

Aprendendo Python

Um guia básico de programação



Prof. André Backes
 @progdescomplicada

Sumário

1	Introdução.....	6
1.1	A linguagem Python	6
1.1.1	Instalação e uso	8
1.1.2	Vantagens e desvantagens do Python.....	9
1.1.3	Aplicações.....	9
1.2	Comentários e Docstrings	10
1.3	Trabalhando com módulos	11
2	Manipulando dados em Python	13
2.1	Variáveis.....	13
2.1.1	Declarando uma variável.....	13
2.1.2	Nomes de variáveis.....	14
2.1.3	Os tipos das variáveis	15
2.1.4	Conversão de tipos	16
2.2	Escrevendo as variáveis.....	16
2.3	Lendo as variáveis.....	18
3	Realizando operações com as variáveis.....	20
3.1	Operadores aritméticos	20
3.2	Operadores relacionais	21
3.3	Operadores lógicos	22
3.4	Operadores bit-a-bit.....	24
3.5	Operadores de atribuição simplificada	25
3.6	Precedência de operadores	27
4	Comandos de controle condicional.....	29
4.1	Definindo uma condição	29
4.2	O comando if	30
4.3	O comando else	31
4.4	Aninhamento de if.....	33
4.5	O comando elif	35
5	Comandos de repetição.....	37
5.1	O comando while.....	37
5.2	O comando for.....	39
5.3	A função range().....	40

5.4	Diferença entre os comandos while e for	41
5.5	Aninhamento de repetições	42
5.6	O comando break	43
5.7	O comando continue	45
6	Listas	47
6.1	Por que utilizar	47
6.2	Sintaxe e criação de uma lista	49
6.3	Acessando os elementos da lista	50
6.4	Acessando uma sub-lista	54
6.5	Manipulando listas	55
6.5.1	Concatenando e repetindo listas	55
6.5.2	Copiando uma lista	56
6.5.3	Removendo elementos da lista	58
6.5.4	Procurando um valor dentro da lista	59
6.5.5	Métodos que manipulam listas	59
6.6	Listas aninhadas	61
6.6.1	Sintaxe e criação de uma lista aninhada	61
6.6.2	Acessando os elementos da lista aninhada	61
6.6.3	Percorrendo os elementos da lista aninhada	63
6.6.4	Removendo elementos da lista aninhada	64
6.7	Compreensão de lista	64
6.8	Aplicações e desempenho	66
7	Tuplas e dicionários	68
7.1	Tuplas	68
7.1.1	Sintaxe e criação de uma tupla	68
7.1.2	Acessando os elementos da tupla	69
7.1.3	Aplicações e desempenho	71
7.2	Dicionários	72
7.2.1	Sintaxe e criação de um dicionário	72
7.2.2	Acessando os elementos do dicionário	73
7.2.3	Funções e métodos de dicionários	74
7.2.4	Aplicações e desempenho	75
8	Strings	77

8.1	Definindo uma string.....	77
8.2	Acessando os elementos da string.....	78
8.3	Acessando uma sub-string	80
8.4	Formatação de string.....	81
8.5	Manipulando strings	83
8.5.1	Concatenando strings	83
8.5.2	Comparando duas strings	83
8.5.3	Procurando uma string dentro de outra.....	84
8.5.4	Métodos que manipulam strings	85
9	Funções.....	87
9.1	Definição e estrutura de uma função.....	87
9.1.1	Declarando uma função	88
9.1.2	O corpo da função.....	89
9.1.3	Os parâmetros da função.....	90
9.1.4	O retorno da função	92
9.2	Cuidados com o escopo de variáveis	94
9.3	Cuidados com a passagem de parâmetros	97
9.4	Recursão	98
9.4.1	Como funciona a recursão	100
9.4.2	Cuidados durante a implementação da recursão	102
10	Arquivos	106
10.1	O que é e os tipos de arquivos existentes.....	106
10.2	Abrindo e fechando um arquivo	107
10.3	Escrevendo dados em um arquivo	111
10.3.1	O método write()	111
10.3.2	O método writelines()	112
10.4	Lendo dados de um arquivo	113
10.4.1	O método read().....	113
10.4.2	O método readline()	113
10.4.3	O método readlines()	114
10.5	Lendo até o final do arquivo	115
10.6	Sabendo a posição atual dentro do arquivo	116
10.7	Movendo-se dentro do arquivo.....	116

11	Trabalhando com classes e objetos	118
11.1	Classe e objeto	119
11.2	Atributos e Métodos	121
11.3	Encapsulamento	124
11.4	Construtor e destrutor	125
11.5	Sobrecarga de operadores.....	127
11.5.1	Somando dois objetos	128
11.5.2	Comparando o conteúdo de dois objetos	129
11.6	Imprimindo um objeto.....	130
11.7	Copiando um objeto	131
11.8	Herança.....	134
11.8.1	Sobreposição de métodos - <i>override</i>	136
11.8.2	Herança Múltipla	138
11.9	Trabalhando com um iterator	140
12	Tratamento de exceções	142
12.1	O bloco try-except	142
12.2	O bloco try-finally	144

1 Introdução

É com grande entusiasmo que apresentamos esse material didático sobre a linguagem de programação Python. Python tem se destacado como uma das linguagens mais populares e versáteis do mundo da programação, conquistando desenvolvedores, cientistas de dados, engenheiros e estudantes de todas as áreas.

A ampla gama de bibliotecas e frameworks disponíveis para Python permite que você construa desde simples scripts até sistemas complexos, aplicativos web dinâmicos e soluções avançadas de análise de dados e inteligência artificial.

Nosso objetivo com este material é capacitar o estudante a dominar suas características, sintaxe e aplicação em diversos contextos.

Seja você um iniciante curioso, um estudante ansioso para ampliar horizontes ou um profissional que busca aprimoramento, nosso material está aqui para guiá-lo e capacitá-lo a dominar as características, sintaxe e aplicação da linguagem Python em diversos contextos.

Ao longo desse material você encontrará diversas caixas com dicas, avisos e vídeo aulas sobre a linguagem, como mostram os exemplos a seguir.



Nessa caixa você encontra alguma informação importante ou lembrete sobre a linguagem Python.



Atenção! Essa caixa indica que devemos tomar cuidado com algum recurso da linguagem Python.



Clique aqui para ver uma vídeo aula sobre o assunto da seção que você está estudando.

Estamos ansiosos para embarcar nessa jornada com você e compartilhar a empolgante aventura de explorar o universo Python. Vamos começar!

1.1 A linguagem Python

Python é uma linguagem de programação versátil e de alto nível, conhecida por sua simplicidade, clareza e facilidade de leitura. Criada por Guido van Rossum e lançada pela primeira vez em 1991, Python rapidamente ganhou popularidade devido à sua sintaxe intuitiva e à ampla gama de aplicações, desde desenvolvimento web e científico até automação de tarefas e aprendizado de máquina.



Clique aqui para ver a vídeo aula de introdução ao Python.

Um dos aspectos notáveis da Python é o seu interpretador híbrido, que combina interpretação e compilação para otimizar o desempenho e a usabilidade da linguagem. Aqui estão alguns pontos-chave sobre o interpretador híbrido de Python:



Python é frequentemente chamado de **interpretado**, mas a realidade é um pouco mais complexa.

Ele utiliza um processo de interpretação e compilação, o que o torna um interpretador híbrido. Aqui está como funciona:

- **Análise Sintática:** O código-fonte Python é lido e analisado para verificar sua sintaxe. Isso é feito pelo interpretador Python.
- **Byte-Code:** Em vez de ser traduzido diretamente para linguagem de máquina, o código-fonte Python é compilado em um formato intermediário chamado de **byte-code**. O byte-code é uma representação de baixo nível que pode ser entendida pela Máquina Virtual Python (PVM), permitindo que o código Python seja interpretado e executado eficientemente.
- **Execução pela Máquina Virtual:** O byte-code é executado pela Máquina Virtual Python (PVM), que é responsável por traduzi-lo para código de máquina específico do sistema operacional em tempo de execução.

Essa abordagem híbrida traz vantagens significativas. O código-fonte é portátil e pode ser executado em diferentes sistemas operacionais sem a necessidade de recompilação. Além disso, a Máquina Virtual Python gerencia aspectos como coleta de lixo e otimização de tempo de execução, o que contribui para a eficiência e a facilidade de desenvolvimento.

O byte-code também traz algumas vantagens:

- **Velocidade:** O byte-code é mais eficiente para a execução em comparação com a interpretação pura do código-fonte.
- **Portabilidade:** O byte-code é independente do sistema operacional e da arquitetura, o que permite a portabilidade do código Python.
- **Proteção de Código:** Embora não seja uma medida de segurança absoluta, o byte-code é mais difícil de ser lido e modificado do que o código-fonte original.

1.1.1 Instalação e uso

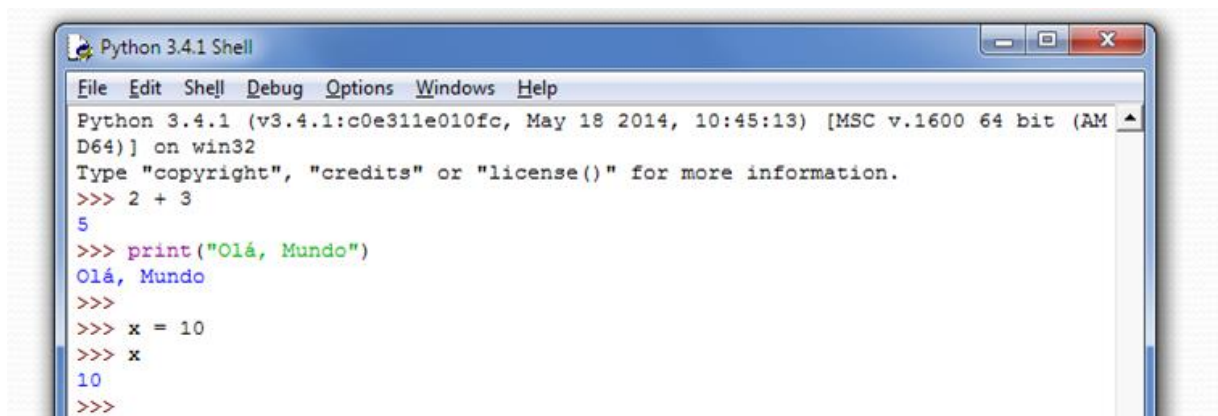
A instalação da versão mais atual do Python está disponível no site

<https://www.python.org/downloads/>



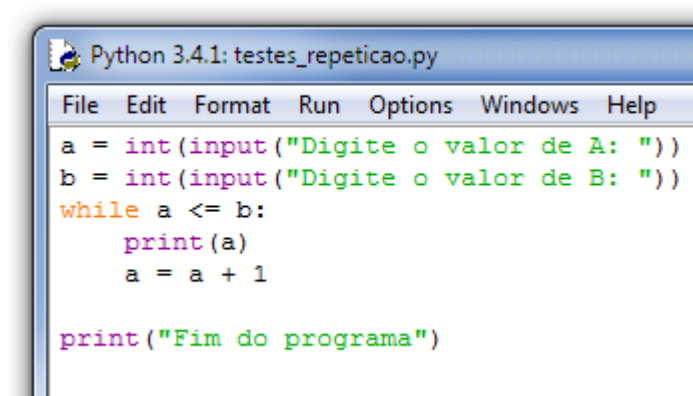
Clique aqui para ver a vídeo aula de introdução ao Python.

Após a sua instalação, podemos executar o programa **IDLE**, um ambiente de desenvolvimento integrado para Python. Ele funciona como um terminal Python onde podemos escrever comandos e ver os resultados imediatamente, como mostra a figura a seguir.



```
Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:45:13) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 3
5
>>> print("Olá, Mundo")
Olá, Mundo
>>>
>>> x = 10
>>> x
10
>>>
```

Também podemos criar um arquivo de *script* onde iremos escrever um programa completo e executar de uma só vez pressionando a tecla **F5**, como mostra a figura a seguir.



```
Python 3.4.1: testes_repeticacao.py
File Edit Format Run Options Windows Help
a = int(input("Digite o valor de A: "))
b = int(input("Digite o valor de B: "))
while a <= b:
    print(a)
    a = a + 1
print("Fim do programa")
```

Os arquivos de programas Python têm extensão `.py` e eles permitem criar códigos mais complexos e salvar o nosso trabalho.

1.1.2 Vantagens e desvantagens do Python

A linguagem Python apresenta uma série de vantagens, como:

- **Sintaxe Clara e Legível:** A sintaxe simples e clara de Python facilita a leitura e escrita de código, tornando-a uma excelente escolha para iniciantes e desenvolvedores experientes.
- **Ampla Comunidade e Suporte:** Python possui uma comunidade ativa e engajada de desenvolvedores, o que significa que há muitos recursos, documentações e bibliotecas disponíveis para facilitar o desenvolvimento.
- **Versatilidade:** Python é uma linguagem versátil, adequada para uma ampla variedade de aplicações, desde desenvolvimento web e científico até automação de tarefas e aprendizado de máquina.
- **Interpretador Híbrido:** O interpretador híbrido combina interpretação e compilação, resultando em código eficiente e portátil.
- **Portabilidade:** Python é executado em diferentes plataformas e sistemas operacionais, o que facilita a portabilidade do código.
- **Bibliotecas e Frameworks:** Python possui uma vasta gama de bibliotecas e frameworks que aceleram o desenvolvimento e permitem a criação de aplicativos complexos com menos esforço.
- **Aprendizado de Máquina e Análise de Dados:** Python é a escolha dominante para aprendizado de máquina, análise de dados e ciência de dados, com bibliotecas populares como NumPy, pandas e scikit-learn.

Apesar disso, a linguagem também possui algumas desvantagens:

- **Performance Relativa:** Python pode ser mais lento em comparação com linguagens de baixo nível, como C e C++, devido à sua natureza interpretada e gerenciamento de memória.
- **GIL (Global Interpreter Lock):** O GIL limita a capacidade de Python de executar várias threads em paralelo, o que pode afetar o desempenho em certas aplicações.
- **Consumo de Memória:** Em comparação com algumas outras linguagens, Python pode consumir mais memória, o que pode ser um problema em sistemas com recursos limitados.

1.1.3 Aplicações

As aplicações de Python incluem:

- **Desenvolvimento Web:** Frameworks como Django e Flask são usados para criar aplicativos web escaláveis e robustos.
- **Ciência de Dados:** Python é amplamente utilizado para análise e visualização de dados, além de modelagem estatística e machine learning.

- **Automação de Tarefas:** Python é uma escolha popular para automação de processos, scripts e tarefas repetitivas.
- **Aplicações de Desktop:** Python é usado para criar aplicativos de desktop com interfaces gráficas usando bibliotecas como Tkinter e PyQt.
- **Desenvolvimento de Aplicativos Móveis:** Python é usado em frameworks como Kivy para desenvolver aplicativos móveis multiplataforma.
- **Aprendizado de Máquina e IA:** Python é a principal linguagem para desenvolvimento de modelos de aprendizado de máquina e inteligência artificial.
- **Desenvolvimento de Protótipos:** A sintaxe legível e a rapidez de desenvolvimento tornam Python ideal para prototipagem de ideias.

1.2 Comentários e Docstrings



Clique aqui para ver a vídeo aula de sobre comentário em Python.

Comentários são trechos de texto no código Python que são ignorados pelo interpretador e não têm efeito na execução do programa. Eles são usados para adicionar notas explicativas ou anotações ao código, facilitando a compreensão para outros desenvolvedores (ou até mesmo para você mesmo no futuro).



Os comentários em Python são criados usando o caractere **#** (cerquilha) e podem ser inseridos em qualquer lugar do código.

O caractere **#** permite comentar apenas uma linha. Tudo após o **#** até o final da linha é considerado um comentário. A seguir podemos ver um exemplo de como usar comentários em Python.

Exemplo: colocando um comentário

```
01 # Isso é um comentário
02 valor = 42  # Isso é um comentário após um comando
```

Docstrings são strings de documentação que fornecem uma explicação mais detalhada sobre módulos, classes, funções e métodos em Python. Eles são cercados por **três aspas** (simples ou duplas) e podem abranger várias linhas.



Um docstring é similar a um comentário, mas permite fornecer uma documentação mais estruturada sobre o código.

Os docstrings são usados para fornecer informações sobre o propósito, parâmetros, retorno e exemplos de uso das estruturas de código. Embora não tenham efeito direto na execução do programa, eles são acessíveis durante o tempo de execução através do atributo especial `__doc__`. A seguir podemos ver um exemplo de como usar um docstring em Python.

Exemplo: criando um docstring

```
01 def soma(a, b):  
02     """  
03     Função para somar dois números.  
04  
05     Args:  
06         a (float): O primeiro número.  
07         b (float): O segundo número.  
08  
09     Returns:  
10         float: A soma dos dois números.  
11     """  
12     return a + b
```

Docstrings permitem criar uma documentação incorporada no próprio código, facilitando a compreensão e o uso de funções e métodos.



Use comentários para explicar o **porquê** e docstrings para explicar o **como** de suas funções e classes.

Comentários e docstrings tornam o código mais fácil de entender e de trabalhar, tanto para você quanto para outros desenvolvedores.

1.3 Trabalhando com módulos



Clique aqui para ver a vídeo aula sobre módulos.

Módulos são arquivos contendo código Python que podem ser reutilizados em diferentes partes de um programa ou em diferentes programas. Eles são uma maneira eficiente de organizar e compartilhar código, permitindo que você divida seu programa em partes menores e mais gerenciáveis.



Os módulos ajudam a evitar a duplicação de código, promovendo a modularidade e a reutilização.

A medida que um programa cresce em tamanho e complexidade, um ou mais módulos são utilizados de forma combinada. Um módulo é criado ao definir um arquivo com extensão **".py"** contendo código Python. O nome do arquivo é o nome do módulo, e ele pode conter variáveis, funções, classes e até mesmo docstrings.



Para usar o código de um módulo em outro arquivo Python, você precisa importá-lo.

Isso é feito usando a palavra-chave **import**, seguida pelo nome do módulo (sem a extensão **".py"**), como mostrado a seguir:

```
import nome_modulo
```

Você também pode renomear um módulo durante a importação usando a palavra-chave **as**:

```
import nome_modulo as novo_nome
```

Isso pode ser útil para evitar conflitos de nome ou para criar apelidos mais curtos. Uma vez que o módulo é importado, você pode acessar seus elementos usando a notação de ponto (**nome_modulo.elemento** ou **novo_nome.elemento**). A seguir podemos um exemplo de importação de módulo.

Exemplo: trabalhando com módulos

```
01 import math
02 import random as rd
03
04 # imprime o valor de PI
05 print(math.pi)
06 # imprime o valor de RAIZ de 2
07 print(math.sqrt(2))
08 # gera um número aleatório entre 0 e 10
09 print(rd.randint(0,10))
```

```
Saída 3.141592653589793
      1.4142135623730951
      6
```

2 Manipulando dados em Python

2.1 Variáveis



Clique aqui para ver a vídeo aula sobre variáveis em Python.

Uma variável é um nome que se refere a um valor na memória do computador. Elas são como rótulos que você atribui a valores para facilitar o acesso e manipulação desses valores em seu programa. As variáveis permitem que você armazene dados de diferentes tipos (como números, textos, listas, etc.) e os utilize ao longo do programa.



As variáveis são usadas para armazenar e gerenciar dados, permitindo que você manipule informações de maneira dinâmica.

2.1.1 Declarando uma variável

A declaração de variáveis em Python possui algumas diferenças significativas devido às características das linguagens. Em várias linguagens, como a linguagem C, você precisa declarar explicitamente o tipo de uma variável antes de usá-la. Isso é importante para que o compilador saiba quanto espaço na memória alocar para essa variável.



Em Python, você não precisa declarar explicitamente o tipo de uma variável.

Isso ocorre porque Python é uma linguagem de **tipagem dinâmica**, o que significa que o tipo da variável é determinado em tempo de execução. A declaração de uma variável em Python segue a sintaxe:

```
nome_da_variavel = valor
```

Em Python, você pode criar uma variável simplesmente atribuindo um valor a ela. O sinal de igual (=) é usado para atribuir um valor a uma variável.



O termo **valor** pode ser qualquer combinação de valores, variáveis ou chamadas de funções utilizando (ou não) os operadores matemáticos.

A seguir podemos ver alguns exemplos de criação de variáveis.

Exemplo: variáveis

```
01 # Variável inteira
02 idade = 25
03
04 # Variável de ponto flutuante
05 altura = 1.75
06
07 # Variável de caractere
08 letra = 'A'
```



Variáveis definidas fora de qualquer função (Capítulo 9) são chamadas de **variáveis globais**.

A variável definida no programa e fora de qualquer função, como vem sendo até o momento, é chamada de variável global. Ela pode ser acessada em qualquer lugar do programa, inclusive dentro de uma função. Seu tempo de vida é o tempo de execução do programa.

2.1.2 Nomes de variáveis

Dentro do nosso programa, as variáveis são identificadas por um nome.



Algumas palavras, ou combinações de caracteres, não podem ser usadas como um nome de variável.

Existem algumas regras para a escolha dos nomes das variáveis na linguagem Python:

- O nome de uma variável é um conjunto de caracteres que podem ser letras, números ou underscores “_”;
- O nome de uma variável deve sempre iniciar com uma letra ou o underscore “_”;
- A linguagem Python é **case-sensitive**: letras maiúsculas e minúsculas são consideradas diferentes;



Palavras reservadas não podem ser usadas como nomes.

As palavras reservadas são um conjunto de 33 palavras que formam a sintaxe da linguagem Python. Essas palavras já possuem funções específicas e não podem ser utilizadas para outro fim. A seguir podemos ver a lista dessas palavras reservadas.

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

2.1.3 Os tipos das variáveis

O tipo da variável define os valores que a variável pode assumir e as operações que podem ser realizadas com ela. Em Python, você não precisa declarar explicitamente o tipo de uma variável. Isso pode levar algumas pessoas a achar que a linguagem não possui tipos.



Python é uma linguagem de **tipagem dinâmica**. Isso significa que o tipo da variável é determinado em tempo de execução, sempre que uma atribuição é executada.

Python possui diversos tipos de dados nativos que você pode usar para armazenar diferentes tipos de informações. Alguns dos principais tipos incluem:

- **Inteiro (int)**: Armazena números inteiros, como 1, -5, 1000, etc.
- **Número de Ponto Flutuante (float)**: Armazena números com casas decimais, como 3.14, -0.5, 2.0, etc.
- **String (str)**: Armazena textos, como "Olá, mundo!" ou "Python é incrível!".
- **Booleano (bool)**: Armazena valores de verdadeiro (**True**) ou falso (**False**), usados em expressões condicionais.
- **Lista (list)**: Armazena uma coleção ordenada de elementos.
- **Tupla (tuple)**: Similar a uma lista, mas imutável (não pode ser alterada após a criação).
- **Dicionário (dict)**: Armazena pares de chave-valor para mapeamento.



O tipo da variável se altera conforme o dado armazenado.

Sempre que fazemos uma atribuição, a variável assume o tipo do dado que ela recebeu. Para saber o tipo da variável usamos o comando **type(x)**, como mostra o exemplo a seguir.

Exemplo: tipos das variáveis

```
01 idade = 25
02 altura = 1.75
03 letra = 'A'
04
05 type(idade)
06 type(altura)
07 type(letra)
```

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

2.1.4 Conversão de tipos

Às vezes, você pode precisar converter um tipo de dado para outro. Por exemplo, converter um número em uma string ou uma string em um número. Isso é conhecido como conversão de tipo. Python oferece várias funções embutidas para realizar essas conversões. Aqui estão algumas delas:

- `int()`: converte um valor para inteiro via truncagem (apenas a parte inteira é considerada).
- `float()`: converte x para em número de ponto flutuante.
- `str()`: converte um valor em texto (string).



Clique aqui para ver a vídeo aula sobre conversão de tipos.

2.2 Escrevendo as variáveis



Clique aqui para ver a vídeo aula sobre a função `print()`.

Para exibir informações na saída padrão (geralmente no console ou terminal), você pode usar a função **`print()`**. A sintaxe básica da função **`print()`** é a seguinte:

```
print(argumento1)
```

ou

```
print(argumento1, argumento2, ...)
```


A função **print()** aceita um ou mais argumentos, separados por vírgulas, e os exibe separados por espaço, seguido de uma quebra de linha. Cada argumento é um nome de variável ou texto a ser escrito em tela, como mostra o exemplo a seguir.

Exemplo: usando a função print()
<pre>01 idade = 25 02 print('Olá, mundo!') 03 print('idade:', idade)</pre>
Saída Olá, mundo! idade: 25

Nesse exemplo, a função **print()** foi usada para escrever um texto simples, mas também para exibir um texto seguido pelo valor de uma variável.



Outra forma de combinar texto e valores formatados é usando **strings literais formatadas**.

Strings literais formatadas são um recurso novo da versão Python 3.6 e que simplificam a forma como formatamos valores. Para usar esse recurso, comece uma string com **f** ou **F** antes de abrir as aspas (simples ou duplas). Agora, ao invés de separar as variáveis com vírgulas, podemos especificar o nome da variável a ser escrita entre chaves, como mostra o exemplo a seguir.

Exemplo: usando strings literais formatadas.
<pre>01 idade = 25 02 altura = 1.76 03 print(f'A altura é {altura} e a idade é {idade}')</pre>
Saída A altura é 1.76 e a idade é 25

Por padrão, a função **print()** exibe os argumentos separados por espaço, seguido de uma quebra de linha. No entanto, podemos configurar esses atributos alterando os parâmetros **sep** e **end** da função **print()**:

```
print(argumento1, sep=separador, end=end)
```

ou

```
print(argumento1, argumento2, ..., sep=separador, end=end)
```

Basicamente, o parâmetro

- **sep**: define o separador a ser utilizado. Por padrão, é um espaço em branco.
- **end**: define o que deve ser impresso ao final da função **print()**. Por padrão, é uma quebra de linha.

A seguir podemos ver um exemplo.

Exemplo: configurando a função print()
<pre>01 idade = 25 02 altura = 1.76 03 print('A altura é ',altura,' e a idade é ',idade) 04 print('A altura é ',altura,' e a idade é ',idade,sep=' ')</pre>
Saída A altura é 1.76 e a idade é 25 A altura é 1.76 e a idade é 25

2.3 Lendo as variáveis



Clique aqui para ver a vídeo aula sobre a função `input()`.

Para receber dados do usuário, você pode usar a função **`input()`**. Ela permite que o usuário insira dados a partir do teclado. A sintaxe básica da função **`input()`** é a seguinte:

```
variável = input()
```

```
variável = input('texto a ser escrito')
```

A função **`input()`** faz a leitura de um texto (string) digitado pelo usuário e o armazena na variável especificada. O programador pode, ou não, especificar um texto a ser exibido para o usuário.



A função **`input()`** retorna a entrada do usuário como uma **string**.

A função **`input()`** sempre retorna uma cadeia de caracteres, mesmo que o que foi digitado contenha apenas números. Se for necessário ler um valor numérico, a melhor solução é forçar a conversão de tipos, como mostrada na Seção 2.1.4. Desse modo, podemos combinar o retorno da função **`input()`** com a conversão de tipos para fazer a leitura de valores numéricos, como mostra o exemplo a seguir.

Exemplo: usando a função input()
<pre>01 # lê uma string 02 nome = input("Digite seu nome: ") 03 # lê e converte para inteiro 04 idade = int(input("Digite sua idade: ")) 05 06 print(f'Seu nome é {nome} e sua idade é {idade}')</pre>

Saída Digite seu nome: Andre Digite sua idade: 42 Seu nome é Andre e sua idade é 42

3 Realizando operações com as variáveis

3.1 Operadores aritméticos



Clique aqui para ver a vídeo aula sobre operadores aritméticos.

Os operadores aritméticos são aqueles que operam sobre valores numéricos (**valores**, **variáveis** ou **chamadas de funções**) e/ou expressões e tem como resultado valores numéricos. A linguagem Python possui um total de sete operadores aritméticos, como mostra a tabela a seguir.

