PYTHON

MATERIAL DE APOIO

Sumário

Introdução ao Python

Python foi criado por Guido van Rossum, um programador holandês, no final dos anos 80 e início dos anos 90. A primeira versão do Python, a 0.9.0, foi lançada em 1991. Guido van Rossum nomeou a linguagem em homenagem ao grupo de comédia britânico Monty Python, do qual era um grande fã.

Python foi projetado com o objetivo de ser uma linguagem fácil de ler e escrever, com uma sintaxe clara e concisa. Ao longo dos anos, evoluiu e ganhou popularidade até se tornar uma das linguagens de programação mais utilizadas no mundo.

Características

Python tem várias características que o tornam atraente tanto para iniciantes quanto para programadores experientes:

Legibilidade

Python utiliza uma sintaxe clara e simples, o que facilita a leitura e compreensão do código. Utiliza indentação (espaços ou tabulações) para delimitar blocos de código, o que promove um estilo de programação estruturado e legível.

Tipagem dinâmica

Em Python, não é necessário declarar explicitamente o tipo de dados das variáveis. Python infere automaticamente o tipo de dados com base no valor atribuído a uma variável, o que simplifica a escrita de código.

Interpretado

Python é uma linguagem interpretada, o que significa que o código é executado linha por linha em tempo real. Isso permite um ciclo de desenvolvimento rápido e facilita a depuração do código.

Multiplataforma

Python pode ser executado em diferentes sistemas operacionais, como Windows, macOS e Linux, sem necessidade de modificar o código. Isso o torna uma linguagem versátil e portátil.

Ampla biblioteca padrão

Python vem com uma extensa biblioteca padrão que fornece uma grande quantidade de módulos e funções para realizar diversas tarefas, como manipulação de arquivos, conexão a bancos de dados, processamento de texto, entre outros.

Comunidade ativa

Python conta com uma comunidade de desenvolvedores grande e ativa que contribui com bibliotecas, *frameworks* e ferramentas adicionais. Isso significa que você encontrará uma grande quantidade de recursos e suporte disponíveis.

Aplicações

Python é utilizado em uma ampla gama de aplicações e domínios, alguns exemplos são:

Desenvolvimento web

Python é amplamente utilizado no desenvolvimento web *backend*, com *frameworks* populares como Django e Flask.

Ciência de dados

Python é a linguagem preferida para análise de dados e ciência de dados devido a bibliotecas como NumPy, Pandas e Matplotlib.

Inteligência artificial e machine learning

Python é a escolha principal para projetos de IA e *machine learning,* graças a bibliotecas como TensorFlow e Scikit-learn.

Automatização de tarefas

Python pode ser utilizado para automatizar tarefas repetitivas, como processamento de arquivos, web *scraping* e testes de *software*.

Desenvolvimento de jogos

Python é utilizado no desenvolvimento de jogos simples, especialmente com bibliotecas como Pygame.

Baixar e instalar Python

Para começar a programar em Python, primeiro você deve baixar e instalar Python no seu computador. Siga estes passos:

1. Vá ao site oficial do Python: https://www.python.org/downloads.
2. Na seção "Download" você encontrará a última versão estável do Python. Selecione a versão adequada para o seu sistema operacional (Windows, macOS ou Linux).
3. Baixe o instalador do Python correspondente ao seu sistema operacional.
4. Uma vez baixado, execute o instalador. Certifique-se de marcar a opção "Add Python to PATH" durante o processo de instalação no Windows. Isso permitirá que você execute Python a partir da linha de comando.
5. Siga as instruções do instalador e espere a instalação ser concluída.

Parabéns! Agora você tem Python instalado no seu computador.

Configuração do ambiente de desenvolvimento

Um ambiente de desenvolvimento integrado (IDE) é uma ferramenta que facilita a escrita, execução e depuração de código. Embora você possa usar um editor de texto simples para escrever código Python, um IDE oferece recursos adicionais que melhoram a produtividade do desenvolvimento.

Alguns IDEs populares para Python são:

- **PyCharm:** é um IDE poderoso e completo desenvolvido pela JetBrains. Oferece funções avançadas, como autocompletar código, depuração, integração com sistemas de controle de versão e mais.
- **Visual Studio Code:** é um editor de código desenvolvido pela Microsoft. Com a extensão de Python, se torna um IDE leve e versátil para o desenvolvimento de Python.
- **Sublime Text:** é um editor de texto rápido e personalizável que suporta programação em Python através do uso de *plugins*.

Para começar, recomendamos usar o Visual Studio Code, pois é leve, fácil de usar e tem uma grande quantidade de extensões disponíveis.

"Olá Mundo"

É uma tradição no mundo da programação começar com um programa simples chamado "Olá Mundo". Este programa simplesmente mostra a mensagem "Olá Mundo" na tela.

- Abra seu IDE ou editor de texto preferido e crie um novo arquivo.
- Nomeie o arquivo como "ola_mundo.py". A extensão ".py" indica que é um arquivo de Python.
- No arquivo, escreva o seguinte código:

```
print ("Olá, Mundo!")
```

- Salve o arquivo e execute o programa. Se estiver utilizando um IDE, procure a opção "Run" ou "Execute".

Você verá que a mensagem "Olá, Mundo!" é impressa na tela.

Conceitos básicos da sintaxe em Python

Antes de mergulharmos em conceitos mais avançados, é importante familiarizar-se com alguns aspectos básicos da sintaxe do Python.

