

Revisão de Python

Python é uma linguagem de programação de alto nível, interpretada, orientada a objetos e de tipagem dinâmica. É conhecida por sua sintaxe simples e legibilidade.

Histórico

- **Guido Van Rossum**, criou o Python. Ele começou em 1989 no Centrum Wiskunde & Informatica (CWI), inicialmente como um projeto de hobby para se manter ocupado durante o **Natal**.
- O nome da linguagem foi inspirado no programa de TV da BBC "**Monty Python's Flying Circus**", porque Guido Van Rossum era um grande fã do programa.



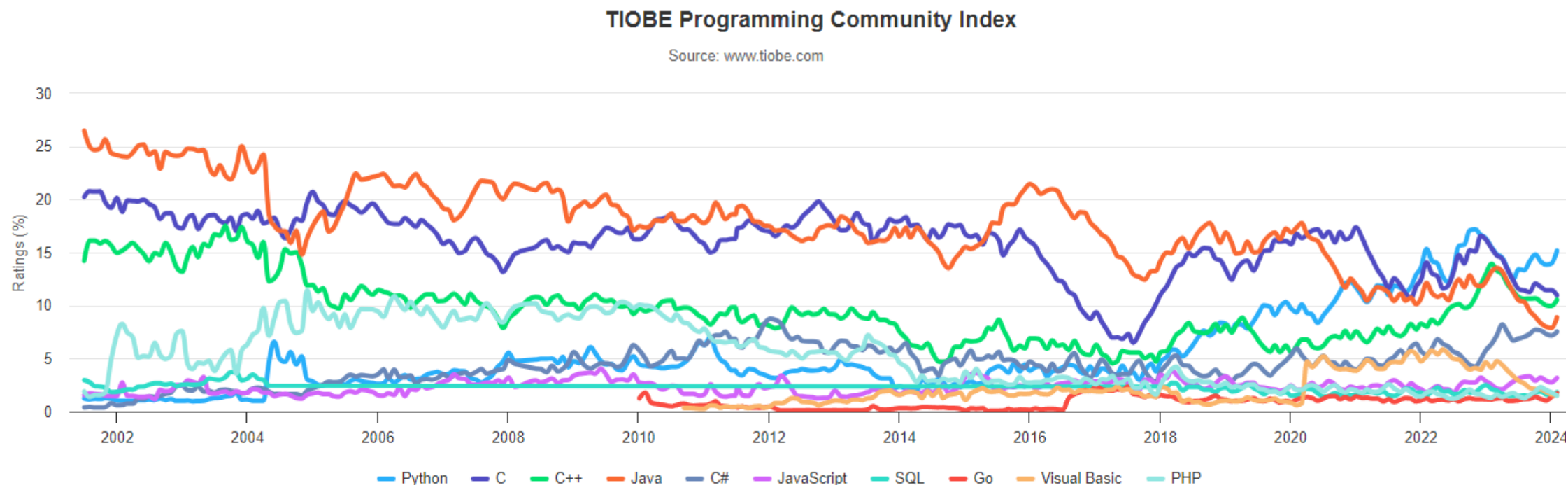
Principais Marcos

- Primeira versão do código Python (versão 0.9.0) foi publicada em **1991**.
- O Python 1.0 foi lançado em **1994** com novas funções para processar facilmente uma lista de dados, como mapear, filtrar e reduzir.
- O Python 2.0 foi lançado em **16 de outubro de 2000**, com novos recursos úteis para programadores, como suporte para caracteres Unicode e um modo mais rápido de percorrer uma lista.
- Em **3 de dezembro de 2008**, foi lançado o Python 3.0.

fonte

Popularidade e Uso

- Python é uma das linguagens de programação mais populares do mundo, conhecida por sua sintaxe simples e legibilidade.



fonte

Características

- **Uma linguagem interpretada:** Executa diretamente o código linha por linha.
- **Uma linguagem fácil de usar:** O Python usa palavras semelhantes às do inglês. Esconde a complexidade de tarefas de baixo nível, como gerenciamento de memória e arquitetura de computadores.
- **Uma linguagem com tipos dinâmicos:** Os programadores não precisam declarar tipos de variáveis ao escrever o código, porque o Python os determina no tempo de execução.
- **Uma linguagem orientada a objetos:** O Python considera tudo como um objeto, mas também aceita outros tipos de programação, como estruturada e funcional.
- **Uma vasta disponibilidade de bibliotecas:** O Python tem uma grande comunidade de desenvolvedores que contribuem com bibliotecas e frameworks para ajudar a resolver problemas comuns.+

Bibliotecas Python

- **NumPy:** Biblioteca para computação numérica, com suporte para arrays e matrizes multidimensionais.
- **Pandas:** Biblioteca para manipulação e análise de dados, com suporte para estruturas de dados como DataFrames e Series.
- **Scikit-learn:** Biblioteca para aprendizado de máquina, com suporte para algoritmos de classificação, regressão e agrupamento.
- **Matplotlib, Seaborn, Plotly:** Biblioteca para criação de visualizações estáticas, como gráficos de linha, barras e dispersão.
- **TensorFlow, Keras:** Biblioteca para aprendizado de máquina e aprendizado profundo, com suporte para construção e treinamento de modelos de redes neurais.
- **muito mais...**

