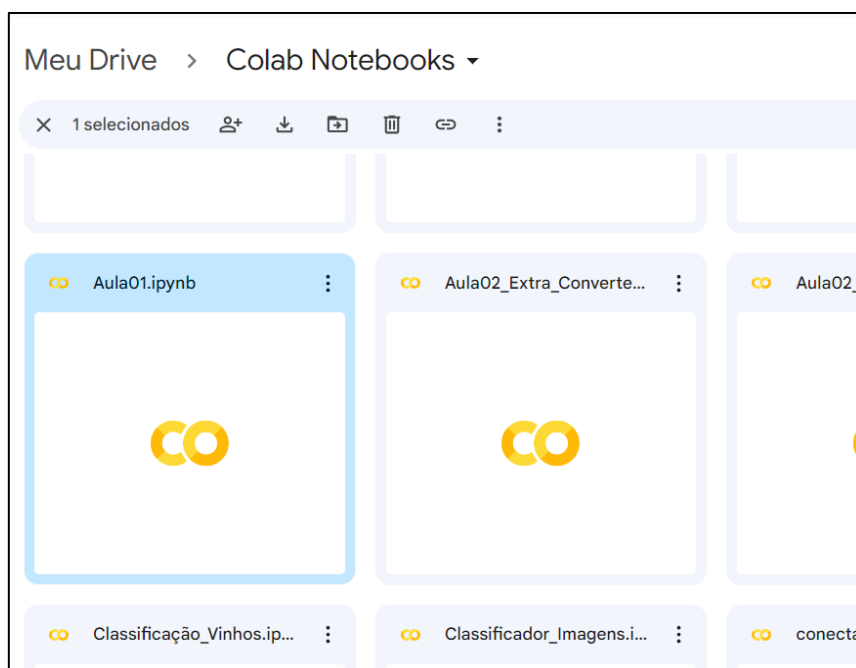


**Figura 14 - Renomeando o arquivo**



Agora o arquivo já está salvo automaticamente dentro da pasta Notebooks em seu Drive do Google.

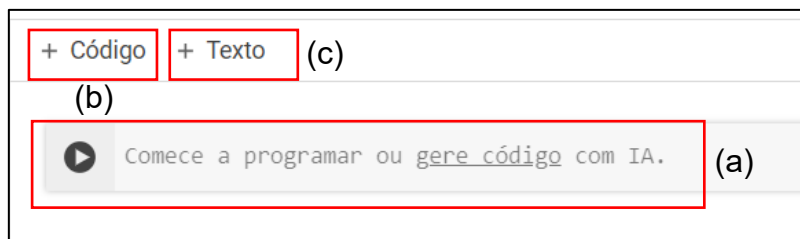
**Figura 15 - Visualizando o arquivo salvo dentro da pasta Colab Notebooks no Drive do Google**



Dentro do notebook, você encontrará células (a), unidades de código (b) ou texto (c) que podem ser executadas individualmente, ilustradas na figura 16.

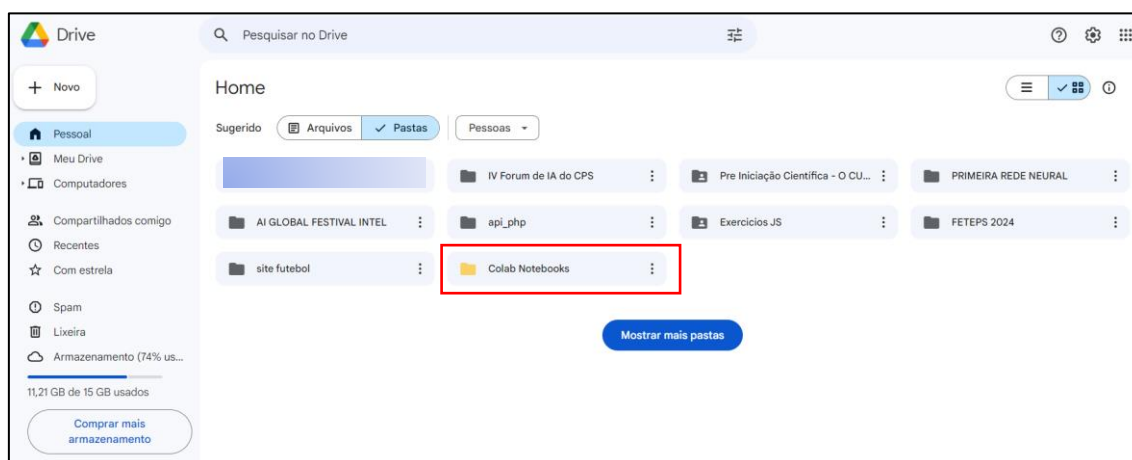


**Figura 16 - Células, unidades de código e texto dentro do Colab**



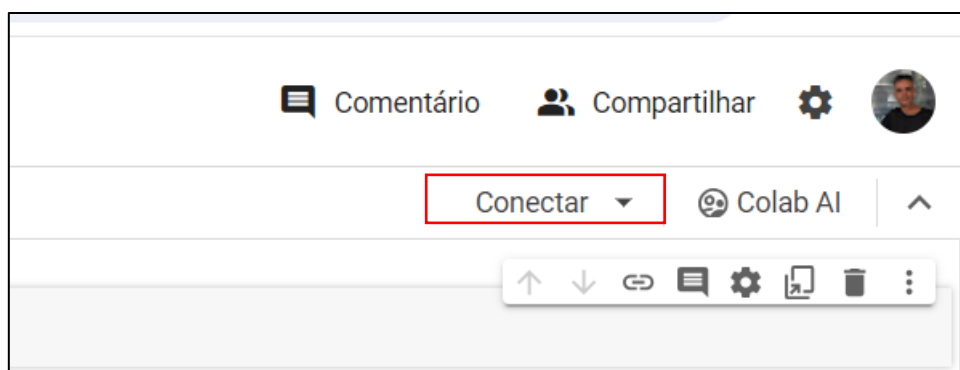
Todo novo notebook criado, ficará salvo em seu drive do Google. Se você não especificar nenhuma pasta específica dentro de seu drive, o local padrão de salvamento de seus arquivos será uma pasta chamada *Colab Notebooks*.

**Figura 17 - Pasta padrão Colab Notebooks dentro do Drive do Google**



Antes de começar, precisamos conectar o arquivo ao ambiente de execução do Google para podermos utilizar a máquina virtual do google. Clique em conectar.

**Figura 18 - Conectando o ambiente de execução**



Aguarde alguns segundos até que o ambiente esteja conectado e pronto para execuções de códigos em ambiente Python.

**Figura 19 - Ambiente conectado. Mostrando ícones de uso de Memória RAM e uso de Disco**



Comece inserindo um comando simples de impressão, utilizando o comando ***print()***.

O `print()` é uma função do Python utilizada para imprimir alguma mensagem na tela. Uma mensagem deve estar delimitada entre aspas simples (") ou duplas (").

Por exemplo, digite ***print("Olá, Mundo!")*** e execute a célula para ver a saída.

**Figura 20 - Executando primeiro comando print**



Neste exemplo, utilizamos uma célula de código e vimos como executá-la. Um outro recurso muito importante do ambiente, é a possibilidade de trabalhar com células ou blocos de Texto, que podem servir para fazer a documentação e explicação de códigos, intercalando-as com células de códigos.

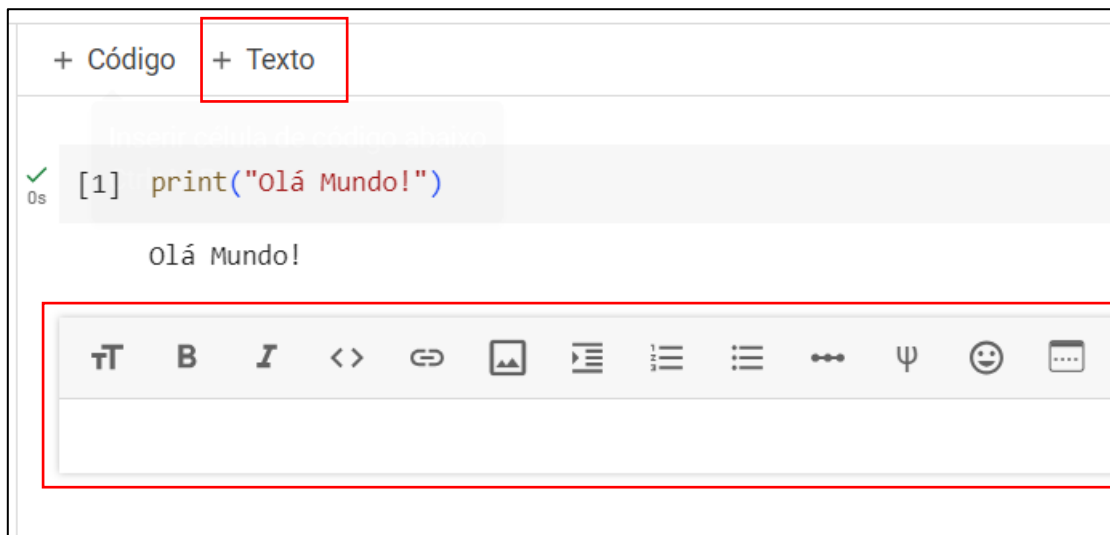
Ao inserir uma célula de texto ou de código, é possível movê-la para cima ou para baixo de outras células para melhor organizar seu notebook.



Vamos inserir um bloco de texto e em seguida reorganizá-lo para que ele ocupe a primeira parte de nosso notebook.

Clique em **+ Texto** para inserir um novo bloco de texto.

**Figura 21 - Inserindo um novo bloco de texto**



Após finalizar a digitação do texto clique na seta para definir a nova posição deste bloco.

**Figura 22 - Seta para movimentação do bloco**

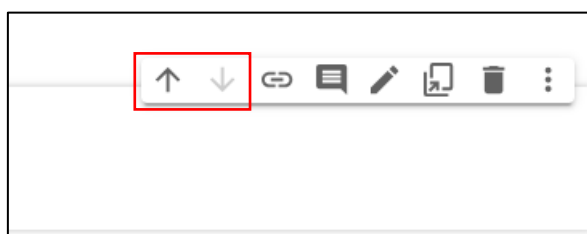
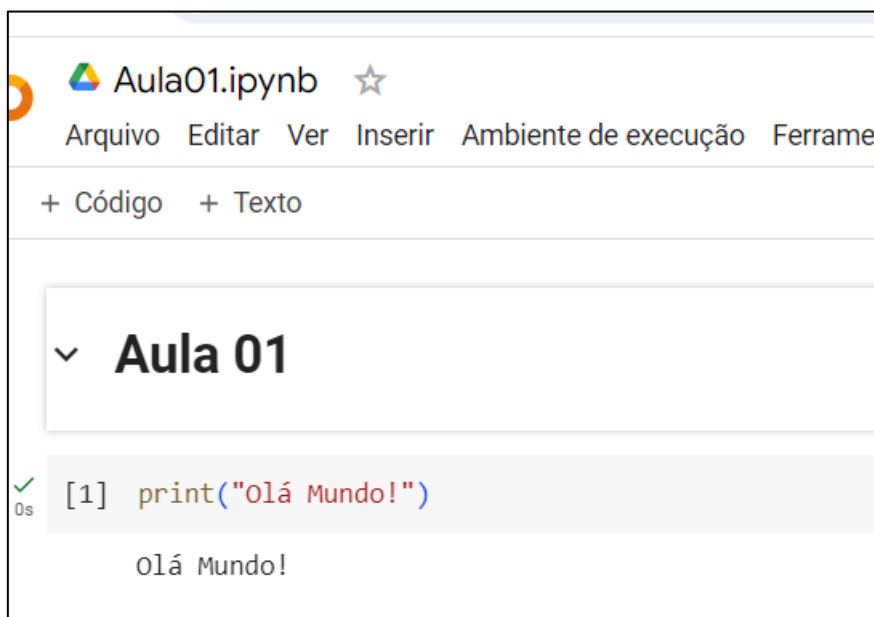




Figura 23 - Bloco de Texto posicionado



**Nota:** O Google Colab permite vários tipos de formatação de textos e imagens que podem ser inseridas neste bloco, porém neste guia de formação básica em Python, não iremos abordar este tipo de formatação que pode ser encontrada como: **Formatação Markdown**.

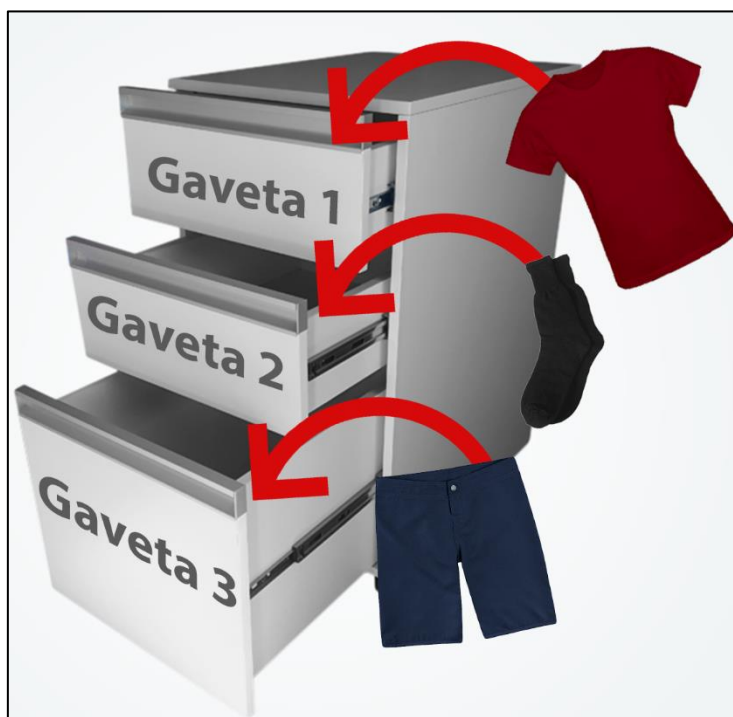
No próximo capítulo, começaremos a abordar o uso de variáveis na linguagem Python.



### 3. VARIÁVEIS

Em linguagem de programação, uma variável é um local de armazenamento com um nome simbólico (um identificador) e um valor associado ou informações. Ela é utilizada para representar e manipular dados na execução de um programa. O conceito de variável permite que programadores atribuam valores a um nome, tornando mais fácil referenciar e manipular esses valores ao longo do código.

**Figura 24 - Comparando variáveis à gavetas de um móvel.**



Cada variável tem um tipo de dado associado, indicando o tipo de informação que ela pode armazenar, como números inteiros, decimais, caracteres, entre outros. A capacidade de atribuir valores diferentes a uma variável durante a execução do programa confere flexibilidade e dinamismo ao código.

Na maioria das linguagens de programação, a criação de uma variável envolve a declaração do tipo de dado que ela irá armazenar, seguida pelo nome da variável e, opcionalmente, a atribuição de um valor inicial. Por exemplo, em



Python, a criação de uma variável que armazena um número inteiro pode ser feita da seguinte forma:

```
idade = 25
```

Nesse exemplo, **idade** é o nome da variável, e **25** é o valor atribuído a ela. Posteriormente, o valor da variável pode ser atualizado, reatribuindo um novo valor a ela:

```
idade = 26
```

Assim, as variáveis são elementos cruciais na construção de algoritmos e programas, proporcionando uma maneira eficiente de lidar com dados e informações durante a execução de um programa.