- Indentação

No Python, a indentação (espaços ou tabulações no início de uma linha) é utilizada para delimitar blocos de código. Diferente de outras linguagens que utilizam chaves ou palavras-chave, o Python utiliza a indentação para determinar o escopo das declarações. Por exemplo:

```
if condition:
  # Bloco de código se a condição for verdadeira
  instrucao1
  instrucao2
else:
  # Bloco de código se a condição for falsa
  instrucao3
  instrucao4
```

É fundamental manter uma indentação consistente em todo o código para evitar erros de sintaxe.

Comentários

Os comentários são linhas de texto no código que são ignoradas pelo interpretador do Python. Eles são utilizados para explicar ou documentar o código. No Python, os comentários de uma única linha começam com o símbolo #, enquanto os comentários de várias linhas são delimitados por três aspas """ . Por exemplo:

```
# Este é um comentário de uma única linha
"""
Este é um comentário
de várias linhas
"""
```

- Sensibilidade a maiúsculas e minúsculas

Python distingue entre maiúsculas e minúsculas.
Portanto, variável, Variável e VARIÁVEL são consideradas variáveis diferentes.

Ponto e vírgula

Diferente de outras linguagens, o Python não requer o uso de ponto e vírgula (;) ao final de cada instrução. No entanto, se você desejar escrever várias instruções em uma única linha, pode separá-las com um ponto e vírgula. Por exemplo:
instrucao1; instrucao2; instrucao3

Uso de parênteses

Os parênteses são utilizados para agrupar expressões, definir funções e realizar chamadas a funções. Por exemplo:
resultado = (a + b) * c

Tipos de dados básicos

Em Python, os tipos de dados básicos são as categorias nas quais podemos classificar os valores que utilizamos em nossos programas. Compreender os diferentes tipos de dados é fundamental para trabalhar com variáveis e realizar operações em Python. Os tipos de dados básicos incluem:

Inteiros (int)

Os números inteiros são aqueles que não têm parte decimal. Em Python, são representados simplesmente escrevendo o número sem aspas nem pontos decimais. Por exemplo:

```
idade = 25
quantidade = 100
```

Flutuantes (float)

Os números flutuantes, também conhecidos como números de ponto flutuante, são aqueles que têm uma parte decimal. Em Python, são representados utilizando um ponto para separar a parte inteira da parte decimal. Por exemplo:

```
preço = 9.99
altura = 1.75
```

Cadeias de texto (strings)

As cadeias de texto, ou simplesmente cadeias, são sequências de caracteres encerradas entre aspas simples ('...') ou duplas ("..."). São utilizadas para representar texto em Python. Por exemplo:

```
nome = "Juan"
mensagem = '¡Hola, mundo!'
```

Você pode incluir caracteres especiais nas cadeias utilizando o caractere de escape \. Por exemplo, para incluir aspas dentro de uma cadeia, você pode usar \' ou \". Também pode utilizar a notação de tripla aspa ('''...''' ou """...""") para criar cadeias de várias linhas.

Booleanos

Os valores booleanos representam os valores de verdade: True (verdadeiro) e False (falso). São comumente utilizados em expressões condicionais e operações lógicas. Por exemplo:

```
é_maior_de_idade = True
tem_desconto = False
```

Os valores booleanos em Python começam com uma letra maiúscula: True e False.

Variáveis

As variáveis são contêineres que nos permitem armazenar e manipular dados em nossos programas. Você pode pensar em uma variável como uma etiqueta à qual você atribui um valor específico. Em Python, não é necessário declarar o tipo de dados de uma variável com antecedência, pois o Python infere o tipo de dados automaticamente com base no valor atribuído.

Declaração e atribuição de variáveis

As variáveis são contêineres que nos permitem armazenar e manipular dados em nossos programas. Para declarar e atribuir um valor a uma variável em Python, utilizamos o operador de atribuição =. O nome da variável vai à esquerda do operador, e o valor que você deseja atribuir vai à direita. Por exemplo:

```
nome = "Juan"
idade = 25
altura = 1.75
é estudante = True
```

No exemplo, declaramos e atribuímos valores às variáveis nome, idade, altura e é_estudante. O Python infere automaticamente o tipo de dados de cada variável com base no valor atribuído.

Você também pode atribuir o mesmo valor a várias variáveis em uma única linha usando o operador de atribuição múltipla:

a = b = c = 10

Neste caso, as variáveis a, b e c terão o valor 10.

Regras para nomear variáveis

Ao nomear variáveis em Python, é importante seguir algumas regras para manter um código legível e evitar erros:

Os nomes das variáveis só podem conter letras (a-z, A-Z), números (0-9) e sublinhados (_). Não podem começar com um número.

O Python diferencia maiúsculas de minúsculas, então nome e Nome são variáveis diferentes.

Não se pode usar palavras-chave reservadas do Python como nomes de variáveis (por exemplo, if, else, for, while, etc.).

Recomenda-se usar nomes descritivos para as variáveis, que indiquem claramente seu propósito: nome, idade, total_vendas, etc.

Seguindo essas regras, alguns exemplos de nomes de variáveis válidos são:

```
idade
nome_completo
total_vendas
_contador
```

E alguns exemplos de nomes de variáveis inválidos são:

```
1idade   # Começa com um número
nome-completo   # Usa um hífen em vez de um sublinhado
if   # Palavra-chave reservada do Python
```

Escolher nomes descritivos para suas variáveis facilita a leitura e compreensão do código, tanto para você quanto para outros desenvolvedores que possam trabalhar no mesmo projeto.