Operador	Significado	Exemplo
+	adição de dois valores	$z = x + y$
-	subtração de dois valores	$z = x - y$
*	multiplicação de dois valores	$z = x * y$
/	quociente de dois valores	$z = x / y$
**	exponenciação de dois valores	$z = x ** y$
//	Quociente da divisão inteira	$z = x // y$
%	resto de uma divisão inteira	$z = x \% y$

Note que os operadores aritméticos são sempre usados em conjunto com o operador de atribuição. Afinal de contas, alguém precisa receber o resultado da expressão aritmética. A seguir podemos ver alguns exemplos de uso desses operadores.

Exemplo: operadores aritméticos

```
01 x = 10
02 y = 20
03 z = x * y
04 print("z = ", z)
05
06 z = y/10
07 print("z = ", z)
08
09 print("x+y = ", x+y)
```

```
Saída z = 200
      z = 2.0
      x+y = 30
```

Note, no exemplo acima, que podemos devolver o resultado de uma expressão aritmética para uma outra variável (linhas 4 e 7), ou para um outro comando ou função que espere receber um valor do mesmo tipo do resultado da operação, no caso, a função **print()** (linha 10).



Em uma expressão, as operações de **exponenciação**, **multiplicação**, **divisão**, e **resto** são executados antes das operações de **adição** e **subtração**. Para forçar uma operação a ser executada antes das demais, colocamos ela entre **(parênteses)**.

Considere a expressão

$$z = x * y + 10$$

Nela, o valor de **x** será multiplicado pelo valor de **y**, e o resultado dessa multiplicação será somado ao valor **10** para só então ser atribuído a variável **z**. Para a operação de adição ser executada antes da de multiplicação, basta colocá-la entre **parêntese**. Assim, na expressão

$$z = x * (y + 10)$$

o valor de **y** será somado ao valor **10** e o resultado dessa adição será multiplicado pelo valor de **x** para só então ser atribuído a variável **z**.



O operador de subtração também pode ser utilizado para inverter o sinal de um número.

De modo geral, os operadores aritméticos são operadores binários, ou seja, atuam sobre dois valores. Mas os operadores de adição e subtração também podem ser aplicados sobre um único valor. Nesse caso, eles são chamados de operadores unários. Por exemplo, na expressão:

$$x = -y$$

a variável **x** receberá o valor de **y** multiplicado por **-1**, ou seja, **x = (-1) * y**.

3.2 Operadores relacionais



Clique aqui para ver a vídeo aula sobre operadores relacionais.

Os operadores relacionais são aqueles que operam sobre dois valores (**valores**, **variáveis** ou **chamadas de funções**) e/ou expressões aritméticas e verificam a magnitude (quem é maior ou menor) e/ou igualdade entre eles.



Os operadores relacionais são operadores de comparação de valores.

A linguagem Python possui um total de seis operadores relacionais, como mostra a tabela a seguir.

Operador	Significado	Exemplo
>	Maior do que	<code>x > 5</code>
>=	Maior ou igual a	<code>x >= 10</code>
<	Menor do que	<code>x < 5</code>
<=	Menor ou igual a	<code>x <= 10</code>
==	Igual a	<code>x == 0</code>
!=	Diferente de	<code>x != 0</code>

Como resultado, esse tipo de operador retorna:

- o valor **True**, se a expressão relacional for considerada **verdadeira**;
- o valor **False**, se a expressão relacional for considerada **falsa**.

O exemplo a seguir apresenta o resultado de algumas expressões relacionais.

Exemplo: expressões relacionais	
<pre>01 x = 5 02 y = 3 03 04 print("Expressão 1: ", x > 4) 05 print("Expressão 2: ", x == 4) 06 print("Expressão 3: ", x != y) 07 print("Expressão 4: ", x != y+2)</pre>	
Saída	<pre>Expressão 1: True Expressão 2: False Expressão 3: True Expressão 4: False</pre>

3.3 Operadores lógicos



Clique aqui para ver a vídeo aula sobre operadores lógicos.

Certas situações não podem ser modeladas apenas utilizando os operadores aritméticos e/ou relacionais. Um exemplo bastante simples disso é saber se uma variável x está dentro de uma faixa de valores. Por exemplo, a expressão matemática

$$0 < x < 10$$

indica que o valor de x deve ser maior do que 0 (zero) e também menor do que 10.

Para modelar esse tipo de situação, a linguagem Python possui um conjunto de 3 operadores lógicos, como mostra a tabela a seguir.

Operador	Significado	Exemplo
and	Operador “E”	x == 5 and x < y
or	Operador “OU”	x != 5 or x < 0
not	Operador “NEGAÇÃO”	not (x > y)

Esses operadores representam situações lógicas, unindo duas ou mais expressões relacionais simples numa composta:

- Operador **E (and)**: o resultado é **True** apenas se **ambas** as expressões unidas por esse operador também forem. Por exemplo, a expressão (x == 5 and x < y) será verdadeira somente se ambas expressões forem verdadeiras;
- Operador **OU (or)**: o resultado é **True** se **alguma** das expressões unidas por esse operador também for. Por exemplo, a expressão (x != 5 or x < 0) será verdadeira se uma de suas duas expressões for verdadeira;
- Operador **NEGAÇÃO (not)**: inverte o valor lógico da expressão a qual se aplica. Por exemplo, a expressão **not**(x > y) se transforma em (x <= y).

O exemplo a seguir mostra o resultado de algumas expressões lógicas.

Exemplo: expressões lógicas	
<pre>01 x = 5 02 y = 3 03 r = (x > 2) and (y < x) 04 print("Resultado: ",r) 05 06 r = (x%2==0) and (y > 0) 07 print("Resultado: ",r) 08 09 r = (x > 2) or (y > x) 10 print("Resultado: ",r) 11 12 r = (x%2==0) or (y < 0) 13 print("Resultado: ",r) 14 15 r = not(x > 2) 16 print("Resultado: ",r) 17 18 r = not(x > 7) and (x > y) 19 print("Resultado: ",r)</pre>	
Saída	Resultado: True
	Resultado: False
	Resultado: True
	Resultado: False
	Resultado: False
	Resultado: True

A seguir é apresentada a **tabela verdade**, onde os termos *A* e *B* representam o resultado de duas expressões relacionais.

Tabela Verdade					
A	B	not(A)	not(B)	A and B	A or B
False	False	True	True	False	False
False	True	True	False	False	True
True	False	False	True	False	True
True	True	False	False	True	True

3.4 Operadores bit-a-bit



Clique aqui para ver a vídeo aula sobre operadores bit-a-bit.

A linguagem Python permite que se faça operações “bit-a-bit” em valores inteiros. Na memória do computador um valor inteiro é sempre representado por sua forma binária. Assim o número **44** é representado pelo seguinte conjunto de 0's e 1's na memória: **00101100**. Os operadores bit-a-bit permitem que o programador faça operações em cada bit do número de maneira direta.



Os operadores bit-a-bit ajudam os programadores que queiram trabalhar com o computador em “baixo nível”.

A linguagem Python possui um total de seis operadores bit-a-bit, como mostra a tabela a seguir.

Operador	Significado	Exemplo
&	E bit-a-bit	<code>x = y & z</code>
	OU bit-a-bit	<code>x = y z</code>
^	OU exclusivo	<code>x = y ^ z</code>
~	complemento bit-a-bit	<code>x = y ~ z</code>
<<	deslocamento de bits à esquerda	<code>x = y << z</code>
>>	deslocamento de bits à direita	<code>x = y >> z</code>

Os operadores ~, &, |, e ^, são operações lógicas que atuam em cada um dos bits do número (por isso, bit-a-bit). Já os operadores de deslocamento, << e >>, servem para rotacionar o conjunto de bits do número à esquerda ou à direita.



Os operadores bit-a-bit só podem ser usados nos tipos inteiros.

Os operadores bit-a-bit não podem ser aplicados sobre valores do tipo **float**. Isso se deve a maneira como um valor **ponto flutuante** é representado nos computadores. A representação desses tipos segue a representação criada por Konrad Zuse, onde um número é dividido numa *mantissa* (**M**) e um *expoente* (**E**). O valor representado é obtido pelo produto: $M * 2^E$. Como se vê, a representação desses tipos é bem mais complexa: não se trata de apenas um conjunto de 0's e 1's na memória.

3.5 Operadores de atribuição simplificada



Clique aqui para ver a vídeo aula sobre operadores de atribuição simplificada.

Como vimos anteriormente, muitos operadores são sempre usados em conjunto com o operador de atribuição. Para tornar essa tarefa mais simples, a linguagem Python permite simplificar algumas expressões, como mostra a tabela a seguir.

Operador	Significado		Exemplo
<code>+=</code>	soma e atribui	<code>x += y</code>	igual <code>x = x + y</code>
<code>-=</code>	subtrai e atribui	<code>x -= y</code>	igual <code>x = x - y</code>
<code>*=</code>	multiplica e atribui	<code>x *= y</code>	igual <code>x = x * y</code>
<code>/=</code>	divide e atribui quociente	<code>x /= y</code>	igual <code>x = x / y</code>
<code>**=</code>	exponenciação e atribui	<code>x **= y</code>	igual <code>x = x ** y</code>
<code>//=</code>	quociente da divisão e atribui	<code>x //= y</code>	igual <code>x = x // y</code>
<code>%=</code>	divide e atribui resto	<code>x %= y</code>	igual <code>x = x % y</code>
<code>&=</code>	E bit-a-bit e atribui	<code>x &= y</code>	igual <code>x = x & y</code>
<code> =</code>	OU bit-a-bit e atribui	<code>x = y</code>	igual <code>x = x y</code>
<code>^=</code>	OU exclusivo e atribui	<code>x ^= y</code>	igual <code>x = x ^ y</code>
<code><<=</code>	desloca à esquerda e atribui	<code>x <<= y</code>	igual <code>x = x << y</code>
<code>>>=</code>	desloca à direita e atribui	<code>x >>= y</code>	igual <code>x = x >> y</code>

A seguir podemos ver alguns exemplos desses operadores de atribuição simplificada.

Exemplo: operadores de atribuição simplificada	
Sem Operador	Com Operador
01 x = 10	x = 10
02 y = 20	y = 20
03	
04 x = x + y - 10	x += y - 10
05 print("x = ", x)	print("x = ", x)
06	
07 x = x - 5	x -= 5
08 print("x = ", x)	print("x = ", x)
09	
10 x = x * 10	x *= 10
11 print("x = ", x)	print("x = ", x)
12	
13 x = x / 15	x /= 15
14 print("x = ", x)	print("x = ", x)

Como se pode notar, esse tipo de operador é muito útil quando a variável que vai receber o resultado da expressão é também um dos operandos da expressão. Por exemplo, a expressão

```
x = x + y - 10;
```

pode ser reescrita usando o operador simplificado como sendo

```
x += y - 10;
```



Apesar de útil, devemos tomar cuidado com esse tipo de operador. Principalmente quando unimos numa mesma expressão operadores com diferentes precedências.

Algumas simplificações podem mudar o sentido da expressão original devido a questão da precedência (por exemplo, multiplicações e divisões são sempre realizadas antes de somas e subtrações), como mostra o exemplo abaixo:

Exemplo: precedência na atribuição simplificada	
Sem Operador	Com Operador
01 x = 10	x = 10
02 y = 20	y = 20
03	
04 x = x * y - 10	x *= y - 10
05 print("x = ", x)	print("x = ", x)
06	
07 x = x - 5 + y	x -= 5 + y
08 print("x = ", x)	print("x = ", x)
Saída x = 190	x = 100
x = 205	x = 75

No exemplo anterior, é fácil supor que a expressão $x = x * y - 10$ será simplificada como sendo $x *= y - 10$. Porém, trata-se de um erro. O operador simplificado atua sobre o resultado da expressão seguinte a ele. Assim,

$x *= y - 10$ equivale a $x = x * (y - 10)$ e não a $x = x * y - 10$
 $x -= 5 + y$ equivale a $x = x - (5 + y)$ e não a $x = x - 5 + y$

3.6 Precedência de operadores

Como podemos ver, a linguagem Python contém muitos operadores e o uso de vários desses operadores em uma única expressão pode tornar confusa a sua interpretação. Por esse motivo, a linguagem Python possui uma série de regras de precedência de operadores.

A precedência de operadores em Python determina a ordem em que os operadores são avaliados em uma expressão. Quando uma expressão contém vários operadores, a precedência define qual operador será avaliado primeiro, qual será o próximo e assim por diante.

A tabela a seguir lista os principais grupos de operadores em ordem de precedência, do mais alto para o mais baixo. Quanto mais alto na tabela, maior o nível de precedência (prioridade) dos operadores em questão.

MAIOR PRECEDÊNCIA	
()	Dentro dos parênteses é sempre avaliado primeiro
[]	Elemento de lista, acesso a índice
.	Elemento de objeto, acesso a atributo
**	Potência
~x	Complemento bit-a-bit
+x, -x	Adição e subtração unária
*, /, %	Multiplicação, divisão e resto
+, -	Adição e subtração
<<, >>	Deslocamento de bits à esquerda e à direita
&	E bit-a-bit
^	OU exclusivo
	OU bit-a-bit
in, not in, is, is not, <, <=, >, >=, ==, !=	Operadores de comparação
not x	NÃO lógico
and	E lógico
or	OU lógico
=	Atribuição
MENOR PRECEDÊNCIA	

É possível notar que alguns operadores ainda são desconhecidos para nós. Esses operadores serão explicados ao longo dos próximos capítulos.

4 Comandos de controle condicional

4.1 Definindo uma condição

A medida que nosso programa cresce em complexidade, há a necessidade de executar um conjunto de comandos apenas se uma determinada condição for verdadeira.



Uma **condição** é uma expressão que resulte numa resposta do tipo **verdadeiro** ou **falso**.

Por exemplo, para a condição $x > 0$ temos que:

- Se o valor de x for um valor **POSITIVO**, a condição será considerada **verdadeira**;
- Se o valor de x igual a **ZERO** ou **NEGATIVO**, a condição será considerada **falsa**.



Uma **condição** pode ser construída utilizando operadores matemáticos, relacionais e lógicos.

Esses operadores permitem criar condições mais complexas, como mostra o exemplo a seguir:

$$x/2 > y - 3$$

Nessa condição, se deseja saber se a divisão de x por 2 é maior do que o valor de y menos o valor 3. A seguir podemos ver mais alguns exemplos de condição.

Exemplo: expressões condicionais

```
01 x é maior ou igual a y?  
02   x >= y  
03  
04 x é maior do que y+2?  
05   x > y+2  
06  
07 x-5 é diferente de y+3?  
08   x-5 != y+3  
09  
10 x é maior do que y e menor do que z?  
11   (x > y) && (x < z)
```

Quando o compilador avalia uma condição, ele quer um valor de retorno (**verdadeiro** ou **falso**) para poder tomar a decisão.

4.2 O comando if

Na linguagem Python, o comando **if** é utilizado quando queremos executar um ou mais comandos que estejam sujeitos ao resultado de um teste baseado em uma condição.



Clique aqui para ver a vídeo aula sobre o comando if.

A forma geral de um comando **if** é:

```
if condição:
    comando 1
    comando 2
    ...
    comando N
continuação do programa
```

Na execução do comando **if** a condição será avaliada e:

- se a condição for **verdadeira (True)** a sequência de comandos será executada;
- se a condição for **falsa (False)** a sequência de comandos não será executada, e o programa irá continuar a partir do primeiro comando seguinte ao final do comando **if**.



Para um comando fazer parte do bloco de comandos do **if**, ele precisar ser indentado. Caso contrário, o comando **if** não irá considerar aquele comando.

Diferente de outras linguagens que usam chaves {} para delimitar blocos de comandos, a linguagem Python usa a indentação para essa tarefa.

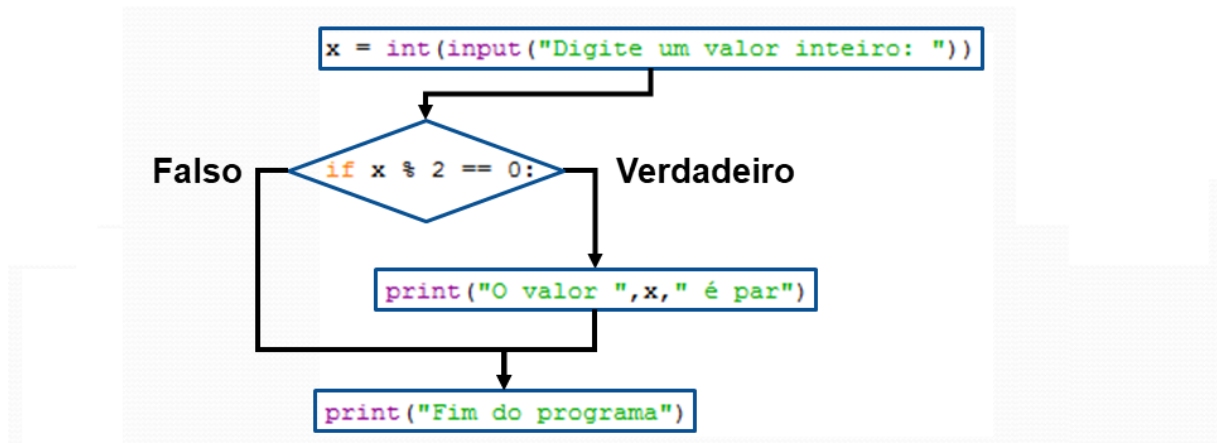
A seguir, tem-se um exemplo de um programa que lê um número inteiro digitado pelo usuário e informa se o mesmo é par.

Exemplo: comando if

```
01 x = int(input("Digite um valor inteiro: "))
02 if x % 2 == 0:
03     print("O valor de ",x," é par")
04
05 print("Fim do programa")
```

Nesse exemplo, a mensagem de que o número é **par** será exibida apenas se a condição for verdadeira. Se a condição for falsa, apenas a mensagem seguinte ao

comando **if** será escrita na tela. Relembrando a ideia de fluxogramas, é possível ter uma boa representação de como os comandos desse exemplo são um-a-um executados durante a execução do programa.



4.3 O comando else



Clique aqui para ver a vídeo aula sobre o comando else.

O comando **else** pode ser entendido como sendo um complemento do comando **if**. Ele auxilia o comando **if** na tarefa de escolher dentre os vários caminhos a serem executados dentro do programa.



O comando **else** é opcional e sua sequência de comandos somente será executada se o valor da condição que está sendo testada pelo comando **if** for **FALSA**.

A forma geral de um comando **else** é:

```
if condição:
    comando 1
    comando 2
    ...
    comando N
else:
    comando 1
    comando 2
    ...
    comando N
continuação do programa
```



Se o comando **if** diz o que fazer quando a condição é **verdadeira**, o comando **else** trata da condição quando ela é **falsa**.

Antes, na execução do comando **if**, a condição era avaliada e:

- se a condição fosse **verdadeira (True)**, a **primeira** sequência de comandos era executada;
- se a condição fosse **falsa (False)**, a sequência de comandos não era executada e o programa seguia o seu fluxo padrão.

Com o comando **else**, temos agora que:

- se a condição for **verdadeira (True)**, a **primeira** sequência de comandos (bloco **if**) será executada;
- se a condição for **falsa (False)**, a **segunda** sequência de comandos (bloco **else**) será executada.



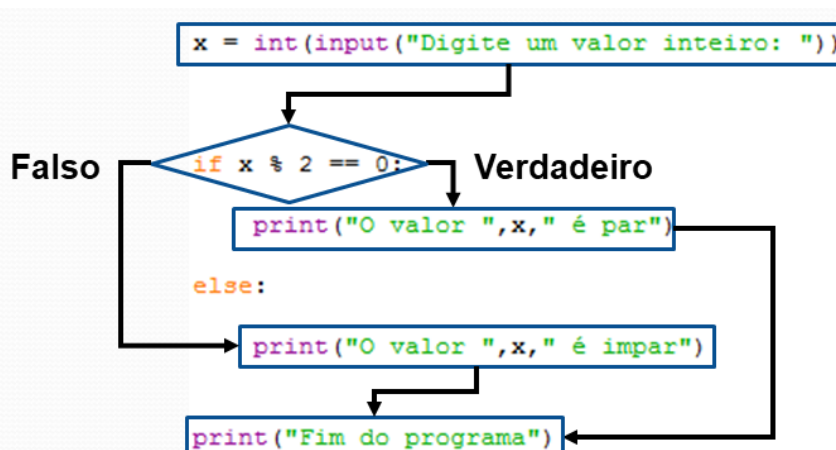
Assim como no comando **if**, é a endentação que define se um comando faz parte do **else**.

A seguir, tem-se um exemplo de um programa que lê um número inteiro digitado pelo usuário e informa se o mesmo é ou não igual a 10.

Exemplo: comando if-else

```
01 x = int(input("Digite um valor inteiro: "))
02 if x % 2 == 0:
03     print("O valor de ",x," é par")
04 else:
05     print("O valor de ",x," é impar")
06
07 print("Fim do programa")
```

O funcionamento desse programa fica mais claro se fizermos o seu fluxograma, como mostra a figura a seguir.





O comando **else** não tem condição. Ele é o caso contrário da condição do **if**.

O comando **else** é complemento do comando **if** e indica quais comandos devem ser executados se a condição do comando **if** for **falsa**. Portanto, não é necessário estabelecer uma nova condição para o comando **else**. Tentar colocar uma condição no **else** irá gerar um erro no programa, como mostra o exemplo a seguir.

Exemplo: comando if-else

```
01 x = int(input("Digite um valor inteiro: "))
02 if x % 2 == 0:
03     print("O valor de \",x,\" é par")
04 else x % 2 != 0: % ERRADO!
05     print("O valor de \",x,\" é impar")
06
07 print("Fim do programa")
```

4.4 Aninhamento de if



[Clique aqui para ver a vídeo aula sobre aninhamento de if.](#)

Um **if** aninhado é simplesmente um comando **if** utilizado dentro do bloco de comandos de um outro **if** (ou **else**) mais externo. Basicamente, é um comando **if** dentro de outro.

A forma geral de um comando **if** aninhado é:

```
if condição_1:
    comando 1
    ...
    comando N
else:
    if condição_2:
        comando 1
        ...
        comando N
    else:
        comando 1
        ...
        comando N
continuação do programa
```

Em um aninhamento de **if's**, o programa começa a testar as condições começando pela **condição_1**. Se a condição for verdadeira (**True**), o programa executará o bloco de comando associados a ela. Caso contrário, irá executar o bloco de comando associados ao comando **else** correspondente, se ele existir. Esse processo se repete para cada comando **if** que o programa encontrar dentro do bloco de comando que ele executar.



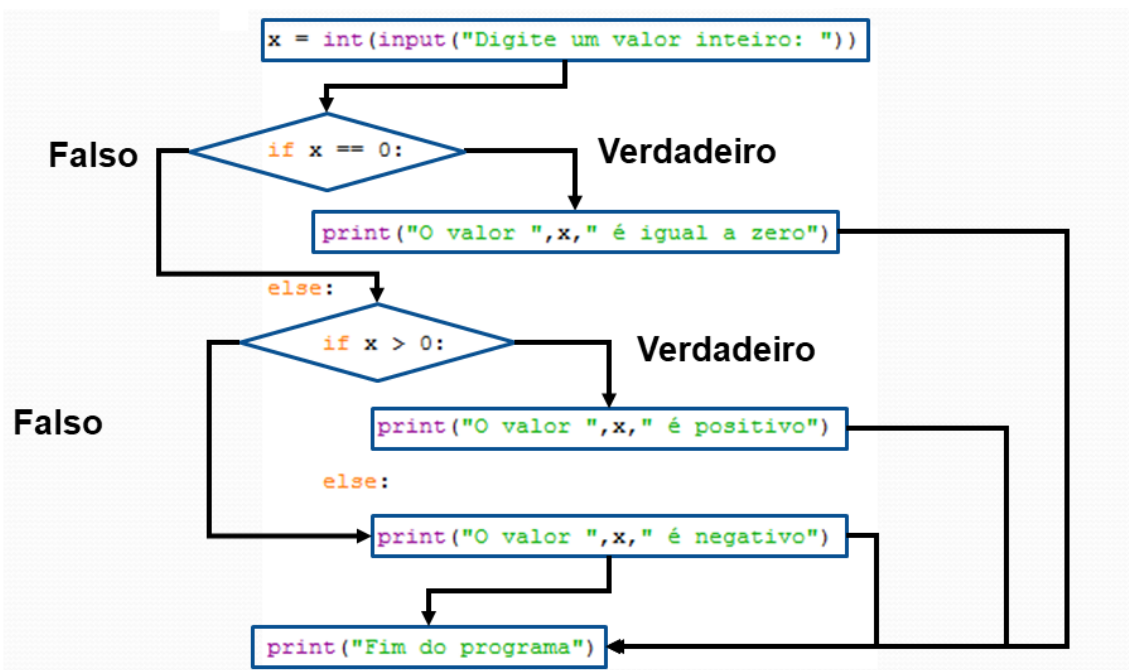
O aninhamento de **if's** é muito útil quando se tem mais do que dois caminhos (ou duas possibilidades) para executar dentro de um programa.

Por exemplo, o comando **if** é suficiente para dizer se um número é maior (ou não) do que outro número. Sozinho ele é incapaz de dizer se esse mesmo número é maior, menor ou igual a outro, como mostra o exemplo abaixo:

Exemplo: aninhamento de if

```
01 x = int(input("Digite um valor inteiro: "))
02 if x == 0:
03     print("O valor de ",x," é igual a zero")
04 else:
05     if x > 0:
06         print("O valor de ",x," é positivo")
07     else:
08         print("O valor de ",x," é negativo")
09
10 print("Fim do programa")
```

Isso fica mais claro quando olhamos a representação do aninhamento de **if's** em um fluxograma, como mostra a figura a seguir.





Não existe aninhamento de **else**'s.

O comando **else** é o caso contrário da condição do comando **if**. Para cada comando **else** deve existir um **if** anterior, porém nem todo **if** precisa ter um **else**:

Exemplo: erro no aninhamento de if

```
01 if condição:
02     sequência de comandos
03 else:
04     sequência de comandos
05 else: %ERRO!
06     sequência de comandos
```

4.5 O comando elif



Clique aqui para ver a vídeo aula sobre o comando elif.

O comando **elif** é uma abreviação de "**else if**" e é usado para verificar condições adicionais após um "**if**" ou uma série de "**if**" consecutivos. Ele permite que você teste várias condições diferentes e execute um bloco de código correspondente à primeira condição que for verdadeira, como mostra o exemplo a seguir.

Sem elif	Com elif
<pre>if condição_1: comando 1 ... comando N else: if condição_2: comando 1 ... comando N else: comando 1 ... comando N continuação do programa</pre>	<pre>if condição_1: comando 1 ... comando N elif condição_2: comando 1 ... comando N else: comando 1 ... comando N continuação do programa</pre>



O comando **elif** pode ser entendido como sendo uma simplificação do aninhamento de um **if** dentro de um **else**.

A seguir podemos ver um exemplo de aninhamento de **if-else** sem usar o **elif**.