Variáveis em Python são elementos fundamentais na construção de algoritmos e programas, proporcionando flexibilidade e dinamismo ao armazenar e manipular dados. Nessa linguagem de programação de alto nível, a variedade de tipos de variáveis reflete-se em sua capacidade de lidar com diferentes formas de informação, tornando Python uma escolha poderosa e versátil para desenvolvedores.

Em Python, a declaração de variáveis é simples e intuitiva. Desde os tipos básicos, como inteiros e decimais, até estruturas mais complexas, como listas e dicionários, a linguagem oferece um leque diversificado de opções para a manipulação eficiente de dados. Os inteiros (*int*), que representam números inteiros sem casas decimais, e os números de ponto flutuante (*float*), que incluem casas decimais, são exemplos de tipos numéricos amplamente utilizados em Python.

Além disso, Python abrange *strings* (*str*), que são sequências de caracteres, e booleanos (*bool*), que indicam verdadeiro ou falso. A flexibilidade é ainda mais evidente na manipulação de estruturas de dados como listas (*list*) e tuplas (*tuple*), proporcionando armazenamento ordenado e imutável, respectivamente. Já os dicionários (*dict*) possibilitam a criação de mapeamentos

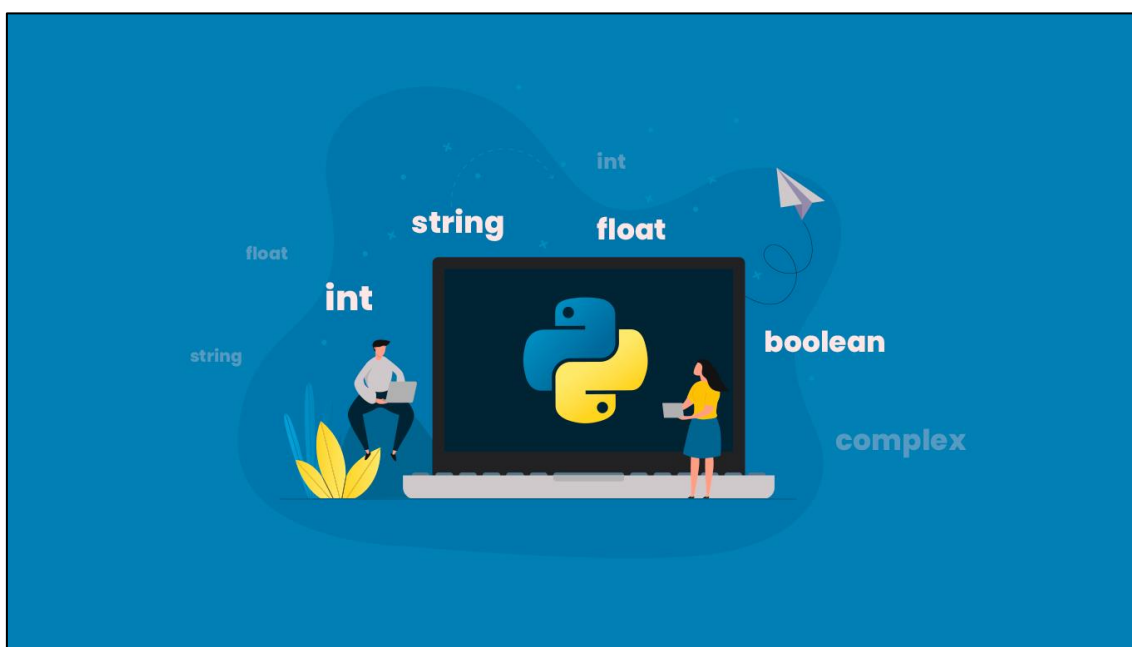


entre chaves e valores, enriquecendo a capacidade de organização e recuperação de dados.

A introdução a variáveis em Python não só apresenta um leque diversificado de tipos de dados, mas também enfatiza a simplicidade da sintaxe. Isso facilita a compreensão e o desenvolvimento de código, especialmente para iniciantes na programação. Em resumo, explorar as variáveis em Python é embarcar em uma jornada pela riqueza de tipos de dados, evidenciando o compromisso da linguagem em proporcionar uma experiência de programação eficiente e abrangente.

### 3.1. Principais tipos de variáveis em Python

Figura 25 - Tipos de Variáveis



Em Python, as variáveis não precisam ter um tipo específico definido previamente, como acontece em outras linguagens de programação. Isso significa que podemos atribuir valores de diferentes tipos a uma mesma variável ao longo do programa. Por exemplo, podemos criar uma variável chamada “nome” e atribuir a ela uma *string*:

```
nome = "João"
```

Mais tarde, podemos atribuir um número à mesma variável:



```
nome = 30
```

No entanto, é importante ter cuidado ao utilizar variáveis em Python para evitar erros de lógica ou confusões. É recomendado utilizar nomes de variáveis que sejam descritivos e facilmente compreensíveis, para facilitar a leitura e manutenção do código.

Em Python, existem diversos tipos de variáveis que podem ser utilizados para armazenar diferentes tipos de dados.

Cada tipo de variável em Python possui características específicas que atendem a diferentes necessidades de programação. Ao utilizar esses tipos de variáveis de maneira apropriada, os programadores podem expressar de forma eficiente uma ampla gama de dados e estruturas em seus programas.

A seguir, apresentaremos os principais tipos de variáveis em Python e como declará-las.

### 3.1.1. Variáveis numéricas

#### a) Inteiro (int):

Utilizado para armazenar números inteiros, como 1, 2, -3, etc. Para declarar uma variável inteira, basta atribuir um valor numérico a ela, sem a necessidade de especificar o tipo. Exemplo:

```
idade = 25
```

```
numero_telefone = 123456789
```

#### b) Ponto Flutuante (float):

Utilizado para armazenar números decimais, como 3.14, 2.5, etc. Para declarar uma variável do tipo float, é necessário adicionar um ponto decimal ao valor. Exemplo:

```
altura = 1.75
```

```
preco_produto = 29.99
```



### 3.1.2. Variáveis de texto

#### a) String (str)

Utilizado para armazenar sequências de caracteres, como palavras, frases etc. Para declarar uma variável do tipo string, é necessário utilizar aspas simples ou duplas. Exemplo:

```
nome = "Alice"
```

```
mensagem = 'Olá, Python!'
```

### 3.1.3. Variáveis lógicas

#### a) Booleano (bool)

Utilizado para armazenar valores lógicos, como verdadeiro (True) ou falso (False). Para declarar uma variável do tipo booleano, basta atribuir o valor True ou False. Exemplo:

```
esta_chovendo = False
```

```
aprovado = True
```

### 3.1.4. Lista (list)

Lista é uma coleção de valores indexada, em que cada valor é identificado por um índice. O primeiro item na lista está no índice 0, o segundo no índice 1 e assim por diante.

Uma lista representa uma coleção ordenada e mutável de itens.

Para criar uma lista com elementos deve-se usar colchetes e adicionar os itens entre eles separados por vírgula. Exemplo:

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

```
nomes = ['Ana', 'Bob', 'Carlos']
```

### 3.1.5. Tupla (tuple)



Tupla é uma estrutura de dados semelhante a lista. Porém, ela tem a característica de ser imutável, ou seja, após uma tupla ser criada, ela não pode ser alterada. Exemplo:

```
coordenadas = (10, 20)
```

```
informacoes_pessoais = ('Maria', 25, 'Feminino')
```

### 3.1.6. Dicionário (dict)

Os dicionários representam coleções de dados que contém na sua estrutura um conjunto de pares chave/valor, nos quais cada chave individual tem um valor associado. Esse objeto representa a ideia de um mapa, que entendemos como uma coleção associativa desordenada. A associação nos dicionários é feita por meio de uma chave que faz referência a um valor. Exemplo:

```
aluno = {"nome": "Carlos", "idade": 22, "curso": "Ciência da  
Computação"}
```

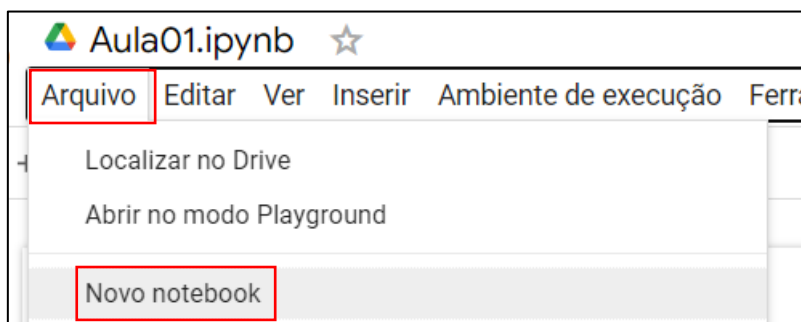
```
endereco = {"rua": "Avenida Principal", "cidade": "Cidade  
Exemplo", "cep": "12345-678"}
```

Pode parecer um pouco confuso neste início, mas não se preocupe, vamos ver em detalhes cada um destes tipos listados acima. Para começar, vamos ver como utilizar variáveis numéricas e executar algumas operações simples.

Vamos começar a entender o funcionamento do uso de variáveis em um novo documento. Inicie um novo documento do Colab e renomeie-o para Aula02.



**Figura 26 - Criando um novo notebook e renomeando-o**



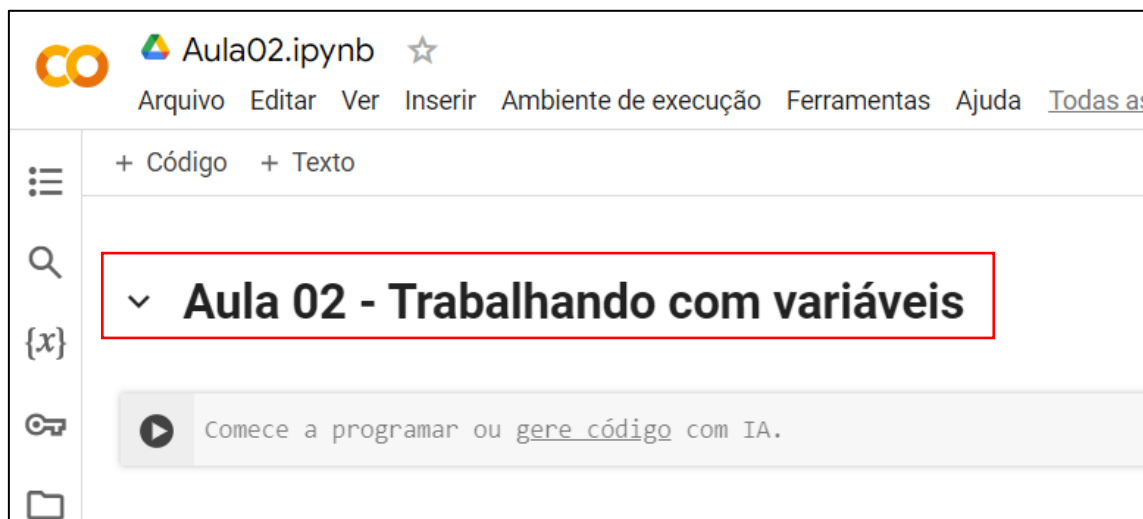
Faça a alteração do novo do arquivo em clique em Conectar

**Figura 27 - Ativando o notebook: conectar**



Coloque uma célula de texto no início do arquivo para deixar mais bem documentado seu exercício.

**Figura 28 - Célula de Texto**

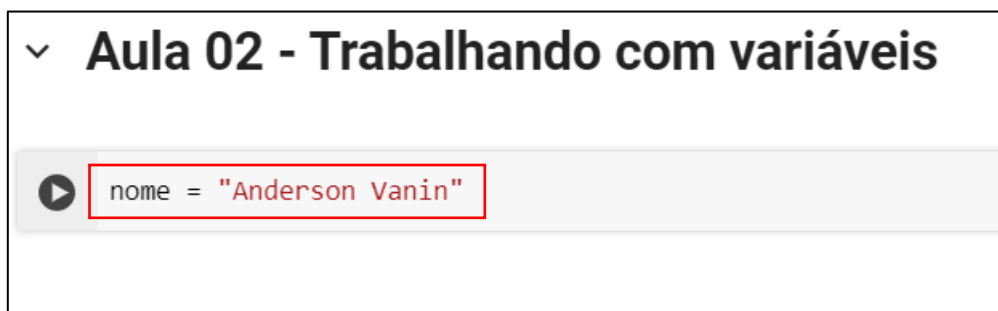


Neste primeiro exemplo, vamos criar uma variável que irá armazenar em memória o seu nome.





Figura 29 - Criando uma variável




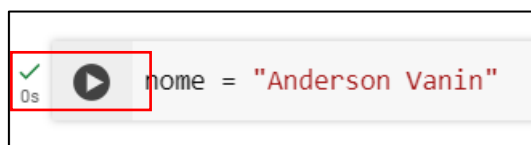
Esse bloco de instruções foi criado, porém ainda não foi executado. Clique no botão  para executar e gravar as informações.

Figura 30 - Executando uma célula



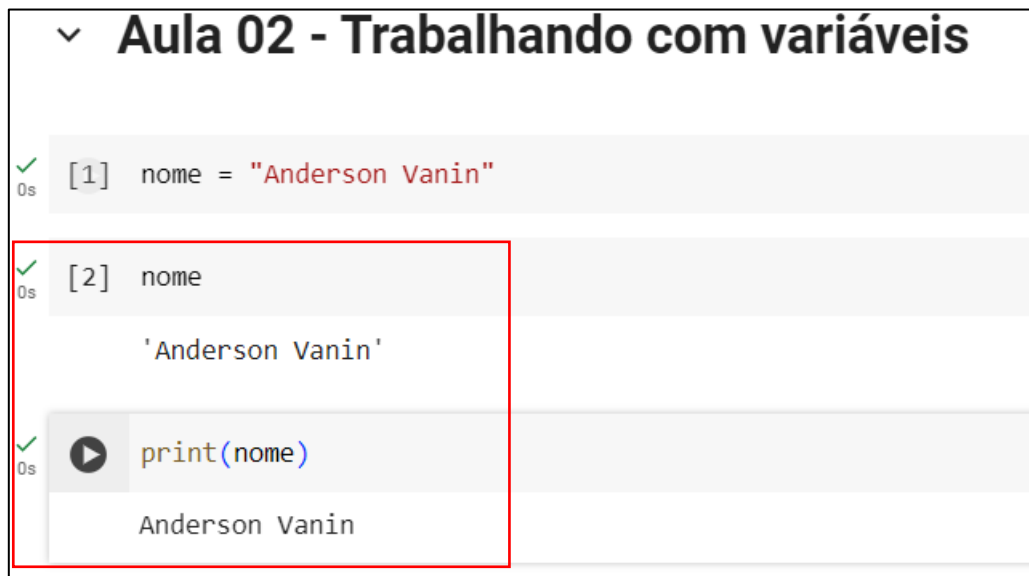
Quando um bloco de células é executado você poderá observar ao lado esquerdo um ticket de ok confirmando a execução daquela célula.