Operadores

Os operadores são símbolos especiais que nos permitem realizar operações em variáveis e valores. Python fornece diferentes tipos de operadores para realizar operações aritméticas, comparações e operações lógicas.

Aritméticos
Os operadores aritméticos são utilizados para realizar operações matemáticas básicas. Os principais operadores aritméticos em Python são:

- Soma (+): soma dois valores.
- Subtração (-): subtrai o segundo valor do primeiro.
- Multiplicação (*): multiplica dois valores.
- Divisão (/): divide o primeiro valor pelo segundo e devolve um resultado de tipo flutuante.
- Divisão inteira (//): divide o primeiro valor pelo segundo e devolve um resultado de tipo inteiro (a parte decimal é descartada).
- Módulo (%): devolve o resto da divisão entre o primeiro valor e o segundo.
- Exponenciação (**): eleva o primeiro valor à potência do segundo.

Exemplos:

```
a = 10
b = 3
soma = a + b   # 13
subtracao = a - b    # 7
multiplicacao = a * b    # 30
divisao = a / b   # 3.333333333
divisao_inteira = a // b   # 3
modulo = a % b   # 1
exponenciacao = a ** b   # 1000
```

De comparação
Os operadores de comparação são utilizados para comparar dois valores e devolvem um valor booleano (True ou False) segundo o resultado da comparação. Os operadores de comparação em Python são:

- Igual a (==): devolve True se ambos os valores são iguais.
- Diferente de (!=): devolve True se os valores são diferentes.
- Maior que (>): devolve True se o primeiro valor é maior que o segundo.
- Menor que (<): devolve True se o primeiro valor é menor que o segundo.
- Maior ou igual que (>=): devolve True se o primeiro valor é maior ou igual que o segundo.
- Menor ou igual que (<=): devolve True se o primeiro valor é menor ou igual que o segundo.

Exemplos:

```
a = 10
b = 3
igual = a == b   # False
diferente = a != b   # True
maior que = a > b   # True
menor que = a < b   # False
maior ou igual = a >= b   # True
menor ou igual = a <= b   # False
```

Lógicos

Os operadores lógicos são utilizados para combinar expressões condicionais e avaliar múltiplas condições. Os operadores lógicos em Python são:

- AND (and): devolve True se ambas as condições são verdadeiras.
- OR (or): devolve True se ao menos uma das condições é verdadeira.
- NOT (not): inverte o valor de uma condição, devolve True se a condição é falsa e False se a condição é verdadeira.

Exemplo:

```
a = 10
b = 3
resultado_and = (a > 5) and (b < 5)   # True
resultado_or = (a > 15) or (b < 5)   # True
resultado_not = not (a > 5)   # False
```

Você pode utilizar esses operadores para realizar cálculos, tomar decisões baseadas em comparações e combinar condições lógicas em seus programas.

Python segue as regras de precedência de operadores, onde certos operadores têm prioridade sobre outros. Em geral, a precedência segue a ordem: parênteses, exponenciação, multiplicação/divisão, soma/subtração, operadores de comparação e operadores lógicos.

Estrutura de Controle

As estruturas de controle nos permitem controlar o fluxo de execução de nossos programas. Em Python, as estruturas de controle mais comuns são as estruturas condicionais e os loops. Essas estruturas nos permitem tomar decisões e repetir blocos de código segundo certas condições.

Estruturas condicionais
As estruturas condicionais nos permitem executar diferentes blocos de código segundo se cumpra ou não uma determinada condição. Em Python, as estruturas condicionais mais utilizadas são if, if-else e if-elif-else.

IF
A estrutura if é utilizada para executar um bloco de código se uma condição for verdadeira. A sintaxe básica é a seguinte:

```
if condicao:
  # Bloco de código a executar se a condição for verdadeira
  instruções
```

```
Exemplo:
idade = 18
if idade >= 18:
  print ("Você é maior de idade.")
```

Neste exemplo, se a variável idade for maior ou igual a 18, será executado o bloco de código dentro do if e será impressa a mensagem "Você é maior de idade."

IF-ELSE

A estrutura if-else nos permite especificar um bloco de código alternativo que será executado se a condição do if for falsa. A sintaxe básica é a seguinte:

```
idade = 15
if idade >= 18:
  print ("Você é maior de idade.")
else:
  print ("Você é menor de idade.")
```

Neste exemplo, se a variável idade for maior ou igual a 18, será executado o bloco de código dentro do if e será impressa a mensagem "Você é maior de idade." Caso contrário, será executado o bloco de código dentro do else e será impressa a mensagem "Você é menor de idade."

IF-ELIF-ELSE

A estrutura if-elif-else nos permite especificar múltiplas condições e blocos de código alternativos. A sintaxe básica é a seguinte:

```
if condicao1:
  # Bloco de código a executar se a condicao1 for verdadeira
  instruções
elif condicao2:
  # Bloco de código a executar se a condicao2 for verdadeira
  instruções
else:
  # Bloco de código a executar se nenhuma condição anterior for verdadeira
  instruções
```

Exemplo:

```
nota = 85
if nota >= 90:
  print ("Excelente")
elif nota >= 80:
  print ("Muito bom")
elif nota >= 70:
  print ("Bom")
else:
  print ("Precisa melhorar")
```

Neste exemplo, são avaliadas múltiplas condições em ordem. Se a variável nota for maior ou igual a 90, será impresso "Excelente". Se não se cumprir a primeira condição, mas nota for maior ou igual a 80, será impresso "Muito bom". Se não se cumprirem as condições anteriores, mas nota for maior ou igual a 70, será impresso "Bom". Se nenhuma das condições anteriores for verdadeira, será executado o bloco else e será impresso "Precisa melhorar".