Frameworks Python

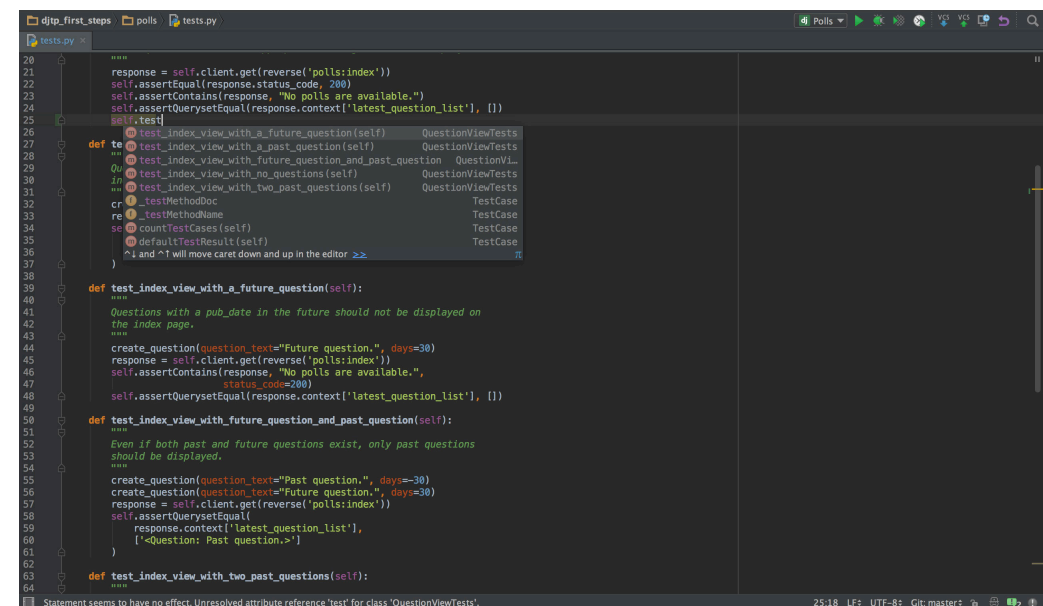
Um framework é uma estrutura de suporte para o desenvolvimento de software. Ele fornece uma base para a criação de aplicativos e oferece uma série de ferramentas e bibliotecas para facilitar o desenvolvimento.

- **Django, Flask, Streamlit:** Framework para desenvolvimento de aplicativos da web, com suporte para criação de sites e APIs.
- **PyTorch:** Framework para aprendizado de máquina e aprendizado profundo, com suporte para construção e treinamento de modelos de redes neurais.
- **Dash:** Framework para criação de aplicativos da web interativos, com suporte para visualizações de dados e painéis de controle.
- **muito mais...**

IDEs Python

Uma IDE (Integrated Development Environment) é um ambiente que fornece ferramentas para escrever, testar e depurar código.

- Visual Studio Code
- PyCharm
- Spyder
- Google Colab
- Jupyter Notebook
- IDLE
- Atom



The screenshot shows a code editor with a file named `tests.py`. The code is a Django test suite for a poll application. It includes several test methods: `test_index_view_with_a_future_question`, `test_index_view_with_a_past_question`, `test_index_view_with_future_question_and_past_question`, `test_index_view_with_no_questions`, and `test_index_view_with_two_past_questions`. The tests use `self.client` to make HTTP requests and `self.assertContains` to verify the response content. The code is written in Python and uses Django's `unittest` framework. The editor has a dark theme and a sidebar on the left showing the file structure.

```
20 response = self.client.get(reverse('polls:index'))
21 self.assertEqual(response.status_code, 200)
22 self.assertContains(response, "No polls are available.")
23 self.assertQuerysetEqual(response.context['latest_question_list'], [])
24
25 self.test()
26
27 def test_index_view_with_a_future_question(self):
28     """
29     Questions with a pub_date in the future should not be displayed on
30     the index page.
31     """
32     create_question(question_text="Future question.", days=30)
33     response = self.client.get(reverse('polls:index'))
34     self.assertContains(response, "No polls are available.")
35     self.assertEqual(response.status_code, 200)
36     self.assertQuerysetEqual(response.context['latest_question_list'], [])
37
38 def test_index_view_with_a_past_question(self):
39     """
40     Questions with a pub_date in the past should be displayed on
41     the index page.
42     """
43     create_question(question_text="Past question.", days=-30)
44     response = self.client.get(reverse('polls:index'))
45     self.assertContains(response, "You have reached this question's poll.")
46     self.assertEqual(response.status_code, 200)
47     self.assertQuerysetEqual(response.context['latest_question_list'], [
48         '<Question: Past question.>'
49     ])
50
51 def test_index_view_with_future_question_and_past_question(self):
52     """
53     Even if both past and future questions exist, only past questions
54     should be displayed.
55     """
56     create_question(question_text="Past question.", days=-30)
57     create_question(question_text="Future question.", days=30)
58     response = self.client.get(reverse('polls:index'))
59     self.assertQuerysetEqual(
60         response.context['latest_question_list'],
61         ['<Question: Past question.>']
62     )
63
64 def test_index_view_with_two_past_questions(self):
65     """
66     Questions with a pub_date in the past should be displayed on
67     the index page.
68     """
69     create_question(question_text="Past question 1.", days=-30)
70     create_question(question_text="Past question 2.", days=-30)
71     response = self.client.get(reverse('polls:index'))
72     self.assertQuerysetEqual(
73         response.context['latest_question_list'],
74         ['<Question: Past question 1.>', '<Question: Past question 2.>']
75     )
76
77 def test_index_view_with_no_questions(self):
78     """
79     If no questions exist, the user should see a message to create
80     a question.
81     """
82     response = self.client.get(reverse('polls:index'))
83     self.assertContains(response, "No polls are available.")
84     self.assertEqual(response.status_code, 200)
85     self.assertQuerysetEqual(response.context['latest_question_list'], [])
86
87 def test_index_view_with_two_past_questions(self):
88     """
89     Questions with a pub_date in the past should be displayed on
90     the index page.
91     """
92     create_question(question_text="Past question 1.", days=-30)
93     create_question(question_text="Past question 2.", days=-30)
94     response = self.client.get(reverse('polls:index'))
95     self.assertQuerysetEqual(
96         response.context['latest_question_list'],
97         ['<Question: Past question 1.>', '<Question: Past question 2.>']
98     )
99
100 def test_index_view_with_no_questions(self):
101     """
102     If no questions exist, the user should see a message to create
103     a question.
104     """
105     response = self.client.get(reverse('polls:index'))
106     self.assertContains(response, "No polls are available.")
107     self.assertEqual(response.status_code, 200)
108     self.assertQuerysetEqual(response.context['latest_question_list'], [])
```