Agora o valor está armazenado na variável nome e pode ser chamado a qualquer momento dentro deste documento.

Para visualizar o conteúdo de uma variável simples como essa, você pode simplesmente, em outra célula, digitar somente o nome da variável ou colocá-la dentro um comando print. Veja abaixo as duas maneiras:



Figura 31 - Comando print



The screenshot shows a code editor window titled "Aula 02 - Trabalhando com variáveis". It contains three lines of Python code, each preceded by a green checkmark and a "0s" execution time indicator. The first line is `[1] nome = "Anderson Vanin"`. The second line is `[2] nome`, followed by the string `'Anderson Vanin'` on the next line. The third line is `[3] print(nome)`, followed by the output `Anderson Vanin` on the next line. A red rectangular box highlights the second and third lines of code.

```
✓ 0s [1] nome = "Anderson Vanin"
✓ 0s [2] nome
      'Anderson Vanin'
✓ 0s [3] print(nome)
      Anderson Vanin
```

**Importante:**

Se durante o seu programa você atribuir um outro valor para a mesma variável, o conteúdo anterior será substituído pelo novo e a informação antiga será perdida!

Exemplo:



Figura 32 - Substituindo o valor de uma variável

✓  
0s

[1] nome = "Anderson Vanin"

✓  
0s

[2] nome  
  
'Anderson Vanin'

✓  
0s

[3] print(nome)  
  
Anderson Vanin

✓  
0s

[4] nome = "Cíntia Pinho"

✓  
0s

▶

 nome  
  
'Cíntia Pinho'

Como no ambiente Colab as células podem ser executadas individualmente, experimente agora clicar para mostrar o nome nas células acima.



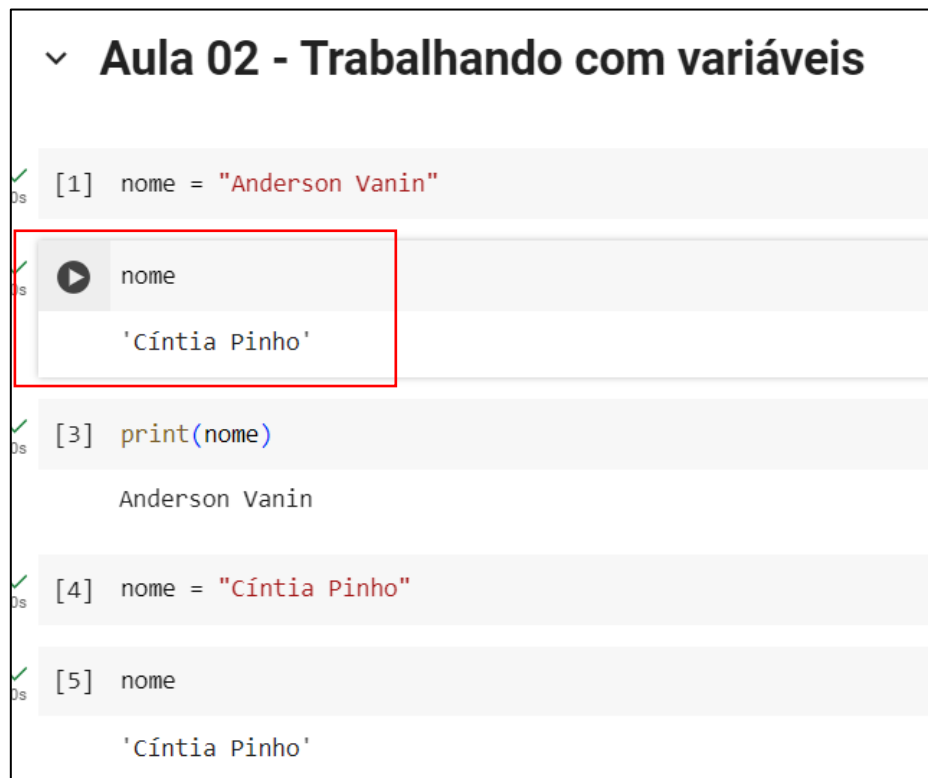
Figura 33 - Células executadas.



Observe que agora o valor da variável `nome`, foi alterado para a última execução.



Figura 34 - Verificando o novo valor de uma variável



```
▼ Aula 02 - Trabalhando com variáveis

[1] nome = "Anderson Vanin"

[2] nome
'Cíntia Pinho'

[3] print(nome)
Anderson Vanin

[4] nome = "Cíntia Pinho"

[5] nome
'Cíntia Pinho'
```

Isso acontece em todas as linguagens de programação não sendo exclusividade do Python, pois uma variável só pode armazenar um valor por vez, mas, no entanto, pode ser substituído a qualquer momento durante a execução do programa.

Vamos ver agora, como é o comportamento do Python com os outros tipos de variáveis.



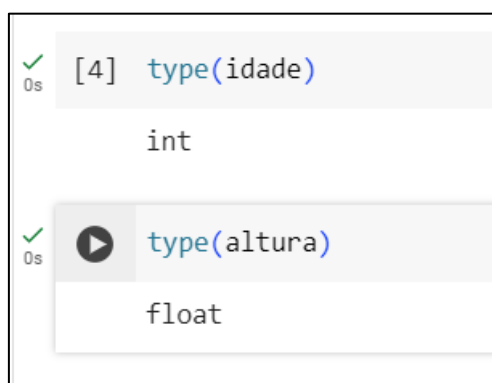
Figura 35 - Outros tipos de variáveis



Veja que no exemplo anterior, você não precisa colocar a exibição de uma variável em um bloco diferente. É possível definir e solicitar o valor de uma variável dentro do mesmo bloco de execução.

O Python oferece uma função em que é possível verificar que tipo de variável estamos trabalhando. Para isso podemos utilizar a função `type()`.

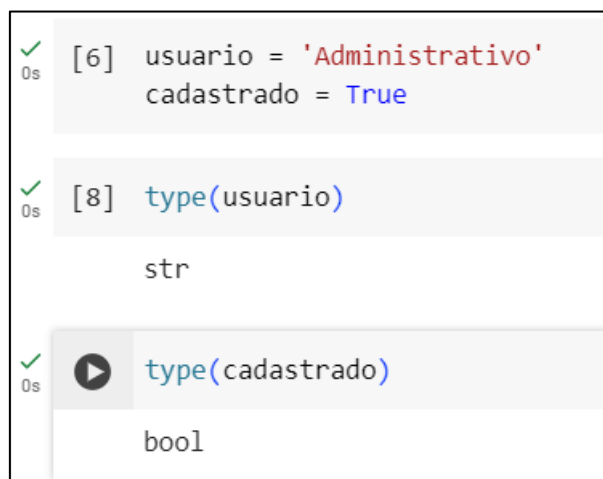
Figura 36 - Verificando o tipo de uma variável



Vamos ver mais alguns exemplos:



Figura 37 - Outros exemplos



```
✓ [6] usuario = 'Administrativo'
0s      cadastrado = True

✓ [8] type(usuario)
0s      str

✓ [9] type(cadastrado)
0s      bool
```

### 3.2. Nomes de Variáveis

Programadores escolhem nomes para variáveis que sejam semânticos e que ao mesmo tempo documentem o código. Esses nomes podem ser bem longos, podem conter letras e números. É uma convenção entre os programadores Python começar uma variável com letras minúsculas e utilizar underscore (`_`) para separar palavras como: ***meu\_nome***, ***numero\_de\_cadastro***, ***tel\_residencial***.

No entanto existem algumas regras importantes que devem ser observadas ao criar uma variável, como por exemplo, as variáveis **não podem**:

- Começar com um número.
- Conter caracteres especiais (`!`, `@`, `#`, `$`, `&`, etc.).
- Variáveis não podem ter nomes reservados, como por exemplo nomes de funções internas como `print`. Em Python existem pelo menos 33 palavras reservadas.



Figura 38 - Palavras reservadas em Python

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	def
for	lambda	return		



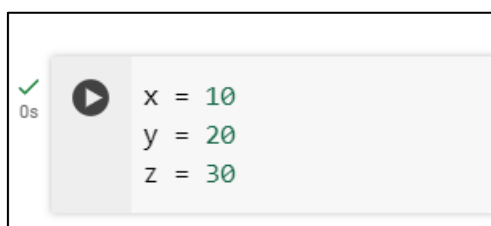


## 4. OPERADORES ARITMÉTICOS

Operadores são símbolos especiais que representam cálculos como adições e multiplicações. Para fazer cálculos com números utilizamos os operadores +, -, \*, / e \*\* que apresentam, respectivamente, adição, subtração, multiplicação, divisão e potenciação.

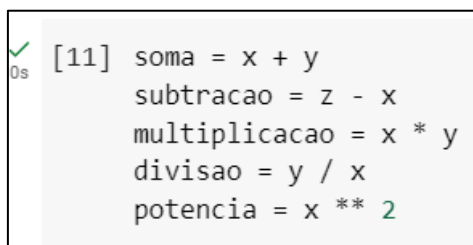
Vamos criar algumas variáveis e atribuir alguns valores para exemplificar o uso dos operadores aritméticos.

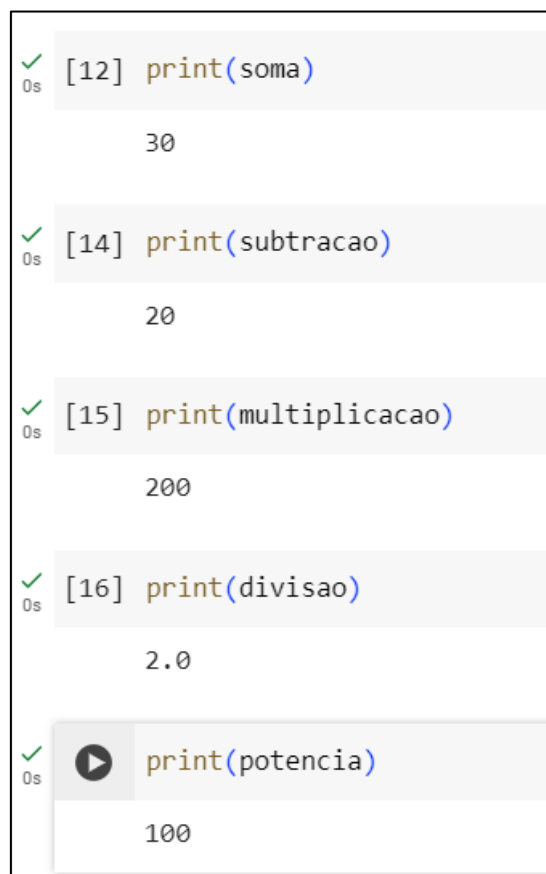
**Figura 39 - Variáveis para testes**



A execução de uma operação aritmética pode ser atribuída diretamente à uma nova variável.

**Figura 40 - Operações Aritméticas com variáveis**



**Figura 41 - Resultados das operações aritméticas**

```
[12] print(soma)
30

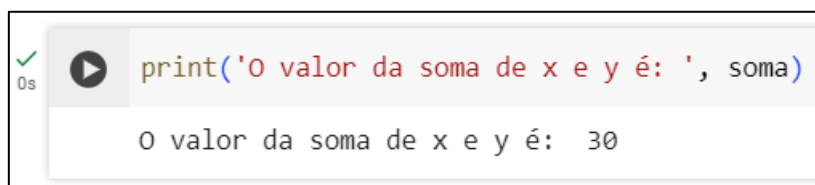
[14] print(subtracao)
20

[15] print(multiplicacao)
200

[16] print(divisao)
2.0

[17] print(potencia)
100
```

Para melhorar a visualização de um resultado utilizando o comando ***print()***, podemos concatenar um texto explicativo junto com o valor desejado.

**Figura 42 - Exibição de resultados com o comando print**

```
print('O valor da soma de x e y é: ', soma)

O valor da soma de x e y é: 30
```

Os principais operadores são:



Figura 43 - Lista dos principais operadores aritméticos

Operação	Nome	Descrição
$a + b$	adição	Soma entre $a$ e $b$
$a - b$	subtração	Diferença entre $a$ e $b$
$a * b$	multiplicação	Produto entre $a$ e $b$
$a / b$	divisão	Divisão entre $a$ e $b$
$a // b$	divisão inteira	Divisão inteira entre $a$ e $b$
$a \% b$	módulo	Resto da divisão entre $a$ e $b$
$a ** b$	exponenciação	$a$ elevado a potência de $b$

Agora um pequeno exemplo mais prático. Digamos que temos armazenado em uma variável o ano de nascimento de uma pessoa e em outra variável o ano atual. Dessa forma queremos calcular a idade desta pessoa.

Figura 44 - Calculando a idade de uma pessoa

**Exemplo: Calcular idade**

✓  
0s

```
[19] ano_nasc = 1974
      ano_atual = 2024
```

✓  
0s

```
[20] idade = ano_atual - ano_nasc
```

✓  
0s

```
print('Sua idade é: ',idade)
```

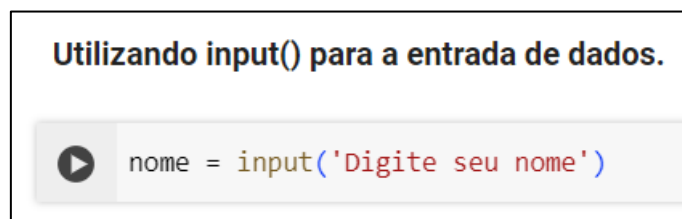
Sua idade é: 50

**Nota:**

Nem sempre os valores de variáveis estarão atribuídos diretamente a elas. Em diversas ocasiões precisaremos solicitar ao usuário que entre com um valor desejado para que a operação seja executada. Neste caso utilizamos a função *input()*.

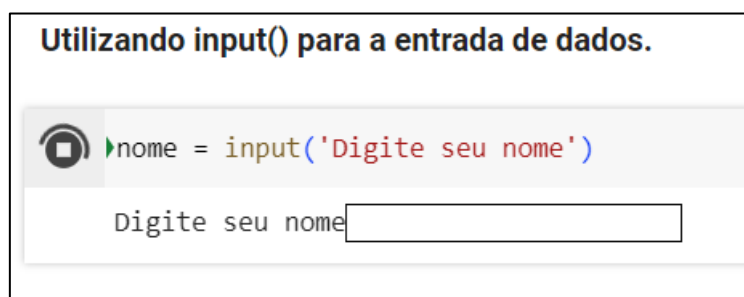


**Figura 45 - Comando input para a entrada de dados**



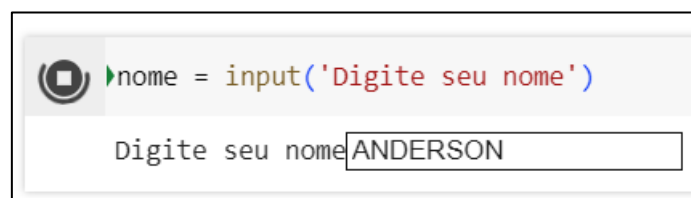
Ao clicar para executar este bloco, será solicitado que o usuário digite um valor que será atribuído à variável **nome**.