Loops / Repetições

Os loops nos permitem repetir um bloco de código várias vezes. Em Python, os loops mais comuns são for e while.

FOR
O loop for é utilizado para iterar sobre uma sequência (como uma lista, uma tupla ou uma string) ou qualquer objeto iterável. A sintaxe básica é a seguinte:

```
for variável in sequência:
    # Bloco de código a repetir
    instruções
```

Exemplo:

```
frutas = ["maçã", "banana", "laranja"]
for fruta in frutas:
    print(fruta)
```

Neste exemplo, o loop for itera sobre a lista frutas. Em cada iteração, a variável fruta assume o valor de um elemento da lista, e o bloco de código dentro do loop é executado. Neste caso, cada fruta é impressa em uma linha separada.

WHILE
O loop while é utilizado para repetir um bloco de código enquanto uma condição for verdadeira. A sintaxe básica é a seguinte:

```
while condição:
    # Bloco de código a repetir
    instruções
```

Exemplo:

```
contador = 0
while contador < 5:
    print(contador)
    contador += 1
```

Neste exemplo, o loop while é executado enquanto a variável contador for menor que 5. Em cada iteração, o valor de contador é impresso e depois incrementado em 1 pela instrução contador += 1. O loop será interrompido quando contador atingir o valor de 5. É importante ter cuidado ao usar o loop while, pois, se a condição nunca se tornar falsa, o loop será executado indefinidamente, o que é conhecido como um loop infinito.

Controle de loops

Python fornece algumas instruções especiais para controlar o fluxo de execução dentro dos loops:

BREAK

A instrução break é utilizada para sair prematuramente de um loop, independentemente da condição. Quando um break é encontrado, o loop é interrompido e o fluxo de execução continua com a próxima instrução fora do loop.
contador = 0

```
while True:
  print(contador)
  contador += 1
  if contador == 5:
    break
```

Neste exemplo, o loop while é executado indefinidamente devido à condição True. No entanto, dentro do loop é utilizada uma estrutura condicional if para verificar se contador é igual a 5. Quando essa condição é satisfeita, a instrução break é executada, fazendo com que o loop seja interrompido e o fluxo de execução continue com a próxima instrução fora do loop.

CONTINUE

A instrução continue é utilizada para pular o restante do bloco de código dentro de um loop e passar para a próxima iteração.

Exemplo:

```
for i in range(10):
  if i % 2 == 0:
    continue
  print(i)
```

Neste exemplo, o loop for itera sobre os números de 0 a 9 utilizando a função range(). Dentro do loop, verifica-se se o número é divisível por 2 utilizando o operador de módulo %. Se o número for divisível por 2 (ou seja, se for par), a instrução continue é executada, fazendo com que o restante do bloco de código seja pulado e passando para a próxima iteração do loop. Como resultado, apenas os números ímpares serão impressos.

PASS

A instrução pass é uma operação nula que não faz nada. É utilizada como um marcador de posição quando uma instrução é sintaticamente necessária, mas nenhuma ação é desejada.

Exemplo:

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

Neste exemplo, o loop for itera sobre os números de 0 a 4, mas nenhuma ação é realizada dentro do loop devido à instrução pass. Isso pode ser útil quando se está desenvolvendo um programa e se deseja reservar um bloco de código para implementá-lo mais tarde.

As estruturas de controle são ferramentas poderosas que nos permitem controlar o fluxo de execução de nossos programas.

Com as estruturas condicionais (if, if-else, if-elif-else) podemos tomar decisões baseadas em condições, enquanto que com os loops (for, while) podemos repetir blocos de código várias vezes.

Além disso, as instruções break, continue e pass nos fornecem um controle adicional sobre o comportamento dos loops.

Estrutura de Dados

As estruturas de dados nos permitem organizar e armazenar dados de maneira eficiente em nossos programas. Python fornece várias estruturas de dados integradas, como listas, tuplas, dicionários e conjuntos, cada uma com suas próprias características e usos.

Listas

Uma lista é uma estrutura de dados mutável e ordenada que permite armazenar uma coleção de elementos. Os elementos de uma lista podem ser de diferentes tipos de dados e são encerrados entre colchetes [], separados por vírgulas.

Criação e acesso

Para criar uma lista, simplesmente encerre os elementos entre colchetes:

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

Para acessar os elementos de uma lista, utilize o índice do elemento entre colchetes. Os índices começam a partir de 0.

```
print(frutas[0])  # Imprime "maçã"
print(frutas[1])  # Imprime "banana"
print(frutas[2])  # Imprime "laranja"
```

Você também pode acessar os elementos a partir do final da lista utilizando índices negativos. O índice -1 representa o último elemento, -2 representa o penúltimo, e assim por diante.

```
print(frutas[-1])  # Imprime "laranja"
print(frutas[-2])  # Imprime "banana"
print(frutas[-3])  # Imprime "maçã"
```

Métodos de listas

As listas em Python têm vários métodos incorporados que nos permitem manipular e modificar os elementos da lista. Alguns métodos comuns são:

- append(elemento): adiciona um elemento ao final da lista.
- insert(indice, elemento): insere um elemento em uma posição específica da lista.
- remove(elemento): remove a primeira ocorrência de um elemento na lista.
- pop(indice): remove e retorna o elemento em uma posição específica da lista.
- sort(): ordena os elementos da lista em ordem ascendente.
- reverse(): inverte a ordem dos elementos na lista.