Exemplo: aninhamento de if-else sem usar o comando elif

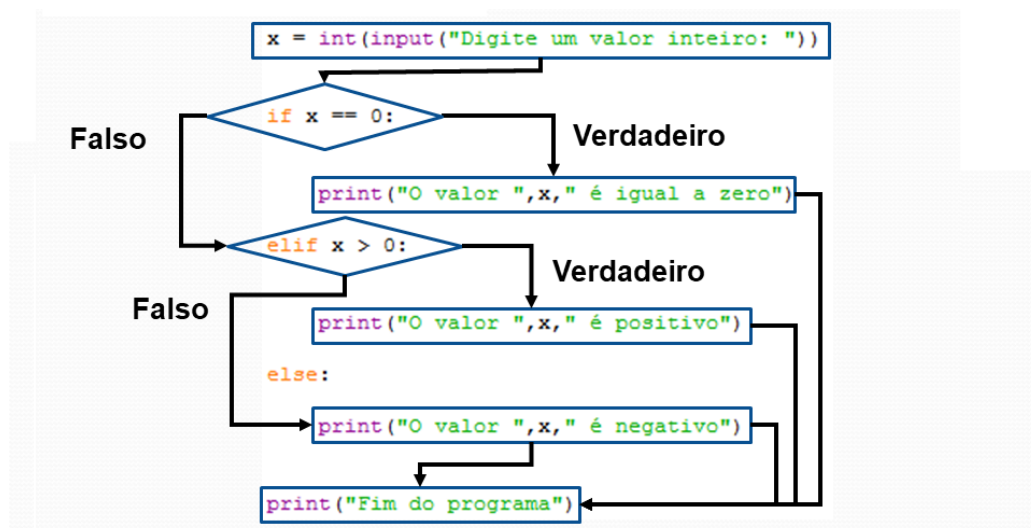
```
01 x = int(input("Digite um valor inteiro: "))
02 if x == 0:
03     print("O valor de ",x," é igual a zero")
04 else:
05     if x > 0:
06         print("O valor de ",x," é positivo")
07     else:
08         print("O valor de ",x," é negativo")
09
10 print("Fim do programa")
```

Usando o comando **elif**, o mesmo exemplo ficaria assim.

Exemplo: aninhamento de if-else usando o comando elif

```
01 x = int(input("Digite um valor inteiro: "))
02 if x == 0:
03     print("O valor de ",x," é igual a zero")
04 elif x > 0:
05     print("O valor de ",x," é positivo")
06 else:
07     print("O valor de ",x," é negativo")
08
09 print("Fim do programa")
```

Se olharmos para a representação em fluxograma, vemos que o **elif** realmente equivale a uma simplificação do aninhamento de um **if** dentro de um **else**, como mostra a figura a seguir.



5 Comandos de repetição

5.1 O comando while



Clique aqui para ver a vídeo aula sobre o comando while.

Na programação, existem situações em que é preciso executar um bloco de comandos mais de uma vez, enquanto uma condição for considerada verdadeira.



Para isso, precisamos de um comando de repetição que permita executar um conjunto de comandos quantas vezes forem necessárias.

Um comando de repetição muito utilizado em programação é o comando **while**. A sua forma geral é:

```
while condição:
    comando 1
    comando 2
    ...
    comando N
continuação do programa
```



De acordo com a condição, os comandos serão repetidos zero (condição falsa) ou mais vezes (enquanto a condição for verdadeira). Essa estrutura normalmente é denominada **laço** ou **loop**.

Na execução do comando **while**, a condição será avaliada e:

- se a condição for considerada **verdadeira (True)**, a sequência de comandos será executada. Ao final da sequência de comandos, o fluxo do programa é desviado novamente para o teste da condição;
- se a condição for considerada **falsa (False)**, a sequência de comandos não será executada.



Para um comando fazer parte do bloco de comandos do **while**, ele precisar ser indentado. Caso contrário, o comando **while** não irá considerar aquele comando.

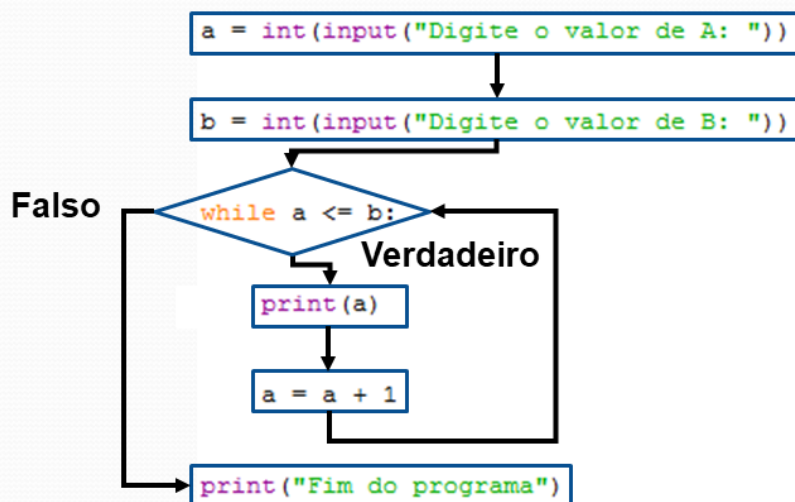
Diferente de outras linguagens que usam chaves {} para delimitar blocos de comandos, a linguagem Python usa a indentação para essa tarefa.

A seguir, tem-se um exemplo de um programa que lê dois números inteiros a e b digitados pelo usuário e imprime na tela todos os números inteiros entre a e b :

Exemplo: comando while

```
01 a = int(input("Digite o valor de A: "))
02 b = int(input("Digite o valor de B: "))
03
04 while a <= b:
05     print(a)
06     a = a + 1
07
08 print("Fim do programa")
```

A figura a seguir apresenta um fluxograma exemplificando como os comandos do exemplo anterior são executados pelo programa.



O comando **while** segue todas as recomendações definidas para o comando **if** quanto a definição da condição usada.

Isso significa que a condição pode ser qualquer expressão que resulte numa resposta do tipo falso (**False**) ou verdadeiro (**True**), e que utiliza operadores dos tipos *matemáticos*, *relacionais* e/ou *lógicos*.



É responsabilidade do programador modificar o valor de algum dos elementos usados na condição para evitar que ocorra um laço infinito.

Um laço infinito (ou **loop** infinito) é uma sequência de comandos em um programa de computador que se repete sem parar. Basicamente, um laço infinito ocorre quando cometemos algum erro ao especificar a condição que controla a repetição.

5.2 O comando for



Clique aqui para ver a vídeo aula sobre o comando for.

O comando **for** em Python é usado para criar um **loop** que percorre uma sequência de elementos, como uma lista, uma string ou um intervalo de números. Ele permite executar um bloco de código repetidamente para cada elemento da sequência.

A forma geral de um comando **for** é:

```
for item in lista_de_elementos:
    comando 1
    comando 2
    ...
    comando N
continuação do programa
```

Na sintaxe do comando **for**, temos que:

- **item** é uma variável que recebe cada elemento da lista em cada iteração do loop;
- **lista_de_elementos** é a sequência de elementos a ser percorrida pelo loop, como uma lista, uma string ou um intervalo.

Durante a execução do loop **for**, o bloco de código dentro dele é repetido para cada **item** da lista. A cada iteração, o elemento atual é atribuído à variável **item**, permitindo que você realize operações ou execute ações específicas com base nesse elemento.



Para um comando fazer parte do bloco de comandos do **while**, ele precisar ser indentado. Caso contrário, o comando **while** não irá considerar aquele comando.

Diferente de outras linguagens que usam chaves **{}** para delimitar blocos de comandos, a linguagem Python usa a indentação para essa tarefa.

A seguir, temos alguns exemplos de uso do comando **for**: um programa que percorre uma lista de valores e imprime a raiz quadrada de cada valor e outro que imprime uma lista de strings.

Exemplo: comando for

```
01 # percorre uma lista de valores e imprime a raiz
02 # quadrada de cada valor
03 import math
04
05 for x in [0,1,2,3,4,5]:
06     print("A raiz de ",x," é igual a ",math.sqrt(x))
07
08
09 # imprime uma lista de strings
10 compras = ["Miojo","Ovo","Leite","Pão"]
11 for item in compras:
12     print("Produto: ",item)
13
```

5.3 A função range()



Clique aqui para ver a vídeo aula sobre a função range().

A função **range()** em Python é usada para gerar uma sequência de números. Ela é comumente usada em loops e iterações, permitindo especificar o início, o fim e o passo da sequência.



A função **range()** permite gerar sequências de valores em progressão aritmética. É muito útil para gerar as listas de valores para o comando **for**.

A sintaxe básica da função **range()** é a seguinte:

range(start, stop, step)

A função recebe até 3 parâmetros de entrada

- **start**: é o número de início da sequência (**opcional**). O valor padrão é 0.
- **stop**: é o número final da sequência (**obrigatório**). O valor final não é incluído na sequência.
- **step**: é o valor de incremento entre os números da sequência (**opcional**). O valor padrão é 1.

Considerando os parâmetros opcionais e obrigatório, a função permite as seguintes formas de uso:

- **range(N)**: gera valores inteiros de **0** até **N-1**
- **range(M,N)**: gera valores inteiros de **M** até **N-1**
- **range(M,N,D)**: gera os valores inteiros **M**, **M+D**, **M+2D**, ... inferiores a **N**.



A função `range()` retorna um objeto do tipo **range**, que pode ser iterado em **loops for** ou convertido em uma lista, se necessário.

A seguir, tem-se alguns exemplos de uso da função **range()** com o laço **for**.

Exemplo: usando a função range()

```
01 # Gerar valores: 0, 1, 2, 3, 4
02 for x in range(5):
03     print(x)
04
05 # Gerar valores: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
06 for x in range(10):
07     print(x)
08
09 # Gerar valores: 3, 4, 5, 6, 7, 8, 9
10 for x in range(3,10):
11     print(x)
12
13 # Gerar valores: 3, 5, 7, 9
14 for x in range(3,10,2):
15     print(x)
16
```

5.4 Diferença entre os comandos while e for



Clique aqui para ver a vídeo aula sobre a diferença entre os comandos **while** e **for**.

Existe uma diferença fundamental no funcionamento dos comandos **while** e **for**. Enquanto o comando **while** permite repetir uma sequência de comandos enquanto uma condição for verdadeira, o comando **for** repete uma sequência de comandos para cada elemento de uma lista.



Isso significa que podemos sempre reescrever um comando **for** com o comando **while**. No entanto, nem sempre podemos reescrever um comando **while** como um comando **for**.

É função do programador se atentar a essa diferença e selecionar o melhor comando para o seu programa.

A seguir podemos ver o mesmo programa implementado usando os comandos **for** e **while**.

Exemplo: for versus while		
	Usando while	Usando for
01	N = int(input("Digite N:"))	N = int(input("Digite N:"))
02	fat = 1	fat = 1
03	i = 1	
04	while (i <= N):	for i in range(1,N+1):
05	fat = fat * i	fat = fat * i
06	i = i + 1	
07		
08	print("Fatorial: ",fat)	print("Fatorial: ",fat)
09		

5.5 Aninhamento de repetições

O aninhamento de repetições em Python refere-se à prática de colocar um loop dentro de outro loop. Essa técnica permite realizar repetições e iterações mais complexas e sofisticadas em seu código. Ao aninhar loops, você pode executar um conjunto de instruções repetidamente dentro de outro conjunto de instruções repetidas.

A forma geral de um comando de repetição aninhado é:

```
loop_1: # loop externo
    # sequência de comandos
    loop_2: # loop interno
        # sequência de comandos
# continuação do programa
```

onde **loop_1** e **loop_2** representam um comando do tipo **while** ou **for**.

Dentro do loop externo, que é executado primeiro, você pode colocar um loop interno, que será executado para cada iteração do loop externo. O número de vezes que o código interno é executado é determinado pela sequência interna, enquanto o número de vezes que o código externo é executado é determinado pela sequência externa.



O aninhamento de comandos de repetição é muito útil quando se tem que percorrer dois conjuntos de valores que estão relacionados dentro de um programa.

Por exemplo, para imprimir uma matriz identidade (composta apenas de 0's e 1's na diagonal principal) de tamanho 4×4 é preciso percorrer as quatro linhas da matriz e, para cada linha, percorrer as suas quatro colunas. Um único comando de repetição não é suficiente para realizar essa tarefa, como mostra o exemplo a seguir.

Exemplo: comandos de repetição aninhados

```
01 for linha in range(4):  
02     for coluna in range(4):  
03         if linha == coluna:  
04             print("1 ", end=" ")  
05         else:  
06             print("0 ", end=" ")  
07     print('')  
08
```

Note, no exemplo anterior, que a impressão de uma matriz identidade pode ser feita com dois comandos **for**, ou dois comandos **while**. É possível ainda fazê-lo usando um comando de cada tipo.



A linguagem Python não proíbe que se misture comandos de repetições de tipos diferentes no aninhamento de repetições.

5.6 O comando break



Clique aqui para ver a vídeo aula sobre o comando break.

O comando **break** serve para interromper a execução de qualquer comando de repetição (**for** ou **while**). O comando **break** faz com que a execução do programa continue na primeira linha seguinte ao laço que está sendo interrompido.



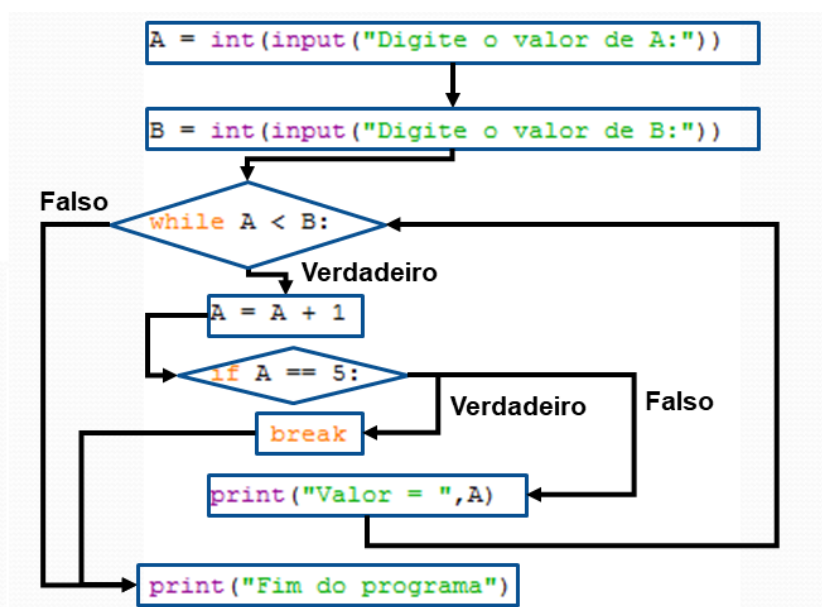
O comando **break** é utilizado para terminar de forma abrupta uma repetição. Por exemplo, se estivermos dentro de uma repetição e um determinado resultado ocorrer, o programa deverá sair da repetição e continuar na primeira linha seguinte a ela.

A seguir, tem-se um exemplo de um programa que lê dois números inteiros a e b digitados pelo usuário e imprime na tela todos os números inteiros entre a e b . Note que, no momento em que o valor de a se torna igual a 5, o comando **break** é executado e o laço termina.

Exemplo: comando break

```
01 A = int(input("Digite o valor de A: "))
02 B = int(input("Digite o valor de B: "))
03
04 while A < B:
05     A = A + 1
06     if A == 5:
07         break
08
09     print("Valor = ",A)
10
11 print("Fim do programa")
```

A figura a seguir apresenta um fluxograma exemplificando como os comandos do exemplo anterior são executados pelo programa.



O comando **break** deverá sempre ser colocado dentro de um comando **if** ou **else** que está dentro da repetição.

Isso ocorre, pois, o comando **break** serve para interromper a execução de qualquer comando de repetição. Porém, esse comando de repetição só deve ser interrompido se um determinado resultado ocorrer. Isso significa que existe uma condição a ser testada com um comando **if** ou **else** antes de chamar o comando **break**. Um

comando **break** colocado dentro da repetição e fora de um comando **if** ou **else** irá SEMPRE terminar a repetição em sua primeira execução.

5.7 O comando continue



Clique aqui para ver a vídeo aula sobre o comando continue.

O comando **continue** é muito parecido com o comando **break**. Tanto o comando **break** quanto o comando **continue** ignoram o restante da sequência de comandos da repetição que os sucedem. A diferença é que, enquanto o comando **break** termina o comando de repetição que está sendo executado, o comando **break** interrompe apenas aquela repetição e passa para a próxima repetição do laço, se ela existir.



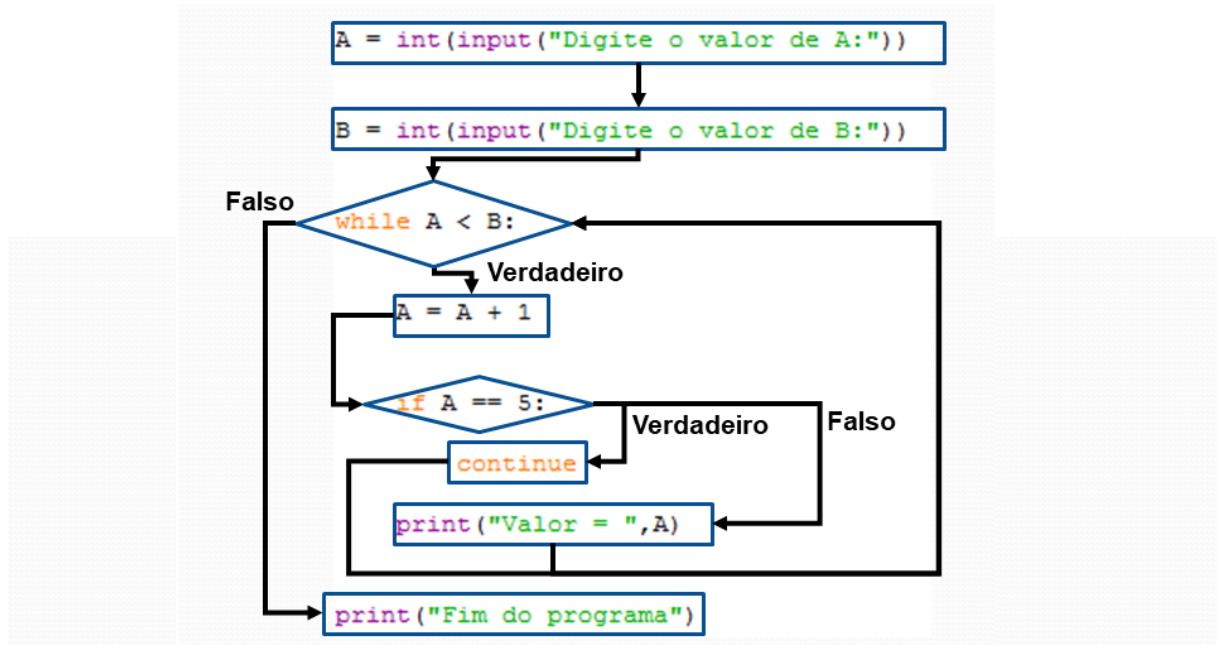
Quando o comando **continue** é executado, os comandos restantes da repetição são ignorados. O programa volta a testar a condição do laço para saber se o mesmo deve ser executado novamente ou não.

A seguir, tem-se um exemplo de um programa que lê dois números inteiros *a* e *b* digitados pelo usuário e imprime na tela todos os números inteiros entre *a* e *b*. Note que, no momento em que o valor de *a* se torna igual a 5, o comando **continue** é executado e essa iteração do laço é ignorada. Consequentemente, o valor 5 não é impresso na tela.

Exemplo: comando continue

```
01 A = int(input("Digite o valor de A: "))
02 B = int(input("Digite o valor de B: "))
03
04 while A < B:
05     A = A + 1
06     if A == 5:
07         continue
08
09     print("Valor = ",A)
10
11 print("Fim do programa")
```

A figura a seguir apresenta um fluxograma exemplificando como os comandos do exemplo anterior são executados pelo programa.



O comando **continue** deverá sempre ser colocado dentro de um comando **if** ou **else** que está dentro da repetição.

Isso ocorre, pois, o comando **continue** serve para ignorar a execução atual de qualquer comando de repetição. Porém, esse comando de repetição só deve ser ignorado se um determinado resultado ocorrer. Isso significa que existe uma condição a ser testada com um comando **if** ou **else** antes de chamar o comando **continue**. Um comando **continue** colocado dentro da repetição e fora de um comando **if** ou **else** irá ignorar TODAS as execuções do comando de repetição, podendo levar até a criação de um LAÇO INFINITO.

6.1 Por que utilizar



Clique aqui para ver a vídeo aula sobre listas.

As variáveis usadas até agora são capazes de armazenar um único valor por vez. Sempre que tentamos armazenar um novo valor dentro de uma variável, o valor antigo é sobrescrito e, portanto, perdido.

Isso pode ser visto claramente no exemplo a seguir.

Exemplo: atribuindo um valor a uma variável

```
01 x = 10
02 print("x = ",x);
03
04 x = 20
05 print("x = ",x);
```

Saída x = 10
x = 20

Isso ocorre por que cada variável representa uma única posição de memória capaz de armazenar apenas um valor do tipo da variável.



Usando apenas variáveis, como seria um programa para ler as notas de cinco alunos e imprimir as notas que são maiores do que a média da turma?

Um algoritmo para resolver esse problema poderia ser o mostrado a seguir.

Exemplo: média das notas de 5 estudantes

```
01 print("Digite a nota de 5 alunos: ");
02 n1 = int(input('Nota do aluno 1: '))
03 n2 = int(input('Nota do aluno 2: '))
04 n3 = int(input('Nota do aluno 3: '))
05 n4 = int(input('Nota do aluno 4: '))
06 n5 = int(input('Nota do aluno 5: '))
07 media = (n1+n2+n3+n4+n5)/5.0;
08
09 if(n1 > media):
10     print("Aprovado: ",n1);
11
```

```

12 if(n2 > media):
13     print("Aprovado: ",n2);
14
15 if(n3 > media):
16     print("Aprovado: ",n3);
17
18 if(n4 > media):
19     print("Aprovado: ",n4);
20
21 if(n5 > media):
22     print("Aprovado: ",n5);

```

O algoritmo anterior apresenta uma solução possível para o problema apresentado. Mas isso apenas se a quantidade de alunos for pequena.



Essa solução é inviável para uma turma de 100 alunos.

O grande inconveniente dessa solução é a grande quantidade de variáveis diferentes que temos que gerenciar e o uso repetitivo de comandos praticamente idênticos. Para 100 alunos, precisamos de:

- Uma variável para armazenar a nota de cada aluno: 100 variáveis
- Um comando de leitura para cada nota: 100 input()
- Um somatório de 100 notas
- Um comando de teste para cada aluno: 100 comandos **if**
- Um comando de impressão na tela para cada aluno: 100 print()

Trata-se de uma solução extremamente engessada, pois implica em reescrever todo o código, repetindo comandos praticamente idênticos.



Como as variáveis da solução têm uma relação entre si (todas representam notas de alunos), podemos declará-las usando um **ÚNICO** nome para todos os 100 alunos.

Uma **lista** é a forma mais simples e comum de dados estruturados da linguagem Python. Voltando ao problema, se usássemos listas, uma solução usando lista poderia ser a mostrada a seguir.

Exemplo: média das notas de 5 estudantes (usando listas)

```

01 N = 5 # número de aluno
02
03 notas = []
04 media = 0
05 print("Digite a nota dos alunos: ");

```



```
06
07 for i in range(N):
08     x = int(input('Nota do aluno 1: '))
09     notas.append(x)
10     media = media + x
11
12 media = media / N
13
14 for x in notas:
15     if(x > media):
16         print("Aprovado: ",x);
```

À primeira vista, temos um código um pouco mais enxuto que o original. Mas a grande vantagem é que agora temos um código mais genérico e flexível. Se quisermos calcular a média de 100 alunos, basta mudar o valor para $N = 100$ (linha 1). Nenhuma outra linha precisa ser alterada, pois o código explora as propriedades e a versatilidade da lista para criar uma solução que independe da quantidade de dados a ser processada.

6.2 Sintaxe e criação de uma lista



Clique aqui para ver a vídeo aula sobre criação de listas.

Em Python, uma **lista** é uma estrutura de dados que permite armazenar um conjunto de elementos em uma única variável. Elas são versáteis e amplamente utilizadas para organizar e manipular coleções de dados. Para criar uma lista em Python, utilizamos colchetes `[]` e separamos os elementos com vírgulas, como mostram os exemplos a seguir:

```
lista1 = [1, 2, 3, 4, 5]
```

```
lista2 = ['gol', 'fusca', 'palio']
```

Como podemos ver, os elementos podem ser de qualquer tipo de dado, inclusive outros objetos como strings, números, listas e até mesmo objetos personalizados.



Também podemos criar uma lista vazia.

Uma lista vazia é definida por um par de colchetes vazio:

```
lista_vazia = []
```

Outra opção é usar a função **range()** para gerar uma sequência de números. Essa função permite gerar sequências de valores em progressão aritmética. É muito útil para gerar as listas de valores

```
# gera uma lista contendo os valores: 0, 1, 2, 3, 4  
lista_de_valores = list(range(5))
```



As listas podem conter qualquer tipo de dados.

Podemos criar uma lista onde cada item é de um tipo diferente. Podemos, inclusive, criar uma lista contendo outra lista:

```
lista1 = [1, 2, 3, 4, 5]  
lista_mista = [10, 'texto', True, 3.14, lista1]
```

6.3 Acessando os elementos da lista



Clique aqui para ver a vídeo aula sobre acesso aos elementos da lista.

As listas são uma sequência arbitrária de elementos. Isso significa que os elementos de uma lista podem ser acessados utilizando um índice inteiro entre colchetes [].



Nas listas a indexação começa na posição de índice 0.

Ou seja, o primeiro elemento tem índice 0, o segundo índice 1 e assim por diante, como mostra o exemplo a seguir.

Exemplo: acessando os elementos de uma lista

```
01 # cria uma lista de carros  
02 carros = ['gol', 'fusca', 'palio', 'onix']  
03  
04 # acessa a lista  
05 print(carros[0])  
06 print(carros[1])
```

```
07
08 # muda o valor da posição 0
09 carros[0] = 'ferrari'
10 print(carros[0])
```

Saída gol
fusca
ferrari

0	1	2	3
gol	fusca	palio	onix
carros			

Como podemos ver nesse exemplo, as listas são estruturas mutáveis. Após criadas, podemos alterar, adicionar ou remover elementos da lista.



As listas são estruturas mutáveis, o que significa que podemos modificar o seu conteúdo após a sua criação.

Cada posição da lista funciona como se fosse uma variável independente e possui todas as características de uma variável. Isso significa que ela pode aparecer em comandos de entrada e saída de dados, expressões e atribuições, como mostra o exemplo a seguir.

Exemplo: acessando os elementos de uma lista

```
01 lista = [10, 20, 30, 40, 50]
02
03 print(lista[0])
04
05 lista[0] = int(input('Digite um valor: '))
06
07 lista[1] = lista[0] + lista[2]
08
09 if(lista[1] > 5):
10     print('Valor = ', lista[1])
```

Nesse exemplo, fica claro que podemos fazer com os elementos de uma lista qualquer tarefa que antes fazíamos com uma variável.



Não podemos acessar um índice da lista que seja maior ou igual ao tamanho da lista. É tarefa do programador garantir que os limites da lista são respeitados.

Os índices de uma lista sempre começam em ZERO e vão até TAMANHO-1. Para se obter o tamanho de uma lista, usamos a função **len()**, como mostra o exemplo a seguir.

Exemplo: tamanho da lista
<pre>01 carros = ['gol', 'fusca', 'palio', 'onix'] 02 tamanho = len(carros) 03 print(tamanho)</pre>
Saída 4

Neste caso, a função retornará o valor 4, que é o número de itens armazenados dentro da lista.



Podemos utilizar índices negativos para acessar os elementos de uma lista.

Neste caso, a contagem começa do último item da lista, como mostra o exemplo a seguir.

Exemplo: usando índices negativos com uma lista													
<pre>01 carros = ['gol', 'fusca', 'palio', 'onix'] 02 03 print(carros[0]) 04 print(carros[-4]) 05 print(carros[-3])</pre>													
Saída	<pre>gol gol fusca</pre>												
<table><tr><td>0</td><td>1</td><td>2</td><td>3</td></tr><tr><td>gol</td><td>fusca</td><td>palio</td><td>onix</td></tr><tr><td>-4</td><td>-3</td><td>-2</td><td>-1</td></tr></table>		0	1	2	3	gol	fusca	palio	onix	-4	-3	-2	-1
0	1	2	3										
gol	fusca	palio	onix										
-4	-3	-2	-1										

Também podemos associar o valor de cada posição da lista a uma variável sem precisar especificar os índices usando o recurso de *unpacking*. Basta definir uma lista de variáveis, entre parênteses, que receberá o conteúdo da lista.