Principais Elementos da Linguagem

- **Saída de Dados:** A função `print()` é usada para exibir dados na tela.
- **Entrada de Dados:** A função `input()` é usada para receber dados do usuário.
- **Variáveis:** São usadas para armazenar dados em memória.
- **Operadores:** São usados para realizar operações em variáveis e valores.
- **Estruturas de Controle:** São usadas para controlar o fluxo de execução do programa.
- **Funções:** São usadas para agrupar um conjunto de instruções em um bloco reutilizável.

Outros Elementos da Linguagem

- **Coleções de Dados:** São usadas para armazenar múltiplos valores em uma única variável.
- **Manipulação de Arquivos:** É usado para ler e escrever dados em arquivos.
- **Orientação a Objetos:** É usado para criar e manipular objetos em Python.
- **Tratamento de Exceções:** É usado para lidar com erros e exceções em Python.
- **Módulos e Pacotes:** São usados para organizar e reutilizar código em Python.
- **Bibliotecas e Frameworks:** São usados para estender as funcionalidades do Python.

Saída de Dados

Saída de dados é a forma como um programa exibe informações para o usuário.

Programa em Python:

```
print("Olá, Mundo!")
```

Resultado:

```
Olá, Mundo!
```

No Colab, o último valor de uma célula é exibido automaticamente.

```
x = 10  
x
```

Resultado:

10

Opções úteis da função `print()`

- **Separador:** O argumento `sep` é usado para definir o separador entre os itens.
- **Final:** O argumento `end` é usado para definir o final da saída.

```
print("Olá", "Mundo", sep=", ", end="!\n")
```

Resultado:

```
Olá, Mundo!
```

Variáveis

Variáveis são usadas para armazenar dados em memória.

Programa em Python:

```
nome = "Albert"  
idade = 30  
altura = 1.75  
print(nome, idade, altura)
```

Resultado:

```
Albert 30 1.75
```

Tipos de Dados

- Inteiros: `int`
- Números de Ponto Flutuante: `float`
- Números Complexos: `complex`
- Booleanos: `bool`
- Cadeias de Caracteres: `str`
- Listas: `list`
- Tuplas: `tuple`
- Conjuntos: `set`
- Dicionários: `dict`

Literais

- Inteiros: `10` , `100` , `1000`
- Números de Ponto Flutuante: `3.14` , `2.718`
 - Notação Científica: `1e3` , `2.5e-4`
- Números Complexos: `3 + 4j` , `5 - 6j`
- Booleanos: `True` , `False`
- Cadeias de Caracteres: `'Olá, Mundo!'` , `"Python"`
- Listas: `[1, 2, 3]` , `['a', 'b', 'c']`
- Tuplas: `(1, 2, 3)` , `('a', 'b', 'c')`
- Conjuntos: `{1, 2, 3}` , `['a', 'b', 'c']`
- Dicionários: `{'a': 1, 'b': 2, 'c': 3}`

Características das Variáveis

- **Nomes de variáveis:** Podem conter letras, números e sublinhados, mas não podem começar com um número.
- **Tipos de variáveis:** O Python é uma linguagem de tipagem dinâmica, o que significa que o tipo de uma variável é determinado no tempo de execução.
- **Atribuição de variáveis:** É feita usando o operador de atribuição `=`.
- **Convenções de nomenclatura:** As variáveis seguem convenções de nomenclatura, como `snake_case` para nomes de variáveis e `CamelCase` para nomes de classes.
- **Palavras-chave reservadas:** Existem palavras-chave reservadas que não podem ser usadas como nomes de variáveis, como `if`, `else`, `for`, `while`, `def`, `class`, etc.
- **Escopo de variáveis:** As variáveis têm escopo local ou global, dependendo de onde são definidas.