**Figura 46 - Execução da célula com o comando input**

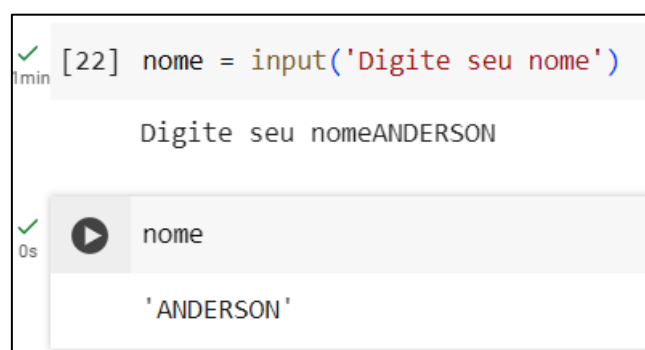


Digitando um valor e teclando Enter, o conteúdo da caixa de texto será atribuído para a variável.

**Figura 47 - Execução da célula com input**



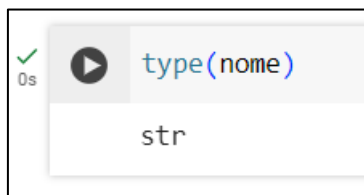
**Figura 48 - Resultado da operação após o input**



Experimente testar o tipo da variável nome.



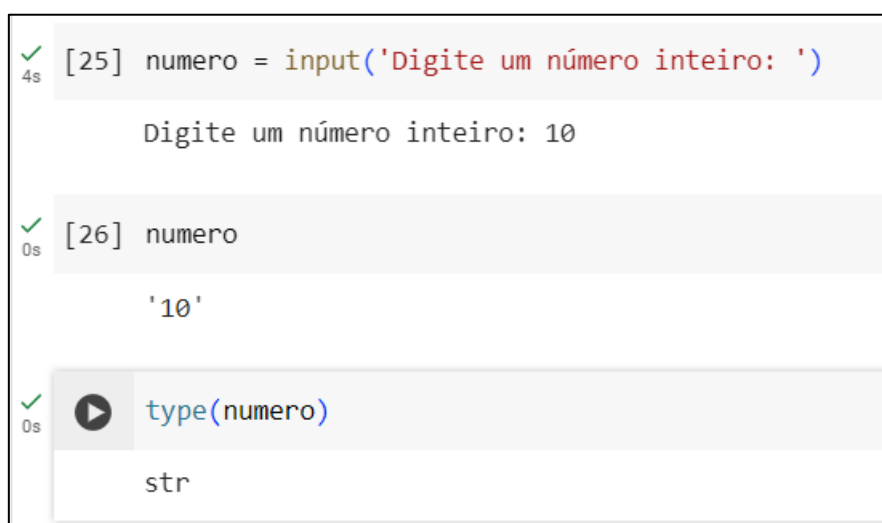
Figura 49 - Testando o tipo da variável



Como esperado, o tipo de conteúdo armazenado na variável `nome` é uma String (texto).

Agora experimente armazenar um número inteiro em uma variável.

Figura 50 - Entrando com um número inteiro e verificando o seu tipo na execução da célula



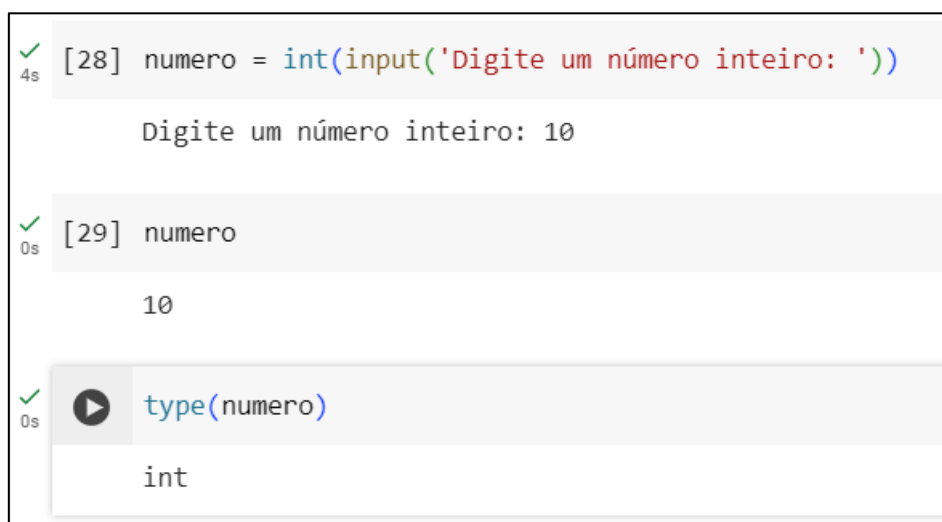
Nesse caso teríamos um grande problema se precisássemos utilizar este número digitado em uma operação aritmética. Observe que o tipo apresentado pelo Python é uma String (texto), assim sendo não é possível realizar operações aritméticas com tipos do tipo String. Isso ocorre porque a função `input()` irá gerar uma caixa de texto para digitação e tudo que for digitado nela, independentemente se for texto ou só um número, será convertido para texto (str).

Para podermos trabalhar com números vindos da digitação em uma caixa de texto, precisamos convertê-lo para um tipo desejado (int, float, etc).



Utilizando o exemplo anterior, vamos utilizar a função `int()`, que faz a conversão para um tipo inteiro.

**Figura 51 - Convertendo uma entrada pelo comando `input` para um número inteiro**



```
[28] numero = int(input('Digite um número inteiro: '))  
      Digite um número inteiro: 10  
[29] numero  
      10  
type(numero)  
      int
```

Agora com o tipo certo, podemos utilizar o valor armazenado na variável **numero** para realizar operações aritméticas.

#### 4.1. Exercícios

- Altere o exemplo da idade, solicitando ao usuário que digite seu nome, o seu ano de nascimento e o ano atual. Crie uma expressão aritmética que irá calcular e mostrar na tela a idade do usuário com os dados fornecidos.
- Solicite ao usuário dois números que ele irá digitar. Apresente a Soma, Subtração, Divisão e Multiplicação destes dois números.
- Solicite a largura e altura de um retângulo. Calcule a área deste retângulo. Lembre-se que a área de um retângulo é dada por:  $\text{área} = \text{largura} * \text{altura}$ .



## 5. OPERADORES RELACIONAIS OU DE COMPARAÇÃO

Python possui também operadores relacionais (de comparação), como os da matemática:  $<$ ,  $>$ ,  $<=$ ,  $>=$ ,  $==$ ,  $!=$ .

Figura 52 - Lista de Operadores Relacionais

Operação	Descrição
$a == b$	$a$ igual a $b$
$a != b$	$a$ diferente de $b$
$a < b$	$a$ menor do que $b$
$a > b$	$a$ maior do que $b$
$a <= b$	$a$ menor ou igual a $b$
$a >= b$	$a$ maior ou igual a $b$

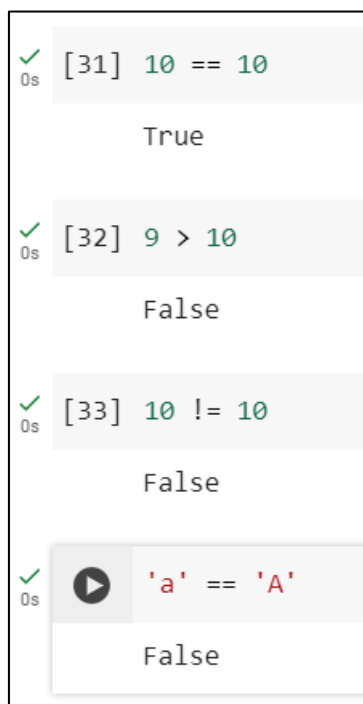
Expressões relacionais são aquelas que realizam uma comparação entre duas expressões e retornam

- **False**, se o resultado é falso
- **True**, se o resultado é verdadeiro.

Exemplos:



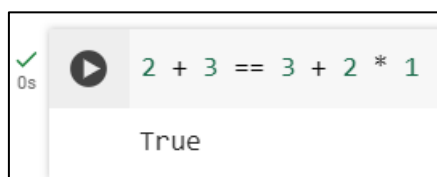
Figura 53 - Exemplos de uso de operadores relacionais



### 5.1. Expressões relacionais

Assim como dizemos que as expressões aritméticas são reduzidas a um valor numérico inteiro ou real, as expressões relacionais são reduzidas a um valor booleano (ou seja, **True** ou **False**). As expressões relacionais podem conter expressões aritméticas, como no seguinte exemplo em Python:

Figura 54 - Exemplo de uma expressão relacional



### 5.2. Precedência de Operações

As regras de precedência indicam qual operador é calculado primeiro. Por exemplo, qual o resultado da expressão **2 - 3 \* 4**?

Como a multiplicação tem maior prioridade que a subtração, o produto **3\*4** é reduzido ao valor **12** e a seguir se calcula o valor da subtração **2 - 12**, resultando em **-10**.





As regras de associatividade indicam a ordem dos cálculos para operadores que tenham a mesma precedência. Por exemplo, qual o resultado da expressão  $2 - 3 + 4$ ?

Como a soma tem a mesma prioridade que a subtração, precisamos aplicar a regra de associatividade. Em Python, a maioria dos operadores binários (que usam dois operandos) tem associatividade “da esquerda para a direita” (ou seja, as operações são realizadas na mesma ordem de leitura). A expressão, portanto, é primeiramente reduzida a  $-1 + 4$  resolvendo a subtração e depois reduzida ao valor  $3$  resolvendo a soma. Observe que, caso a soma  $3+4$  fosse calculada primeiro, o resultado final seria  $-5 = 2-(3+4)$ .

A tabela a seguir mostra a associatividade das principais operações aritméticas em Python, em ordem decrescente de precedência (da maior para a menor):

**Figura 55 - Precedência de operações**

Operador	descrição	Associatividade
()	parênteses	da esquerda para a direita
**	potência	da direita para a esquerda
+, -	positivo e negativo unário	da direita para a esquerda
*, /, //, %	multiplicação, divisão, divisão inteira e resto	da esquerda para a direita
+, -	soma e subtração	da esquerda para a direita

#### Observações:

- **Níveis de precedência:**
  - A tabela mostra que há grupos de operadores com o mesmo nível, como soma e subtração.
  - Quanto mais “alto” o nível na tabela, maior a precedência.
- **Operadores com mesmo nível de precedência são resolvidos segundo a sua associatividade.**



- exemplo: para reduzir a expressão **12 / 2 / 3 / 4**, o Python calcula **12/2**, e segue dividindo o resultado por **3** e depois por **4**.
- **Os operadores unários (+ e -) tornam explícitos o sinal do operando.**
  - experimente colocar uma sequência de operadores unários como **+-+3** para ver o que acontece (será que resulta em erro de sintaxe?).
- **Use parênteses caso deseje alterar a precedência ou torná-la explícita**
  - exemplo: **12 / 2 / 3 / 4 => 0.5**
  - exemplo: **(12 / 2) / ( 3 / 4) => 8.0**



## 6. OPERADORES LÓGICOS

O próximo tipo de operador da linguagem Python que estudaremos são os operadores lógicos. Os mais importantes são: **and**, **not**, e **or**.

A tabela abaixo resume o funcionamento dos operadores lógicos em Python:

Figura 56 - Tabela Verdade dos Operadores Lógicos

AND		
exp da esquerda	exp da direita	resultado
True	True	True
True	False	False
False	True	False
False	False	False

OR		
exp da esquerda	exp da direita	resultado
True	True	True
True	False	True
False	True	True
False	False	False

NOT	
exp	resultado
True	False
False	True

Operadores lógicos são ferramentas fundamentais na programação utilizadas para realizar operações de comparação entre valores ou expressões, resultando em um valor lógico de verdadeiro (True) ou falso (False). Eles são comumente usados em estruturas de controle de fluxo, como condicionais e loops, para tomar decisões baseadas em condições.

Os operadores lógicos mais comuns são:



- **AND (E):** Retorna True se ambas as condições forem verdadeiras.
- **OR (OU):** Retorna True se pelo menos uma das condições for verdadeira.
- **NOT (NÃO):** Inverte o valor lógico da expressão.

Aqui estão alguns exemplos de como esses operadores são utilizados na linguagem Python:

Figura 57 - Exemplo de uso de operadores lógicos

```
x = True
y = False

# Operação AND
resultado_and = x and y
print(resultado_and) # Saída: False

# Operação OR
resultado_or = x or y
print(resultado_or) # Saída: True

# Operação NOT
resultado_not_x = not x
print(resultado_not_x) # Saída: False
```

### 6.1. Exemplo de uso de operadores lógicos

Vamos considerar um exemplo de verificação de elegibilidade com base em critérios específicos, sem utilizar expressões condicionais, listas, tuplas ou dicionários.

**Exercício:** Verificar se um estudante é elegível para participar de um programa acadêmico. Suponha que um programa acadêmico tenha os seguintes critérios de elegibilidade:

- O estudante deve ter uma média mínima de 7.0.



- O estudante não pode ter nenhuma falta durante o semestre.

Figura 58 - Exemplo de uso de operadores lógicos

```
# Dados do estudante
media = 8.5
faltas = 0

# Critérios de elegibilidade
media_minima = 7.0
faltas_maximas = 0

# Verificando a elegibilidade do estudante usando operadores lógicos
elegivel = media >= media_minima and faltas <= faltas_maximas

# Exibindo o resultado
print("O estudante é elegível para participar do programa acadêmico?", elegivel)
```

Neste exemplo, utilizamos operadores lógicos para verificar se o estudante atende aos critérios de elegibilidade. A expressão `media >= media_minima` verifica se a média do estudante é maior ou igual à média mínima exigida, enquanto `faltas <= faltas_maximas` verifica se o número de faltas do estudante é menor ou igual ao número máximo de faltas permitidas. Se ambas as condições forem verdadeiras, o estudante é considerado elegível para participar do programa acadêmico.

## 6.2. Sua vez

Tente executar todos os exemplos mostrados neste capítulo no Colab. Modifique os valores e veja os novos resultados obtidos.

## 6.3. Exercícios

Sendo `v1 = 15`, `v2 = 10`, `v3 = 5`, `v4 = 0`, defina qual será a resposta (**VERDADEIRO OU FALSO**) para as sentenças lógicas abaixo.

a) `(v1 = 10) E (v2 = 10)`

b) `(v1 = 15) E (v2 = 10)`

c) `(v1 = 15) E (v3 = 10)`



d)  $(v_2 > 5) \text{ OU } (v_3 < 10)$

e)  $(v_4 = 1) \text{ OU } (v_3 = 4)$

f)  $(v_2 = 10) \text{ OU } (v_4 = 5)$

g)  $(v_1 > 10 \text{ E } v_2 < 15) \text{ E } (v_3 < 10 \text{ E } v_4 = 0)$

h)  $(v_1 < 10 \text{ E } v_2 > 15) \text{ OU } (v_3 > 5 \text{ OU } v_4 = 0)$



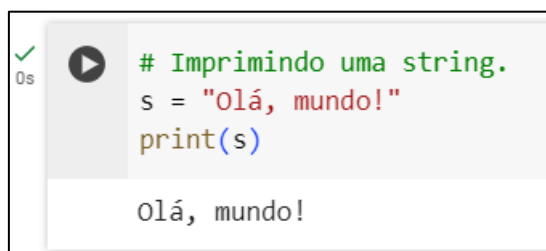
## 7. MANIPULAÇÃO DE STRINGS

Um tipo de dados bastante usado no dia a dia são as *strings*, ou cadeias de caracteres (ou sequências de caracteres). O tipo de dados *string*, ou **str** como é chamado em Python, possui várias operações úteis associadas a ele. Essas operações tornam Python uma linguagem bastante propícia para manipulação de textos.