Clique aqui para ver a vídeo aula sobre o recurso de *unpacking*.

Exemplo: acessando os elementos de uma lista
<pre>01 numeros = [1, 2, 3] 02 [x, y, z] = numeros 03 04 print(x) 05 print(y) 06 print(z)</pre>
Saída 1 2 3

Esse recurso de *unpacking* faz com que as listas sejam comumente utilizadas em funções para retornar múltiplos valores. Uma função pode retornar uma lista e os valores podem ser atribuídos a variáveis separadas, como mostra o exemplo a seguir.

Exemplo: função com lista como retorno
<pre>01 # Função com lista como retorno 02 def func(): 03 return [10, 20, 30] 04 05 # Atribuindo valores retornados a variáveis 06 a, b, c = func() 07 print(a, b, c)</pre>

Podemos percorrer os elementos de uma lista utilizando um comando de repetição, como o **loop for**. O exemplo a seguir mostra que podemos usar um comando de repetição (preferencialmente o comando **for**) para percorrer a lista de 2 formas. Podemos percorrer os índices e elementos da lista ou podemos percorrer apenas os elementos. Isso evita manipular explicitamente o índice.

Exemplo: iterando os elementos de uma lista
<pre>01 carros = ['gol', 'fusca', 'palio', 'onix'] 02 03 # Percorre apenas os elementos 04 for item in carros: 05 print(item) 06 07 # Percorre os índices e elementos 08 for indice in range(len(carros)): 09 print(indice, carros[indice])</pre>

6.4 Acessando uma sub-lista



Clique aqui para ver a vídeo aula sobre acesso a sub- listas.

Em Python, o operador ":" pode ser usado para acessar uma sub-lista, também conhecido como fatiamento de lista. Ele permite extrair uma parte específica de uma lista com base em índices de início e fim. A sintaxe básica para usar o operador ":" para fatiar uma string é a seguinte

```
lista[início:fim:passo]
```

onde

- **início**: é o índice em que a sub-lista deve começar (inclusive).
- **fim**: é o índice em que a sub-lista deve terminar (exclusivo).
- **passo**: é o valor de incremento entre os números da sequência (**opcional**). O valor padrão é 1.



Vale ressaltar que o índice de **início** é inclusivo, ou seja, a posição correspondente é incluída na sub-lista, enquanto o índice de **fim** é exclusivo, ou seja, a posição correspondente não é incluída na sub-lista.

A seguir podemos ver alguns exemplos de uso do operador ":" para fatiar uma lista.

Exemplo: fatiando uma string

```
01 lista = [ 10, 20, 30, 40, 50]
02
03 # seleciona as posições 2 e 3
04 print(lista[2:4])
05
06 # seleciona as posições 2 até o final
07 print(lista[2:])
08
09 # seleciona as posições do início até a 2
10 print(lista[:3])
11
12 # seleciona todas as posições
13 print(lista[:])
14
15 # seleciona as posições 0, 2 e 4
16 print(lista[0:5:2])
```

```
Saída [30, 40]
       [30, 40, 50]
```

```
[10, 20, 30]
[10, 20, 30, 40, 50]
[10, 30, 50]
```

6.5 Manipulando listas

A linguagem Python possui diversas funções especialmente desenvolvidas para a manipulação de listas. A seguir são apresentadas algumas das funções e operações mais utilizadas.

6.5.1 Concatenando e repetindo listas



Clique aqui para ver a vídeo aula sobre concatenação e repetição de listas.

A operação de concatenação é uma tarefa muito comum ao se trabalhar com listas. Basicamente, essa operação consiste em copiar uma lista para o final de outra. Na linguagem Python, para se fazer a concatenação de duas listas, usamos o operador de soma, “+”, como mostra o exemplo a seguir.

Exemplo: concatenação de listas

```
01 lista1 = [1,2,3]
02 lista2 = [4,5,6]
03 lista3 = lista1 + lista2
04 print(lista3)
```

Saída [1, 2, 3, 4, 5, 6]

Também podemos criar uma lista a partir da repetição de outra usando o operador de “*”, como mostra o exemplo a seguir.

Exemplo: repetição de listas

```
01 lista1 = [1,2,3]
02 lista3 = 2 * lista1
03 print(lista3)
```

Saída [1, 2, 3, 1, 2, 3]

6.5.2 Copiando uma lista



Clique aqui para ver a vídeo aula sobre cópia de listas.

Em Python, existem várias maneiras de copiar uma lista.



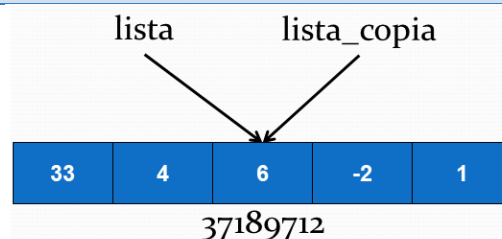
A operação de atribuição não cria uma cópia do objeto.

Em Python, listas são objetos. Se atribuirmos uma lista a outra, ambas irão se referir ao mesmo objeto, como mostra o exemplo a seguir.

Exemplo: atribuição de listas

```
01 lista = [1, 2, 3, 4, 5]
02 lista_copia = lista
03
04 print(id(lista))
05 print(id(lista_copia))
06
07 lista[0] = 100
08
09 print(lista)
10 print(lista_copia)
```

```
Saída 37189712
      37189712
      [100, 2, 3, 4, 5]
      [100, 2, 3, 4, 5]
```



Nesse exemplo, é possível ver que ambas as listas possuem o mesmo id, ou seja, são a mesma lista. Alterar o valor em uma lista, significa alterar o valor na outra também. Precisamos garantir que temos duas listas diferentes, mas com o mesmo conteúdo.



Para fazer a cópia de uma lista é essencial entender a diferença entre uma **cópia rasa** (*shallow copy*) e uma **cópia profunda** (*deep copy*).

A diferença entre os dois tipos de cópia depende muito do conteúdo da lista a ser copiada:

- **Cópia Rasa** (*Shallow Copy*): cria uma nova lista, mas os elementos dessa nova lista ainda se referem aos mesmos objetos dos elementos da lista original. Ou seja, apenas o objeto da lista é copiado, não seus elementos internos. Para fazer uma cópia rasa, você pode usar a função **copy()** ou o método **[:]**.
- **Cópia Profunda** (*Deep Copy*): cria uma nova lista e também cria novos objetos para os elementos internos. Assim, os elementos da lista copiada são independentes da lista original. Para fazer uma cópia profunda, você precisa usar o módulo **copy** e a função **deepcopy()**.



Basicamente, a **cópia rasa** faz somente uma cópia superficial do objeto enquanto a cópia profunda copia não somente o objeto, mas também todo e qualquer objeto embutido neste objeto.

A cópia rasa funciona bem quando temos uma lista que contém elementos mais simples. Esse método não é capaz de copiar objetos embutidos dentro de outros objetos, como outra lista, como mostra o exemplo a seguir.

Exemplo: cópia rasa de listas

```
01 import copy
02
03 lista = [1, 2, 3, [4, 5]]
04 lista_copia = lista.copy()
05 # outra possibilidade de cópia rasa
06 # lista_copia = lista[:]
07
08 print(id(lista))
09 print(id(lista_copia))
10
11 lista[0] = 100
12 lista_copia[3][0] = 40
13
14 print(lista)
15 print(lista_copia)
```

```
Saída 36998160
      37415712
      [100, 2, 3, [40, 5]]
      [1, 2, 3, [40, 5]]
```

Como podemos ver, a **cópia rasa** criou uma cópia da lista de modo que os três primeiros elementos são independentes. Alterar a posição 0 de uma lista não afeta a outra. Porém, o último elemento é uma lista contendo outros dois elementos. A **cópia rasa** não criou uma cópia dessa lista interna, de modo que alterar o valor nela

significa alterar o valor na outra também. Para resolver esse problema, precisamos fazer uma **cópia profunda** da lista, como mostra o exemplo a seguir.

Exemplo: cópia profunda de listas

```
01 import copy
02
03 lista = [1, 2, 3, [4, 5]]
04 lista_copia = copy.deepcopy(lista)
05
06 print(id(lista))
07 print(id(lista_copia))
08
09 lista[0] = 100
10 lista_copia[3][0] = 40
11
12 print(lista)
13 print(lista_copia)
```

```
Saída 36998160
      37415712
      [100, 2, 3, [4, 5]]
      [1, 2, 3, [40, 5]]
```

Observe a diferença entre os resultados da **cópia rasa** e da **cópia profunda**. Na cópia rasa, as alterações nos elementos internos são refletidas tanto na lista original quanto na cópia, pois os elementos ainda estão se referindo aos mesmos objetos. Já na cópia profunda, as alterações nos elementos internos não afetam a lista original, pois os elementos são independentes.

6.5.3 Removendo elementos da lista



Clique aqui para ver a vídeo aula sobre remoção em listas.

Para remover um ou mais elementos de uma lista podemos usar a declaração **del**. Essa declaração é usada para remover um elemento (ou uma fatia de elementos) com base no seu índice, como mostra o exemplo a seguir.

Exemplo: removendo elementos da lista

```
01 lista = [1, 2, 3, 4, 5, 6]
02 # remove o elemento da posição 0
03 del lista[0]
04 print(lista)
05 # remove os elementos das posições 0 e 1
06 del lista[:2]
07 print(lista)
```

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

6.5.4 Procurando um valor dentro da lista



Clique aqui para ver a vídeo aula sobre o operador *in*.

Podemos procurar um elemento em uma lista utilizando o operador **in**. A sintaxe básica para usar o operador **in** é a seguinte

valor in lista

O resultado será **True** se o valor existir dentro da lista, e **False**, caso contrário. A seguir podemos ver um exemplo de uso desse operador.

Exemplo: procurando um elemento dentro da lista

```
01 lista = [1, 2, 3, 4, 5, 6]
02 x = 2
03
04 if(x in lista):
05     print('X existe na lista')
06 else:
07     print('X NÃO existe na lista')
```

6.5.5 Métodos que manipulam listas

Uma lista é um objeto da classe **list** e, portanto, possui diversos métodos já definidos. Um dos jeitos mais simples de manipular uma lista é utilizar os métodos que já fazem parte dela. Esses métodos permitem executar diversas tarefas, como a ordenação, inserção, remoção, etc.



Clique aqui para ver a vídeo aula sobre métodos para manipular listas.



Esses métodos modificam o conteúdo original da lista.

A sintaxe básica para usar esses métodos é a seguinte

```
lista.nome-método()
```

A lista abaixo apresenta alguns dos métodos disponíveis

- **sort()**: ordena os elementos da lista
- **append(x)**: insere um elemento x no final da lista
- **insert(pos,x)**: insere um elemento x na posição pos
- **remove(valor)**: remove o primeiro elemento da lista que corresponde a um valor específico
- **pop(pos)**: remove e retorna o elemento da posição **pos**. Se nenhum índice for especificado, o último elemento será removido.
- **reverse()**: Inverte a ordem dos elementos na lista

A seguir podemos ver alguns exemplos de usos desses métodos.

Exemplo: usando os métodos da lista

```
01 lista = [21, 12, 3, 44, 5]
02 lista.sort()
03 print(lista)
04
05 lista.remove(3)
06 print(lista)
07
08 lista.append(10)
09 print(lista)
10
11 lista.insert(0,-1)
12 print(lista)
```

```
Saída [3, 5, 12, 21, 44]
      [5, 12, 21, 44]
      [5, 12, 21, 44, 10]
      [-1, 5, 12, 21, 44, 10]
```

6.6 Listas aninhadas



Clique aqui para ver a vídeo aula sobre listas aninhadas.

Uma lista aninhada em Python é uma lista que contém outras listas ou outras sequências como seus elementos. Isso significa que cada elemento da lista aninhada pode ser uma lista por si só. Essa estrutura permite criar estruturas de dados multidimensionais, como matrizes, tabelas ou listas de listas, para armazenar dados de forma organizada e hierárquica.

6.6.1 Sintaxe e criação de uma lista aninhada

Criar uma lista aninhada é simples, basta incluir outras listas como elementos dentro da lista principal. Os elementos das listas internas podem ser de diferentes tipos, como números, strings, outras listas ou até mesmo objetos mais complexos.



Para criar uma lista aninhada, basta definir cada elemento da lista como uma nova lista.

Em linguagem Python, a declaração de uma lista aninhada ficaria assim

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

Neste caso, nossa lista aninhada contém três outras listas dentro dela, cada uma contendo três valores inteiros.

6.6.2 Acessando os elementos da lista aninhada

As listas são uma sequência arbitrária de elementos. Isso significa que os elementos de uma lista podem ser acessados utilizando um índice inteiro entre colchetes []. Para acessar os elementos de uma lista aninhada, é necessário usar índices adicionais para indicar a posição do elemento nas listas internas.



Tanto nas listas, quanto nas listas internas, a indexação começa na posição de índice 0.

Ou seja, para acessar a posição 0 da lista interna armazenada na posição 0 da lista aninhada, usamos [0][0], como mostra o exemplo a seguir.

Exemplo: acessando os elementos de uma lista aninhada

```
01 lista_aninhada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
02
03 # Acessa o primeiro item da primeira lista interna
04 print(lista_aninhada[0][0])
05
06 # Acessa o segundo item da segunda lista interna
07 print(lista_aninhada[1][1])
08
09 # Acessa o último item da última lista interna
10 print(lista_aninhada[2][2])
11
12
```

Saída 1
5
9

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Devemos lembrar que as listas são estruturas mutáveis. Após criadas, podemos alterar, adicionar ou remover elementos da lista aninhada assim como das listas internas, como mostra o exemplo a seguir.

Exemplo: manipulando uma lista aninhada

```
01 lista_aninhada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
02
03 # adiciona uma nova lista ao final
04 lista_aninhada.append([10, 11, 12])
05 print(lista_aninhada)
06
07 # Altera o primeiro item da primeira lista interna
08 lista_aninhada[0][0] = 'python'
09 print(lista_aninhada)
```

Saída [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
[['python', 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

Como podemos ver, podemos alterar o conteúdo de uma lista interna, assim como incluir novas linhas e colunas em cada lista.



Recomenda-se, neste caso, usar os métodos que manipulam as listas ao invés do operador de concatenação.

Além disso, não é necessário que as linhas internas tenham sempre o mesmo tamanho. Elas funcionam de forma independente.

6.6.3 Percorrendo os elementos da lista aninhada



Clique aqui para ver a vídeo aula sobre como percorrer listas aninhadas.

Você também pode realizar iterações para percorrer os elementos da lista aninhada usando **loops for**, tanto para as listas externas quanto para as listas internas.



Todas as operações feitas em listas aninhadas devem considerar o fato de que temos agora uma lista dentro de outra.

O exemplo a seguir mostra que podemos percorrer a lista de 2 formas. Podemos percorrer os índices e elementos da lista ou podemos percorrer apenas os elementos (isso evita manipular explicitamente o índice).

Exemplo: iterando os elementos de uma lista aninhada

```
01 lista_aninhada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
02 # Percorre apenas os elementos
03 for lista in lista_aninhada:
04     for item in lista:
05         print(item)
06
07 # Percorre os índices e elementos
08 for y in range(len(lista_aninhada)):
09     for x in range(len(lista_aninhada[y])):
10         print(y,x,lista_aninhada[y][x])
```

6.6.4 Removendo elementos da lista aninhada



Clique aqui para ver a vídeo aula sobre remoção em listas aninhadas.

Para remover um ou mais elementos de uma lista aninhada podemos usar a declaração **del**. Essa declaração é usada para remover um elemento (ou uma fatia de elementos) com base no seu índice, como mostra o exemplo a seguir. Como estamos trabalhando com listas aninhadas, podemos remover toda uma lista interna ou apenas um elemento da lista interna.

Exemplo: removendo elementos da lista

```
01 lista_aninhada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
02
03 # remove o elemento da posição 0 da lista 0
04 del lista_aninhada[0][0]
05 print(lista_aninhada)
06
07 # remove toda a lista da posição 1
08 del lista_aninhada[1]
09 print(lista_aninhada)
```

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

6.7 Compreensão de lista



Clique aqui para ver a vídeo aula sobre compreensão de listas.

Compreensão de lista (*list comprehension*) é uma poderosa e concisa construção em Python que permite criar novas listas a partir de listas existentes ou de outras sequências, aplicando expressões e/ou condições em uma única linha de código. É uma forma elegante e eficiente de transformar, filtrar e combinar elementos de uma lista em uma nova lista.



A compreensão de lista é uma construção sintática para criação de listas. Ela permite mapear e filtrar uma lista em uma única expressão.

A estrutura geral da compreensão de lista é a seguinte:

```
nova_lista = [expressão for elemento in lista if condição]
```

onde

- **nova_lista**: É a lista resultante após a aplicação da compreensão de lista.
- **expressão**: É uma operação ou cálculo que será aplicado a cada elemento da lista original (**lista**) para obter os elementos da nova lista.
- **elemento**: É a variável temporária que representa cada item da lista original durante o processo de compreensão.
- **lista**: É a lista de origem a partir da qual a nova lista será criada.
- **condição (opcional)**: É uma condição que pode ser aplicada para filtrar os elementos da lista original que serão incluídos na nova lista. Se a condição não for definida, todos os elementos serão incluídos.



A compreensão de lista geralmente leva a um código mais eficiente e performático, pois é otimizada internamente pelo interpretador Python.

A compreensão de lista é uma técnica muito útil e amplamente utilizada em Python, pois permite escrever código mais legível e conciso, evitando a necessidade de usar **loops** explícitos. No entanto, é importante usá-la com moderação, pois o uso excessivo de compreensões de lista muito complexas pode tornar o código menos legível e difícil de entender.

Imagine que queremos criar uma lista com os quadrados de valores contidos em outra lista. Normalmente, teríamos que percorrer a lista original com um **loop for** e calcular o quadrado para cada número. Como compreensão de lista isso pode ser facilmente feito como mostrado a seguir.

Exemplo: compreensão de lista - quadrados dos números

```
01 numeros = [1, 2, 3, 4, 5]
02
03 # usando um comando de repetição
04 quadrados = []
05 for x in numeros:
06     quadrados.append(x ** 2)
07 print(quadrados)
08
09 # usando compreensão de lista
10 quadrados = [x ** 2 for x in numeros]
11 print(quadrados)
```

Saída [1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]

Vamos a outro exemplo. Imagine agora que queremos filtrar os elementos de uma lista já existente. Para isso, precisamos percorrer a lista original com um **loop for** e selecionar os elementos desejados com um comando **if**. Como compreensão de lista, basta incluir a condição de seleção dos elementos, como mostrado a seguir.

Exemplo: compreensão de lista – seleção de elementos	
<pre>01 numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 02 03 # usando um comando de repetição 04 pares = [] 05 for x in numeros: 06 if(x % 2 == 0): 07 pares.append(x) 08 print(pares) 09 10 # usando compreensão de lista 11 pares = [x for x in numeros if x % 2 == 0] 12 print(pares)</pre>	
Saída	<pre>[2, 4, 6, 8, 10] [2, 4, 6, 8, 10]</pre>

6.8 Aplicações e desempenho

As listas são uma das estruturas de dados mais versáteis e úteis disponíveis na linguagem Python. Elas fornecem uma série de vantagens ao se manipular coleções de elementos:

- **Flexibilidade:** Listas são dinâmicas e mutáveis, o que permite adicionar, modificar e remover elementos de forma fácil e eficiente.
- **Organização de Dados:** são úteis para organizar e armazenar coleções de elementos, tornando-as ideais para lidar com grandes quantidades de dados.
- **Acesso por Índices:** Os elementos em uma lista podem ser acessados por índices numéricos, o que possibilita recuperar valores específicos de maneira rápida.



No entanto, é importante estar ciente do consumo de memória em situações de grandes conjuntos de dados.

As maiores desvantagens do uso de listas são:

- **Desempenho:** O desempenho de algumas operações em listas pode ser afetado quando lidando com listas muito grandes, especialmente quando ocorrem inserções ou remoções frequentes.

- **Consumo de Memória:** Listas podem ocupar mais espaço na memória quando comparadas com estruturas de dados mais eficientes, como *arrays* em algumas linguagens.

As listas são amplamente utilizadas em aplicações do dia a dia, desde tarefas simples como listas de compras até implementações mais complexas de algoritmos e estruturas de dados como filas e pilhas, que são amplamente utilizadas em algoritmos e processamento de dados.

7 Tuplas e dicionários

7.1 Tuplas

As tuplas são uma estrutura de dados fundamental em Python, permitindo armazenar coleções ordenadas de elementos, muito parecidas com as listas.



No entanto, as tuplas possuem uma importante diferença em relação às listas: são imutáveis.

Isso significa que, uma vez criada, uma tupla não pode ser alterada, adicionando ou removendo elementos. Essa característica torna as tuplas especialmente úteis para armazenar conjuntos de dados que não precisam ser modificados ao longo do programa.

7.1.1 Sintaxe e criação de uma tupla



Clique aqui para ver a vídeo aula sobre criação de tuplas.

Uma tupla é criada utilizando parênteses () ou simplesmente separando os elementos por vírgulas, como mostram os exemplos a seguir:

```
tupla1 = (1, 2, 3, 4, 5)
```

```
tupla2 = 6, 7, 8, 9, 10
```



Também podemos criar uma tupla vazia.

Uma tupla vazia é definida por um par de parênteses vazio:

```
tupla_vazia = ()
```

Tuplas com 1 elemento devem possuir uma vírgula depois do elemento para serem consideradas tuplas. Caso contrário serão consideradas como do tipo do elemento:

```
tupla_um_elemento = (5,)
```



Assim como as listas, as tuplas podem conter qualquer tipo de dados.

Podemos criar um tupla onde cada item é de um tipo diferente. Podemos, inclusive, criar uma tupla contendo outra tupla:

```
tupla1 = (1, 2, 3, 4, 5)
tupla2 = (tupla1, 'abc', 6, 7.5)
```

Podemos também usar as operações de concatenação e repetição para criar um tupla:

```
tupla1 = (1, 2, 3, 4, 5)
tupla2 = (6, 7, 8, 9, 10)
tupla3 = tupla1 + tupla2
tupla4 = 2 * tupla2
```

7.1.2 Acessando os elementos da tupla



Clique aqui para ver a vídeo aula sobre acesso aos elementos da tupla.

Assim como as listas, as tuplas também são uma sequência arbitrária de elementos. Isso significa que os elementos de uma tupla podem ser acessados utilizando um índice inteiro entre colchetes [], da mesma forma que as listas.



Assim como as listas, a indexação das tuplas começa na posição de índice 0.

Ou seja, o primeiro elemento tem índice 0, o segundo índice 1 e assim por diante, como mostra o exemplo a seguir.

Exemplo: acessando os elementos de uma tupla

```
01 numeros = (1, 2, 3, 4, 5)
02
03 print(numeros[0])
04 print(numeros[2])
```

```
Saída 1
      3
```



As tuplas em Python são imutáveis, o que significa que não podem ser modificadas depois de criadas.

As tuplas são imutáveis, o que significa que, uma vez criadas, não é possível alterar, adicionar ou remover elementos. Seu conteúdo permanece constante durante toda a execução do programa, como mostra o exemplo a seguir.

Exemplo: acessando os elementos da string

```
01 # Tupla imutável
02 tupla = (1, 2, 3)
03
04 # Tentativa de alterar o valor da tupla
05 # resultará em erro
06 tupla[0] = 10
```

Saída Erro: 'tuple' object does not support item assignment

Como nas listas, nas tuplas também podemos associar o valor de cada posição a uma variável sem precisar especificar os índices usando o recurso de *unpacking*. Basta definir uma lista de variáveis, entre parênteses, que receberá o conteúdo da tupla, como mostra o exemplo a seguir.

Exemplo: acessando os elementos de uma tupla

```
01 numeros = (1, 2, 3)
02 (x, y, z) = numeros
03
04 print(x)
05 print(y)
06 print(z)
```

Saída 1
2
3



Clique aqui para ver a vídeo aula sobre o recurso de *unpacking* em tuplas.

Esse recurso de *unpacking* faz com que as tuplas sejam comumente utilizadas em funções para retornar múltiplos valores. Uma função pode retornar uma tupla e os valores podem ser atribuídos a variáveis separadas, como mostra o exemplo a seguir.

Exemplo: função com tupla como retorno

```
01 # Função com tupla como retorno
02 def func():
03     return (10, 20, 30)
04
05 # Atribuindo valores retornados a variáveis
06 a, b, c = func()
07 print(a, b, c)
```

Assim como nas listas, podemos percorrer os elementos de uma tupla utilizando um comando de repetição, como o **loop for**.

Exemplo: iterando os elementos de uma tupla

```
01 numeros = (1, 2, 3, 4, 5)
02 for item in numeros:
03     print(item)
```

7.1.3 Aplicações e desempenho

Por serem imutáveis, as tuplas podem ter um desempenho ligeiramente melhor do que as listas, especialmente em operações de leitura e iteração.



A principal vantagem das tuplas é sua imutabilidade. Uma vez criadas, seus elementos não podem ser alterados, o que garante a integridade dos dados ao longo do programa.

As tuplas são frequentemente usadas quando precisamos garantir a integridade de dados, isto é, quando os dados precisam ser fixos e não modificados durante a execução do programa, como coordenadas geográficas, dias da semana, informações que não mudam ao longo do tempo, como mostra o exemplo a seguir.

Exemplo: aplicações de uma tupla

```
01 # Manutenção de coordenadas geográficas
02 coordenadas = (40.7128, -74.0060)
03
04 # Manutenção de informações imutáveis
05 historico_eventos = (("2023-07-20", "Evento A"),
06                      ("2023-07-21", "Evento B"),
07                      ("2023-07-22", "Evento C"))
08
09 # Necessidade de definir constantes no programa
10 DIAS_SEMANA = ("Segunda", "Terça", "Quarta",
11 "Quinta", "Sexta", "Sábado", "Domingo")
```

7.2 Dicionários

Os dicionários são uma estrutura de dados muito poderosa em Python, permitindo armazenar coleções de pares chave-valor. Ao contrário de listas ou tuplas, que armazenam elementos ordenados por índices, os dicionários associam uma chave única a um valor correspondente. Isso torna os dicionários ideais para armazenar informações que precisam ser facilmente acessadas e recuperadas através de chaves específicas.

7.2.1 Sintaxe e criação de um dicionário



Clique aqui para ver a vídeo aula sobre a criação de um dicionário.

Os dicionários são criados utilizando chaves { } e cada par chave-valor é separado por dois pontos :. Diferentes pares de chave-valor são separados por vírgula, conforme a sintaxe a seguir:

```
dicionario={chave1:valor1,chave2:valor2,...,chaveN:valorN}
```



Assim como as listas e tuplas, dicionários podem conter qualquer tipo de dados.

Podemos criar um dicionário onde cada par chave-valor é de um tipo diferente:

```
dicionario={'nome':'João','idade':30,1:'Itu'}
```

Nesse exemplo, três associações são criadas:

- Uma chave string é associada a um valor string: 'nome': 'João'
- Uma chave string é associada a um valor inteiro: 'idade': 30
- Uma chave inteira é associada a um valor string: 1: 'Itu'

De fato, quaisquer tipos de dados podem ser combinados para criar um par chave-valor. Podemos, inclusive, guardar um dicionário dentro de outro:

```
banco_de_dados = {  
    'id_001':{'nome':'Alice','idade': 25,'cidade': 'São Paulo'},  
    'id_002':{'nome':'Bob','idade': 30,'cidade':'Rio de Janeiro'},  
    'id_003':{'nome':'Carol','idade':28,'cidade':'Belo Horizonte'}  
}
```




Também podemos criar um dicionário vazio.