Entrada de Dados

Entrada de dados é a forma como um programa recebe informações do usuário.

Programa em Python:

```
nome = input("Digite seu nome: ")  
print("Olá,", nome)
```

Resultado:

```
Digite seu nome: Albert  
Olá, Albert
```

Entrada de Valores Numéricos

A função `input()` retorna uma string, que pode ser convertida em um número usando as funções `int()` e `float()`.

Programa em Python:

```
idade = int(input("Digite sua idade: "))  
altura = float(input("Digite sua altura: "))  
print(idade, altura)
```

Resultado:

```
Digite sua idade: 30  
Digite sua altura: 1.75  
30 1.75
```

Operadores Aritméticos

- Adição: `+`
- Subtração: `-`
- Multiplicação: `*`
- Divisão: `/`
- Divisão Inteira: `//`
- Resto da Divisão: `%`
- Exponenciação: `**`

- Atribuição com Operação: `+=` , `-=` , `*=` , `/=` , `//=` , `%=` , `**=`
- Operadores de Comparação: `==` , `!=` , `>` , `<` , `>=` , `<=`
- Operadores Lógicos: `and` , `or` , `not`
- Operadores de Associação: `in` , `not in`
- Operadores de Identidade: `is` , `is not`
- Operadores Ternários: `if` , `else`
- Operadores Bit a Bit: `&` , `|` , `^` , `~` , `<<` , `>>`
- Precedência de Operadores: `()` , `**` , `*` , `/` , `//` , `%` , `+` , `-`

| Python não possui operadores de incremento e decremento (`++` e `--`).

Exemplo 1

```
# Cálculo de IMC
peso = float(input("Digite seu peso (kg): "))
altura = float(input("Digite sua altura (m): "))
imc = peso / altura ** 2
print("Seu IMC é:", imc)
```

Exemplo 2

```
# Atribuição com Operação  
x = 5  
x += 3  
print(x)
```


Exemplo 3

```
# Divisão Inteira vs Divisão Real  
a = 10  
b = 3  
print(a // b)  # Divisão Inteira  
print(a / b)   # Divisão Real
```

Exemplo 4

```
# Operador ternário  
idade = 18  
maioridade = "Maior de Idade" if idade >= 18 else "Menor de Idade"  
print(maioridade)
```

Blocos de Código

- Ao contrário de outras linguagens de programação *C-like*, Python não usa chaves para definir blocos de código.
- **Indentação:** Python usa a indentação para definir blocos de código.
- **Blocos de Código:** São usados para agrupar instruções em um bloco.

```
if x > 5:  
    x = 5  
    print(x)  
print("bola")
```

- Portanto, a indentação é muito importante em Python, pois altera o significado do código.

Quebras de Linha

- Python usa quebras de linha para indicar o fim de uma instrução.
- Se uma instrução é muito longa, pode ser dividida em várias linhas usando a barra invertida `\`.
- Se uma instrução está entre parênteses, colchetes ou chaves, ela pode ser dividida em várias linhas sem usar a barra invertida.

```
x = 1 + 2 + 3 + 4 + 5 + \  
    6 + 7 + 8 + 9 + 10
```

```
cores = [  
    "vermelho",  
    "verde",  
    "azul"  
]
```

Comentários

- Comentários são usados para explicar o código e torná-lo mais legível.
- Em Python, os comentários são precedidos pelo caractere `#`.
- Comentários de várias linhas podem ser feitos usando aspas triplas `'''` ou `"""`.

```
# Isto é um comentário  
x = 5 # Isto é outro comentário
```

```
'''  
Isto é um comentário  
de várias linhas  
'''  
x = 5
```

Estruturas de Controle

Estruturas de controle são usadas para controlar o fluxo de execução do programa.

- Estruturas Condicionais: `if` , `elif` , `else`
- Estruturas de Repetição: `for` , `while`
- Estruturas de Controle de Loop: `break` , `continue`
- Estruturas de Controle de Função: `return` , `yield` , `pass`
- Estruturas de Controle de Exceção: `try` , `except` , `finally`
- Estruturas de Controle de Contexto: `with`

IF...ELSE

```
x = 15
if x > 5:
    x = 5
print(x)
```

```
idade = 18
if idade >= 18:
    print("Maior de Idade")
else:
    print("Menor de Idade")
```

ELIF

```
idade = 18
if idade < 18:
    print("Criança")
elif idade < 60:
    print("Adulto")
else:
    print("Idoso")
```


WHILE

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

FOR

```
for i in range(5):  
    print(i)
```

- Se a variável de controle não for usada no bloco de código, pode ser substituída por um sublinhado `_`.

```
for _ in range(5):  
    print("Olá, Mundo!")
```

Função Range

- `range` é uma função que gera uma sequência de números, ela pode receber um, dois ou três argumentos.
- Se receber um argumento, gera uma sequência de 0 até o número anterior ao argumento.
- Se receber dois argumentos, gera uma sequência do primeiro argumento até o número anterior ao segundo argumento.
- Se receber três argumentos, gera uma sequência do primeiro argumento até o número anterior ao segundo argumento, com um intervalo definido pelo terceiro argumento.