Os exemplos abaixo ilustram o tipo de dados *string* e as operações associadas a ele.

- Imprimindo uma String

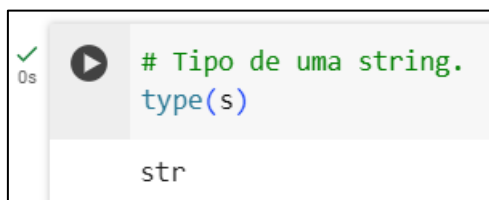
Figura 59 - Imprimindo uma String



```
✓ 0s # Imprimindo uma string.  
s = "Olá, mundo!"  
print(s)  
  
Olá, mundo!
```

- Obtendo o tipo de uma variável

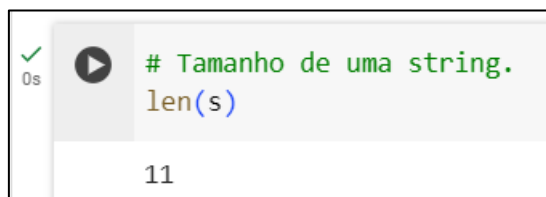
Figura 60 - Obtendo o tipo de uma variável string



```
✓ 0s # Tipo de uma string.  
type(s)  
  
str
```

- Tamanho de uma string

Figura 61 - Tamanho de uma string



```
✓ 0s # Tamanho de uma string.  
len(s)  
  
11
```

- Concatenação



Figura 62 - Concatenação de Strings

```
✓ [4] # Concatenação
0s print("Meu Brasil " + "brasileiro")

Meu Brasil brasileiro

✓ 0s ▶ a = 'Pequenos'
b = ' '
c = 'Talentos'
print(a+b+c)

Pequenos Talentos
```

- Substitui uma *substring* por alguma outra coisa

Figura 63 - Substrings

```
✓ 0s ▶ # Substitui uma substring por alguma outra coisa.
print(s)
s1 = s.replace("mundo", "meu abacate")

print(s1)

Olá, mundo!
Olá, meu abacate!
```

- A *string* **s** começa com "Olá"?

Figura 64 - Encontrando uma parte de uma string

```
✓ 0s ▶ # A string s começa com "Olá"?
print(s.startswith("Olá"))

True
```

- A *string* **s** termina com "mundo"?





**Figura 65 - Encontrando um último caractere de uma string**

```

✓ 0s # A string s termina com "mundo"?
print(s)
print(s.endswith("mundo"))

Olá, mundo!
False

```

- Quantas ocorrências da palavra "abacate" a string s1 possui?

**Figura 66 - Quantidade de ocorrências de uma string**

```

✓ 0s # Quantas ocorrências da palavra "abacate" a string s1 possui?
print(s1)
print(s1.count("abacate"))

Olá, meu abacate!
1

```

**Figura 67 - Exemplo para ocorrências de uma string**

```

✓ 0s txt = "I love apples, apple are my favorite fruit"
print(txt)
x = txt.count("apple")
print(x)

I love apples, apple are my favorite fruit
2

```

Strings em Python são muito flexíveis e nos permitem executar as mais diversas operações de maneira simples. Vejamos mais exemplos:

- Transformar a primeira letra da primeira palavra em maiúscula

**Figura 68 - Capitalize**

```

✓ 0s # Como "capitalizar" (transformar a primeira letra da primeira palavra em maiúscula).
s = "ordem e progresso"
print(s.capitalize())

Ordem e progresso

```

- Como verificar se uma string só possui números



Figura 69 - Verificando se uma string possui somente números

```

✓ 0s # Como verificar se uma string só possui números.
primeiro = '12345'
segundo = '12345abc'
print(primeiro.isdigit())
print(segundo.isdigit())

True
False

```

- Como verificar se uma string é alfanumérica (só possui letras e números)

Figura 70 - Verificando se uma string é alfanumérica

```

✓ 0s # Como verificar se uma string é alfanumérica (só possui letras e números).
print(segundo)
print(segundo.isalnum())

12345abc
True

```

### 7.1. Substrings em Python (Slicing)

Além das operações vistas acima, podemos acessar caracteres específicos de uma *string* em Python usando a notação `[ ]`. Neste esquema de acesso a caracteres de uma *string*, o primeiro caractere está no índice 0, o segundo no índice 1, e assim por diante, conforme ilustrado no exemplo abaixo.

Figura 71 - Slicing em strings

```

✓ 0s s = "Olá, mundo!"
print(s[0])
print(s[2])
print(s[6])

O
á
u

```

No caso da *string* "Olá, mundo!", temos a seguinte associação entre índices e caracteres:

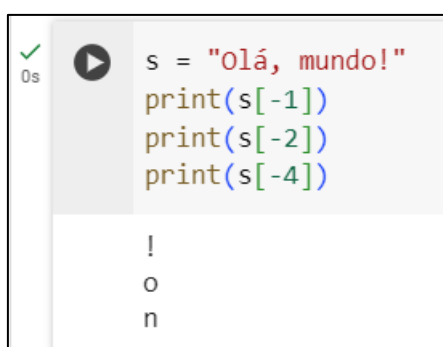


Figura 72 - Índices de uma string

O	l	á	,		m	u	n	d	o	!
0	1	2	3	4	5	6	7	8	9	10

Podemos também acessar os elementos em ordem reversa usando índices negativos. Neste esquema, o último caractere de uma *string* está no índice  $-1$ , o penúltimo no índice  $-2$ , e assim por diante, como mostrado no exemplo abaixo.

Figura 73 - Exibindo índices específicos de uma string



```

✓ 0s ▶ s = "Olá, mundo!"
      print(s[-1])
      print(s[-2])
      print(s[-4])
    
```

!

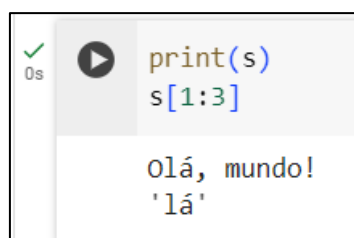
o

n

Podemos também acessar fatias ou "*slices*" de uma *string* ou lista em Python. Esta notação é muito concisa e poderosa, então é importante que você a entenda bem.

Segundo essa notação, uma fatia de uma *string*, ou seja, uma *substring*, pode ser acessada se fornecermos os índices do começo e do final da fatia que desejamos analisar, como mostrado abaixo:

Figura 74 - Fatiando uma string



```

✓ 0s ▶ print(s)
      s[1:3]
    
```

Olá, mundo!

'lá'

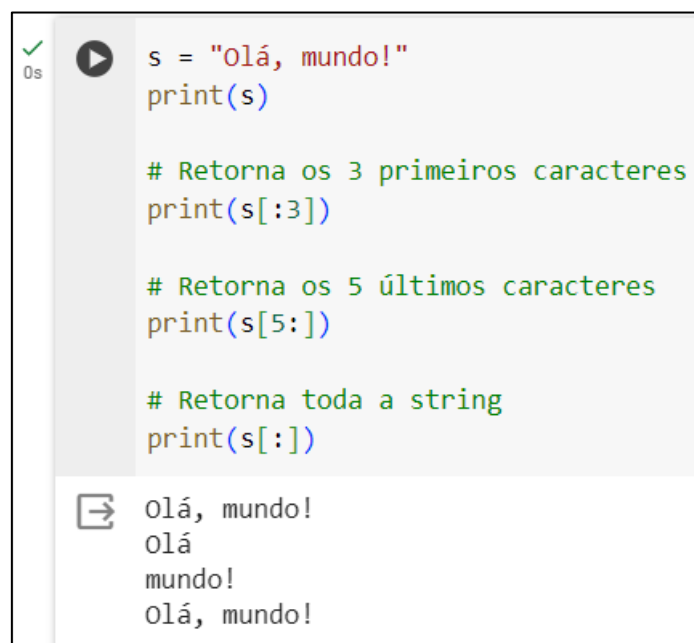
Note que, como mencionamos anteriormente, os índices de uma string começam do 0 e não do 1.



Além disso, perceba que o índice do final da fatia não é incluído nela. No exemplo acima, o [1:3] nos retornou dois caracteres e não três. Foram retornados o caractere no índice 1 e o caractere no índice 2, mas não o caractere no índice 3.

Se omitirmos o índice de início da fatia ou o de final (ou ambos), o início e o final da *string* serão considerados, respectivamente. Veja os exemplos:

**Figura 75 - Outros exemplos de fatias de strings**



```

✓ 0s ▶ s = "Olá, mundo!"
      print(s)

      # Retorna os 3 primeiros caracteres
      print(s[:3])

      # Retorna os 5 últimos caracteres
      print(s[5:])

      # Retorna toda a string
      print(s[:])
  
```

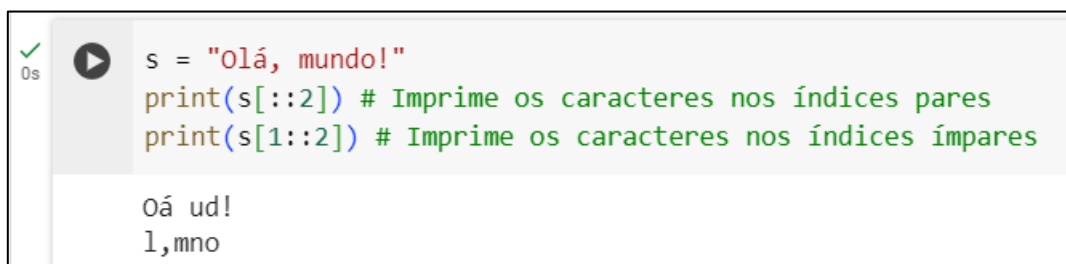
Olá, mundo!  
Olá  
mundo!  
Olá, mundo!

É possível ainda especificar um parâmetro que indica quantos caracteres devem ser processados de cada vez. Por exemplo, se quisermos imprimir somente os caracteres nos índices pares ou ímpares de uma *string*, podemos fazer assim:

O	l	á	,		m	u	n	d	o	!
0	1	2	3	4	5	6	7	8	9	10



Figura 76 - Fatiando strings



```

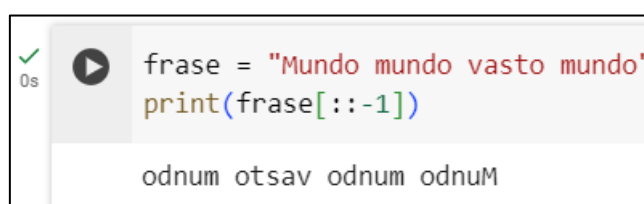
✓ 0s ▶ s = "Olá, mundo!"
      print(s[::2]) # Imprime os caracteres nos índices pares
      print(s[1::2]) # Imprime os caracteres nos índices ímpares

Oá ud!
l,mno

```

Um outro exemplo útil do uso da técnica de *slicing* para manipulação de *strings* é inverter uma palavra ou frase usando somente operações de *slicing*:

Figura 77 - Invertendo os índices de uma string



```

✓ 0s ▶ frase = "Mundo mundo vasto mundo"
      print(frase[::-1])

odnum otsav odnum odnuM

```

No exemplo acima, usamos um terceiro parâmetro do recurso de *slicing* para indicar que retornamos toda a frase (os `::`) e logo em seguida dizemos que faremos isso de trás para frente (por meio do `-1` no final). Mais especificamente, o `-1` indica que estamos saltando um caractere de cada vez, começando de trás para frente (o que é feito por meio do sinal de menos).

Então, para resumir, a sintaxe de *slicing* de *strings* é a seguinte **[início:fim:salto]**, onde:

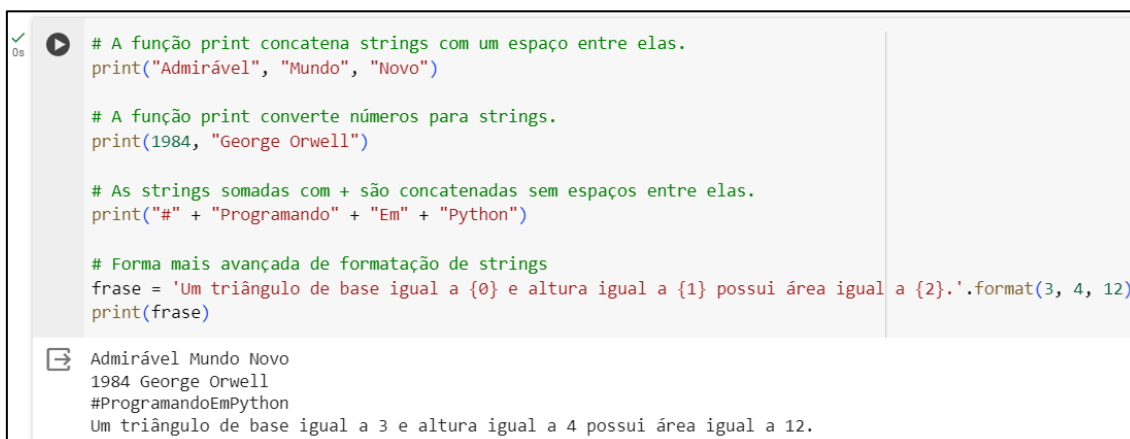
- início é o primeiro índice a ser considerado (o primeiro caractere da *string* é considerado caso este valor seja omitido);
- fim - 1 é o último índice a ser considerado (o último caractere da *string* é considerado caso este valor seja omitido); e
- salto indica quantos caracteres devem ser saltados em cada etapa (o valor 1 é considerado por padrão, e um sinal de menos deve ser usado para percorrer a *string* em ordem reversa).



## 7.2. Formatação de strings em Python

Os exemplos abaixo mostram as várias opções de formatação de *strings*.

Figura 78 - Formatação de strings em Python



```
✓ 0s # A função print concatena strings com um espaço entre elas.
print("Admirável", "Mundo", "Novo")

# A função print converte números para strings.
print(1984, "George Orwell")

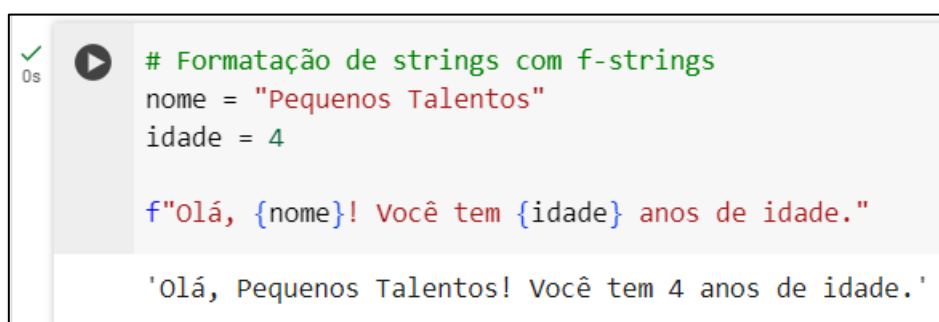
# As strings somadas com + são concatenadas sem espaços entre elas.
print("#" + "Programando" + "Em" + "Python")

# Forma mais avançada de formatação de strings
frase = 'Um triângulo de base igual a {0} e altura igual a {1} possui área igual a {2}.'.format(3, 4, 12)
print(frase)
```

Admirável Mundo Novo  
1984 George Orwell  
#ProgramandoEmPython  
Um triângulo de base igual a 3 e altura igual a 4 possui área igual a 12.