Um dicionário vazio é definido por um par de chaves vazio:

```
dicionario_vazio = {}
```

7.2.2 Acessando os elementos do dicionário



Clique aqui para ver a vídeo aula sobre acesso aos elementos do dicionário.

Os valores em um dicionário podem ser acessados utilizando suas **chaves**, fornecendo o nome da chave entre colchetes []. Da mesma forma, podemos modificar ou adicionar novos valores associados a uma chave, como mostra o exemplo a seguir.

Exemplo: acessando os elementos de um dicionário

```
01 # Cria um dicionário
02 dic = {'nome':'João','idade':30,'cidade':'Itu'}
03
04 # Acessando um valor no dicionário
05 print(dic['nome'])
06
07 # Modificando um valor no dicionário
08 dic['idade'] = 35
09
10 # Adicionando um novo par chave-valor
11 dic['profissão'] = 'Engenheiro'
```

Perceba, nesse exemplo, que é possível adicionar um novo par de chave-valor ao dicionário através do operador de colchetes (linha 11).



Novos itens são adicionados a um dicionário quando fazemos uma atribuição a uma chave ainda não definida.

Se a chave ainda não existe no dicionário, ela será criada e adicionada ao dicionário com o valor atribuído a ela. Caso ela já exista, seu valor será atualizado.

Assim como nas listas e tuplas, podemos percorrer os elementos de um dicionário utilizando um comando de repetição, como o **loop for**. Podemos percorrer tanto o conjunto das chaves quanto os dos valores, como mostra o exemplo a seguir.

Exemplo: iterando os elementos de um dicionário

```
01 # Cria um dicionário
02 dic = {'nome':'João','idade':30,'cidade':'Itu'}
03
04 # Iterando nas chaves do dicionário
05 for chave in dic:
06     print(chave)
07
08 # Iterando nos valores do dicionário
09 for valor in dic.values():
10     print(valor)
11
12 # Iterando nos pares chave-valor do dicionário
13 for chave, valor in dic.items():
14     print(f'{chave}: {valor}')
```

7.2.3 Funções e métodos de dicionários



Clique aqui para ver a vídeo aula sobre funções e métodos de dicionários.

Um dicionário é um objeto da classe **dict** e, portanto, possui diversas funções e métodos já definidos. Esses métodos permitem executar diversas tarefas e facilitam o trabalho com dicionários. A sintaxe básica para usar esses métodos é a seguinte

dicionario.nome-método()

A lista abaixo apresenta alguns dos métodos disponíveis.

- **clear()**: remove todos os elementos do dicionário;
- **copy()**: cria uma cópia do dicionário (atribuição não cria cópia);
- **get(chave,valor)**: obtém o conteúdo de chave. Caso a chave não exista, retorna valor;
- **items()**: retorna uma lista com todos os pares chave/conteúdo do dicionário;
- **keys()**: retorna uma lista com todas as chaves do dicionário;
- **values()**: retorna uma lista com todos os valores do dicionário;
- **pop(chave)**: obtém o valor correspondente a chave e remove o par chave/valor do dicionário;

- **popitem()**: retorna e remove um par chave/valor aleatório do dicionário. Pode ser usado para iterar sobre todos os elementos do dicionário.



Além disso, podemos usar o operador **in** para saber se uma chave existe no dicionário e a função **len()** para saber o tamanho de um dicionário.

A seguir podemos ver alguns exemplos de usos desses métodos.

Exemplo: usando os métodos do dicionário

```
01 dic = {'nome': 'João', 'idade': 30, 'cidade': 'Itu'}
02
03 # Verificando se uma chave existe no dicionário
04 if 'idade' in dic:
05     print("Chave 'idade' encontrada!")
06
07 # Obtendo o número de pares chave-valor no dicionário
08 num_pares = len(dic)
09
10 print(dic.keys())
11 print(dic.values())
```

```
Saída Chave 'idade' encontrada!
dict_keys(['nome', 'idade', 'cidade'])
dict_values(['João', 30, 'Itu'])
```

7.2.4 Aplicações e desempenho

Os dicionários são extremamente versáteis e têm uma ampla variedade de aplicações em programação Python. Eles são especialmente úteis quando é necessário associar informações entre chaves e valores, facilitando a organização e acesso eficiente aos dados.



No entanto, é importante estar ciente do consumo de memória em situações de grandes conjuntos de dados.

De modo geral, dicionários podem ser mais pesados em termos de consumo de memória quando comparados com outras estruturas de dados, como listas ou tuplas.



Dicionários possuem um desempenho ótimo para operações de busca e acesso a elementos.

Dicionários permitem um acesso rápido aos valores através das chaves. Em média, o tempo de busca é constante, $O(1)$, tornando-os ideais para operações de busca,

devido à rapidez em encontrar valores associados às chaves. Além disso, eles são ideais para representar estruturas de dados complexas, como configurações, bancos de dados em memória, ou mapeamento de informações em geral, como mostra o código a seguir.

Exemplo: aplicações de um dicionário

```
01 # Armazenando configurações de um programa de forma
02 organizada
03 configuracoes = {
04     'tamanho_fonte': 12,
05     'cor_fundo': 'branco',
06     'modo_noturno': False
07 }
08
09 # Contagem de frequência de letras
10 texto = "abracadabra"
11 frequencia = {}
12 for letra in texto:
13     if letra in frequencia:
14         frequencia[letra] += 1
15     else:
16         frequencia[letra] = 1
17
18 print(frequencia)
```

Saída {'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}

8.1 Definindo uma string



Clique aqui para ver a vídeo aula sobre strings.

String é o nome que usamos para definir uma sequência de caracteres adjacentes na memória do computador, como letras, números e símbolos. Ela é tratada como um único objeto da classe **str** e é usada para representar texto, como uma palavra ou frase, em Python.



As strings são definidas entre **aspas simples** (") ou **aspas duplas** (").

Por exemplo, "Olá, mundo!" é uma string válida. Podemos também atribuir uma string para uma variável:

```
texto = "Python"
```



Podemos utilizar **três aspas simples** na inicialização de uma string. Neste caso, será possível criar uma string contendo mais de uma linha.

Neste tipo de inicialização, as quebras de linha também serão armazenadas dentro da string. O exemplo a seguir mostra as formas de inicialização de uma string.

Exemplo: inicialização da string

```
01 # inicialização com aspas duplas
02 str = "Python"
03
04 # inicialização com aspas simples
05 str = 'Python'
06
07 # inicialização com 3 aspas simples
08 str = '''Aprender Python
09 é muito
10 fácil'''
```

8.2 Acessando os elementos da string



Clique aqui para ver a vídeo aula sobre acesso a elementos da string.

Podemos tratar uma string como uma entidade única. Mas também podemos acessar seus caracteres individualmente usando colchetes e o índice da posição.



Uma string funciona como um tipo de lista. A diferença é que cada posição da lista contém um único caractere.

Um ponto importante na manipulação de strings é que, por se tratar de um tipo de lista, cada caractere pode ser acessado individualmente por indexação como em qualquer lista.



As strings em Python são imutáveis, o que significa que não podem ser modificadas depois de criadas.

Ou seja, podemos acessar os elementos de uma string, mas não podemos mudar o seu conteúdo, como mostra o exemplo a seguir.

Exemplo: acessando os elementos da string

```
01 str = "Python"
02
03 # imprime a letra 'P'
04 print(str[0])
05
06 # imprime a letra 'y'
07 print(str[1])
08
09 # Erro: não podemos modificar o conteúdo da string
10 str[0] = 'X'
```

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



Não podemos acessar um índice da string que seja maior ou igual ao tamanho da string.

Os índices dos caracteres de uma string sempre começam em ZERO e vão até TAMANHO-1. Para se obter o tamanho de uma string, usamos a função **len()**, como mostra o exemplo a seguir.

Exemplo: tamanho da string
<pre>01 str = "Python" 02 tamanho = len(str) 03 print(tamanho)</pre>
Saída 6

Neste caso, a função retornará o valor 6, que é o número de caracteres na palavra “Python”.



Podemos utilizar índices negativos para acessar os caracteres de uma string.

Neste caso, a contagem começa do último caractere da string com o valor **-1** e vai até a primeira posição da string, com valor **-TAMANHO**, como mostra o exemplo a seguir.

Exemplo: usando índices negativos com uma string

```
01 str = "Python"
02
03 # imprime a letra 'P'
04 print(str[0])
05
06 # imprime a letra 'P'
07 print(str[-6])
08
09 # imprime a letra 'n'
10 print(str[-1])
```

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

Podemos percorrer os elementos de uma string utilizando um comando de repetição, como o **loop for**. O exemplo a seguir mostra que podemos usar um comando de repetição (preferencialmente o comando **for**) para percorrer a string de 2 formas. Podemos percorrer os índices e elementos da string ou podemos percorrer apenas os elementos. Isso evita manipular explicitamente o índice.

Exemplo: iterando os elementos de uma string

```
01 texto = 'Python'
02
03 # Percorre apenas os elementos
04 for letra in texto:
05     print(letra)
06
07 # Percorre os índices e elementos
08 for indice in range(len(texto)):
09     print(indice, texto[indice])
```



Clique aqui para ver a vídeo aula sobre como percorrer uma string.

8.3 Acessando uma sub-string



Clique aqui para ver a vídeo aula sobre acesso a uma sub-string.

Em Python, o operador ":" pode ser usado para acessar uma sub-string, também conhecido como fatiamento de strings. Ele permite extrair uma parte específica de uma string com base em índices de início e fim.



Como nas listas, as strings também suportam acesso a sub-strings ou sub-cadeias de caracteres.

A sintaxe básica para usar o operador ":" para fatiar uma string é a seguinte

string[início:fim:passo]

onde

- **início**: é o índice em que a sub-string deve começar (inclusivo).
- **fim**: é o índice em que a sub-string deve terminar (exclusivo).
- **passo**: é o valor de incremento entre os números da sequência (**opcional**). O valor padrão é 1.



Vale ressaltar que o índice de **início** é inclusivo, ou seja, a posição correspondente é incluída na sub-string, enquanto o índice de **fim** é exclusivo, ou seja, a posição correspondente não é incluída.

A seguir podemos ver alguns exemplos de uso do operador ":" para fatiar uma string.

Exemplo: fatiando uma string

```
01 str = "Aprender Python é muito fácil"
02
03 # seleciona da posição 9 até a 14:
04 # 'Python'
05 print(str[9:15])
06
07 # seleciona da posição 9 até o final:
08 # 'Python é muito fácil'
09 print(str[9:])
10
11 # seleciona da posição 0 até a 14:
12 # 'Aprender Python'
13 print(str[:15])
14
15 # seleciona as posições [0,2,4,6,8,10,12,14]:
16 # 'Arne yhn'
17 print(str[0:15:2])
```

8.4 Formatação de string



Clique aqui para ver a vídeo aula sobre formatação de strings.

Às vezes, pode ser necessário formatar uma string e adicionar a ela os valores de outras variáveis, como números inteiros ou reais. Isso pode ser feito convertendo essas variáveis para texto e fazendo a concatenação de strings.



Uma forma mais fácil de fazer é utilizar o operador % para fazer a formatação da string.

A sintaxe básica para usar o operador “%” para formatar uma string é a seguinte

```
str = string-a-ser-formatada % (lista-de-valores)
```

onde

- **str**: string gerada pelo operador “%”
- **string-a-ser-formatada**: uma string contendo o texto a ser formatado e a posição de cada valor a ser inserido
- **lista-de-valores**: valores que deverão ser formatado e inseridos na string.

Basicamente, esse operador escreve na string **str** um conjunto de valores, caracteres e/ou sequência de caracteres de acordo com o código de formatação especificado na tabela a seguir:

Código	Valor formatado
%c	caractere
%s	string
%d	valor inteiro
%u	valor inteiro sem sinal
%f	valor real (ponto flutuante)
%.Nf	valor real (ponto flutuante) com N casas decimais
%%	símbolo de %

Na string da esquerda, o conjunto de caracteres depois do % define o tipo de formatação a ser executada. Assim, todo conteúdo da string da esquerda precedido por um % é substituído por um valor a direita (entre parênteses), como mostra o exemplo a seguir.

Exemplo: formatando uma string
<pre>01 reajuste = 10 02 inflacao = 12.5 03 str = "O reajuste foi de %d %% e a inflação de %.2f 04 %%" % (reajuste, inflacao) 05 06 print(str)</pre>
Saída O reajuste foi de 10% e a inflação de 12.50%

É possível também desabilitar o funcionamento dos código de formatação. Para fazer isso, comece uma string com **r** antes de abrir as aspas (simples ou duplas).



O prefixo **r** indica uma **raw string**, ou seja, tudo deve ser tratado literalmente como uma string e não como código de formatação.

A seguir podemos ver um exemplo de **raw string**.

Exemplo: definindo uma raw string
<pre>01 # string normal 02 print('Linguagem \ Python') 03 04 # raw string 05 print(r'Linguagem \ Python') 06</pre>
Saída Linguagem \ Python Linguagem \ Python

8.5 Manipulando strings

A linguagem Python possui diversas funções especialmente desenvolvidas para a manipulação de strings.



As strings em Python são imutáveis, o que significa que não podem ser modificadas depois de criadas.

No entanto, você pode realizar várias operações em strings, como concatenação (junção de duas ou mais strings), comparação, etc. A seguir são apresentadas algumas das funções e operações mais utilizadas.

8.5.1 Concatenando strings



Clique aqui para ver a vídeo aula sobre concatenação de strings.

A operação de concatenação é outra tarefa bastante comum ao se trabalhar com strings. Basicamente, essa operação consiste em copiar uma string para o final de outra string. Na linguagem Python, para se fazer a concatenação de duas strings, usamos o operador de soma, “+”, como mostra o exemplo a seguir.

Exemplo: concatenação de strings

```
01 texto_1 = "Linguagem"
02 texto_2 = "Python"
03 str = texto_1 + texto_2
04 print(str)
```

Saída LinguagemPython

8.5.2 Comparando duas strings



Clique aqui para ver a vídeo aula sobre comparação de strings.

Dado duas strings, podemos utilizar os operadores relacionais (==, !=, <, <=, >, >=) para compara-las.



A comparação é feita usando os códigos numéricos dos caracteres presentes na Tabela ASCII.

Por esse motivo, as letras maiúsculas e minúsculas são consideradas **diferentes** e maiúsculas vem **antes** das minúsculas. Podemos ver os códigos numéricos dos caracteres usando as seguintes funções

- **ord(ch)**: código numérico de um caractere **ch**
- **chr(x)**: caractere de um código numérico **x**

A seguir podemos ver alguns exemplos de comparação de strings.

Exemplo: comparando duas strings

```
01 str1 = "Andre"
02 str2 = "Ricardo"
03
04 if(str1 < str2):
05     print(str1,"vem antes de ",str2)
06 else:
07     print(str2,"vem antes de ",str1)
08
09 if(str1 == str2):
10     print("Strings iguais")
11 else:
12     print("Strings diferentes")
```

8.5.3 Procurando uma string dentro de outra



Clique aqui para ver a vídeo aula sobre o operador *in*.

Podemos procurar uma string menor dentro de uma string maior (por exemplo, palavra dentro de uma frase) utilizando o operador **in**. A sintaxe básica para usar o operador **in** é a seguinte

string1 in string2

O resultado será **True** se a string1 existir dentro da string2, e **False**, caso contrário. A seguir podemos ver um exemplo de uso desse operador.

Exemplo: procurando uma string dentro de outra

```
01 str = "Aprender Python é muito fácil"
02
03 if("Python" in str):
04     print("String encontrada")
05 else:
06     print("String não encontrada")
```

Saída String encontrada

8.5.4 Métodos que manipulam strings



Clique aqui para ver a vídeo aula sobre métodos que manipulam strings.

Uma string é um objeto da classe **str** e, portanto, possui diversos métodos já definidos. Um dos jeitos mais simples de manipular strings é utilizar os métodos que já fazem parte da string. Esses métodos permitem executar diversas tarefas, como a conversão maiúsculo/minúsculo, localizar e substituir substrings, etc.



Esses métodos geram uma nova string com a alteração. Eles nunca modificam o conteúdo original da string.

A sintaxe básica para usar esses métodos é a seguinte

`string.nome-método()`

A lista abaixo apresenta alguns dos métodos disponíveis.

- **lower()**: converte a string para minúsculo;
- **upper()**: converte a string para maiúsculo;
- **replace(ch1,ch2)**: troca os caracteres **ch1** por **ch2**
- **strip()**: remove espaços em branco do início e final da string
- **split()**: separa uma string por espaços e devolve uma lista de strings
- **split(ch)**: separa uma string usando o caractere **ch** e devolve uma lista de strings

A seguir podemos ver alguns exemplos de usos desses métodos.

Exemplo: usando os métodos da string

```
01 str = "Aprender Python é muito fácil"
02
03 # aprender python é muito fácil
04 print(str.lower())
05
06 # APRENDER PYTHON É MUITO FÁCIL
07 print(str.upper())
08
09 # ['Aprender', 'Python', 'é', 'muito', 'fácil']
10 print(str.split())
11
12 # Aprender-Python-é-muito-fácil
13 print(str.replace(' ','-'))
```

9 Funções

A linguagem Python possui muitas funções já implementadas e nós temos utilizadas elas constantemente. Um exemplo delas são as funções de entrada e saída: **input()** e **print()**.

Basicamente, uma função é uma sequência de comandos que recebe um nome e pode ser chamada de qualquer parte do programa, quantas vezes forem necessárias, durante a execução do programa.

9.1 Definição e estrutura de uma função



Clique aqui para ver a vídeo aula sobre funções.

Existem duas razões principais para usar funções:

- **estruturação dos programas;**
- **reutilização de código.**

A **estruturação dos programas** nos diz que o programa será construído usando pequenos blocos de código (as funções). Cada bloco tem uma tarefa específica e bem definida. Isso facilita a compreensão do programa.



Programas grandes e complexos são construídos bloco a bloco com a ajuda de funções.

A **reutilização de código** significa que não iremos ficar repetindo o mesmo código ao longo do programa. O código para uma tarefa será definido uma única vez e na forma de uma função, a qual poderá ser utilizada quantas vezes forem necessárias dentro do programa.



O uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros.

9.1.1 Declarando uma função



Clique aqui para ver a vídeo aula sobre a declaração de uma função.

Na linguagem Python, a definição de uma função segue a seguinte estrutura:

```
def nome_da_função(parametro1, parametro2, ...):  
    # Instruções e operações realizadas pela função  
    return resultado
```

Toda função começa com o comando **def**, seguido de um nome associado a aquela função: **nome_da_função**. O nome deve seguir as regras de nomenclatura em Python e é como o seu trecho de código será conhecido dentro do programa.



Para um comando fazer parte do bloco de comandos da função, ele precisar ser indentado. Caso contrário, a função não irá considerar aquele comando.

Diferente de outras linguagens que usam chaves **{}** para delimitar blocos de comandos, a linguagem Python usa a indentação para essa tarefa.



Funções devem ser definidas antes de serem utilizadas.

Com relação ao seu local no código do programa, uma função deve ser sempre definida antes de ser utilizada. Caso contrário, isso resultará em um erro. A seguir é mostrado um exemplo de declaração de função.

Exemplo: declaração de uma função

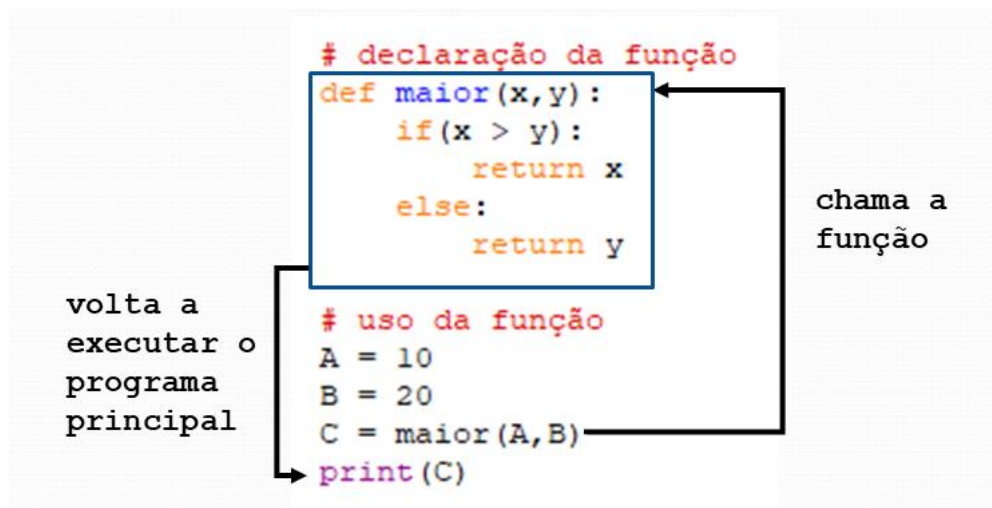
```
01 # declaração da função  
02 def maior(x,y):  
03     if(x > y):  
04         return x  
05     else:  
06         return y  
07  
08 # uso da função  
09 A = 10  
10 B = 20  
11 C = maior(A,B)  
12 print(C)
```

Saída 20



Ao chamar uma função, o programa que a chamou é pausado até que a função termine a sua execução.

Quando um programa chama uma função, ele é interrompido temporariamente, e o fluxo do programa vai para a função. Os comandos da função são executados e quando ela termina, o programa volta ao ponto onde ele foi interrompido e continua sua execução normal. A figura a seguir representa esse processo.



Clique aqui para ver a vídeo aula sobre a ordem de execução da função.

9.1.2 O corpo da função



Clique aqui para ver a vídeo aula sobre o corpo da função.

O corpo da função define a tarefa que a função irá realizar quando for chamada.



O corpo da função é formado pelos comandos que a função deve executar.

O corpo da função pode conter declarações de variáveis, objetos, comandos condicionais, de repetição, chamada de outras funções, etc., qualquer comando que o programador quiser. Ele recebe os parâmetros (se houver), realiza o processamento e gera as saídas (se necessário).



Todos os exemplos de programas que fizemos até aqui, também podem ser feitos na forma de uma função.

Qualquer código pode ser modularizado na forma de uma função. O código a seguir mostra um exemplo de uma função que recebe um valor N e devolva a soma dos valores de 1 até N.

Exemplo: função para somar de 1 a N

```
01 def somaN(N) :  
02     # declara uma variável  
03     S = 0  
04  
05     # executa uma repetição  
06     for i in range(1,N+1) :  
07         S = S + i  
08  
09     # retorna o resultado  
10     return S  
11  
12 print(somaN(5))
```

Nesse exemplo é possível perceber que não foi feito nada de diferente do que temos feito até o momento. A função é apenas uma ferramenta que ajuda a organizar o código em unidades lógicas e bem definidas, facilitando a leitura e manutenção.



Uma função criada pelo programador pode utilizar qualquer outra função, inclusive as que foram criadas.

9.1.3 Os parâmetros da função



Clique aqui para ver a vídeo aula sobre os parâmetros da função.

Os parâmetros permitem ao programador informa quais dados a função deve processar quando for chamada. Eles são definidos como uma lista de variáveis, separadas por vírgula, como mostra o exemplo a seguir:

```
def soma(x,y) :
```

Neste caso, a função soma possui dois parâmetros: **x** e **y**. Mas podemos definir quantos parâmetros forem necessários. Se nenhum dado precisa ser passado para a função, podemos deixar a lista de parâmetros vazia

```
def imprime():
```



Mesmo que a função não tenha parâmetros, os parênteses ainda são necessários.

O exemplo a seguir mostra a implementação de duas funções, uma que recebe parâmetros e outra que não.

Exemplo: parâmetros da função	
Função sem parâmetros	Função com parâmetros
01 def imprime(): 02 print("Teste") 03 04 05 # programa principal 06 imprime() 07 08 09	def soma(x, y): z = x + y return z # programa principal A = 10 B = 20 C = soma(A, B) print(C)

Nesse exemplo, a função **imprime()** apenas exige uma mensagem, então nenhum parâmetro é necessário. Já a função **soma()** necessita que se informe quais valores serão somados quando a função é chamada (linha 8).



Podemos definir **valores padrão** para os parâmetros de uma função.

De modo geral, os valores dos parâmetros de uma função são informados no momento em que chamamos ela. Porém, podemos definir parâmetros que já possuam um valor pré-definido, um **valor padrão**. Isso faz com que o parâmetro se torne opcional, ou seja, se não for definido o valor padrão será usado, como mostra o exemplo a seguir:

```
def reajuste(salario, juros = 0.25):
```

Neste exemplo, a função reajuste possui dois parâmetros: **salário** e **juros**. O valor do parâmetro salário deve ser sempre informado quando a função for chamada, é um **parâmetro obrigatório**. Já o valor de **juros** pode ou não ser informado. Se não for informado, o valor padrão de **0.25** será usado.



Parâmetros com **valores padrão** devem vir sempre depois dos parâmetros com valores obrigatórios.

A seguir podemos ver um exemplo de uso de função com valor padrão.

Exemplo: função com parâmetro com valor padrão

```
01 def reajuste(salario, juros = 0.25):  
02     return salario + salario * juros  
03  
04 # chama a função e especifica o valor do juros  
05 A = reajuste(100,10)  
06  
07 # chama a função e usa o valor padrão do juros  
08 B = reajuste(100)  
09  
10 print(A)  
11 print(B)
```

Saída 110.0
125.0

9.1.4 O retorno da função



Clique aqui para ver a vídeo aula sobre o retorno da função.

Na linguagem Python, uma função definida pelo programador pode ou não retornar um valor.



O retorno da função é a maneira como a função devolve o resultado da sua execução para o trecho de código que a chamou.

O retorno da função é definido pela clausula **return**, especificada dentro da função, como mostra o exemplo a seguir:

```
def função():  
  
    # Instruções e operações  
    # realizadas pela função  
  
    return valor
```

O termo **valor** pode ser qualquer combinação de valores, variáveis ou chamadas de funções utilizando (ou não) os operadores matemáticos



A cláusula **return** é **opcional**.

Não é obrigatório definir uma cláusula **return** para a sua função. Ela é necessária se queremos que a função devolva o resultado de algum cálculo realizado dentro da função. Uma função pode apenas executar um conjunto de comandos e não gerar nenhum tipo de resultado. Por exemplo, podemos querer uma função que recebe uma lista e salva ela em um arquivo. Neste caso, não há nenhum valor para ser retornado e podemos omitir a cláusula **return**.



Se a cláusula **return** for definida, então alguém deverá receber o resultado da função.

Como dito antes, a cláusula **return** devolve o resultado de algum cálculo realizado dentro da função. Assim, o código que chamou a função deverá receber esse resultado com um comando de atribuição, como mostra o exemplo a seguir.

Exemplo: retorno da função	
Função sem retorno	Função com retorno
<pre>01 def imprime(lis): 02 for valor in lis: 03 print(valor) 04 05 lista = [1,2,3,4,5] 06 imprime(lista) 07 08 09</pre>	<pre>def soma(lis): s = 0 for valor in lis: s = s + valor return s lista = [1,2,3,4,5] res = soma(lista) print(res)</pre>

Nesse exemplo, a função **imprime()** não possui o comando **return**. Para executá-la, basta chamar o seu nome. Já a função **soma()** retorna um valor (linha 5) e necessita que uma variável receba o seu resultado (linha 8).



Quando o comando **return** é executado, a função termina imediatamente.

Além de retornar um valor, o comando **return** também encerra a chamada da função, parecido com o comando **break** em um laço **for** ou **while**. Comando definidos após o **return** são ignorados, como mostra o exemplo a seguir.

Exemplo: finalizando a função com return

```
01 def soma(lis):
02     s = 0
03     for valor in lis:
04         s = s + valor
05     return s
06     print('Fim da função')
07
08 lista = [1,2,3,4,5]
09 res = soma(lista)
10 print(res)
```

Nesse exemplo, a função irá terminar quando o comando **return** for executado. A mensagem “Fim da funcao” (linha 6) jamais será exibida pois ela se encontra após o comando **return**.