- Exemplos

```
for i in range(5):  
    print(i)
```

Saida: 0 1 2 3 4

```
for i in range(2, 5):  
    print(i)
```

Saida: 2 3 4

```
for i in range(1, 10, 2):  
    print(i)
```

Saida: 1 3 5 7 9

- Exemplo com incremento negativo

```
for i in range(5, 0, -1):  
    print(i)
```

Saida: 5 4 3 2 1

for é usado para iterar sobre uma sequência (como uma lista, tupla, dicionário, conjunto ou string) ou outros objetos iteráveis. Veremos mais sobre isso em aulas futuras.

BREAK e CONTINUE

- **Break:** Interrompe a execução do loop.

```
for i in range(5):  
    if i == 3:  
        break  
    print(i)
```

- **Continue:** Interrompe a execução atual do loop e continua com a próxima iteração.

```
for i in range(5):  
    if i == 3:  
        continue  
    print(i)
```

For...Else

- O bloco `else` é executado quando o loop termina sem interrupção.

```
x = 113
for i in range(2, x):
    if x % i == 0:
        print("Não é primo")
        break
else:
    print("É primo")
```

- Também pode ser usado com `while`.

Python não possui do...while.

- Exemplo de simulação do...while

```
while True:
    x = int(input("Digite um número: "))
    x += 1
    if x == 5:
        break
```

Funções

Funções são blocos de código reutilizáveis que realizam uma tarefa específica.

```
def saudacao(nome):  
    print("Olá, ", nome)  
saudacao("Albert")
```

- Uma função é composta por um cabeçalho e um corpo.
- O cabeçalho contém o nome da função e seus parâmetros.
- O corpo contém as instruções que a função executa.
- A função é chamada com um argumento que é passado para o parâmetro.
- A função pode retornar um valor usando a instrução `return`.

Parâmetros

- Python permite definir funções com parâmetros posicionais e parâmetros nomeados.
- Parâmetros posicionais são passados na ordem em que a função foi definida.
- Parâmetros nomeados são passados com um nome e um valor.

```
def saudacao(nome, mensagem="Olá"):  
    print(mensagem + ", ", nome)  
saudacao("Albert")  
  
saudacao("Albert", "Bom dia")
```

Parâmetros Arbitrários

- Python permite definir funções com um número arbitrário de parâmetros.
- Parâmetros arbitrários são passados como uma tupla.
- Parâmetros arbitrários nomeados são passados como um dicionário.

```
def saudacao(*nomes):  
    for nome in nomes:  
        print("Olá,", nome)  
saudacao("Albert", "Maria", "João")
```

```
def saudacao(**nomes):  
    for nome, mensagem in nomes.items():  
        print(mensagem + ",", nome)  
saudacao(Albert="Bom dia", Maria="Boa tarde", João="Boa noite")
```

Regras de Sintaxe para Parâmetros

- Parâmetros posicionais devem vir antes de parâmetros nomeados.
- Parâmetros arbitrários devem vir por último.
- Parâmetros arbitrários nomeados devem vir após parâmetros arbitrários.

```
def saudacao(dia, *nomes, **mensagens):  
    print("Hoje é", dia)  
    for nome in nomes:  
        print(mensagens.get(nome, "Olá") + ", ", nome)  
saudacao("Albert", "Maria", "João", Albert="Bom dia", Maria="Boa tarde", João="Boa noite")
```

Retorno de Valores

- Uma função pode retornar um valor usando a instrução `return`.
- Uma função pode retornar múltiplos valores usando uma tupla.
- `'return'` interrompe a execução da função e retorna um valor.
- Se não houver uma instrução `return`, a função retorna `None`.

```
def soma(a, b):  
    return a + b  
print(soma(2, 3))  
  
def divisao_e_resto(a, b):  
    return a // b, a % b  
print(divisao_e_resto(10, 3))
```

Retorno de Valores `yield`

- A instrução `yield` é usada para retornar um valor de uma função geradora.
- A instrução `yield` suspende a execução da função e retorna um valor.
- Quando a função é chamada novamente, a execução é retomada a partir do ponto onde foi suspensa.

```
def contador():  
    yield 1  
    yield 2  
    yield 3  
for i in contador():  
    print(i)
```

Saída: 1 2 3

Escopo de Variáveis

- O escopo de uma variável é a parte do programa onde a variável é acessível.
- Variáveis definidas dentro de uma função têm escopo local.
- Variáveis definidas fora de uma função têm escopo global.
- Variáveis locais têm precedência sobre variáveis globais.
- Estruturas condicionais e de repetição **não** criam escopo local.

```
x = 5
def funcao():
    x = 10
    print(x)
funcao()
print(x)
```


Acesso a Variáveis Globais

- Variáveis globais podem ser acessadas e modificadas dentro de uma função usando a instrução `global`.

```
x = 5
def funcao():
    global x
    x = 10
    print(x)
funcao()
print(x)
```

Por que evitar variáveis globais?

- **Poluição do Espaço de Nomes:** Variáveis globais podem ser acessadas e modificadas em qualquer lugar do programa, o que pode levar a erros difíceis de depurar.
- **Legibilidade:** Variáveis globais podem dificultar a compreensão do código, pois o comportamento de uma função pode depender do estado de variáveis globais.
- **Manutenção:** Variáveis globais podem dificultar a manutenção do código, pois o comportamento de uma função pode depender do estado de variáveis globais.