Exemplo:

```
frutas = ["maçã", "banana", "laranja"]
frutas.append("pera")
print(frutas)  # Imprime ["maçã", "banana", "laranja", "pera"]
frutas.insert(1, "uva")
print(frutas)  # Imprime ["maçã", "uva", "banana", "laranja", "pera"]
frutas.remove("banana")
print(frutas)  # Imprime ["maçã", "uva", "laranja", "pera"]
fruta_removida = frutas.pop(2)
print(frutas)  # Imprime ["maçã", "uva", "pera"]
print(fruta_removida)  # Imprime "laranja"
frutas.sort()
print(frutas)  # Imprime ["maçã", "pera", "uva"]
frutas.reverse()
print(frutas)  # Imprime ["uva", "pera", "maçã"]
```

Listas de compreensão

As listas de compreensão são uma forma concisa de criar novas listas baseadas em uma sequência existente. Permitem filtrar e transformar os elementos de uma lista em uma única linha de código.

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

Exemplo:

```
números = [1, 2, 3, 4, 5]
quadrados = [x ** 2 for x in números if x % 2 == 0]
print(quadrados)  # Imprime [4, 16]
```

Neste exemplo, é criada uma nova lista chamada quadrados, que contém os quadrados dos números pares da lista números. A expressão x ** 2 eleva cada elemento ao quadrado, e a condição if x % 2 == 0 filtra apenas os números pares.

Tuplas

Uma tupla é uma estrutura de dados imutável e ordenada que permite armazenar uma coleção de elementos. Os elementos de uma tupla são encerrados entre parênteses (), separados por vírgulas.

Criação e acesso

Para criar uma tupla, encerre os elementos entre parênteses:

```
ponto = (3, 4)
```

Para acessar os elementos de uma tupla, utilize o índice do elemento entre colchetes, similar às listas:

```
print(ponto[0])  # Imprime 3
print(ponto[1])  # Imprime 4
```

Ao contrário das listas, as tuplas são imutáveis, o que significa que não podem ser modificadas uma vez criadas. Não se pode adicionar, eliminar ou alterar elementos em uma tupla existente.

As tuplas são úteis quando você precisa armazenar uma coleção de elementos que não devem ser modificados, como coordenadas ou dados de configuração.

Métodos de tuplas

Embora as tuplas sejam imutáveis, Python fornece vários métodos úteis para trabalhar com elas:

- **count(elemento):** devolve o número de vezes que um elemento aparece na tupla.
- **index(elemento):** devolve o índice da primeira aparição de um elemento na tupla. Opcionalmente, pode-se especificar o início e fim da busca.
- **len(tupla):** embora não seja um método de tupla propriamente dito, esta função incorporada devolve o comprimento da tupla.

```
minha_tupla = (1, 2, 3, 2, 4, 2)
print (minha_tupla.index(2))   # Saída: 1
print (minha_tupla.index(2, 2))   #Saída: 3
print (minha_tupla.index(2, 2, 4))   #Saída: 3
```

Dicionários

Um dicionário é uma estrutura de dados mutável e não ordenada que permite armazenar pares de chave-valor. Cada elemento em um dicionário consiste em uma chave única e seu valor correspondente. Os dicionários são delimitados por chaves {}, e os pares chave-valor são separados por vírgulas.

Criação e acesso

Para criar um dicionário, utilize chaves e separe as chaves e valores com dois pontos.

```
pessoa = {"nome": "João", "idade": 25, "cidade": "Madri"}
```

Para acessar os valores de um dicionário, utilize a chave correspondente entre colchetes:

```
print(pessoa["nome"])  # Imprime "João"
print(pessoa["idade"])   # Imprime 25
print(pessoa["cidade"])  # Imprime "Madri"
```

Você também pode utilizar o método get() para obter o valor de uma chave. Se a chave não existir, retorna um valor padrão (por padrão, None).

Métodos de dicionários

Os dicionários em Python têm vários métodos incorporados para manipular e acessar os elementos. Alguns métodos comuns são:

- **keys():** retorna uma visualização de todas as chaves do dicionário.
- **values():** retorna uma visualização de todos os valores do dicionário.
- **items():** retorna uma visualização de todos os pares chave-valor do dicionário.
- **update(outro_dicionario):** atualiza o dicionário com os pares chave-valor de outro dicionário.
-

Exemplo:

```
pessoa = {"nome": "João", "idade": 25, "cidade": "Madri"}
print(pessoa.keys())   # Imprime dict_keys(["nome", "idade", "cidade"])
print(pessoa.values())  # Imprime dict_values(["João", 25, "Madri"])
print(pessoa.items())   # Imprime dict_items([("nome", "João"), ("idade", 25), ("cidade", "Madri")])
pessoa.update({"profissao": "Engenheiro"})
print(pessoa)  # Imprime {"nome": "João", "idade": 25, "cidade": "Madri", "profissao": "Engenheiro"}
```

Conjuntos (SET) – Lembra join do SQL ?