Uma função pode ter mais de um comando **return**.

Podemos fazer uso de vários comandos **return** dentro da função, cada um devolvendo um resultado diferente dependendo das condições executadas, como mostra o exemplo a seguir.

Exemplo: função com vários return

```
01 def maior(x, y):
02     if(x > y):
03         return x
04     else:
05         return y
```

9.2 Cuidados com o escopo de variáveis



Clique aqui para ver a vídeo aula sobre o escopo das variáveis na função.

O escopo é o conjunto de regras que determinam o uso e a validade da variável ao longo do programa. Ele define **onde** no programa a variável pode ser usada e depende do local onde a variável foi definida.



A variável definida fora de qualquer função pertence ao **escopo global** e é chamada de **variável global**.

A **variável global** pode ser acessada em qualquer lugar do programa, inclusive dentro de uma função. Seu tempo de vida é o tempo de execução do programa. A seguir podemos ver um exemplo de variável local.

Exemplo: usando uma variável global

```
01 def func():
02     print('X = ', x)
03
04 x = 10
05 func()

Saída X = 10
```

Nesse exemplo, a variável **x** foi definida sem indentação, fora de qualquer função. É uma **variável global** e pode ser acessada por qualquer parte do código, inclusive dentro de uma função (linha 2).



Uma **variável global** pode ser acessada por uma função, mas não pode ser alterada pela função.

Atribuições dentro da função geram novos objetos. Ao tentar alterar uma variável global dessa forma, o que ocorre é a criação de uma **variável local** que ofusca completamente a variável global. A seguir podemos ver um exemplo.

Exemplo: tentando modificar uma variável global

```
01 def funcao():
02     x = 20
03     print('X = ', x)
04
05 x = 10
06 funcao()
07 print('X = ', x)

Saída X = 20
      X = 10
```

Nesse exemplo, a variável global **x** foi modificada dentro da função usando uma atribuição. Isso gera um novo objeto que oculta o original, preservando o seu valor original fora da função (linha 7).



Para atribuir um novo valor a uma variável global precisamos utilizar o comando **global**.

O comando **global** informa ao interpretador Python que a variável manipulada dentro da função pertence ao escopo global. Isso impede que um novo objeto seja criado e garante que as alterações serão realizadas na variável global, como mostra o exemplo a seguir.

Exemplo: usando o comando global	
01	def funcao():
02	global x
03	x = 20
04	print('X = ',x)
05	
06	x = 10
07	funcao()
08	print('X = ',x)
Saída X = 20	
X = 20	

Nesse exemplo, com o uso do comando **global**, as modificações na variável global **x** também se refletem fora da função (linha 8).



A variável definida dentro de uma função ou na sua lista de parâmetros pertence ao **escopo local** e é chamada de **variável local**.

A **variável local** somente pode ser acessada dentro da função, nunca fora dela. Seu tempo de vida é o tempo de execução da função. A seguir podemos ver um exemplo.

Exemplo: usando uma variável local	
01	def funcao():
02	y = 20
03	print('Y = ',y)
04	
05	funcao()
06	print('Y = ',y)
Saída Y = 20	
Traceback (most recent call last):	
File "example.py", line 6, in <module>	
print('Y = ',y)	
NameError: name 'y' is not defined	

Nesse exemplo, a variável global **y** foi criada dentro da função e somente lá ela pode ser manipulada. Tentar acessar essa variável fora da função irá gerar um erro (linha 6).

9.3 Cuidados com a passagem de parâmetros



Clique aqui para ver a vídeo aula sobre a passagem de parâmetros para a função.

Os parâmetros de uma função é o meio que o programador utiliza para passar dados de um trecho de código para dentro da função.



Nas diferentes linguagens de programação, o tipo de passagem de parâmetros define como as modificações realizadas nos valores dos parâmetros irão se refletir no programa principal.

Em Python, sempre que passamos um parâmetro para a função, estamos passando a **referência** a um objeto via atribuição. Esse parâmetro pode ou não ser modificado, sendo definido como **mutável** ou **imutável**.



Atribuições dentro da função geram novos objetos, fazendo com que o valor do parâmetro passado se torne **imutável**.

Um parâmetro é dito **imutável** quando o seu valor é modificado dentro da função via atribuição. A atribuição gera um novo objeto, que oculta o objeto original. Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função, como mostra o exemplo a seguir.

Exemplo: parâmetro imutável

```
01 def incrementa(N):  
02     print('valor = ',N)  
03     N = N + 1  
04     print('valor = ',N)  
05  
06 x = 1  
07 incrementa(x)  
08 print('valor = ',x)
```

```
Saída valor = 1  
      valor = 2  
      valor = 1
```

Nesse exemplo, o valor de **x** é copiado para o parâmetro **N**, que é modificado dentro da função por uma atribuição. Isso gera um novo objeto que oculta o objeto original, preservando o seu valor original fora da função (linha 8).



Um parâmetro é considerado **mutável** se o seu valor é modificado sem usar o operador de atribuição.

Um parâmetro é dito **mutável** quando o seu valor é modificado dentro da função sem usar a operação de atribuição. Assim, um novo objeto não é gerado e as alterações no seu valor são percebidas fora da função, como mostra o exemplo a seguir.

Exemplo: parâmetro mutável

```
01 def incrementa(lis):
02     print('lis = ', lis)
03     lis.append(100)
04     print('lis = ', lis)
05
06 lista = [1, 2, 3, 4, 5]
07 incrementa(lista)
08 print('lis = ', lista)
```

Saída lis = [1, 2, 3, 4, 5]
lis = [1, 2, 3, 4, 5, 100]
lis = [1, 2, 3, 4, 5, 100]

Nesse exemplo, o conteúdo da lista foi modificado usando um método de manipulação de lista, **append()**. O conteúdo da lista foi modificado dentro da função, mas sem fazer uso de uma atribuição. Como essa tarefa não gerou um novo objeto dentro da função, as modificações na lista se refletem na lista original, definida fora da função (linha 8).

9.4 Recursão



[Clique aqui para ver a vídeo aula sobre recursão \(parte 1\).](#)

Na linguagem Python, assim como em outras linguagens, uma função pode chamar a outra função. No exemplo a seguir, a função **imprime()** faz uma chamada a função **print()**.

Exemplo: chamadas de funções

```
01 def imprime(lista):
02     for item in lista:
03         print(item)
04
05 li = [1, 2, 3, 4, 5]
06 imprime(li)
```

Quando uma função é chamada dentro de si mesma, ela cria uma pilha de chamadas (*stack*) que armazena cada chamada da função em ordem. Cada chamada da função é resolvida separadamente e resulta em uma etapa do problema sendo resolvida. A recursão continua até que a condição de parada seja atingida, momento em que as chamadas da função começam a ser resolvidas a partir da pilha de chamadas.

Um exemplo clássico de recursão é o cálculo do fatorial de um número.



Como calcular o fatorial de 4 (definido como 4!)?

Solução: para calcular o fatorial de 4, multiplica-se o número 4 pelo fatorial de 3 (definido como 3!). Nesse ponto, já é possível perceber que esse problema é muito semelhante ao do vaso de flores. Generalizando esse processo temos que: o fatorial de N é igual a N multiplicado pelo fatorial de $(N-1)$, ou seja, $N! = N * (N - 1)!$. No caso do vaso de flores, o processo termina quando não há mais flores no vaso. No caso do fatorial, o processo irá terminar quando atingirmos o número zero. Nesse caso, o valor do fatorial de 0 ($0!$) é definido como sendo igual a 1. Temos então que a função fatorial é definida matematicamente como

$$0! = 1$$

$$N! = N * (N - 1)!$$

Note novamente que o fatorial de N , $N!$, é definido em termos do fatorial de $N - 1$, $(N - 1)!$, ou seja, trata-se de uma definição circular do problema: precisamos saber o fatorial de um número para calcular o de outro. Esse processo continua até que se chegue a um caso mais simples e que é a base do cálculo do fatorial: o fatorial de zero, $0!$. Nesse momento, a recursão para pois temos um valor constante já definido para o fatorial. A seguir, é possível ver a função recursiva para o cálculo do fatorial.

Exemplo: função recursiva para calcular o fatorial

```
01 def fatorial(N) :  
02     if(N == 0):  
03         return 1  
04     else:  
05         return N * fatorial(N-1)  
06
```

9.4.1 Como funciona a recursão

A ideia básica da recursão é **dividir e conquistar**:

- Um problema maior é dividido em um conjunto de problemas menores;
- Esses problemas menores são resolvidos de forma independente;
- As soluções dos problemas menores são combinadas para gerar a solução final.

Essa lógica fica evidente no cálculo do fatorial.



Como calcular o fatorial do número 4 (definido como 4!)?

Matematicamente, o fatorial de um número **N** é definido como

$$0! = 1$$

$$N! = N * (N - 1)!$$

O fatorial de **N** é o produto de todos os números inteiros entre 1 e N. Por exemplo, o fatorial de 4 é **1 * 2 * 3 * 4**. Aplicando a ideia da recursão, temos que

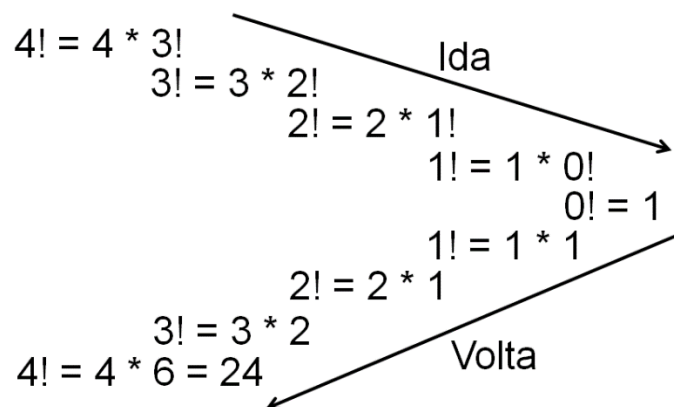
- o fatorial de 4 é definido em função do fatorial de 3;
- o fatorial de 3 é definido em função do fatorial de 2;
- o fatorial de 2 é definido em função do fatorial de 1;
- o fatorial de 1 é definido em função do fatorial de 0;
- o fatorial de 0 é definido como sendo igual a 1.

Perceba que o cálculo do fatorial prossegue até chegarmos no fatorial de 0, nosso **caso base**, que é igual a 1.



Quando a função recursiva chega no seu **caso base**, ela para.

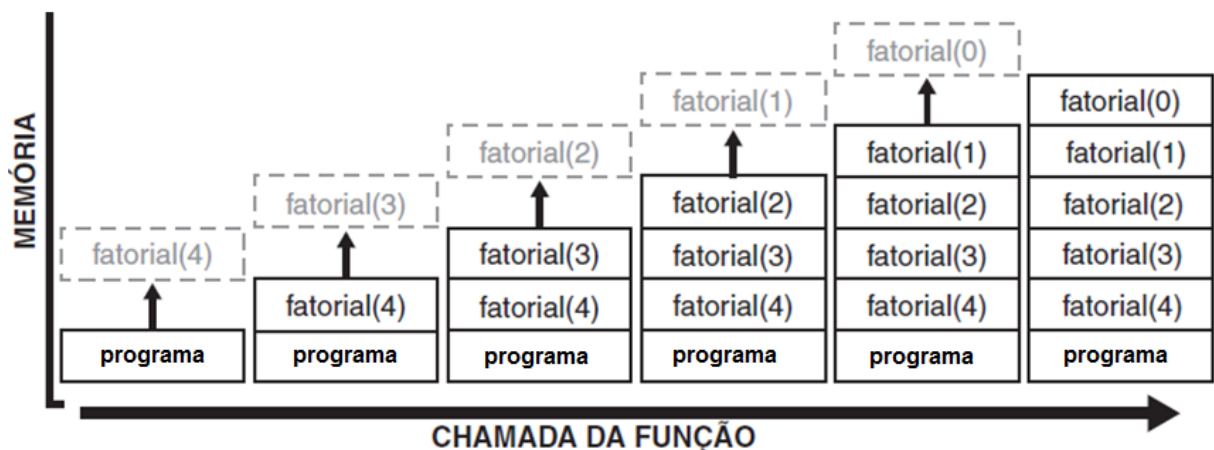
Vamos chamar essa primeira etapa do cálculo de **caminho de ida da recursão**, que é onde as chamadas do fatorial são executadas até chegar no caso base. Quando a recursão para, é chegado o momento de fazer o **caminho de volta da recursão** que, basicamente, consiste em fazer o caminho inverso, mas agora devolvendo o valor obtido para quem fez aquela chamada do fatorial e assim por diante, até chegarmos na primeira chamada do fatorial, como mostra a figura a seguir.



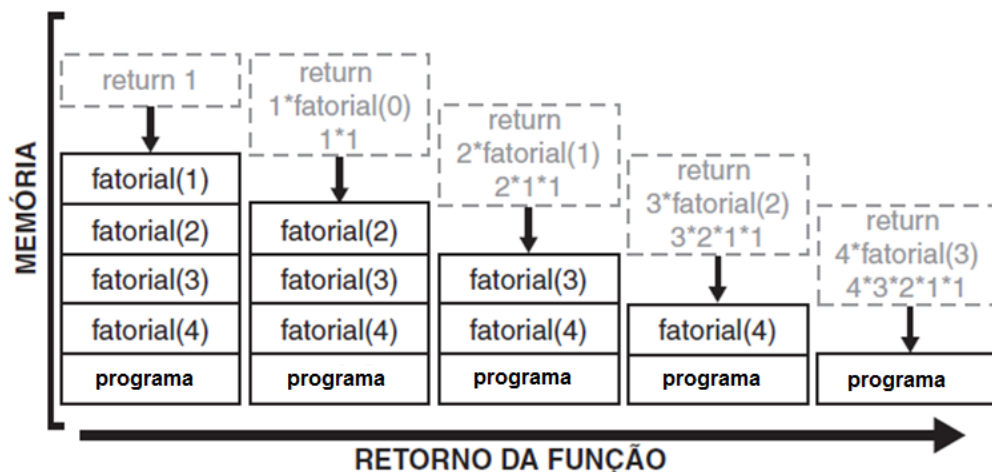
Os mesmos princípios de **caso base**, **caminho de ida da recursão** e **caminho de volta da recursão**, presentes no exemplo matemático da recursão, também estão também presentes na recursão de uma função computacional. Considere o exemplo a seguir.

Exemplo: calculando o fatorial	
01	<code>def fatorial(N):</code>
02	<code> if(N == 0):</code>
03	<code> return 1</code>
04	<code> else:</code>
05	<code> return N * fatorial(N-1)</code>
06	
07	<code>F = fatorial(4)</code>
08	<code>print(F)</code>

Sempre que fazemos uma chamada de função, seja ela qual for, o código atualmente em execução é pausado, e a função chamada é carregada na memória. Assim, o programa Python em execução é pausado quando ele chama a função **fatorial()**. Note que a função **fatorial()** também chama a si própria. Nesse caso, ela é pausada e carrega na memória uma cópia de si mesma, mas com valor de parâmetro diferente (**N-1**). Esse processo se repete até que a função seja chamada com o valor de **N=0**. Este é o **caso base** onde a função para de chamar a si mesma, como mostra a figura a seguir.



Uma vez que chegamos ao **caso base**, é hora de fazer o **caminho de volta da recursão**. Sempre que uma função termina de executar, ela devolve (se existir) seu resultado para quem a chamou e, em seguida, é removida da memória. Assim, a chamada da função **fatorial(0)** irá devolver o valor **1** para quem a chamou, **fatorial(1)**; a chamada da função **fatorial(1)** irá devolver o valor **1*1** para quem a chamou, **fatorial(2)**; e assim por diante, como mostra a figura a seguir.



Saber identificar o **caso base**, **caminho de ida da recursão** e **caminho de volta da recursão**, torna a construção de uma função com recursão muito mais simples.

9.4.2 Cuidados durante a implementação da recursão




Clique aqui para ver a vídeo aula sobre recursão (parte 2).

O exemplo a seguir apresenta duas funções para o cálculo do fatorial: com e sem recursão.

Exemplo: fatorial	
Com Recursão	Sem Recursão
<pre> 01 def fatorial(N): 02 if(N == 0): 03 return 1 04 else: 05 return N * fatorial(N-1) 06 </pre>	<pre> def fatorial(N): fat = 1 for i in range(1,N+1): fat = fat * i return fat </pre>

Como podemos ver, a forma recursiva dos algoritmos são, em geral, consideradas “mais enxutas” e “mais elegantes” do que suas formas iterativas. Isso facilita a interpretação do código. Porém, esses algoritmos apresentam maior dificuldade na detecção de erros. Além disso, muitas vezes eles apresentam uma eficiência computacional menor que a versão iterativa.



Todo cuidado é pouco ao criar uma função recursiva. Duas coisas devem sempre ficar bem definidas: o *critério de parada* e o *parâmetro da chamada recursiva*.

Durante a implementação de uma função recursiva temos sempre que ter em mente duas coisas: o **critério de parada** e o **parâmetro da chamada recursiva**:

- **Critério de parada:** determina quando a função deve parar de chamar a si mesma, ou seja, quando a recursão termina. Se ele não existir, a função irá continuar executando até esgotar a memória do computador. No cálculo de fatorial, o critério de parada ocorre quando tentamos calcular o fatorial do número zero: $0! = 1$.
- **Parâmetro da chamada recursiva:** quando fazemos uma chamada recursiva, devemos sempre mudar o valor do parâmetro passado. Dessa forma, a recursão poderá chegar ao seu final. Se o valor do parâmetro for sempre o mesmo a função irá continuar executando até esgotar a memória do computador. No cálculo de fatorial, a mudança no parâmetro da chamada recursiva ocorre quando definimos o fatorial de N em termos no fatorial de $(N-1)$: $N! = N * (N - 1)!$.

O exemplo a seguir deixa bem claro o **critério de parada** e o **parâmetro da chamada recursiva** na função recursiva implementada em linguagem Python.

Exemplo: fatorial	
<pre> 01 def fatorial(N): 02 if(N == 0): # critério de parada 03 return 1 04 else: #parâmetro do fatorial sempre muda 05 return N * fatorial(N-1) </pre>	

Note que a implementação da função recursiva do fatorial em linguagem Python segue exatamente o que foi definido matematicamente.



Algoritmos recursivos tendem a necessitar de mais tempo e/ou espaço do que algoritmos iterativos.

Sempre que chamamos uma função, é necessário um espaço de memória para armazenar os parâmetros, variáveis locais e endereço de retorno da função. Numa função recursiva, essas informações são armazenadas para cada chamada da recursão, sendo, portanto, a memória necessária para armazená-las proporcional ao número de chamadas da recursão. Por exemplo, para calcular o fatorial do número 4 são necessárias 5 chamadas da função fatorial.

Além disso, todas essas tarefas de alocar e liberar memória, copiar informações, etc. envolvem tempo computacional, de modo que uma função recursiva sempre gasta mais tempo que sua versão iterativa (sem recursão).

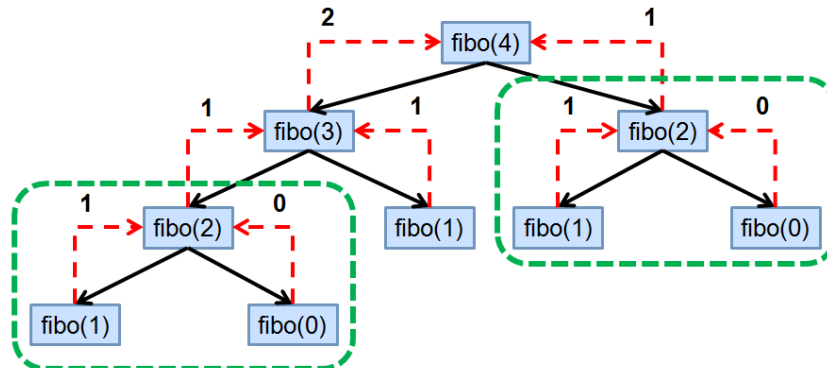
Outro exemplo clássico de recursão é a sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula:

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{outros casos} \end{cases}$$

O exemplo a seguir apresenta as funções para o cálculo da sequência de Fibonacci implementadas com e sem recursão.

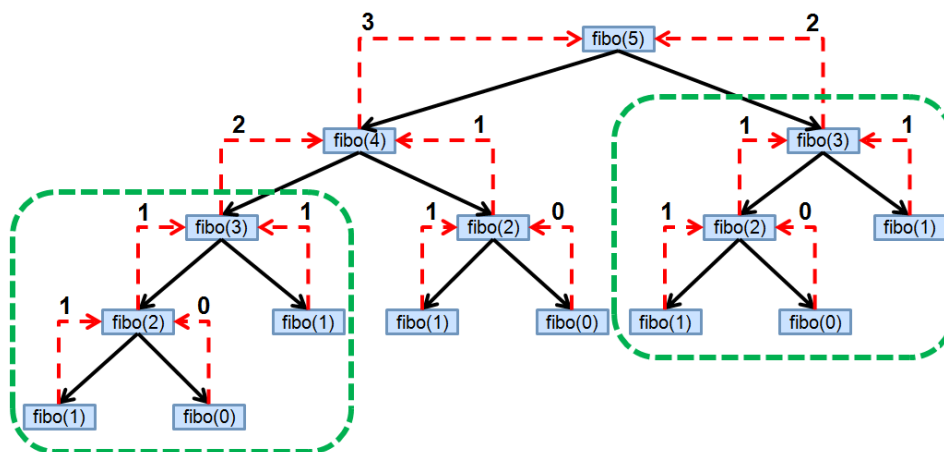
Exemplo: sequência de Fibonacci	
Com Recursão	
01	def fibo(n):
02	if (n == 0 or n == 1):
03	return n
04	else :
05	return fibo(n-1) + fibo(n-2)
Sem Recursão	
01	def fibo(n):
02	if (n == 0 or n == 1):
03	return n
04	else :
05	A = 0
06	B = 1
07	cont = 1
08	while (cont < n):
09	C = A + B
10	cont = cont + 1
11	A = B
12	B = C
13	return C

Como se nota, a solução recursiva para a sequência de Fibonacci é mais enxuta e muito elegante. Infelizmente, ela contém duas chamadas recursivas para si mesma. Logo sua elegância não significa eficiência, como se verifica no diagrama a seguir.



Nesse diagrama, as setas pretas indicam quando uma nova chamada da função é realizada. Já as setas vermelhas indicam o processo inverso, ou seja, quando a função devolve o valor do comando **return** para quem a chamou. O maior problema dessa solução recursiva está nos quadrados marcados com o pontilhado verde. Está claro que parte do cálculo é realizado duas vezes pelas duas chamadas recursivas: **um grande desperdício de tempo e espaço!**

Se, ao invés de calcularmos **fibo(4)** quisermos calcular **fibo(5)**, teremos um desperdício ainda maior de tempo e espaço, como mostra o diagrama a seguir.



Por esse motivo, devemos sempre ter muito cuidado com o uso da recursão na programação.



Recursão é realmente um tópico difícil de entender em programação. Para entender melhor a recursão, leia o a Seção 9.3.

10.1 O que é e os tipos de arquivos existentes



Clique aqui para ver a vídeo aula sobre arquivos.

Em programação, um arquivo é uma coleção de dados armazenados em um dispositivo de armazenamento, como disco rígido, memória USB ou até mesmo em um servidor remoto. Esses dados podem ser qualquer tipo de informação, como

- texto
- imagens
- áudio
- vídeo
- e muito mais.

Arquivos são uma forma essencial de persistência de dados, permitindo que os programas armazenem informações entre execuções.



Os arquivos desempenham um papel fundamental em muitos programas e sistemas. Eles permitem que os dados sejam mantidos após a finalização de um programa, possibilitando a continuidade do trabalho em sessões futuras.

Além disso, os arquivos possibilitam a troca de informações entre diferentes aplicativos e até mesmo entre diferentes sistemas.

As vantagens de trabalhar com arquivos:

- **Persistência de Dados:** Arquivos permitem que os dados sejam armazenados permanentemente, garantindo que as informações não sejam perdidas após o encerramento do programa.
- **Compartilhamento de Informações:** Arquivos facilitam a transferência e compartilhamento de dados entre diferentes aplicativos, dispositivos ou usuários.
- **Manipulação de Grandes Volumes de Dados:** Arquivos são ideais para armazenar grandes volumes de dados que podem não caber totalmente na memória do sistema.
- **Histórico e Logs:** Arquivos de histórico e logs são usados para registrar eventos e atividades, o que é crucial para diagnósticos e solução de problemas.

- **Configurações e Preferências:** Muitos programas usam arquivos para armazenar configurações, preferências do usuário e outras informações personalizadas.
- **Backup e Recuperação:** Arquivos são facilmente copiados e transferidos, o que facilita a criação de cópias de backup para proteger contra perda de dados.
- **Integração com Bancos de Dados:** Arquivos frequentemente servem como uma camada intermediária para importar/exportar dados entre aplicativos e bancos de dados.

Basicamente, a linguagem Python trabalha com dois tipos de arquivos: **arquivos texto** e **arquivos binários**.



Um **arquivo texto** armazena dados legíveis por humanos, como documentos, scripts e configurações.

Num arquivo texto, os dados gravados podem ser mostrados diretamente na tela ou modificados por um editor de textos simples, como o Bloco de Notas. Os dados são gravados como caracteres de 8 bits utilizando a tabela ASCII. Durante a gravação e leitura dos dados existe uma etapa de “conversão” dos dados, o que pode fazer com que o espaço em disco ocupado pelo arquivo fique maior do que o seu espaço na memória RAM.



Um **arquivo binário** armazena dados em formato binário, sendo usados para armazenar informações não legíveis diretamente, como imagens, áudio e executáveis.

Um arquivo binário armazena uma sequência de bits que está sujeita as convenções dos programas que o gerou. Os dados são gravados exatamente como estão organizados na memória do computador, sem etapa de conversão

10.2 Abrindo e fechando um arquivo



Clique aqui para ver a vídeo aula sobre como abrir e fechar arquivos.

A primeira coisa que devemos fazer ao se trabalhar com arquivos é abri-lo. Para isso utilizamos a função **open()**, cujo protótipo é:

```
objeto_arquivo = open(nome_do_arquivo, modo)
```

A função **open()** recebe 2 parâmetros de entrada

- **nome_do_arquivo**: uma string contendo o nome do arquivo que deverá ser aberto;
- **modo**: uma string contendo o modo de abertura do arquivo.

e retorna

- um objeto para o arquivo aberto com sucesso.
- Um erro caso o arquivo não possa ser aberto.



No parâmetro **nome_do_arquivo** pode-se trabalhar com caminhos **absolutos** ou **relativos**.

Imagine que o arquivo com que desejamos trabalhar esteja no seguinte local:

`"C:\Projetos\NovoProjeto\arquivo.txt"`

O **caminho absoluto** de um arquivo é uma sequência de diretórios separados pelo caractere barra ('\'), que se inicia no diretório raiz e termina com o nome do arquivo. Nesse caso, o **caminho absoluto** do arquivo é a string

`"C:\Projetos\NovoProjeto\arquivo.txt"`

Já o **caminho relativo**, como o próprio nome diz, é relativo ao local onde o programa se encontra. Nesse caso, o sistema inicia a pesquisa pelo nome do arquivo a partir do diretório do programa. Se tanto o programa quanto o arquivo estiverem no mesmo local, o caminho relativo até esse arquivo será

`".\arquivo.txt"`

ou

`"arquivo.txt"`

Se o programa estivesse no diretório "C:\Projetos", o **caminho relativo** até o arquivo seria

`".\NovoProjeto\arquivo.txt"`



O modo de abertura do arquivo determina que tipo de uso será feito do arquivo.