As versões mais novas do Python introduziram um recurso chamado **f-strings**, que melhoram ainda mais a formatação de *strings*. Vejamos um exemplo:

Figura 79 - Formatação com f-strings



```
✓ 0s # Formatação de strings com f-strings
nome = "Pequenos Talentos"
idade = 4

f"Olá, {nome}! Você tem {idade} anos de idade."

'Olá, Pequenos Talentos! Você tem 4 anos de idade.'
```



## 8. LISTAS, TUPLAS E DICIONÁRIOS

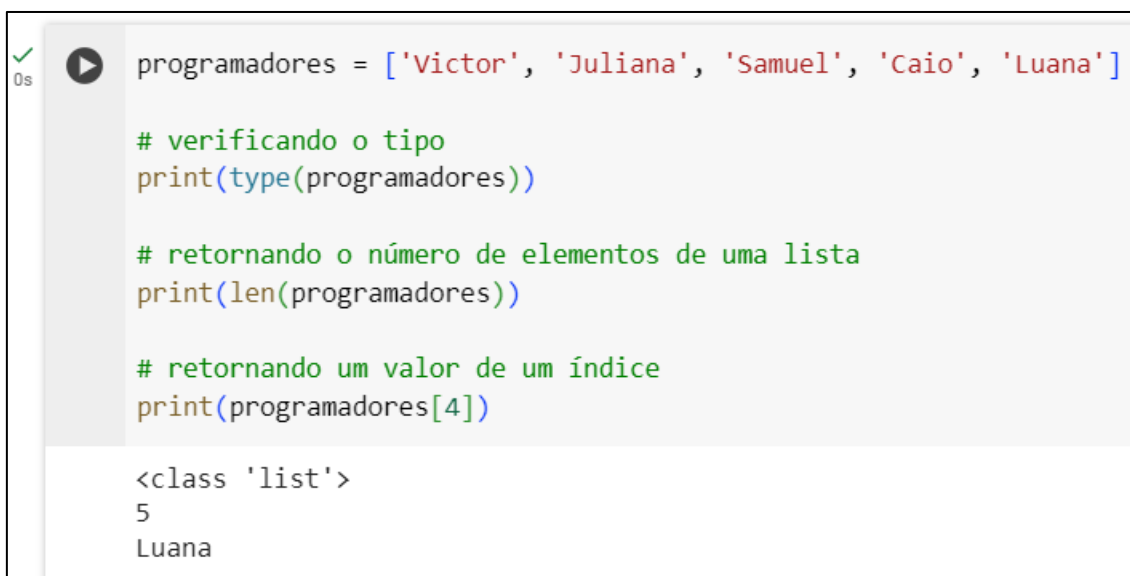
Neste capítulo abordaremos mais tipos de dados na linguagem Python. Aqui veremos os tipos que trabalham como coleções de dados, como as listas, tuplas e dicionários. As coleções permitem armazenar múltiplos itens dentro de uma única unidade, que funciona como um container.

### 8.1. Listas

Lista é uma coleção de valores indexada, em que cada valor é identificado por um índice. O primeiro item na lista está no índice 0, o segundo no índice 1 e assim por diante.

Para criar uma lista com elementos deve-se usar colchetes e adicionar os itens entre eles separados por vírgula, como mostra o exemplo a seguir:

Figura 80 - Exemplo de Lista



```
✓ 0s ▶ programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']

# verificando o tipo
print(type(programadores))

# retornando o número de elementos de uma lista
print(len(programadores))

# retornando um valor de um índice
print(programadores[4])

<class 'list'>
5
Luana
```

Outra característica das listas no Python é que elas são mutáveis, podendo ser alteradas depois de terem sido criadas. Em outras palavras, podemos adicionar, remover e até mesmo alterar os itens de uma lista.

No exemplo a seguir vemos um exemplo de como alterar uma lista.



Figura 81 - Alterando uma Lista

```
✓ 0s ▶ programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']  
print(programadores)  
  
programadores[1] = 'Carolina'  
print(programadores)  
  
['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']  
['Victor', 'Carolina', 'Samuel', 'Caio', 'Luana']
```

Além de alterar elementos em listas, também é possível adicionar itens nelas, pois já vêm com uma coleção de métodos predefinidos que podem ser usados para manipular os objetos que ela contém. No caso de adicionar elementos, podemos usar o método **append()**, como veremos no exemplo a seguir:

Figura 82 - Utilizando o Append

```
✓ 0s ▶ programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']  
print(programadores)  
  
programadores.append('Renato')  
print(programadores)  
  
['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']  
['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana', 'Renato']
```

Assim como podemos adicionar itens em nossa lista, também podemos retirá-los. E para isso temos dois métodos: **remove()** para a remoção pelo valor informado no parâmetro, e **pop()** para remoção pelo índice do elemento na lista. Vejamos como isso funciona na lista de programadores que estamos usando como exemplo:





Figura 83 - Uso do Remove

```

0s [✓] ▶ programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
    print(programadores)

    programadores.remove('Victor')
    print(programadores)

    ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
    ['Juliana', 'Samuel', 'Caio', 'Luana']

```

Também é possível remover um item pelo seu índice:

Figura 84 - Utilizando o Remove pelo índice

```

0s [✓] ▶ programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
    print(programadores)

    programadores.pop(0)
    print(programadores)

    ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
    ['Juliana', 'Samuel', 'Caio', 'Luana']

```

No caso de tentarmos remover um item de uma posição inexistente, obtemos o erro ***IndexError: pop index out of range*** como veremos a seguir:

Figura 85 - Erro na ausência de um índice

```

0s [✗] ▶ programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']

    programadores.pop(5)

-----
IndexError                                Traceback (most recent call last)
<ipython-input-6-1239a7fc1201> in <cell line: 3>()
      1 programadores = ['Victor', 'Juliana', 'Samuel', 'Caio', 'Luana']
      2
----> 3 programadores.pop(5)

IndexError: pop index out of range

```



Outra característica das listas é que elas podem possuir diferentes tipos de elementos na sua composição. Isso quer dizer que podemos ter *strings*, booleanos, inteiros e outros tipos diferentes de objetos na mesma lista.

**Figura 86 - Uso de diferentes objetos**

```
aluno = ['Murilo', 19, 1.79] # Nome, idade e altura

print(type(aluno))
print(aluno)
```

<class 'list'>  
['Murilo', 19, 1.79]

## 8.2. Tuplas

Tupla é uma estrutura de dados semelhante a lista. Porém, ela tem a característica de ser imutável, ou seja, após uma tupla ser criada, ela não pode ser alterada. Vejamos o uso desse objeto no exemplo a seguir:

**Figura 87 - Exemplo Tuplas**

```
times_rj = ('Botafogo', 'Flamengo', 'Fluminense', 'Vasco')

print(type(times_rj))
print(times_rj)
```

<class 'tuple'>  
('Botafogo', 'Flamengo', 'Fluminense', 'Vasco')

Assim como é feito nas listas, podemos acessar um determinado valor na tupla pelo seu índice.

**Figura 88 - Acessando um elemento de uma Tupla**

```
times_rj = ('Botafogo', 'Flamengo', 'Fluminense', 'Vasco')

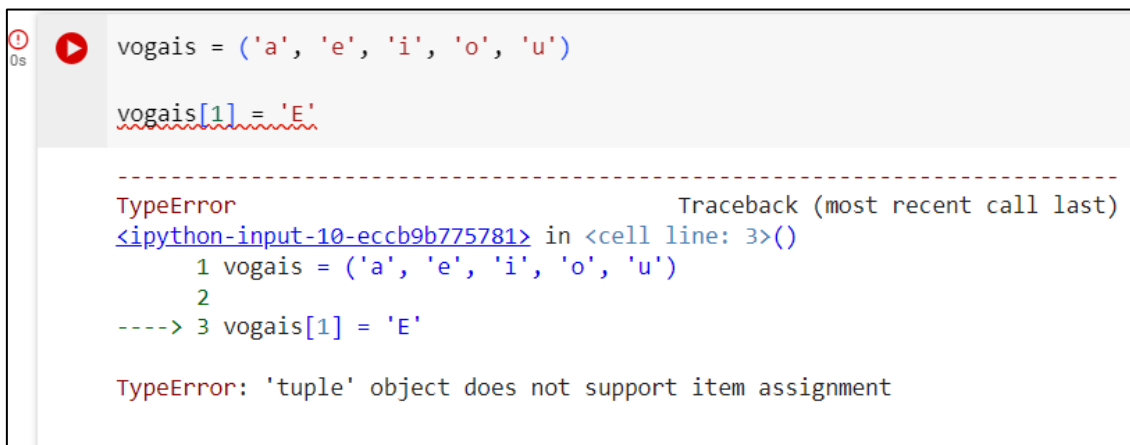
print(times_rj[2])
```

Fluminense



O fato da tupla ser imutável faz com que os seus elementos não possam ser alterados depois dela já criada. Vamos usar a tupla vogais para mostrar um exemplo desse tipo. Veja no exemplo a seguir:

**Figura 89 - Erro de acesso em uma Tupla**



```
0s [icon] vogais = ('a', 'e', 'i', 'o', 'u')

vogais[1] = 'E'

-----
TypeError                                Traceback (most recent call last)
<ipython-input-10-eccb9b775781> in <cell line: 3>()
      1 vogais = ('a', 'e', 'i', 'o', 'u')
      2
----> 3 vogais[1] = 'E'

TypeError: 'tuple' object does not support item assignment
```

Veja que não é possível fazer alteração nas tuplas. Diferentemente do que acontece com as listas, não podemos trocar os elementos de um objeto do tipo tupla, pois se trata de uma sequência imutável.

As tuplas devem ser usadas em situações em que não haverá necessidade de adicionar, remover ou alterar elementos de um grupo de itens. Exemplos bons seriam os meses do ano, os dias da semana, as estações do ano etc. Nesses casos, não haverá mudança nesses itens (pelo menos isso é muito improvável).

### 8.3. Dicionários

Os dicionários representam coleções de dados que contém na sua estrutura um conjunto de pares chave/valor, nos quais cada chave individual tem um valor associado. Esse objeto representa a ideia de um mapa, que entendemos como uma coleção associativa desordenada. A associação nos dicionários é feita por meio de uma chave que faz referência a um valor.



Figura 90 - Exemplo Dicionário

```

[11] dados_cliente = {
    'Nome': 'Renan',
    'Endereco': 'Rua Cruzeiro do Sul',
    'Telefone': '982503645'
}

print(dados_cliente['Nome'])

Renan

print(dados_cliente['Telefone'])

982503645

```

**“Nas listas e tuplas acessamos os dados por meio dos índices. Já nos dicionários, o acesso aos dados é feito por meio da chave associada a eles.”**

Para adicionar elementos num dicionário basta associar uma nova chave ao objeto e dar um valor a ser associado a ela.

Figura 91 - Adicionando elementos em um Dicionário

```

dados_cliente = {
    'Nome': 'Renan',
    'Endereco': 'Rua Cruzeiro do Sul',
    'Telefone': '982503645'
}

print(dados_cliente)

dados_cliente['Idade'] = 40

print(dados_cliente)

{'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul', 'Telefone': '982503645'}
{'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul', 'Telefone': '982503645', 'Idade': 40}

```

Para remover um item do dicionário, podemos usar o método **pop()**.



Figura 92 - Removendo um elemento de um dicionário

```
0s dados_cliente = {  
    'Nome': 'Renan',  
    'Endereco': 'Rua Cruzeiro do Sul',  
    'Telefone': '982503645'  
}  
  
print(dados_cliente)  
  
dados_cliente.pop('Telefone', None)  
  
print(dados_cliente)  
  
{'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul', 'Telefone': '982503645'}  
{'Nome': 'Renan', 'Endereco': 'Rua Cruzeiro do Sul'}
```

#### 8.4. Funções para coleções

O Python conta com funções úteis quando se trabalha com coleções. Vejamos algumas delas:

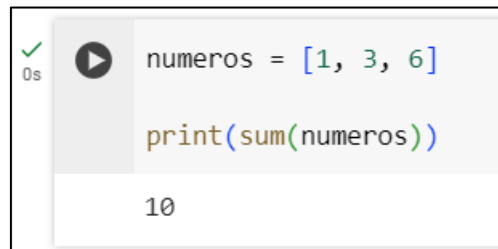
- **min() e max()**

Figura 93 - Funções min() e max()

```
0s numeros = [15, 5, 0, 20, 10]  
nomes = ['Caio', 'Alex', 'Renata', 'Patrícia', 'Bruno']  
  
print(min(numeros))  
print(max(numeros))  
print(min(nomes))  
print(max(nomes))  
  
0  
20  
Alex  
Renata
```

- **sum()**



**Figura 94 - Função sum()**A screenshot of a code editor showing a Python list named 'numeros' with values [1, 3, 6]. The code uses the sum() function to calculate the total, which is printed as 10. A green checkmark and '0s' indicate successful execution.

```
numeros = [1, 3, 6]
print(sum(numeros))
```

10

- len()

**Figura 95 - Função len()**A screenshot of a code editor showing a list named 'países' containing four country names: 'Argentina', 'Brasil', 'Colômbia', and 'Uruguai'. The code uses the len() function to count the number of elements, which is printed as 4. A green checkmark and '0s' indicate successful execution.

```
países = ['Argentina', 'Brasil', 'Colômbia', 'Uruguai']
print(len(países))
```

4



## 9. DESVIOS CONDICIONAIS (IF, ELIF E ELSE)

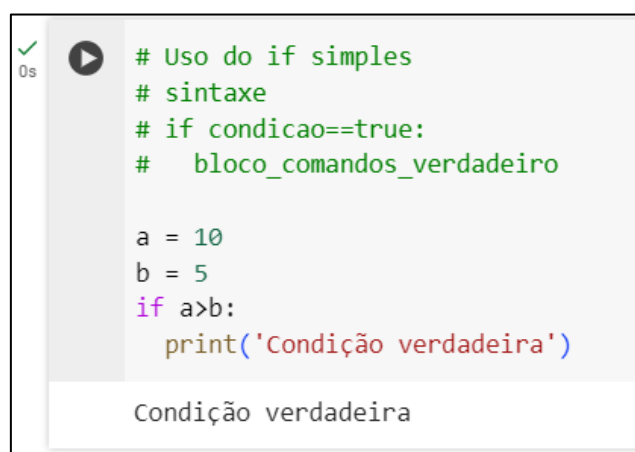
Neste capítulo, são apresentados os operadores condicionais. Isto é, os operadores que permitem o programa realizar verificações condicionais para uma lógica mais robusta e que proporcione melhores soluções ao problema a ser resolvido.