Um conjunto é uma estrutura de dados mutável e não ordenada que permite armazenar uma coleção de elementos únicos. Os conjuntos são delimitados por chaves {} ou são criados utilizando a função set().

Criação e operações básicas

Para criar um conjunto, utilize chaves ou a função set():

```
frutas = {"maçã", "banana", "laranja"}
numeros = set([1, 2, 3, 4, 5])
```

Os conjuntos suportam operações matemáticas de conjuntos, como a união (|), a interseção (&), a diferença (-) e a diferença simétrica (^).

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
uniao = conjunto1 | conjunto2
print(uniao)  # Imprime {1, 2, 3, 4, 5}
intersecao = conjunto1 & conjunto2
print(intersecao)  # Imprime {3}
diferenca = conjunto1 - conjunto2
print(diferenca)  # Imprime {1, 2}
diferenca_simetrica = conjunto1 ^ conjunto2
print(diferenca_simetrica)  # Imprime {1, 2, 4, 5}
```

Métodos de conjuntos

Os conjuntos em Python têm vários métodos incorporados para manipular e acessar os elementos. Alguns métodos comuns são:

- add(elemento): adiciona um elemento ao conjunto.
- remove(elemento): remove um elemento do conjunto. Se o elemento não existir, gera um erro.
- discard(elemento): remove um elemento do conjunto se estiver presente. Se o elemento não existir, não faz nada.
- clear(): remove todos os elementos do conjunto.

Exemplo:

```
frutas = {"maçã", "banana", "laranja"}
frutas.add("pera")
print(frutas)  # Imprime {"maçã", "banana", "laranja", "pera"}
frutas.remove("banana")
print(frutas)  # Imprime {"maçã", "laranja", "pera"}
frutas.discard("uva")
print(frutas)  # Imprime {"maçã", "laranja", "pera"}
frutas.clear()
print(frutas)  # Imprime set()
```

As estruturas de dados em Python nos oferecem grande flexibilidade e potência para armazenar e manipular dados em nossos programas. As listas são úteis para coleções ordenadas e mutáveis, as tuplas para coleções ordenadas e imutáveis, os dicionários para armazenar pares de chave valor e os conjuntos para coleções não ordenadas de elementos únicos.

Funções

As funções são blocos de código reutilizáveis que nos permitem encapsular tarefas específicas e executá-las quando necessário. As funções nos ajudam a organizar nosso código, evitar a repetição e fazer com que nossos programas sejam mais modulares e fáceis de manter.

Definição e chamada de funções
Para definir uma função em Python, utilizamos a palavra-chave def seguida do nome da função e parênteses. Opcionalmente, podemos especificar parâmetros dentro dos parênteses. O bloco de código da função é indentado após os dois pontos.

Para chamar uma função, simplesmente escrevemos o nome da função seguido de parênteses:

```
def saudacao():
  print("Olá, mundo!")
saudacao()  # Imprime "Olá, mundo!"
```

Parâmetros e argumentos
As funções podem aceitar parâmetros, que são valores que são passados para a função quando ela é chamada. Os parâmetros são especificados dentro dos parênteses na definição da função.

```
def saudacao(nome):
  print(f"Olá, {nome}!")
```

Ao chamar a função, fornecemos os argumentos correspondentes aos parâmetros:

```
saudacao("João")  # Imprime "Olá, João!"
saudacao("Maria")  # Imprime "Olá, Maria!"
```

Valores de retorno
As funções podem retornar valores usando a palavra-chave *return*. O valor de retorno pode ser usado pelo código que chama a função.

```
def soma(a, b):
  return a + b
resultado = soma(3, 4)
print(resultado)  # Imprime 7
```

Funções anônimas (lambda)

Python permite criar funções anônimas ou funções lambda, que são funções sem nome definidas em uma única linha. São comumente usadas para funções pequenas e concisas.

```
quadrado = lambda x: x ** 2
print(quadrado(5))  # Imprime 25
```

Escopo das variáveis (local vs. global)

As variáveis definidas dentro de uma função têm um escopo local, o que significa que só são acessíveis dentro da função. Por outro lado, as variáveis definidas fora de qualquer função têm um escopo global e podem ser acessadas de qualquer parte do programa.

```
def funcao():
    variavel_local = 10
    print(variavel_local)  # Acessível dentro da função
variavel_global = 20
def funcao2():
    print(variavel_global)  # Acessível de qualquer lugar

funcao()  # Imprime 10
funcao2()  # Imprime 20
print(variavel_global)  # Imprime 20
print(variavel_local)  # Gera um erro, a variável não está definida neste escopo.
```

- **Documentação de funções (docstrings)**

É uma boa prática documentar nossas funções utilizando docstrings. Os docstrings são cadeias de texto que descrevem o propósito, os parâmetros e o valor de retorno de uma função. São colocados imediatamente após a definição da função e são encerrados entre aspas duplas triplas.

```
def area_retangulo(base, altura):
    """
    Calcula a área de um retângulo.
    Args:
        base (float): A base do retângulo.
        altura (float): A altura do retângulo.
    Returns:
        float: A área do retângulo.
    """
    return base * altura
```

- **Funções com número variável de argumentos**

Python permite definir funções que aceitem um número variável de argumentos. Isso é feito utilizando o operador * antes do nome do parâmetro.