O modo de abertura do arquivo diz à função **open()** como usaremos o arquivo. Pode-se, por exemplo, querer escrever em um arquivo binário ou ler de um arquivo texto. A tabela a seguir mostra os modos válidos de abertura de um arquivo:

Modo	Arquivo	Função
"r"	Texto	Leitura. Arquivo deve existir.
"w"	Texto	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a"	Texto	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"rb"	Binário	Leitura. Arquivo deve existir.
"wb"	Binário	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"ab"	Binário	Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+"	Texto	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+"	Texto	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+"	Texto	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").
"r+b"	Binário	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
"w+b"	Binário	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
"a+b"	Binário	Leitura/Escrita. Os dados serão adicionados no fim do arquivo ("append").

Note que para cada tipo de ação que o programador deseja realizar existe um modo de abertura de arquivo mais apropriado.



O arquivo deve sempre ser aberto em um modo que permita executar as operações desejadas.

Imagine que desejemos gravar uma informação em um arquivo texto. Obviamente, esse arquivo deve ser aberto em um modo que permita escrever nele. Já um arquivo aberto para leitura não irá permitir outra operação que não seja a leitura de dados.

Exemplo: abrindo um arquivo texto para escrita

```
01 f = open('teste.txt', 'w')
02 f.write('teste de escrita')
03 f.close()
```

No exemplo anterior, o comando **open()** tenta abrir um arquivo de nome "teste.txt" no modo de escrita para arquivos textos, "w". Note que foi utilizado o **caminho relativo** do arquivo.



Sempre que terminamos de usar um arquivo, devemos fechá-lo.

Para realizar essa tarefa, usamos o método **close()**, cujo protótipo é:

```
objeto_arquivo.close()
```

Basicamente, o método **close()** fecha o arquivo indicado pelo objeto_arquivo e aberto com a função **open()**.



Por que devemos fechar o arquivo?

Ao fechar um arquivo, todo caractere que tenha permanecido no “buffer” é gravado. O “buffer” é uma área intermediária entre o arquivo no disco e o programa em execução. Trata-se de uma região de memória que armazena temporariamente os caracteres a serem gravados em disco. Apenas quando o “buffer” está cheio é que seu conteúdo é escrito no disco.



Por que utilizar um “buffer” durante a escrita em um arquivo?

O uso de um “buffer” é uma questão de **eficiência**. Para ler e escrever arquivos no disco rígido é preciso posicionar a cabeça de gravação em um ponto específico do disco rígido. E isso consome tempo. Se tivéssemos que fazer isso para cada caractere lido ou escrito, as operações de leitura e escrita de um arquivo seriam extremamente lentas. Assim a gravação só é realizada quando há um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.

Exemplo: abrindo e fechando um arquivo texto para escrita

```
01 f = open('compras.txt', 'w')
02 compras = ['pão', 'leite', 'manteiga', 'queijo']
03
04 f.write('Lista de compras\n')
05 for item in compras:
06     f.write('Produto:'+item+'\n')
07
08 f.close()
```

No exemplo acima, se retirarmos do programa o comando **close()** (linha 8), não teremos nenhuma garantia de que o conteúdo foi realmente escrito no arquivo.



Podemos utilizar a instrução **with** para garantir o fechamento do arquivo ao final do seu uso..

A instrução **with** garante a aquisição e liberação adequada de recursos. Neste caso, não há necessidade de chamar o método **close()** para fechar um arquivo ao usar a instrução **with**. Um exemplo é mostrado a seguir.

Exemplo: usado a instrução with para fechar um arquivo

```
01 with open("compras.txt", "r") as f:  
02     str = f.read()  
03     print(str)
```

10.3 Escrevendo dados em um arquivo



Clique aqui para ver a vídeo aula sobre como escrever dados em arquivos.

10.3.1 O método write()

O método **write()** é usado para escrever dados em um arquivo em Python. Ele pertence aos objetos de arquivo, e seu protótipo é:

```
objeto_arquivo.write(string)
```

Basicamente, esse método recebe como parâmetro um objeto **string** que será escrita no arquivo aberto especificado por **objeto_arquivo**.



O método **write()** trabalha apenas com a escrita de strings. Cabe ao programador tratar os dados que serão escritos no arquivo.

Para gravar outros tipos de dados, como valores inteiros e pontos flutuantes, é necessário antes fazer a conversão destes para uma string. A seguir podemos ver um exemplo de uso do método **write()**.

Exemplo: usando o método write()

```
01 f = open('compras.txt','w')
02 compras = ['pão','leite','manteiga','queijo']
03
04 f.write('Lista de compras\n')
05 for item in compras:
06     f.write('Produto:'+item+'\n')
07
08 qtd = len(compras)
09 f.write('Quantidade: '+str(qtd))
10
11 f.close()
```

Nesse exemplo, usamos o método **write()** para escrever um conjunto de strings e a quantidade de strings (linhas 8 e 9), um valor inteiro e que foi convertido para string usando a função **str()**.

10.3.2 O método writelines()

O método **writelines()** é usado para escrever múltiplas strings de uma única vez em um arquivo. Ele pertence aos objetos de arquivo, e seu protótipo é:

objeto_arquivo.writelines(lista_de_string)

Basicamente, esse método recebe como parâmetro uma lista contendo vários objetos **string** que serão escritos no arquivo aberto especificado por **objeto_arquivo**.

A seguir podemos ver um exemplo de uso do método **writelines()**.

Exemplo: usando o método writelines()

```
01 f = open('compras.txt','w')
02 compras = ['pão','leite','manteiga','queijo']
03
04 f.writelines(compras)
05
06 f.close()
```

Nesse exemplo, usamos o método **writelines()** para escrever um conjunto de strings definidas dentro da lista **compras** (linha 2).

10.4 Lendo dados de um arquivo



Clique aqui para ver a vídeo aula sobre como ler dados de arquivos.

10.4.1 O método `read()`

O método **`read()`** é usado para ler uma sequência de bytes de um arquivo, os quais são retornados na forma de uma string. Ele pertence ao objeto arquivo, e seu protótipo é:

```
string_lida = objeto_arquivo.read(N)
```

ou

```
string_lida = objeto_arquivo.read()
```

Basicamente, esse método retorna **N** caracteres lidos (a partir do ponto em que atualmente se encontra) do arquivo especificado por **objeto_arquivo**. Caso o valor de **N** não seja especificado, o método lê os caracteres restantes até o final do arquivo.

A seguir podemos ver um exemplo de uso do método **`read()`**.

Exemplo: usando o método `read()`

```
01 f = open('compras.txt', 'r')
02
03 str = f.read(5)
04 print(str)
05
06 str = f.read(5)
07 print(str)
08
09 f.close()
```

Nesse exemplo, usamos o método **`read()`** para ler 5 caracteres (linha 3) e depois, usamos novamente esse método para ler o restante dos caracteres, até o final do arquivo (linha 6).

10.4.2 O método `readline()`

O método **`readline()`** é usado para ler uma sequência de bytes de um arquivo, os quais são retornados na forma de uma string. Ele lê tantos caracteres forem

necessários até encontrar o caractere '\n'. Ele pertence ao objeto arquivo, e seu protótipo é:

```
string_lida = objeto_arquivo.readline()
```

Basicamente, esse método é usado para ler uma linha do arquivo texto.



O caractere de nova linha (\n ou ENTER) lido com o método **readline()** fará parte da string retornada.

A seguir podemos ver um exemplo de uso do método **readline()**.

Exemplo: usando o método **readline()**

```
01 f = open('compras.txt', 'r')
02
03 str = f.readline()
04 print(str)
05
06 f.close()
```

10.4.3 O método **readlines()**

O método **readlines()** é semelhante ao método **readline()** visto na seção anterior. No entanto, ao invés de ler uma única linha do arquivo, esse método lê todas as linhas e as retorna como strings armazenadas em uma lista. Ele pertence ao objeto arquivo, e seu protótipo é:

```
lista_string_lida = objeto_arquivo.readlines()
```

Basicamente, esse método é usado para ler todo o conteúdo do arquivo de uma única vez.



Enquanto o método **read()** lê todo o conteúdo do arquivo e o retorna como uma única string, o método **readlines()** devolve uma lista onde cada posição contém uma linha do arquivo lido.

A seguir podemos ver um exemplo de uso do método **readlines()**.

Exemplo: usando o método **readlines()**

```
01 f = open('compras.txt', 'r')
02
03 str = f.readlines()
04 print(str)
05
06 f.close()
```

10.5 Lendo até o final do arquivo



Clique aqui para ver a vídeo aula sobre como ler até o final do arquivo.

A linguagem Python possui alguns métodos que permitem automaticamente ler e retornar todo o conteúdo do arquivo, como os métodos **read()** e **readlines()**. No entanto, em algumas situações, pode ser necessário ler o arquivo um pedacinho por vez, processar, e somente então mover para o próximo trecho do arquivo. Neste caso, precisamos saber quando chegamos ao final do arquivo.



Diferente de outras linguagens, a linguagem Python não possui uma função específica para informar se já chegamos ao final do arquivo.

A informação sobre o final do arquivo (**EOF, End Of File**) já está codificada dentro de outros métodos, como o **read()**. Esse método lê até o final do arquivo, então depois que terminar com sucesso você saberá que o arquivo está em **EOF**. Não há necessidade de verificar. Se não conseguir atingir o **EOF**, o método irá gerar uma exceção.



Para arquivos texto, podemos verificar se o resultado da leitura é uma string vazia. Esse valor é retornado sempre que chegamos ao final do arquivo, **EOF**.

O código a seguir mostra como ler um arquivo uma linha por vez usando o método **readline()**. Para fazer isso, a leitura é realizada dentro de um laço **while**, o qual é finalizado sempre que o resultado da leitura é uma string vazia (linha 5).

Exemplo: lendo até o final de um arquivo texto

```
01 f = open('compras.txt', 'r')
02
03 while True:
04     str = f.readline()
05     if(str == ''):
06         break
07     print(str)
```

10.6 Sabendo a posição atual dentro do arquivo

Outra operação bastante comum é saber onde estamos dentro de um arquivo. Para realizar essa tarefa, usamos o método **tell()**. Ele pertence ao objeto arquivo, e seu protótipo é:

```
posição = objeto_arquivo.tell()
```

Basicamente, esse método retorna um valor inteiro indicando o número de bytes lidos a partir do início do arquivo. A seguir temos um exemplo de um programa que utiliza o método **tell()** para retornar a posição do arquivo após ler a sua primeira linha.

Exemplo: usando o método tell()

```
01 f = open('compras.txt', 'r')
02
03 str = f.readline()
04 print('Posição: ', f.tell())
05
06 f.close()
```

10.7 Movendo-se dentro do arquivo

De modo geral, o acesso a um arquivo é quase sempre feito de modo sequencial. Porém, a linguagem Python permite realizar operações de leitura e escrita randômica. Para isso, usamos o método **seek()**. Esse método pertence ao objeto arquivo, e seu protótipo é:

```
posição = objeto_arquivo.seek(numero_bytes, origem)
```



Basicamente, o método **seek()** move a posição atual de leitura ou escrita no arquivo para um byte específico, a partir de um ponto especificado.

O método **seek()** recebe 2 parâmetros de entrada

- **numero_bytes**: é o total de bytes que devem ser pulados a partir de **origem**;
- **origem**: determina a partir de onde os **numero_bytes** de movimentação serão contados. Se não for especificado, esse parâmetro assume o valor 0 (ZERO).

Os valores possíveis para o parâmetro **origem** são:

Valor	Significado
0	Movimenta a partir do início do arquivo
1	Movimenta a partir da posição atual no arquivo
2	Movimenta a partir do final do arquivo

Portanto, para movermos **numero_bytes** a partir do início do arquivo, a **origem** deve ser igual a 0. Se quisermos mover a partir da posição atual em que estamos no arquivo, devemos usar o valor 1. E, por fim, se quisermos nos mover a partir do final do arquivo, o valor 2 deverá ser usado.



Os valores posição atual / final do arquivo para o parâmetro **origem** só podem ser usados com o arquivo aberto em modo binário.

Um exemplo de uso do método **seek()** é mostrado a seguir.

Exemplo: usando o método seek()

```
01 f = open('compras.txt', 'r')
02
03 # pula 20 bytes a partir do início do arquivo
04 f.seek(20)
05
06 #imprime a posição atual
07 print('Posição: ', f.tell())
08
09 str = f.readline()
10 print('Lido: ', str)
11
12 f.close()
```



O valor do parâmetro **numero_bytes** pode ser negativo dependendo do tipo de movimentação que formos realizar.

Por exemplo, se quisermos se mover no arquivo a partir do ponto atual (**origem = 1**) ou do seu final (**origem = 2**), um valor negativo de bytes é possível. Nesse caso, estaríamos voltando dentro do arquivo a partir daquele ponto.

11 Trabalhando com classes e objetos

A programação orientada a objetos (POO, do inglês *Object Oriented Programming*) é um paradigma de programação que se baseia nos conceitos de **objetos**. Esses podem ser entendidos como entidades que possuem um estado e comportamento definidos.

O objetivo da POO é modelar a realidade de forma mais próxima possível, utilizando conceitos de herança, polimorfismo e encapsulamento para representar de maneira mais fiel as relações e interações entre os objetos em um sistema.



Enquanto a programação procedimental, também chamada de programação procedural, decompõe um problema em subproblemas por meio de funções, a programação orientada a objetos decompõe um problema em objetos que interagem entre si, sendo cada objeto uma unidade de software.

A programação estruturada é um paradigma de programação que se baseia em dividir o código em funções ou procedimentos, cada um deles responsável por realizar uma tarefa específica. Seu objetivo é aumentar a legibilidade e manutenção do código, tornando-o mais organizado e fácil de entender.

A programação orientada a objetos é um paradigma que se baseia no conceito de **objetos**, que são entidades que possuem estado e comportamento. Diferente da programação procedimental, onde a ênfase do desenvolvimento está nas operações desenvolvidas (funções e módulos), aqui a ênfase está na criação de unidades de software (objetos), cada qual contendo seus dados e operações possíveis, e na interação entre elas.

A POO é uma forma de organizar o código em um programa de maneira a facilitar a manutenção e a reutilização, tornando-o mais legível e mais fácil de manter a longo prazo.



A POO é amplamente utilizada em diversas linguagens de programação, como Java, C#, Python, C++, entre outras.

Os principais conceitos da POO são:

- **Classe:** é a representação de um tipo de objeto, definindo seus atributos (características) e métodos (comportamentos e ações que ele pode realizar).
- **Objeto:** é uma instância de uma classe, ou seja, é um objeto concreto criado a partir da classe. Cada objeto possui seu próprio estado (valores dos atributos) e comportamento (métodos).

- **Herança:** é a relação entre classes onde uma classe (filha) herda as características de outra classe (mãe). Isso permite que uma classe possa reutilizar o código de outra classe, evitando a duplicação de código.
- **Polimorfismo:** é a capacidade de uma classe **filha** sobrescrever ou estender o comportamento de uma classe **mãe**. Isso permite que as classes filhas possam ter comportamentos diferentes, mesmo possuindo a mesma base de código.
- **Encapsulamento:** é uma técnica usada para proteger os detalhes de implementação de uma classe. Dessa forma, é exposto apenas os métodos e atributos necessários para o funcionamento do sistema. Isso permite que o código seja mais modular e fácil de manter.

Algumas das principais vantagens da POO são:

- **Abstração:** a POO permite que os objetos sejam representados de forma mais abstrata, ocultando os detalhes de implementação e expondo apenas os métodos e atributos necessários para o funcionamento do sistema. Isso aumenta a modularidade e facilita a manutenção do código.
- **Reutilização de código:** é possível reutilizar um código através da herança, onde uma classe herda os atributos e métodos de outra classe. Isso evita a duplicação de código e aumenta a produtividade.



Uma desvantagem da POO é a sua **complexidade**, pois envolve o uso de conceitos mais avançados (como herança e polimorfismo) que a programação estruturada. Isso pode dificultar o aprendizado inicial, mas a longo prazo pode tornar o código mais legível e fácil de manter.

11.1 Classe e objeto



Clique aqui para ver a vídeo aula sobre classes e objetos.

Classes e objetos são conceitos fundamentais da programação orientada a objetos (POO) em Python. Uma **classe** é uma estrutura de código que atua como um modelo ou um plano para criar objetos. Ela define a estrutura e o comportamento de um objeto, especificando os atributos (dados) e métodos (funções) que o objeto terá.



A classe é um molde para a criação de objetos. É como a planta arquitetônica que serve de modelo para a construção de uma casa.

As classes são a base para criar objetos que compartilham características comuns. Para definir uma classe, usamos a palavra-chave **class**, seguida pelo nome da classe e, opcionalmente, uma classe base entre parênteses. A definição da classe é

um bloco de código indentado que contém os atributos e métodos da classe, como mostrado a seguir:

```
class nome_da_classe:

    # Métodos e atributos da classe
```

Imagine que você quer criar uma classe para representar pessoas em um sistema. Essa classe poderia ter atributos como nome, idade, sexo, altura e peso, e métodos como andar, correr, falar e comer.

Um exemplo simples de definição de classe em Python é mostrado a seguir.

Exemplo: definindo uma classe	
01	class Pessoa:
02	def __init__(self, nome, idade):
03	self.nome = nome
04	self.idade = idade
05	
06	def saudacao(self):
07	return f"Meu nome é {self.nome} e eu tenho {self.idade} anos."

Nesse exemplo, criamos uma classe chamada **Pessoa**. Ela possui dois atributos (**nome** e **idade**) e um método (**saudacao()**). O método **__init__()** é um construtor especial que é executado automaticamente quando um objeto da classe é criado. Ele é usado para inicializar os atributos do objeto com os valores passados como argumentos.



Um objeto é uma instância da classe. É como a casa construída com base em uma planta arquitetônica.

Um objeto é uma instância específica de uma classe, criada a partir do modelo definido pela classe. Em outras palavras, um objeto é uma representação concreta de uma classe, com seus próprios valores de atributos e a capacidade de executar os métodos definidos na classe.



Um objeto possui identidade própria e é distinguível de qualquer outro objeto mesmo que seus atributos sejam idênticos.

Considere nosso exemplo anterior: uma classe para representar pessoas em um sistema. Mesmo que duas pessoas tenham os mesmos atributos (como nome, idade, sexo, altura e peso), cada pessoa é única.

Para criar um objeto em Python, basta chamar o nome da classe seguido de parênteses vazios, como mostrado a seguir:

```
nome_objeto = nome_da_classe()
```

Isso cria uma instância da classe, que é um objeto único com suas próprias características e comportamentos, como mostra o exemplo a seguir.

Exemplo: definindo uma classe e objetos	
<pre>01 class Pessoa: 02 def __init__(self, nome, idade): 03 self.nome = nome 04 self.idade = idade 05 06 def saudacao(self): 07 return f"Meu nome é {self.nome} e eu tenho {self.idade} anos." 08 09 # Criando objetos da classe Pessoa 10 pessoa1 = Pessoa("João", 30) 11 pessoa2 = Pessoa("Maria", 25) 12 13 # Acessando os atributos e métodos dos objetos 14 print(pessoa1.nome) 15 print(pessoa2.idade) 16 print(pessoa1.saudacao()) 17 print(pessoa2.saudacao())</pre>	
Saída Meu nome é João e eu tenho 30 anos. Meu nome é Maria e eu tenho 25 anos.	

Nesse exemplo, criamos dois objetos (**pessoa1** e **pessoa2**) a partir da classe **Pessoa**. Cada objeto possui seus próprios valores de atributos (**nome** e **idade**) e pode chamar o método **saudacao()** para exibir uma saudação personalizada.

11.2 Atributos e Métodos

Em Python, os atributos são variáveis associadas a uma classe ou a um objeto. Eles representam as características dos objetos criados a partir dessa classe.



Os atributos são os valores internos do objeto. Eles definem as propriedades que queremos para o nosso objeto.



Clique aqui para ver a vídeo aula sobre atributos e métodos.

Existem dois tipos de atributos em Python:

- **Atributos de Classe:** São atributos que são compartilhados por todas as instâncias (objetos) de uma classe. Eles são definidos diretamente dentro da classe, fora de qualquer método, e são acessados usando o nome da classe.
- **Atributos de Instância:** São atributos exclusivos de cada objeto criado a partir de uma classe. Eles são definidos dentro do construtor da classe (método `__init__`) usando a notação **self.nome_atributo**. Cada objeto possui sua própria cópia dos atributos de instância.

Um exemplo simples de definição de atributos é mostrado a seguir.

Exemplo: atributos de classe e de instância

```
01 class Pessoa:
02     # Atributo de Classe
03     especie = "Humana"
04
05     def __init__(self, nome, idade):
06         # Atributos de Instância
07         self.nome = nome
08         self.idade = idade
09
10 # Criando um objeto da classe Pessoa
11 pessoal = Pessoa("João", 30)
12
13 # Acessando os atributos de instância
14 print(pessoal.nome)
15 print(pessoal.idade)
16
17 # Acessando o atributo de classe
18 print(pessoal.especie)
```

Nesse exemplo, a classe **Pessoa** possui um atributo de classe chamado **especie**, que é compartilhado por todas as instâncias da classe. Além disso, a classe tem dois atributos de instância, **nome** e **idade**, que são únicos para cada objeto criado a partir da classe.



O acesso aos atributos e métodos de um objeto é feito usando o operador **ponto** (`.`).

Métodos são funções definidas dentro de uma classe que definem o comportamento dos objetos criados a partir dessa classe. Eles são usados para realizar operações ou manipulações nos atributos do objeto e podem ou não retornar um valor.



Os métodos são o conjunto de funcionalidades da classe. Eles definem as habilidades que queremos para nosso objeto.

Existem três tipos de métodos em Python:

- **Métodos de Instância:** São métodos que operam em instâncias específicas da classe. Eles têm **self** como primeiro parâmetro, que representa o objeto chamador, permitindo acesso aos atributos dessa instância. Os métodos de instância são os mais comuns e são definidos dentro da classe.
- **Métodos de Classe:** São métodos que operam na classe como um todo, em vez de instâncias específicas. Eles têm **cls** como primeiro parâmetro, que representa a classe, permitindo acesso aos atributos de classe. Eles são definidos usando o decorador **@classmethod** antes da definição do método.
- **Métodos Estáticos:** São métodos que não dependem de instâncias ou classes e, portanto, não têm parâmetros especiais como **self** ou **cls**. Eles são definidos usando o decorador **@staticmethod** antes da definição do método.



Um **método de classe** pode acessar ou modificar o estado da classe (isto é, acessar e modificar os **atributos de classe**) enquanto um **método estático** não sabe nada sobre o estado da classe.

Um exemplo simples de definição de atributos é mostrado a seguir.

Exemplo: diferentes métodos em Python

```
01 class Matematica:
02     # Método Estático
03     @staticmethod
04     def somar(x, y):
05         return x + y
06     # Método de Classe
07     @classmethod
08     def multiplicar(cls, x, y):
09         return x * y
10     # Método de Instância
11     def dividir(self, x, y):
12         return x / y
13
14 # Chamando um métodos estático
15 print(Matematica.somar(3, 5))
16 # Chamando um métodos da classe
17 print(Matematica.multiplicar(2, 4))
18 # Criando um objeto da classe Matematica
19 calc = Matematica()
20 # Chamando o método de instância
21 print(calc.dividir(10, 2))
```

11.3 Encapsulamento



Clique aqui para ver a vídeo aula sobre encapsulamento de atributos e métodos.

O encapsulamento é um conceito fundamental da programação orientada a objetos. Ele se refere à ideia de esconder os detalhes de implementação dos usuários de um objeto ou classe.



É uma forma de proteger os dados e as operações de um objeto de acesso e manipulação diretos pelo código que os utiliza.

Em várias situações, o acesso direto aos atributos de um objeto não é aconselhável. O encapsulamento ajuda a garantir a integridade dos dados e promove uma melhor organização do código, uma vez que apenas as interfaces públicas da classe são acessíveis externamente.



Em Python, o encapsulamento é alcançado principalmente através do uso de convenções de nomenclatura.

Essas convenções ajudam a indicar a intenção do desenvolvedor em relação à visibilidade dos atributos e métodos da classe. Existem duas principais convenções para o encapsulamento em Python:

- **Atributos e métodos públicos:** São atributos e métodos que podem ser acessados livremente a partir de outras partes do programa.
- **Atributos e métodos privados:** São atributos e métodos que são destinados a serem acessados somente dentro da própria classe e não devem ser acessados por outras partes do programa. Em Python, o uso de **dois sublinhados** (__) no início do nome do atributo ou método indica que eles são privados.

Um exemplo simples de atributos público e privado é mostrado a seguir.

Exemplo: atributos públicos e privados

```
01 class Pessoa:
02     def __init__(self, nome, idade):
03         # Atributo público
04         self.nome = nome
05         # Atributo privado
06         self.__idade = idade
07
08 pessoa1 = Pessoa("João", 30)
09 print(pessoa1.nome)
10 print(pessoa1.__idade)
```

```
Saída João
Traceback (most recent call last):
AttributeError: 'Pessoa' object has no attribute
'__idade'
```

Nesse exemplo, o atributo `__idade` é considerado privado por convenção (usando dois sublinhados no início). Embora seja possível acessá-los diretamente fora da classe, a convenção sugere que eles não devem ser acessados externamente para evitar possíveis efeitos indesejados no programa.

11.4 Construtor e destrutor



Clique aqui para ver a vídeo aula sobre construtor e destrutor.

O construtor e o destrutor são métodos especiais que permitem a criação e a destruição de objetos de uma classe de maneira controlada. Eles são parte da classe, mesmo que não sejam definidos.



Eles são úteis para inicializar os atributos de um objeto quando ele é criado e executar tarefas de limpeza ou liberação de recursos quando o objeto é removido da memória.

O **construtor** é um método especial que é executado automaticamente quando um objeto de uma classe é criado.



O **construtor** é responsável pela alocação de recursos necessários ao funcionamento do objeto e da definição inicial dos estados dos atributos.

Ele é usado para inicializar os atributos do objeto e realizar qualquer outra configuração necessária antes que o objeto seja utilizado. No construtor, você pode definir os valores iniciais dos atributos da classe com base nos argumentos fornecidos ao criar o objeto.



O **construtor** funciona como um inicializador de atributos, de modo a garantir que o atributo sempre existe.

Dentro do construtor podemos definir e inicializar (com um valor pré-definido ou passado por parâmetro) todos os atributos do objeto. Em Python, o construtor é definido por um método especial chamado `__init__`, como mostra o exemplo a seguir.

Exemplo: definindo um construtor

```
01 class Pessoa:
02     def __init__(self, nome, idade):
03         self.nome = nome
04         self.idade = idade
05         print(f'Pessoa criada: {nome}')
06
07 # Criando um objeto da classe Pessoa e
08 # passando argumentos para o construtor
09 pessoa1 = Pessoa("João", 30)
10 pessoa2 = Pessoa("Maria", 25)
```

```
Saída Pessoa criada: João
        Pessoa criada: Maria
```

Nesse exemplo, o método `__init__` é executado automaticamente quando um objeto de uma classe é criado (linhas 9 e 10), gerando a impressão da mensagem que a pessoa foi criada corretamente. Não há necessidade de chamar esse método manualmente, ele é sempre chamado, automaticamente, quando um novo objeto da classe é criado.

O **destrutor** é um método especial que é executado automaticamente quando um objeto de uma classe é removido da memória, ou seja, quando ele não é mais referenciado e está pronto para ser liberado.



O **destrutor** é responsável pela liberação de recursos do objeto.

O destrutor é menos comumente usado do que o construtor, pois a linguagem Python possui mecanismos de gerenciamento automático de memória (*garbage collection*) que geralmente cuidam da liberação de recursos de forma transparente.



O **destrutor** permite executar alguma tarefa antes que o objeto seja eliminado.

Podemos, por exemplo, definir um destrutor para salvar dados em um arquivo. Dessa forma, a função de salvamento dos dados será chamada pelo destrutor sempre que o objeto é eliminado da memória.

Em Python, o destrutor é definido por um método especial chamado `__del__`, como mostra o exemplo a seguir.