Em Python, os operadores condicionais são o **if** e o **else**. A partir destes, existe a ramificação **elif**, que é a junção do **if** com o **elif** para indentar os condicionais em sequência lógica.

Quando um teste condicional é realizado e o resultado for verdadeiro, executa-se todos os comandos abaixo do comando **if**. Caso seja falso, serão executados os comandos escritos no **else** ou **elif**.

Exemplo 1 – Uso do **if** simples:

Figura 96 - Uso do if simples



```
# Uso do if simples
# sintaxe
# if condicao==true:
#     bloco_comandos_verdadeiro

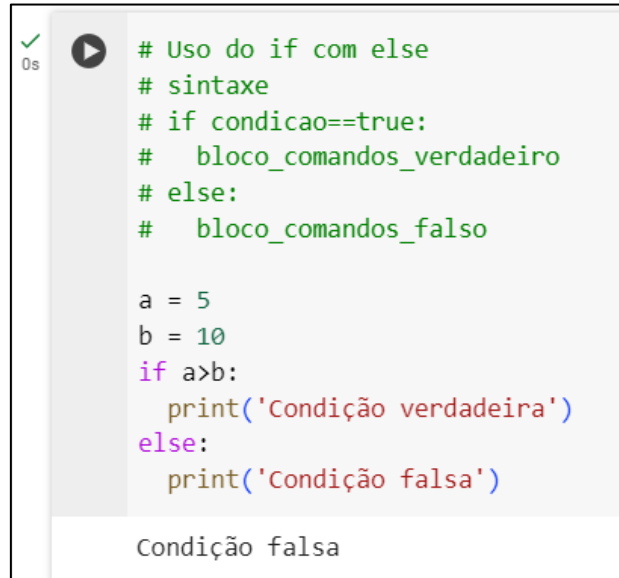
a = 10
b = 5
if a>b:
    print('Condição verdadeira')
```

Condição verdadeira

Exemplo 2: Uso do **if** com **else**:



Figura 97 - Uso do if com else



```
# Uso do if com else
# sintaxe
# if condicao==true:
#     bloco_comandos_verdadeiro
# else:
#     bloco_comandos_falso

a = 5
b = 10
if a>b:
    print('Condição verdadeira')
else:
    print('Condição falsa')
```

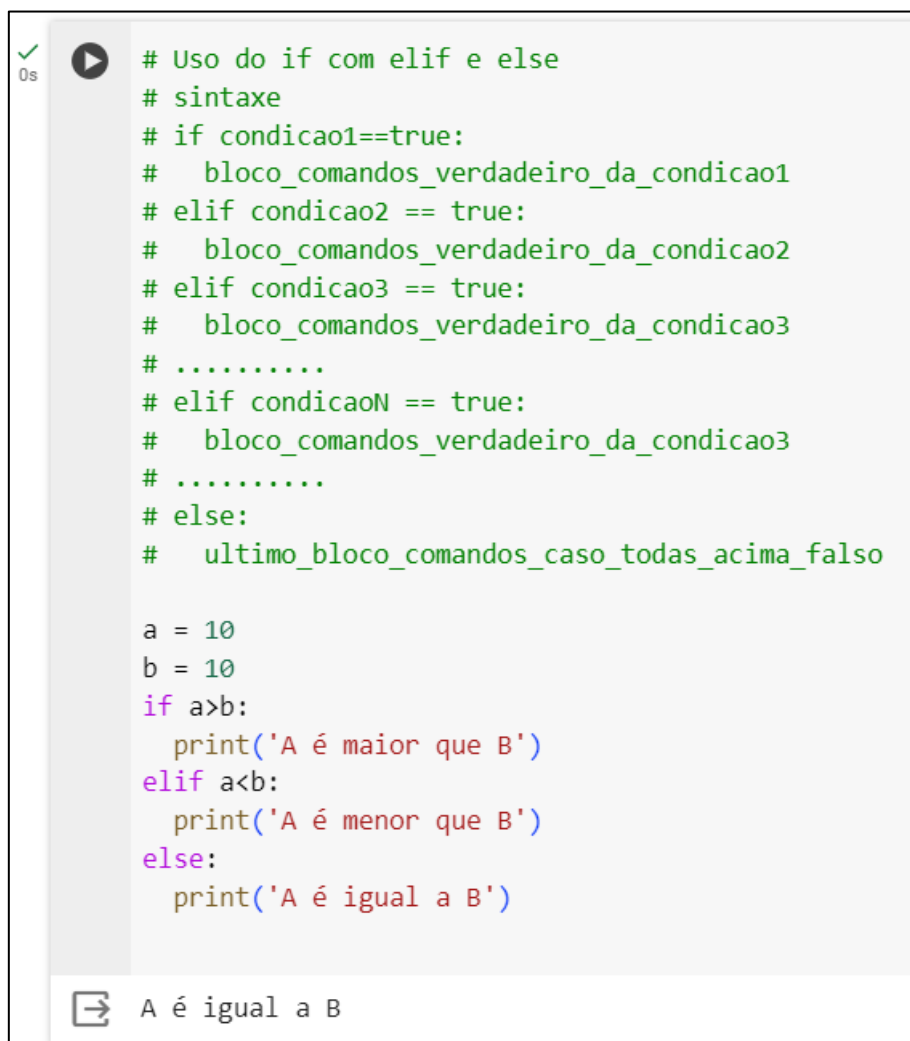
Condição falsa

Exemplo 3: Uso do **if** com **elif** e **else**:





Figura 98 - Uso do if com elif e else



```

# Uso do if com elif e else
# sintaxe
# if condicao1==true:
#     bloco_comandos_verdadeiro_da_condicao1
# elif condicao2 == true:
#     bloco_comandos_verdadeiro_da_condicao2
# elif condicao3 == true:
#     bloco_comandos_verdadeiro_da_condicao3
# .....
# elif condicaoN == true:
#     bloco_comandos_verdadeiro_da_condicao3
# .....
# else:
#     ultimo_bloco_comandos_caso_todas_acima_falso

a = 10
b = 10
if a>b:
    print('A é maior que B')
elif a<b:
    print('A é menor que B')
else:
    print('A é igual a B')

```

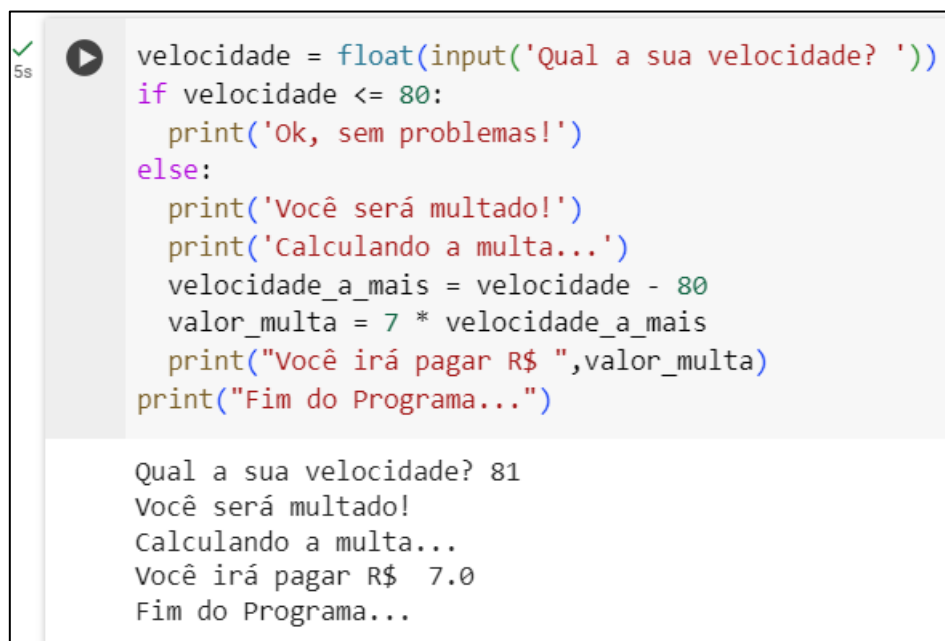
A é igual a B

Vejamos um outro exemplo de uma outra situação:

O programa abaixo multa um infrator de trânsito que ultrapassou 80 km por hora. O valor por cada km acima de 80 é R\$7,00. Veja a solução abaixo:



Figura 99 - Exemplo de uma outra situação com uso de uma estrutura condicional



```
✓ 5s ▶ velocidade = float(input('Qual a sua velocidade? '))
if velocidade <= 80:
    print('Ok, sem problemas!')
else:
    print('Você será multado!')
    print('Calculando a multa...')
    velocidade_a_mais = velocidade - 80
    valor_multa = 7 * velocidade_a_mais
    print("Você irá pagar R$ ",valor_multa)
    print("Fim do Programa...")
```

Qual a sua velocidade? 81  
Você será multado!  
Calculando a multa...  
Você irá pagar R\$ 7.0  
Fim do Programa...

### 9.1. Dividindo um problema em partes

Vejamos outro exemplo, você foi convidado a criar um programa para calcular a média aritmética das notas de um aluno qualquer e, em seguida, apresentar o resultado de aprovação do aluno com base nas seguintes regras: se a média aritmética for maior ou igual a 60 o aluno é considerado aprovado, caso contrário estará reprovado. Como você resolveria isso? Vamos lá. Seu programa deve receber duas notas, as quais você armazenaria em duas variáveis distintas, por exemplo, nota1 e nota2. Em seguida, calcularia a média aritmética das notas e, só agora, se preocuparia com os testes da nota para aprovação ou reprovação.



**Figura 100 - Dividindo um programa em partes utilizando o Google Colab**

```

✓ 5s [8] nota1 = int(input("Informe a nota do bimestre 1 (0-100): "))
      nota2 = int(input("Informe a nota do bimestre 2 (0-100): "))
      media = (nota1 + nota2) / 2
      print("Media: ", media)
      estado_aprovacao = media >= 60
      print(estado_aprovacao)
      # O que fazer?

Informe a nota do bimestre 1 (0-100): 50
Informe a nota do bimestre 2 (0-100): 60
Media: 55.0
False

```

Perceba que, até agora, conseguimos calcular a média e até saber o estado de aprovação, com base em um teste realizado com a variável `media`. No entanto, não conseguimos avançar com respeito ao resultado da aprovação. Como faremos para apresentar o resultado "Aprovado" ou "Reprovado"?

É aí que entram as estruturas condicionais, ou seja, a partir do resultado de um teste condicional, executaremos comandos e/ou deixaremos de executar outros. Certamente, agora é aquele momento que você pensa: "Humm?". Calma! Agora, vamos contar com o auxílio das palavras chaves `if` e `else`. Com elas, conseguiremos decidir se apresentaremos o resultado "Aprovado" ou "Reprovado".

**Figura 101 - Utilizando a Estrutura Condicional**

```

✓ 7s [8] nota1 = int(input("Informe a nota do bimestre 1 (0-100): "))
      nota2 = int(input("Informe a nota do bimestre 2 (0-100): "))
      media = (nota1 + nota2) / 2
      if media >= 60:
          print("Aprovado")
      else:
          print("Reprovado")

Informe a nota do bimestre 1 (0-100): 50
Informe a nota do bimestre 2 (0-100): 60
Reprovado

```



Observe que o código reflete nossa estratégia mental de solução do problema. Após a declaração e leitura das variáveis, calculamos a média e testamos esse valor para decidir se o aluno foi aprovado ou reprovado. O referido teste

Figura 102 - Uso do if e else

```
if media >= 60:  
    print("Aprovado")  
else:  
    print("Reprovado")
```

pode ser interpretado da seguinte maneira: Se (**if**) a média for maior ou igual a 60, imprima a palavra Aprovado, senão (**else**), imprima a palavra Reprovado. Portanto, a palavra-chave **if** é um teste que vai decidir que comandos serão executados: **print("Aprovado")** ou **print("Reprovado")**.

## 9.2. Exercício

Refaça o exemplo anterior, identificando o conceito aprovado (média superior a 60), exame (média entre 40 e 60) ou reprovado (média inferior a 40).



## 10. ESTRUTURAS REPETIÇÃO

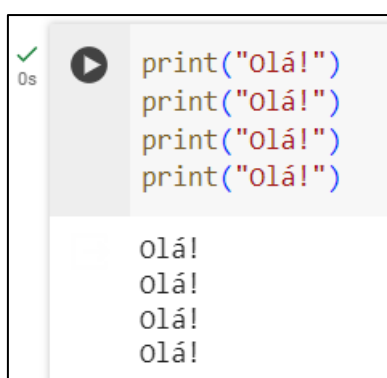
Há situações do nosso cotidiano que, às vezes, a repetição de uma atividade é necessária. Imagine, por exemplo, que você está no final de semana e resolveu acessar uma plataforma de vídeos sob demanda para assistir a um filme. Você não sabe ao certo a qual filme assistirá e, por isso, lê diversas sinopses a fim de encontrar aquele filme que você espera ser interessante. Ao encontrá-lo, você o seleciona e começa uma sessão de cinema na confortável poltrona da sua sala.

No exemplo anterior, a atividade que foi repetida diversas vezes foi a leitura da sinopse do filme. Após repetidas leituras, o fato de você ter encontrado o filme ideal para o momento fez você parar a leitura de várias sinopses, ou seja, isto foi a sua condição de parada.

Como em situações do mundo real, em um programa, pode ser necessário repetir um trecho diversas vezes até que uma determinada condição seja satisfeita. Em programação, para casos como esse, são usadas estruturas conhecidas como iteração (não é interação!), repetição, laço ou loop. Python implementa duas estruturas de repetição: `while` e `for`.

Vamos construir um programa para imprimir quatro vezes a saudação “Olá!” da maneira que aprendemos até o momento:

**Figura 103 - Repetição sem o uso de uma estrutura de repetição**



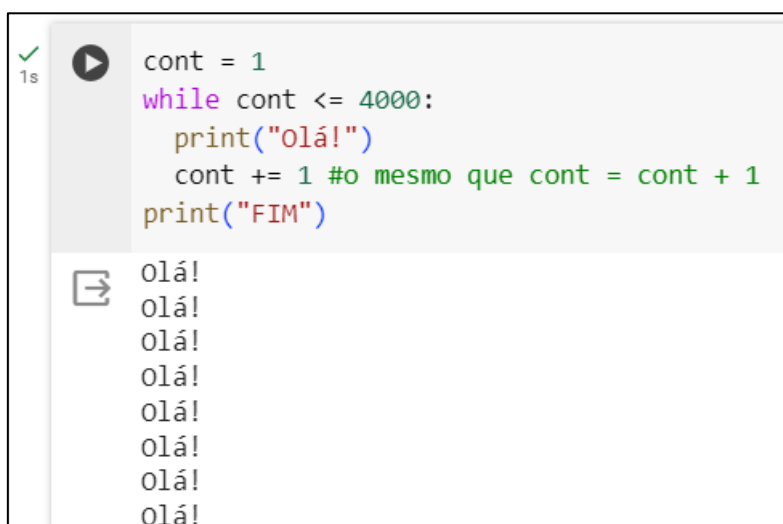
```
✓ 0s ▶ print("Olá!")  
print("Olá!")  
print("Olá!")  
print("Olá!")  
  
Olá!  
Olá!  
Olá!  
Olá!
```



Mas, se agora pedíssemos para você construir um programa para escrever essa saudação quatro mil vezes, você usaria a mesma estratégia da situação anterior, isto é, escreveria quatro mil vezes o comando `print("Olá!")`?