```
def soma_variavel(*numeros):
    total = 0
    for numero in numeros:
        total += numero
    return total
print(soma_variavel(1, 2, 3))  # Imprime 6
print(soma_variavel(4, 5, 6, 7))  # Imprime 22
```

As funções são uma ferramenta fundamental na programação e nos permitem estruturar e modularizar nosso código. Com a capacidade de definir funções personalizadas, podemos encapsular tarefas específicas e reutilizá-las em diferentes partes do nosso programa.

Além das funções definidas pelo usuário, Python também fornece uma ampla gama de funções incorporadas que podemos utilizar diretamente, como print(), len(), range(), entre outras.

Tratamento de erros e exceções

Quando escrevemos programas, é comum nos depararmos com situações inesperadas ou erros durante a execução. Python fornece um mecanismo para lidar com esses erros de maneira controlada utilizando o tratamento de exceções. Isso nos permite capturar e lidar com erros específicos sem que o programa pare abruptamente.

Erros comuns em Python
Antes de mergulharmos no tratamento de exceções, vejamos alguns erros comuns que você pode encontrar em Python

Erro de sintaxe (SyntaxError)
Ocorre quando o código não segue as regras de sintaxe do Python, como esquecer dois pontos após uma declaração de função ou um loop.

```
def minha_funcao() # Faltam os dois pontos
  print("Olá")
```

Erro de nome (NameError)
Ocorre quando se faz referência a uma variável ou função que não foi definida.

```
print(variavel_nao_definida)
```

Erro de tipo (TypeError)
Ocorre quando se realiza uma operação com tipos de dados incompatíveis, como tentar somar um número e uma string.

```
resultado = 5 + "10"
```

Erro de índice (IndexError)
Ocorre quando se tenta acessar um índice fora do intervalo válido de uma lista ou sequência.

```
lista = [1, 2, 3]
print(lista[3])  # O índice 3 está fora do intervalo
```

Estes são apenas alguns exemplos de erros comuns. Quando ocorre um erro, Python gera uma exceção e exibe uma mensagem de erro que inclui o tipo de exceção e uma descrição do problema.

Manejo de exceções

O manejo de exceções nos permite capturar e lidar com erros de maneira controlada utilizando as declarações try, except e opcionalmente finally.

Try
O bloco try contém o código que pode gerar uma exceção. Se ocorrer uma exceção dentro do bloco try, o fluxo de execução é transferido para o bloco except correspondente.

```
try:
    # Código que pode gerar uma exceção
    resultado = 10 / 0  # Divisão por zero
    print(resultado)
except ZeroDivisionError:
    print("Erro: Divisão por zero")
```

Except
O bloco except especifica o tipo de exceção que se deseja capturar e lidar. Você pode ter múltiplos blocos except para lidar com diferentes tipos de exceções.

```
try:
    # Código que pode gerar uma exceção
    resultado = 10 / 0  # Divisão por zero
    print(resultado)
except ZeroDivisionError:
    print("Erro: Divisão por zero")
except ValueError:
    print("Erro: Valor inválido")
```

Finally
O bloco finally é opcional e é executado sempre, independentemente de ter ocorrido uma exceção ou não. É comumente utilizado para realizar tarefas de limpeza ou liberação de recursos.

```
try:
    # Código que pode gerar uma exceção
    arquivo = open("arquivo.txt", "r")
    # Realizar operações com o arquivo
except FileNotFoundError:
    print("Erro: Arquivo não encontrado")
finally:
    arquivo.close()  # Fechar o arquivo sempre, mesmo se ocorrer uma exceção
```

Exceções personalizadas

Além das exceções incorporadas no Python, você também pode criar suas próprias exceções personalizadas. Isso é útil quando deseja lidar com situações específicas do seu programa.
Para criar uma exceção personalizada, você deve criar uma classe que herde da classe base Exception ou de uma de suas subclasses.

```
def funcao():
  # Código que pode gerar uma exceção personalizada
  if condicao:
    raise Exception("Descrição do erro")
try:
  funcao()
except Exception as e:
  print(f"Erro: {str(e)}")
```

Neste exemplo, define-se uma função chamada funcao(). Dentro da função, verifica-se uma condição e, se for satisfeita, gera-se uma exceção utilizando a declaração raise. Em vez de criar uma classe personalizada, utiliza-se diretamente a classe base Exception para gerar a exceção.

Depois, utiliza-se um bloco try-except para capturar e lidar com a exceção. A variável e é utilizada para acessar a descrição do erro fornecida ao gerar a exceção.
O tratamento de erros e exceções é uma parte fundamental da programação em Python. Permite lidar com situações inesperadas de maneira controlada e evitar que seu programa trave ou pare abruptamente.

Quando ocorre um erro no seu código, o Python gera uma exceção. Ao utilizar blocos try-except, você pode capturar e lidar com essas exceções de maneira adequada. Pode especificar diferentes blocos except para lidar com diferentes tipos de exceções e realizar ações específicas em cada caso.

Além disso, o bloco finally permite executar código de limpeza ou liberação de recursos, independentemente de ter ocorrido uma exceção ou não. Isso é útil para garantir que certas ações sejam sempre realizadas, como fechar arquivos ou conexões de banco de dados.

Considere os possíveis erros que podem ocorrer no seu código e utilize o tratamento de exceções adequado para lidar com eles de maneira apropriada. Isso tornará seus programas mais robustos e confiáveis.

Entradas/saídas (manual input)

Em Python, a entrada e saída de dados nos permite interagir com o usuário e manipular arquivos. Podemos solicitar informações ao usuário, mostrar resultados na tela e ler ou escrever dados em arquivos externos.

Entrada de dados do usuário

Para obter informações do usuário durante a execução do programa, podemos utilizar a função input(). Esta função mostra uma mensagem na tela e espera que o usuário insira um valor.