Exemplo: definindo um destrutor

```
01 class Pessoa:
02     def __init__(self, nome, idade):
03         self.nome = nome
04         self.idade = idade
05
06     def __del__(self):
07         print(f"Objeto {self.nome} foi removido da
memória.")
08
09 def func():
10     pessoa1 = Pessoa("João", 30)
11     pessoa2 = Pessoa("Maria", 25)
12
13 func()
```

```
Saída Objeto João foi removido da memória.
      Objeto Maria foi removido da memória.
```

Nesse exemplo, o método `__del__` é executado automaticamente quando um objeto de uma classe é eliminado da memória (linha 7), gerando a impressão da mensagem que a pessoa foi removida da memória. Neste exemplo, os objetos foram criados dentro da função `func()` (linhas 9-11), e são removidos da memória quando a função termina a sua execução.

11.5 Sobrecarga de operadores



Clique aqui para ver a vídeo aula de sobrecarga de operadores.

A sobrecarga de operadores é uma poderosa funcionalidade que permite que você defina o comportamento de operadores comuns (+, -, *, /, etc.) em classes personalizadas.



Sobrecarga de operador significa a possibilidade de definir o comportamento de alguns operadores básicos da linguagem para novos tipos de dados.

Com essa capacidade, você pode estender as operações padrão para objetos criados a partir de suas classes, tornando-os mais intuitivos e facilitando o uso em operações aritméticas, de comparação e muito mais.

Com a sobrecarga, podemos escrever a igualdade entre dois objetos assim

```
p1 == p2
```

ao invés de

```
p1.igual(p2)
```

Para implementar a sobrecarga de operadores, você deve definir os métodos especiais que representam cada operador. Esses métodos especiais começam e terminam com dois underscores (*dunder methods*) e têm nomes específicos de acordo com o operador que você deseja sobrecarregar. A tabela a seguir apresenta alguns exemplos.

Operador	Método	Exemplo
+	<code>__add__</code>	<code>A + B</code>
-	<code>__sub__</code>	<code>A - B</code>
*	<code>__mul__</code>	<code>A * B</code>
/	<code>__truediv__</code>	<code>A / B</code>
<code>==</code>	<code>__eq__</code>	<code>A == B</code>
<code>!=</code>	<code>__ne__</code>	<code>A != B</code>
<code>></code>	<code>__gt__</code>	<code>A > B</code>
<code><</code>	<code>__lt__</code>	<code>A < B</code>

11.5.1 Somando dois objetos

Vamos fazer a sobrecarga do operador de adição, `+`. Para isso, vamos criar o método especial `__add__` dentro da nossa classe, como mostra o exemplo a seguir.

Exemplo: usando o método <code>__add__</code>
<pre> 01 class Ponto: 02 def __init__(self, x, y): 03 self.x = x 04 self.y = y 05 06 # Sobrecarga do operador de adição 07 def __add__(self, outro_ponto): 08 novo_x = self.x + outro_ponto.x 09 novo_y = self.y + outro_ponto.y 10 return Ponto(novo_x, novo_y) 11 12 # Criando dois objetos da classe Ponto 13 ponto1 = Ponto(2, 3) 14 ponto2 = Ponto(5, 7) 15 16 ponto3 = ponto1 + ponto2 17 print(ponto3.x, ponto3.y) </pre>
Saída 7 10

Como podemos ver, fazer a sobrecarga do operador de adição é simples. É importante também notar que isso é apenas uma conveniência.

Com a sobrecarga, podemos escrever a adição de dois objetos assim

`p1 + p2`

ao invés de

`p1.soma(p2)`

Isso torna o código mais legível e mais próximo do que seria naturalmente esperado pelos programadores, facilitando o desenvolvimento e o entendimento dos programas em Python.

11.5.2 Comparando o conteúdo de dois objetos



Clique aqui para ver a vídeo aula de comparação de objetos.

Comparar dois objetos não é uma tarefa tão simples quanto possa parecer.



Por definição, o operador de igualdade, `==`, testa se os dois argumentos são o mesmo objeto.

Isso significa que o operador de igualdade verifica se dois objetos referenciam, na verdade, o mesmo objeto na memória. Nenhuma comparação entre os atributos dos objetos é realizada, como mostra o exemplo a seguir.

Exemplo: comparando objetos

```
01 class Ponto:
02     def __init__(self, x, y):
03         self.x = x
04         self.y = y
05
06 ponto1 = Ponto(2, 3)
07 ponto2 = Ponto(2, 3)
08
09 if(ponto1 == ponto2):
10     print('Objetos iguais')
11 else:
12     print('Objetos diferentes')
```

Saída: `Objetos diferentes`

Nesse exemplo, temos dois objetos diferentes, mas com o mesmo conteúdo. O operador de igualdade compara os objetos, não o conteúdo deles. Para fazer a comparação do conteúdo dos objetos, o mais indicado é definir um método na classe para testar se dois objetos possuem os mesmos valores de atributos.

Vamos fazer a sobrecarga do operador de igualdade, `==`. Para isso, vamos criar o método especial `__eq__` dentro da nossa classe, como mostra o exemplo a seguir. Note que temos que especificar exatamente como a comparação deve ser feita.

Exemplo: comparando o conteúdo dos objetos

```
01 class Ponto:
02     def __init__(self, x, y):
03         self.x = x
04         self.y = y
05     def __eq__(self, po):
06         return self.x == po.x and self.y == po.y
07
08 ponto1 = Ponto(2, 3)
09 ponto2 = Ponto(2, 3)
10
11 if(ponto1 == ponto2):
12     print('Objetos iguais')
13 else:
14     print('Objetos diferentes')
```

Saída: `Objetos iguais`

11.6 Imprimindo um objeto



Clique aqui para ver a vídeo aula de imprimir um objeto.

O método `__str__` é um método especial, também conhecido como método de *stringificação*. Ele permite que você defina uma representação em formato de string para objetos de uma classe customizada.



O método `__str__` permite definir o que será impresso sempre que o objeto de uma classe for impresso. Funciona como se fosse uma conversão customizada do objeto para texto.

O objetivo principal do método `__str__` é fornecer uma descrição amigável e informativa do objeto, tornando-o mais compreensível para os programadores e permitindo que os objetos sejam exibidos de forma mais legível em saídas de impressão.

Quando chamado, o método `__str__` retorna uma representação legível em formato de texto do objeto, o que facilita sua leitura e depuração. A seguir podemos ver a estrutura do método `__str__`:

```
def __str__(self):  
    return "string retornada pelo objeto"
```



Dentro do corpo do método `__str__`, você pode definir como deseja que o objeto seja representado em formato de string.

Isso geralmente envolve retornar informações relevantes e descritivas sobre os atributos do objeto, como mostra o exemplo a seguir. Neste exemplo, a classe `Pessoa` possui o método `__str__`, que retorna uma representação personalizada em formato de string contendo o nome e a idade da pessoa.

Exemplo: usando o método `__str__`

```
01 class Pessoa:  
02     def __init__(self, nome, idade):  
03         self.nome = nome  
04         self.idade = idade  
05  
06     def __str__(self):  
07         return f"Nome: {self.nome}, Idade:  
08 {self.idade}"  
09  
10 pessoa = Pessoa("João", 30)  
11 print(pessoa)
```

Saída Nome: João, Idade: 30

11.7 Copiando um objeto



Clique aqui para ver a vídeo aula de cópia de objetos.

Em Python, existem várias maneiras de copiar um objeto.



Assim como no caso das listas, a operação de atribuição não cria uma cópia do objeto de uma classe que criamos.

Nos vimos isso na Seção 6.5.2 que as listas são um tipo de objeto. Se atribuirmos uma lista a outra, ambas irão se referir ao mesmo objeto. O mesmo princípio vale para objetos de classes que nós criamos. Precisamos garantir que temos dois objetos diferentes, mas com o mesmo conteúdo.



Para fazer a cópia de um objeto é essencial entender a diferença entre uma **cópia rasa** (*shallow copy*) e uma **cópia profunda** (*deep copy*).

A diferença entre os dois tipos de cópia depende do conteúdo do objeto:

- **Cópia Rasa** (*Shallow Copy*): cria um novo objeto da classe, mas não faz cópias internas dos objetos contidos dentro do objeto original. Em vez disso, ele cria uma nova referência para os objetos internos do objeto original. Isso significa que as referências internas são compartilhadas entre o objeto original e sua cópia rasa. Portanto, qualquer alteração nos objetos internos afetará tanto o objeto original quanto sua cópia. Para fazer uma cópia rasa, você pode usar a função **copy()**.
- **Cópia Profunda** (*Deep Copy*): cria um novo objeto da classe e faz cópias internas de todos os objetos contidos dentro do objeto original. Nesse caso, todos os objetos internos são copiados de forma independente, criando novos objetos com os mesmos valores dos objetos originais. Isso garante que as alterações em objetos internos de uma cópia profunda não afetem o objeto original e vice-versa. Para fazer uma cópia profunda, você precisa usar o módulo **copy** e a função **deepcopy()**.



Basicamente, a **cópia rasa** faz somente uma cópia superficial do objeto enquanto a cópia profunda copia não somente o objeto, mas também todo e qualquer objeto embutido neste objeto.

A cópia rasa funciona bem quando temos um objeto que contém elementos mais simples. Esse método não é capaz de copiar objetos embutidos dentro de outros objetos, como mostra o exemplo a seguir.

Exemplo: cópia rasa de objetos

```
01 import copy
02
03 class Pessoa:
04     def __init__(self, nome, idade, valores):
05         self.nome = nome
06         self.idade = idade
07         self.lista = valores
08
09     def __str__(self):
10         return f"Nome: {self.nome}, Idade:
    {self.idade}, Lista: {self.lista}"
```

```

11
12 pessoa1 = Pessoa("João", 30, [10,20])
13 pessoa2 = copy.copy(pessoa1)
14 pessoa1.idade = 20
15 pessoa1.lista[0] = 100
16
17 print(pessoa1)
18 print(pessoa2)
19 print(id(pessoa1))
20 print(id(pessoa2))

```

```

Saída Nome: João, Idade: 20, Lista: [100, 20]
      Nome: João, Idade: 30, Lista: [100, 20]
      2248638592016
      2248638592272

```

Como podemos ver, a **cópia rasa** criou uma cópia do objeto de modo que os atributos **nome** e **idade** são independentes. Porém, a **cópia rasa** não criou uma cópia da lista interna, de modo que alterar o valor nela significa alterar o valor na outra lista também. Para resolver esse problema, precisamos fazer uma **cópia profunda** do objeto, como mostra o exemplo a seguir.

Exemplo: cópia profunda de objetos

```

01 import copy
02
03 class Pessoa:
04     def __init__(self, nome, idade, valores):
05         self.nome = nome
06         self.idade = idade
07         self.lista = valores
08
09     def __str__(self):
10         return f"Nome: {self.nome}, Idade:
11         {self.idade}, Lista: {self.lista}"
12
13 pessoa1 = Pessoa("João", 30, [10,20])
14 pessoa2 = copy.deepcopy(pessoa1)
15 pessoa1.idade = 20
16 pessoa1.lista[0] = 100
17
18 print(pessoa1)
19 print(pessoa2)
20 print(id(pessoa1))
21 print(id(pessoa2))

```

```

Saída Nome: João, Idade: 20, Lista: [100, 20]
      Nome: João, Idade: 30, Lista: [10, 20]
      1774312223248
      1774312224080

```

Observe a diferença entre os resultados da **cópia rasa** e da **cópia profunda**. Na cópia rasa, as alterações nos objetos internos são refletidas tanto no objeto original quanto na cópia, pois os atributos ainda estão se referindo aos mesmos objetos. Já na cópia profunda, as alterações nos objetos internos não afetam o objeto original, pois os atributos são independentes.

11.8 Herança



Clique aqui para ver a vídeo aula de herança.

A herança em Python é um conceito fundamental da programação orientada a objetos (POO) que permite criar novas classes (classes derivadas ou subclasses) com base em classes já existentes (classes base ou superclasses).



A herança permite que a nova classe herde atributos e métodos da classe base, permitindo reutilização de código e estabelecendo uma relação de hierarquia entre as classes.

Basicamente, a herança permite criar uma nova classe a partir de outra já existente. Isso permite construir objetos especializados que herdam os atributos/métodos de objetos mais genéricos.



A herança é uma forma de reutilizar código.

Para criar uma classe derivada, basta declarar a nova classe com a classe base entre parênteses na definição da classe, como mostrado a seguir:

```
class ClasseBase:

    # Métodos e atributos da classe base


class ClasseDerivada(ClasseBase):

    # Métodos e atributos da classe derivada
```

A nova classe, **ClasseDerivada**, pode acessar e utilizar todos os atributos e métodos da classe base, **ClasseBase**, além de adicionar seus próprios atributos e métodos exclusivos, como mostra o exemplo a seguir.

Exemplo: usando a herança

```
01 # classe base
02 class Animal:
03     def __init__(self, nome):
04         self.nome = nome
05
06     def fazer_som(self):
07         pass
08 # classes derivadas
09 class Cachorro(Animal):
10     def fazer_som(self):
11         return "Au Au!"
12
13 class Gato(Animal):
14     def fazer_som(self):
15         return "Miau!"
16
17 # Criando objetos das classes derivadas
18 doguinho = Cachorro("Rex")
19 gatinho = Gato("Mimi")
20
21 print(doguinho.nome)
22 print(doguinho.fazer_som())
23 print(gatinho.nome)
24 print(gatinho.fazer_som())
```

```
Saída Rex
      Au Au!
      Mimi
      Miau!
```

Nesse exemplo, **Cachorro** e **Gato** são classes derivadas da classe base **Animal**. Ambas as classes derivadas herdam o atributo **nome** e o método **fazer_som()** da classe base. Note que esse método não foi implementado pela classe base (por isso foi usada a palavra **pass**). Podemos também adicionar métodos específicos para cada animal, se for necessário.



A herança permite criar uma hierarquia de classes, o que é útil quando há características em comum entre diferentes classes, mas também características exclusivas para cada uma delas.

Com a herança, você pode reutilizar e organizar o código de forma eficiente, evitando duplicação e facilitando a manutenção do código em projetos complexos. No entanto, é importante lembrar que o uso adequado da herança requer uma modelagem cuidadosa das classes para garantir que a hierarquia seja coerente e que a relação de "é um" entre as classes seja realmente aplicável.



Em alguns casos, a composição (usando objetos de outras classes como atributos) pode ser uma abordagem mais apropriada para evitar problemas de herança excessiva.

11.8.1 Sobreposição de métodos - *override*



Clique aqui para ver a vídeo aula de sobreposição.

A sobreposição de métodos, também conhecida como sobrescrita de métodos, ocorre quando uma classe derivada define um método com o mesmo nome de um método da classe base. Quando uma instância da classe derivada chama esse método, a implementação da classe derivada é executada em vez da implementação da classe base.



A sobreposição de métodos permite que a classe derivada substitua ou altere o comportamento de um método herdado da classe base, adicionando sua própria lógica específica.

Essa é uma das principais formas de personalização de comportamento em herança e é amplamente utilizada para adaptar classes às necessidades específicas de um programa, como mostra o exemplo a seguir.

Exemplo: sobreposição de métodos

```
01 # classe base
02 class Animal:
03     def fazer_som(self):
04         return "Som genérico de animal."
05
06 # classes derivadas
07 class Cachorro(Animal):
08     def fazer_som(self):
09         return "Au Au!"
10
11 # Criando objetos das classes derivadas
12 animal_generico = Animal()
13 doguinho = Cachorro()
14
15 print(animal_generico.fazer_som())
16 print(doguinho.fazer_som())
```

Saída Som genérico de animal.
Au Au!

Nesse exemplo, a classe **Animal** tem um método **fazer_som()** que fornece uma implementação genérica para o som de um animal. A classe derivada **Cachorro** sobrepõe o método **fazer_som()**, fornecendo sua própria implementação específica para o som de um cachorro.



Ao usar a herança, a classe derivada pode ter seu próprio construtor, além de herdar o construtor da classe base.

Nesse tipo de sobreposição, o construtor da classe derivada é responsável por inicializar os atributos específicos da classe derivada, enquanto o construtor da classe base é responsável por inicializar os atributos herdados da classe base. Se a classe derivada não possui um construtor, então ela irá herdar o definido na classe base automaticamente.

Para chamar o construtor da classe base a partir do construtor da classe derivada, você pode utilizar a função **super()**. Essa função retorna um objeto proxy que representa a classe base e permite acessar os seus métodos e atributos, como mostra o exemplo a seguir.

Exemplo: usando construtor com a herança

```
01 # classe base
02 class Animal:
03     def __init__(self, nome):
04         self.nome = nome
05
06     def fazer_som(self):
07         return "Som genérico de animal."
08
09 # classe derivada
10 class Cachorro(Animal):
11     def __init__(self, nome, raca):
12         # Chama o construtor da classe base
13         super().__init__(nome)
14         self.raca = raca
15
16     def fazer_som(self):
17         return "Au Au!"
18
19 # Criando um objeto da classe derivada
20 doguinho = Cachorro("Rex", "Labrador")
21
22 print(doguinho.nome)
23 print(doguinho.raca)
24 print(doguinho.fazer_som())
```

Saída Rex
Labrador
Au Au!

Nesse exemplo, a classe **Cachorro** possui seu próprio construtor, que chama o construtor da classe base usando a função **super()** para inicializar o atributo **nome**. Além disso, as classes derivadas têm seus próprios atributos específicos (**raca**), o qual é inicializado pelo construtor da classe derivada.



A função **super()** permite que você acesse qualquer método da classe base mesmo quando o método é sobreposto na classe derivada.

Você pode usar a função **super()** para chamar a implementação de qualquer método da classe base. Isso é útil quando você deseja estender a funcionalidade original do método em vez de substituí-la completamente.

11.8.2 Herança Múltipla



[Clique aqui para ver a vídeo aula de herança múltipla.](#)

A herança múltipla ocorre quando a classe derivada herda atributos/métodos de mais de uma classe existente. Para implementar a herança múltipla, basta declarar a classe derivada com várias classes base separadas por vírgulas na definição da classe, como mostrado a seguir:

```
class ClasseBase1:

    # Métodos e atributos da classe base 1

class ClasseBase2:

    # Métodos e atributos da classe base 2

class ClasseDerivada(ClasseBase1,ClasseBase2):

    # Métodos e atributos da classe derivada
```

A nova classe, **ClasseDerivada**, é uma classe derivada que herda tanto de **ClasseBase1** quanto de **ClasseBase2**. Isso significa que a classe derivada tem acesso a todos os métodos e atributos definidos em ambas as classes base, como mostra o exemplo a seguir.

Exemplo: usando herança múltipla

```
01 # classes bases
02 class Animal:
03     def __init__(self, nome):
04         self.nome = nome
05
06     def fazer_som(self):
07         pass
08
09 class Voador:
10     def voar(self):
11         return "Estou voando!"
12
13 # classe derivada
14 class Pato(Animal, Voador):
15     def fazer_som(self):
16         return "Quack!"
17
18 # Criando um objeto da classe Pato
19 pato = Pato("Donald")
20
21 print(pato.nome)
22 print(pato.fazer_som())
23 print(pato.voar())
```

```
Saída Donald
      Quack!
      Estou voando!
```

Nesse exemplo, a classe **Pato** herda tanto da classe **Animal** quanto da classe **Voador**. Isso significa que a classe **Pato** tem acesso ao método **fazer_som()** da classe **Animal** e ao método **voar()** da classe **Voador**.



A ordem em que as classes base são especificadas afeta a ordem de resolução de métodos em casos de sobrescrita de métodos com o mesmo nome em classes diferentes.

É importante ter cuidado ao usar a herança múltipla, pois ela pode levar a problemas de ambiguidade ou conflitos em casos de sobrescrita de métodos com o mesmo nome em diferentes classes base.

A codificação da herança múltipla é, em geral, muito mais difícil e seu uso costuma gerar uma codificação confusa. Além disso, o uso da função **super()** não é tão trivial como na herança simples, o que gera uma maior dificuldade na manutenção do código.

11.9 Trabalhando com um iterator



Clique aqui para ver a vídeo aula sobre iterator.

Um *iterator* (ou iterador) é um objeto que permite percorrer um conjunto finito de elementos sequenciais, como listas, tuplas, dicionários, e até mesmo objetos personalizados, de forma sequencial e ordenada.



Um *iterator* permite que seja executada uma iteração sobre ele, ou seja, podemos percorrer todos os seus valores de forma simples e fácil.

Um exemplo de estrutura que implementa um *iterator* é a lista. Isso nos permite acessar cada um de seus elementos de forma fácil usando um comando **for**, como mostra o exemplo a seguir.

Exemplo: iterando uma lista

```
01 lista = [10, 20, 30, 40, 50]
02
03 for v in lista:
04     print('valor = ',v)
```

Como podemos, não precisamos especificar quem é o próximo elemento a ser acessado dentro da lista pelo comando **for**. O *iterator* implementado pela lista se encarrega disso.



A linguagem Python nos permite criar um *iterator* personalizado para uma classe que estamos desenvolvendo.

Podemos criar um *iterator* para uma nova classe. Para isso precisamos implementar dois métodos essenciais que permitem a iteração sobre os elementos de forma eficiente. São eles:

- Método **__iter__()**: Esse método é chamado quando o *iterator* é inicializado e deve retornar o próprio objeto *iterator*. É usado para tornar o objeto iterável, ou seja, permitir que ele seja percorrido usando um **loop for** ou através da função **next()**.
- Método **__next__()**: Esse método é chamado a cada iteração para obter o próximo elemento do *iterator*. Ele retorna o próximo elemento da sequência e

avança o cursor interno do *iterator*. Quando todos os elementos são percorridos, o método deve lançar uma exceção **StopIteration** para indicar que a iteração foi concluída.

A seguir podemos ver como implementar uma classe contendo um iterator.

Exemplo: criando um iterator	
01 class MeuIterator: 02 def __init__(self, maximo): 03 self.maximo = maximo 04 self.atual = 0 05 06 def __iter__(self): 07 return self 08 09 def __next__(self): 10 if self.atual < self.maximo: 11 resultado = self.atual 12 self.atual += 1 13 return resultado 14 else: 15 raise StopIteration 16 17 # Usando o iterator personalizado 18 meu_iterador = MeuIterator(5) 19 for numero in meu_iterador: 20 print(numero)	
	Saída 0
	1
	2
	3
	4

Nesse exemplo, criamos um *iterator* **MeuIterator**, que permite iterar de 0 até um número máximo especificado. Cada vez que o método **__next__()** é chamado, o *iterator* retorna o próximo número na sequência. Quando todos os números são percorridos, a exceção **StopIteration** é lançada, indicando que a iteração foi concluída.



Em Python, muitas estruturas de dados internas, como listas e dicionários, são iteráveis, o que significa que você pode percorrê-las diretamente usando um **loop for**.

12 Tratamento de exceções

Em Python, existem dois tipos principais de erros:

- **Erros de Sintaxe (*Syntax Errors*):** São erros causados por um código que não segue a sintaxe correta da linguagem. Esses erros são detectados pelo interpretador durante a análise do código e geralmente resultam em um encerramento do programa. Exemplos incluem a falta de um dois pontos em uma estrutura **if** ou uma aspa de fechamento ausente em uma string.
- **Exceções (*Exceptions*):** São erros que ocorrem durante a execução do programa devido a circunstâncias imprevistas. Exceções podem ser tratadas para evitar que o programa seja interrompido abruptamente.



A ideia do tratamento de exceção é poder tornar o código mais claro com relação ao que é considerado “normal” e o que é considerado “exceção” (ou anormal).

O tratamento de exceções é uma parte fundamental da escrita de código robusto e confiável. Ele permite que você lide com situações excepcionais ou inesperadas de maneira controlada, garantindo que seu programa não seja encerrado de forma inesperada e possa fornecer *feedback* útil em caso de erros.

12.1 O bloco try-except



Clique aqui para ver a vídeo aula sobre o bloco try-except.

O tratamento de exceções em Python é realizado usando um bloco **try-except**. A forma geral de um bloco **try-except** é:

```
try:
    comando 1
    comando 2
    ...
    comando N
except Exceção:
    comando 1
    comando 2
    ...
    comando N
```

A ideia é envolver o código que pode causar uma exceção dentro de um bloco **try**, e então definir como o programa deve reagir a essa exceção usando um ou mais blocos **except**.



Um bloco **except** pode estar associado a qualquer tipo de erro ou a um erro específico.

Podemos definir a exceção a ser tratada quando definimos um bloco **except**. Se esse parâmetro não for informado, então o bloco **except** irá tratar qualquer tipo de exceção que ocorra no trecho de código definido pelo bloco **try**. Algumas exceções já pré-definidas pela linguagem Python:

- **IOError**: Erros de leitura/escrita de arquivos
- **ValueError**: Parâmetros fora do domínio (exemplo, `sqrt(-1)`)
- **IndexError**: Índice fora de limites
- **TypeError**: Erro de tipos
- **KeyError**: Chave não encontrada no dicionário
- **NameError**: Nome de variável não encontrado
- **RecursionError**: Foi alcançada a profundidade máxima da recursão
- **IndentationError**: indentação incorreta detectada

O código a seguir mostra um exemplo de bloco **try-except**.

Exemplo: tratando uma exceção

```
01 try:
02     # Código que pode causar exceções
03     # Tentando dividir por zero
04     resultado = 10 / 0
05 except ZeroDivisionError:
06     print("Erro: Divisão por zero!")
```

Saída Erro: Divisão por zero!

Nesse exemplo, o bloco **try** tenta realizar a divisão, mas como é uma divisão por zero, isso resulta em uma exceção do tipo **ZeroDivisionError**. O bloco **except** pega essa exceção específica e imprime uma mensagem de erro.



Um bloco **try** pode possuir mais de um bloco **except**.

Dependendo da complexidade do nosso programa, um único trecho de código pode estar sujeito a vários tipos de problemas. Nesse caso, podemos definir vários blocos **except**, onde cada um deles pode tratar um tipo diferente de exceção. O código a seguir mostra um exemplo.

Exemplo: tratando diferentes tipos de exceções

```
01 try:
02     arquivo = open("arquivo.txt", "r")
03     conteudo = arquivo.read()
04     arquivo.close()
05 except FileNotFoundError:
06     print("Erro: O arquivo não foi encontrado.")
07 except IOError:
08     print("Erro: Ocorreu um erro de E/S.")
```

12.2 O bloco try-finally



Clique aqui para ver a vídeo aula sobre o bloco **try-finally**.

O funcionamento do bloco **try-finally** é similar ao **try-except**. A forma geral de um bloco **try-finally** é:

```
try:
    comando 1
    comando 2
    ...
    comando N
finally:
    comando 1
    comando 2
    ...
    comando N
```

No entanto, ele é usado para definir um código que deve ser executado, independentemente do resultado do bloco **try**.



O bloco **finally** é executado independentemente de ter ocorrido uma exceção ou não. Isso é muito útil quando queremos liberar algum recurso utilizado.

Diferente do bloco **except**, o código que estiver no bloco **finally** sempre será executado, independentemente se ocorre ou não uma exceção. Isso nos permite liberar algum recurso utilizado, como fechar arquivo utilizado, como mostra o exemplo a seguir.

Exemplo: criando um bloco try-finally

```
01 try:  
02     arquivo = open("arquivo.txt", "r")  
03     conteudo = arquivo.read()  
04 finally:  
05     arquivo.close()
```

Nesse exemplo, mesmo que ocorra um erro durante a leitura do arquivo, temos garantia de que o mesmo será fechado pelo bloco **finally**.



Também podemos usar os blocos **try**, **except** e **finally** em conjunto.

Isso significa que podemos definir um bloco de comando para ser executado, um bloco para tratar as exceções que possam ocorrer e um terceiro bloco para liberação de recursos, como mostra o exemplo a seguir.

Exemplo: criando um bloco try-except-finally

```
01 try:  
02     arquivo = open("arquivo.txt", "r")  
03     conteudo = arquivo.read()  
04 except:  
05     print("Erro na leitura do arquivo.")  
06 finally:  
07     arquivo.close()  
08
```