Escopo estático

- Python não tem um equivalente direto ao escopo estático, mas é possível simular criando uma variável de função que mantém seu valor entre chamadas.

```
def contador():  
    contador.count += 1  
    return contador.count  
contador.count = 0  
print(contador())  
print(contador())  
print(contador())
```

- No exemplo acima, a variável `contador.count` mantém seu valor entre chamadas da função `contador`.
- Embora `contador.count` seja uma variável global, seu nome associa-se à função `contador` evitando as desvantagens das variáveis globais.

Strings

Strings são usadas para armazenar uma coleção de caracteres. São imutáveis, o que significa que não podem ser alteradas após a criação.

- **Criação de Strings:** As strings podem ser criadas usando aspas simples ou duplas.

```
nome = 'Albert'  
mensagem = "Olá, Mundo!"
```

- **Concatenação de Strings:** As strings podem ser concatenadas usando o operador `+`.

```
a = "Olá"  
b = "Mundo"  
c = a + " " + b
```

Alguns Métodos de Strings

- Métodos de Formatação: `format()`, `f-strings`

```
nome = "Albert"
idade = 30
print("Olá, meu nome é {} e tenho {} anos.".format(nome, idade))
print(f"Olá, meu nome é {nome} e tenho {idade} anos.")
```

- **Métodos de Manipulação:** `upper()` , `lower()` , `capitalize()` , `title()` , `swapcase()`

```
nome = "tutorial de Python"  
print(nome.upper())  
print(nome.capitalize())  
print(nome.title())  
print(nome.swapcase())
```

- Métodos de Busca: `find()`, `index()`, `count()`

```
nome = "tutorial de Python"  
print(nome.find("de"))  
print(nome.index("de"))  
print(nome.count("t"))
```

- **Métodos de Verificação:** `startswith()`, `endswith()`, `isalpha()`, `isdigit()`, `isalnum()`, `isspace()`

```
nome = "tutorial de Python"  
print(nome.startswith("t"))  
print(nome.endswith("n"))  
print(nome.isalpha())  
print(nome.isdigit())  
print(nome.isalnum())  
print(nome.isspace())
```


- **Métodos de Substituição:** `replace()`, `strip()`, `lstrip()`, `rstrip()`

```
nome = "tutorial de Python"
print(nome.replace("Python", "Java"))
print(nome.strip("t"))
print(nome.lstrip("t"))
print(nome.rstrip("n"))
```

- Métodos de Separação: `split()`, `partition()`, `rpartition()`

```
nome = "tutorial de Python"  
print(nome.split())  
print(nome.partition("de"))  
print(nome.rpartition("de"))
```

- **Métodos de União:** `join()`

O método `join()` é usado para unir uma lista de strings em uma única string. É muito útil para formatar saídas de dados e por apresentar melhor desempenho do que a concatenação de strings.

```
palavras = ["tutorial", "de", "Python"]  
print(" ".join(palavras))
```

- Concatenação vs `join()` usando `%timeit`

```
palavras = ["tutorial", "de", "Python"]  
%timeit x = " ".join(palavras)  
%timeit x = palavras[0] + " " + palavras[1] + " " + palavras[2]
```

Caracteres de Escape

São usados para representar caracteres especiais em strings.

- `\n` : Nova linha
- `\t` : Tabulação
- `\\` : Barra invertida
- `\'` : Aspas simples
- `\"` : Aspas duplas
- `\r` : Retorno de carro
- `\v` : Tabulação vertical
- `\xhh` : Caractere ASCII em hexadecimal
- `\uXXXX` : Caractere Unicode

Coleções de Dados

Coleções de dados são usadas para armazenar múltiplos valores em uma única variável.

- **Listas:** São usadas para armazenar uma coleção ordenada de itens.
- **Tuplas:** São usadas para armazenar uma coleção ordenada e imutável de itens.
- **Conjuntos:** São usados para armazenar uma coleção não ordenada e sem duplicatas de itens.
- **Dicionários:** São usados para armazenar uma coleção de pares chave-valor.

Listas

Listas são usadas para armazenar uma coleção ordenada de itens. Estes itens podem ser de diferentes tipos.

- **Criação de Listas:** As listas podem ser criadas usando colchetes `[]` ou a função `list()`.

```
cores = ["vermelho", "verde", "azul"]  
numeros = list((1, 2, 3, 4, 5))
```

- **Acesso a Itens:** Os itens de uma lista podem ser acessados por índice.

```
cores = ["vermelho", "verde", "azul"]  
print(cores[0])  
print(cores[1])  
print(cores[2])
```

- **Alteração de Itens:** Os itens de uma lista podem ser alterados por índice.

```
cores = ["vermelho", "verde", "azul"]  
cores[0] = "amarelo"  
print(cores)
```

- **Adição de Itens:** Há vários métodos para adicionar itens a uma lista.
 - `append()` : Adiciona um item ao final da lista.
 - `insert()` : Adiciona um item em uma posição específica da lista.
 - `extend()` : Adiciona os itens de uma lista a outra lista.
 - `+=` : Adiciona os itens de uma lista a outra lista.
 - `+` : Concatena duas listas.
 - `*` : Repete os itens de uma lista.

```
cores = ["vermelho", "verde", "azul"]
cores.append("amarelo")
cores.insert(1, "laranja")
cores.extend(["roxo", "rosa"])
cores += ["preto", "branco"]
cores = cores + ["cinza", "marrom"]
cores *= 2
print(cores)
```