As quatro mil saudações podem ser escritas de uma maneira não enfadonha, utilizando uma estrutura de repetição, conforme o exemplo abaixo:

Figura 104 - Exemplo com uso da estrutura While



```
✓ 1s ▶ cont = 1
while cont <= 4000:
    print("Olá!")
    cont += 1 #o mesmo que cont = cont + 1
print("FIM")
```

Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!

Obs.: Neste exemplo cortamos a imagem pois o programa escreveu na tela 4000 vezes a saudação: Olá! 😊

## 10.1. ESTRUTURA WHILE

O comportamento da estrutura de repetição **while** é muito simples e a sua ideia é a seguinte: enquanto uma condição (ou um conjunto delas) for verdadeira, uma instrução (ou um conjunto de instruções) que está dentro do laço deverá ser executada. Veja a sintaxe dessa estrutura no pseudocódigo:

Figura 105 - Sintaxe While

```
1 while <condição (ou um conjunto delas) for verdadeira>:
2     #Instruções a serem executadas
3     #Instruções a serem executadas após o encerramento do loop
```



Retornando das 4000 saudações, percebemos que as instruções das linhas 3 e 4 estão deslocados para a direita, em relação à instrução da linha 2. Isso significa que as duas instruções estão dentro do fluxo de execução da estrutura **while** e, assim, elas só serão processadas enquanto a condição determinada pelo laço de repetição for verdadeira, isto é, **cont <= 4000**. Por outro lado, a linha 5 está com a mesma indentação da linha 2 e, portanto, não está dentro do laço. A seguir, detalharemos esse programa, linha por linha.

- Na linha 1, é definida uma variável chamada **cont**, que tem como papel contar a quantidade de vezes que o laço **while** está sendo executado. Em um programa, geralmente variáveis que têm esse papel são chamadas de variáveis contadoras.
- Na linha 2, é utilizada a estrutura de repetição **while**, cuja condição de execução é o valor da variável contadora **cont** ser menor ou igual a 4000. Portanto, quando o valor de **cont** passar a ser maior que 4000, a condição de execução deixará de ser verdadeira e, consequentemente, as instruções definidas nas linhas 3 e 4 não serão executadas.
- Na linha 3, a **string** “Olá!” será exibida várias vezes porque a instrução **print("Olá!")** está dentro da estrutura **while**.
- Na linha 4, a variável **cont** é incrementada em uma unidade. Em outras palavras, todas as vezes que o programa entrar no laço, a variável contadora terá o seu valor aumentado em 1, isto é, **cont += 1** (vale lembrar que esta instrução poderia ser substituída por **cont = cont + 1**). Ao fim da execução da linha 4, o fluxo de execução retornará para a linha 2, verificando se o novo valor de **cont** é menor ou igual a 4000. Sendo verdadeiro, as linhas 3 e 4 serão executadas novamente (esse “vai e vem” entre as linhas 2, 3 e 4, é o motivo de a estrutura de repetição também ser chamada de laço ou de loop). Quando **cont** passar a ser maior que 4000, o



laço não será mais executado e o fluxo do programa será deslocado para a linha 5.

- Na linha 5, finalmente o programa exibirá a **string FIM** e será encerrado.

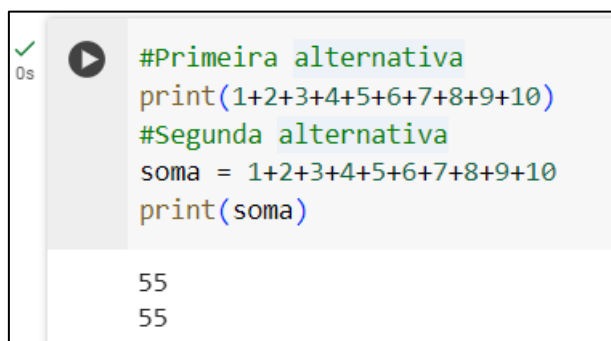
Em algumas situações é necessária a construção de trecho cujo comportamento seja semelhante a de um loop infinito. Uma maneira de escrevê-lo é utilizando a palavra reservada **True**, conforme o exemplo a seguir:

Figura 106 - Sintaxe de um While com loop infinito

```
1 while True:  
2     print("Execução de um loop infinito")
```

Agora, vamos apresentar um novo exemplo, porém com um número finito de iterações. Imaginemos a situação na qual se deseja calcular e exibir a soma dos 10 primeiros números inteiros positivos. Duas possíveis alternativas (porém, bastante mal elaboradas) são apresentadas no programa abaixo:

Figura 107 - Exemplo de soma de 10 números sem o uso de estrutura de repetição



```
#Primeira alternativa  
print(1+2+3+4+5+6+7+8+9+10)  
#Segunda alternativa  
soma = 1+2+3+4+5+6+7+8+9+10  
print(soma)
```

55  
55

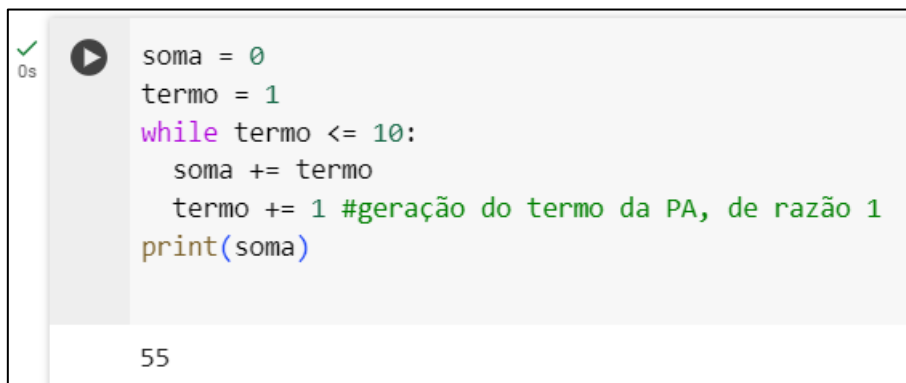
Neste somatório, podemos perceber a existência de uma sequência numérica finita que cresce, de forma padronizada, em uma unidade. Observa-se que o primeiro elemento é o 1 e o último é o 10 (1, 2, 3, ..., 10). Na Matemática, essa sequência numérica é chamada de progressão aritmética (PA).

Uma nova alternativa para somar os termos dessa PA, diferentemente do que foi apresentado no exemplo anterior, é utilizar uma estrutura de repetição, como visto no Programa a seguir:





Figura 108 - Soma de 10 números com uso do While



```
soma = 0
termo = 1
while termo <= 10:
    soma += termo
    termo += 1 #geração do termo da PA, de razão 1
print(soma)
```

55

## 10.2. FUNÇÃO RANGE

Se fizermos uma tradução literal da língua inglesa para a língua portuguesa, a palavra range teria como significado “faixa”. Na prática, essa tradução faz sentido porque, em Python, a função range() gera uma sequência de números dentro de uma faixa especificada.

É bastante comum a utilização dessa função em conjunto com a estrutura de repetição for, a qual falaremos na próxima seção. A seguir, mostraremos duas maneiras de invocar a função range():

Figura 109 - Sintaxe da função Range()

1. `range(<quantidade_de_números_a_serem_gerados>)`
2. `range(<início_da_faixa>, <fim_da_faixa>[, <incremento>])`

Na primeira maneira, a função range() recebe um parâmetro o qual indica a quantidade N de números a serem gerados, onde N deve ser maior que 0. Para ficar claro como se dá a execução da primeira maneira, apresentaremos um exemplo utilizando uma lista, assunto que será abordado no próximo capítulo. Veja:

Figura 110 - Uso do list() com o Range()



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

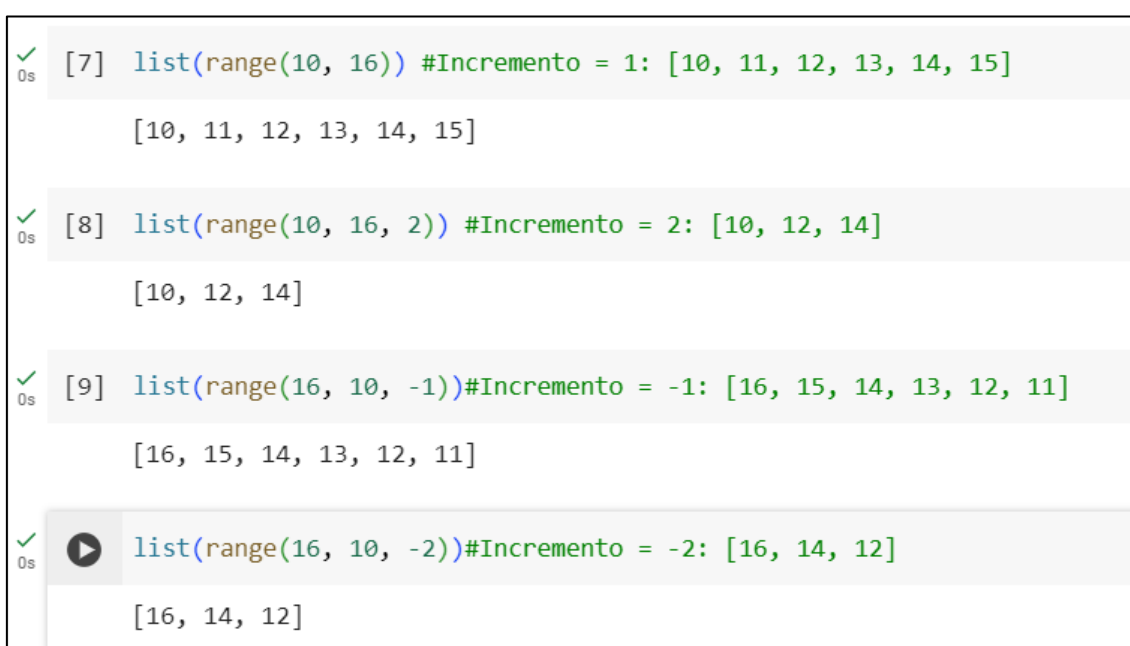
[0, 1, 2]



A saída dessa instrução é uma lista contendo 3 números: 0, 1 e 2. O detalhe importante é que o primeiro número da sequência gerada é o 0 e o último é o 2. Portanto, a instrução `range(N)` gera uma lista com N números, onde o primeiro número é 0 e o último é  $N - 1$ . Percebe? Então, agora, execute a instrução `list(range(10))` e observe que o primeiro número é 0 e o último 9.

Por outro lado, a segunda maneira de executar a função `range()` possui três parâmetros distintos: `<início_da_faixa>`, `<fim_da_faixa>` e `<incremento>` (opcional). Embora os nomes dos parâmetros sejam bastante significativos, `<início_da_faixa>` determina o primeiro número que deve ser gerado na faixa, `<incremento>` indica a razão (positiva ou negativa) da PA e, quando não especificado, seu valor padrão é 1. Por fim, o parâmetro `<fim_da_faixa>` está associado à razão da sequência. Quando `<incremento>` for positivo, o último número da sequência corresponde a `<fim_da_faixa> - <incremento>`. No caso de `<incremento>` ser negativo, o último número da sequência é calculado da seguinte forma: `<fim_da_faixa> + <incremento> * (-1)`.

**Figura 111 - Exemplos de faixas e incrementos com o range()**



```

[7] list(range(10, 16)) #Incremento = 1: [10, 11, 12, 13, 14, 15]
[10, 11, 12, 13, 14, 15]

[8] list(range(10, 16, 2)) #Incremento = 2: [10, 12, 14]
[10, 12, 14]

[9] list(range(16, 10, -1)) #Incremento = -1: [16, 15, 14, 13, 12, 11]
[16, 15, 14, 13, 12, 11]

[10] list(range(16, 10, -2)) #Incremento = -2: [16, 14, 12]
[16, 14, 12]
  
```

Perceba que quando `<incremento>` é positivo `<início_da_faixa>` é menor que `<fim_da_faixa>`. No entanto, quando `<incremento>` é negativo,



<início\_da\_faixa> é maior que <fim\_da\_faixa>. Agora, já podemos falar da outra estrutura de repetição: o for.

### 10.3. ESTRUTURA FOR

É comum programadores iniciantes se questionarem sobre qual das duas estruturas de repetição é mais adequada: while ou for. Costumamos dizer que depende da situação. Em geral, quando a quantidade de iterações é indeterminada, a estrutura while é uma boa alternativa. Por outro lado, quando o número de iterações é definido, a estrutura for é bastante adequada, cuja sintaxe é a seguinte:

Figura 112 - Sintaxe da estrutura for

```
1   for <variável> in range([maneira_1|maneira_2]):  
2       #Instruções a serem executadas  
3       #Instruções a serem executadas após o fim do loop
```

Agora, vamos recodificar o programa que faz as 4000 saudações de uma maneira diferente, modificando os programas para que utilizem a estrutura de repetição for ao invés da estrutura while.

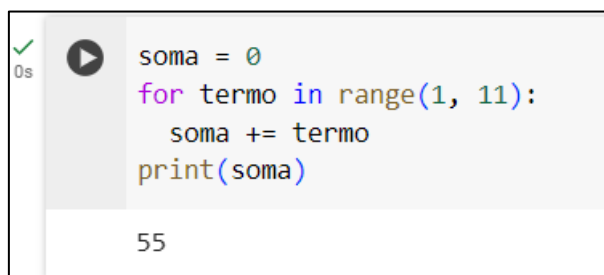
Figura 113 - Exemplo da estrutura for para 4000 repetições



```
✓ 1s ▶ for cont in range(4000):  
    print("Olá!")  
    print("FIM")
```

Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!  
Olá!



**Figura 114 - Exemplo do uso do for para a soma dos números de 1 a 10**

```
soma = 0
for termo in range(1, 11):
    soma += termo
print(soma)
```

55

#### 10.4. EXERCÍCIO RESOLVIDO

Crie um programa no qual o usuário informe um número inteiro positivo N e armazene-o em uma variável. Em seguida, o usuário deve digitar N números. Ao fim, o programa deve imprimir a média aritmética dos N números digitados.

- **Comentários sobre a resolução**

Antes de começar a resolver esta questão, é importante destacar que isso se trata de um processo de repetição. Observe que a repetição ocorre porque o usuário informará N vezes um determinado número, considerando N um número inteiro e maior que 0. Nós poderíamos utilizar a estrutura while para resolver essa questão. No entanto, como sabemos a quantidade de vezes que o laço será executado, optamos pelo laço for.

Vamos ao código:



Figura 115 - Resolução do Exercício Resolvido utilizando o for

```
✓ [12] N = int(input("Digite a quantidade de números a informar: "))  
9s soma = 0  
for cont in range(N):  
    num = float(input("Digite um número: "))  
    soma += num  
media = soma / N  
print("Média = ", media)
```

```
Digite a quantidade de números a informar: 5  
Digite um número: 1  
Digite um número: 2  
Digite um número: 3  
Digite um número: 4  
Digite um número: 5  
Média = 3.0
```