- **Remoção de Itens:** Há vários métodos para remover itens de uma lista.
 - **remove()** : Remove o primeiro item com um valor específico.
 - **pop()** : Remove um item em uma posição específica da lista.
 - **del** : Remove um item em uma posição específica da lista.
 - **clear()** : Remove todos os itens da lista.

```
cores = ["vermelho", "verde", "azul"]
cores.remove("verde")
cores.pop(1)
del cores[0]
cores.clear()
print(cores)
```

- `sort()` : Ordena os itens de uma lista em ordem crescente.
- `reverse()` : Inverte a ordem dos itens de uma lista.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]  
numeros.sort()  
numeros.reverse()  
print(numeros)
```

- `index()` : Retorna a posição de um item específico.
- `count()` : Retorna o número de vezes que um item aparece na lista.
- `len()` : Retorna o número de itens na lista.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]  
print(numeros.index(5))  
print(numeros.count(5))  
print(len(numeros))
```

Slice

O slice é usado para acessar um subconjunto de itens de uma lista. Possui lógica semelhante à função `range()`.

- **Sintaxe:** `lista[início:fim:passo]`
- **Início:** Índice de início do slice.
- **Fim:** Índice de fim do slice.
- **Passo:** Tamanho do passo do slice.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
print(numeros[2:5])
print(numeros[:5])
print(numeros[5:])
print(numeros[::2])
print(numeros[::-1])
```

- É possível usar slice para alterar itens de uma lista.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]  
numeros[2:5] = [0, 0, 0]  
print(numeros)
```

Indexação Negativa

- A indexação negativa é usada para acessar itens de uma lista a partir do final.
- O índice `-1` refere-se ao último item da lista.
- O índice `-2` refere-se ao penúltimo item da lista.
- E assim por diante.

```
numeros = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]  
print(numeros[-1])  
print(numeros[-2])  
print(numeros[-3])
```

List Comprehension

É uma maneira eficiente de criar listas sem usar loops.

```
numeros = [1, 2, 3, 4, 5]
quadrados = [x ** 2 for x in numeros]
print(quadrados)
```

- List Comprehension pode ser usado para filtrar itens de uma lista.

```
numeros = [1, 2, 3, 4, 5]
pares = [x for x in numeros if x % 2 == 0]
print(pares)
```

Tuplas

Tuplas são usadas para armazenar uma coleção ordenada e imutável de itens.

- **Criação de Tuplas:** As tuplas podem ser criadas usando parênteses `()` ou a função `tuple()`.

```
cores = ("vermelho", "verde", "azul")  
numeros = tuple([1, 2, 3, 4, 5])
```

- **Acesso a Itens:** Os itens de uma tupla podem ser acessados por índice.

```
cores = ("vermelho", "verde", "azul")  
print(cores[0])  
print(cores[1])  
print(cores[2])
```


- **Alteração de Itens:** Os itens de uma tupla **não** podem ser alterados após a criação.

```
cores = ("vermelho", "verde", "azul")  
cores[0] = "amarelo" # Erro!
```

Por que usar tuplas?

- **Imutabilidade:** As tuplas são imutáveis, o que significa que seus itens não podem ser alterados após a criação.
- **Desempacotamento:** As tuplas podem ser desempacotadas em variáveis individuais.
- **Retorno Múltiplo:** As funções podem retornar múltiplos valores como uma tupla.
- **Iteração Eficiente:** As tuplas são mais eficientes para iteração do que listas.
- **Chaves de Dicionários:** As tuplas podem ser usadas como chaves de dicionários.
- **Segurança:** As tuplas são mais seguras do que listas em ambientes concorrentes.

Desempacotamento de Tuplas

- O desempacotamento de tuplas é usado para atribuir os itens de uma tupla a variáveis individuais.

```
cores = ("vermelho", "verde", "azul")  
r, g, b = cores  
print(r, g, b)
```

- Troca de valores de variáveis (swap).

```
a = 1
b = 2
a, b = b, a
print(a, b)
```

- Atribuição de valores a múltiplas variáveis.

```
a, b, c = 1, 2, 3
print(a, b, c)
```

- Retorno múltiplo de funções.

```
def divisao_e_resto(a, b):
    return a // b, a % b
div, res = divisao_e_resto(10, 3)
print(div, res)
```

- Ignorar valores de uma tupla.

```
a, _, c = (1, 2, 3)
print(a, c)
```

- Trocar valores de uma lista ou array.

```
numeros = [1, 2, 3, 4, 5]
numeros[0], numeros[-1] = numeros[-1], numeros[0]
print(numeros)
```

Conjuntos

Conjuntos são usados para armazenar uma coleção não ordenada e sem duplicatas de itens.

- **Criação de Conjuntos:** Os conjuntos podem ser criados usando chaves `{}` ou a função `set()`.

```
cores = {"vermelho", "verde", "azul"}  
numeros = set([1, 2, 3, 4, 5])  
vazio = {} # Dicionário, não conjunto!  
valido = set() # Conjunto vazio
```

- **Acesso a Itens:** Os itens de um conjunto **não** podem ser acessados por índice.

```
cores = {"vermelho", "verde", "azul"}  
print(cores[0]) # Erro!
```

- **Adição de Itens:** Há vários métodos para adicionar itens a um conjunto.
 - **add()** : Adiciona um item ao conjunto.
 - **update()** : Adiciona os itens de um conjunto a outro conjunto.
 - **|** : Adiciona os itens de um conjunto a outro conjunto.
 - **union()** : Adiciona os itens de um conjunto a outro conjunto.

```
cores = {"vermelho", "verde", "azul"}  
cores.add("amarelo")  
cores.update(["roxo", "rosa"])  
cores |= {"preto", "branco"}  
cores = cores.union(["cinza", "marrom"])  
print(cores)
```

- **Remoção de Itens:** Há vários métodos para remover itens de um conjunto.
 - **remove()** : Remove um item do conjunto.
 - **discard()** : Remove um item do conjunto.
 - **pop()** : Remove um item do conjunto.
 - **clear()** : Remove todos os itens do conjunto.

```
cores = {"vermelho", "verde", "azul"}  
cores.remove("verde")  
cores.discard("verde")  
cores.pop()  
cores.clear()  
print(cores)
```


- **Operações de Conjuntos:** Há vários operadores para realizar operações de conjuntos.
 - `|` : União
 - `&` : Interseção
 - `-` : Diferença
 - `^` : Diferença Simétrica

```
pares = {2, 4, 6, 8, 10}
primos = {2, 3, 5, 7, 11}
print(pares | primos)
print(pares & primos)
print(pares - primos)
print(pares ^ primos)
```

Dicionários

Dicionários são usados para armazenar uma coleção de pares chave-valor. As chaves de um dicionário devem ser únicas e imutáveis.

- **Criação de Dicionários:** Os dicionários podem ser criados usando chaves `{}` ou a função `dict()`.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
numeros = dict([(1, "um"), (2, "dois"), (3, "três")])
```

- **Acesso a Itens:** Os itens de um dicionário podem ser acessados por chave.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
print(cores["r"])  
print(cores["g"])  
print(cores["b"])
```

- **Adição de Itens:** Os itens de um dicionário podem ser adicionados por chave.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
cores["y"] = "amarelo"  
print(cores)
```

- **Remoção de Itens:** Há vários métodos para remover itens de um dicionário.
 - **pop()** : Remove um item do dicionário por chave.
 - **popitem()** : Remove o último item do dicionário.
 - **del** : Remove um item do dicionário por chave.
 - **clear()** : Remove todos os itens do dicionário.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
cores.pop("g")  
cores.popitem()  
del cores["r"]  
cores.clear()  
print(cores)
```

- **Métodos de Dicionários:** Há vários métodos para manipular dicionários.
 - **keys()** : Retorna uma lista de chaves do dicionário.
 - **values()** : Retorna uma lista de valores do dicionário.
 - **items()** : Retorna uma lista de pares chave-valor do dicionário.
 - **get()** : Retorna o valor de uma chave específica.
 - **update()** : Atualiza o dicionário com outro dicionário.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
print(cores.keys())  
print(cores.values())  
print(cores.items())  
print(cores.get("g"))  
cores.update({"y": "amarelo", "p": "roxo"})  
print(cores)
```

- **Valor Padrão:** O método `get()` pode retornar um valor padrão se a chave não existir.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
print(cores.get("y", "não encontrado"))
```

- **Verificação de Chave:** O operador `in` pode ser usado para verificar se uma chave existe em um dicionário.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
print("r" in cores)  
print("y" in cores)
```

Compreensão de Dicionários

A compreensão de dicionários é usada para criar dicionários de maneira eficiente.

```
numeros = [1, 2, 3, 4, 5]  
quadrados = {x: x ** 2 for x in numeros}  
print(quadrados)
```

Comparação de Coleções

- **Listas:** São usadas para armazenar uma coleção ordenada de itens.
- **Tuplas:** São usadas para armazenar uma coleção ordenada e **imutável** de itens.
- **Conjuntos:** São usados para armazenar uma coleção **não ordenada** e sem duplicatas de itens.
- **Dicionários:** São usados para armazenar uma coleção de pares **chave-valor**.

Iteração sobre Coleções

Iteração é o processo de acessar itens de uma coleção de dados.

- **For Loop:** É usado para iterar sobre uma sequência (como uma lista, tupla, conjunto ou string) ou outros objetos iteráveis.

```
cores = ["vermelho", "verde", "azul"]  
for cor in cores:  
    print(cor)
```

- **Iteração com Índice:** A função `enumerate()` pode ser usada para iterar sobre uma sequência com índices.

```
cores = ["vermelho", "verde", "azul"]  
for i, cor in enumerate(cores):  
    print(i, cor)
```

- **Iteração em Dicionários:** Os dicionários podem ser iterados por chave, valor ou ambos.

```
cores = {"r": "vermelho", "g": "verde", "b": "azul"}  
for chave in cores:  
    print(chave, cores[chave])  
for chave, valor in cores.items():  
    print(chave, valor)
```

Observe que a ordem de iteração em um dicionário é arbitrária.

No segundo exemplo, a função `items()` retorna uma lista de pares chave-valor que é desempacotado.