```
nome = input("Insira seu nome: ")
idade = input("Insira sua idade: ")
print("Olá, " + nome + "!")
print("Você tem " + idade + " anos.")
```

Neste exemplo, solicita-se ao usuário que insira seu nome e idade utilizando a função input(). Os valores inseridos são armazenados nas variáveis nome e idade, respectivamente. Em seguida, essas variáveis são utilizadas para mostrar uma saudação personalizada na tela.

A função input() sempre retorna uma cadeia de texto. Se você deseja trabalhar com outros tipos de dados, como números inteiros ou flutuantes, deve realizar uma conversão explícita utilizando funções como int() ou float().

```
idade = int(input("Insira sua idade: "))
if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

Neste exemplo, solicita-se ao usuário que insira sua idade e converte o valor inserido para um número inteiro utilizando int(). Em seguida, utiliza-se uma estrutura condicional para verificar se a idade é maior ou igual a 18 e mostrar uma mensagem correspondente.

Saída de dados

Para mostrar informações na tela, utilizamos a função print(). Esta função recebe um ou mais argumentos e os mostra no console.
Podemos utilizar a f-string (formatação de cadeias) para inserir variáveis diretamente dentro de uma cadeia de texto.

```
nome = "Juan"
idade = 25
print(f"Olá, meu nome é {nome} e tenho {idade} anos.")
```

Neste caso, as variáveis são inseridas dentro da cadeia utilizando chaves {} e a cadeia é precedida pela letra f para indicar que é uma f-string.

Leitura e escrita de arquivos

Python nos permite ler e escrever dados em arquivos externos. Podemos abrir arquivos em diferentes modos, como leitura ("r"), escrita ("w") ou anexar ("a"), e realizar operações de leitura e escrita.

Leitura de arquivos

Para ler o conteúdo de um arquivo, primeiro devemos abri-lo utilizando a função open() em modo de leitura ("r"). Depois, podemos ler o conteúdo do arquivo utilizando métodos como read() ou readlines().

```
arquivo = open("dados.txt", "r")
conteudo = arquivo.read()
print(conteudo)
arquivo.close()
```

Neste exemplo, o arquivo "dados.txt" é aberto em modo de leitura utilizando open(). Depois, todo o conteúdo do arquivo é lido utilizando o método read() e armazenado na variável conteudo. Finalmente, o conteúdo é mostrado na tela e o arquivo é fechado utilizando o método close().

Escrita de arquivos

Para escrever dados em um arquivo, abrimos em modo de escrita ("w") utilizando a função open(). Se o arquivo não existir, será criado automaticamente. Se o arquivo já existir, seu conteúdo será sobrescrito.

```
arquivo = open("dados.txt", "w")
arquivo.write("Olá, mundo!")
arquivo.close()
```

Neste exemplo, o arquivo "dados.txt" é aberto em modo de escrita utilizando open(). Depois, a string "Olá, mundo!" é escrita no arquivo utilizando o método write(). Finalmente, o arquivo é fechado utilizando o método close().

É importante fechar sempre os arquivos depois de utilizá-los para liberar os recursos do sistema.

Você também pode utilizar a declaração with para manejar a abertura e fechamento de arquivos de maneira automática.

```
with open("dados.txt", "r") as arquivo:
    conteudo = arquivo.read()
    print(conteudo)
```

Neste caso, o arquivo é aberto utilizando a declaração with e é fechado automaticamente uma vez que se sai do bloco with, mesmo se ocorrer uma exceção.

A entrada e saída de dados em Python nos oferece uma grande flexibilidade para interagir com o usuário e manipular arquivos externos. Podemos solicitar informações ao usuário, mostrar resultados na tela e ler ou escrever dados em arquivos de texto.

Lembre-se sempre de manejar adequadamente a abertura e fechamento de arquivos, e considerar as possíveis exceções que podem ocorrer durante as operações de entrada/saída.

Importação e criação de módulos

Em Python, um módulo é um arquivo que contém definições de funções, classes e variáveis que podem ser utilizadas em outros programas. A importação de módulos nos permite acessar a funcionalidade definida em outros arquivos e reutilizar código de maneira eficiente. Além disso, podemos criar nossos próprios módulos para organizar e modularizar nosso código.

Python vem com uma ampla biblioteca padrão de módulos que fornecem funcionalidades adicionais. Esses módulos estão disponíveis sem a necessidade de instalá-los separadamente.

Importar módulos

Para utilizar um módulo em nosso programa, devemos importá-lo utilizando a declaração import. Podemos importar um módulo completo ou funções específicas de um módulo.

```
import math
resultado = math.sqrt(25)
print(resultado)  # Imprime 5.0
```

Neste exemplo, importa-se o módulo math utilizando a declaração import. Em seguida, utiliza-se a função sqrt() do módulo math para calcular a raiz quadrada de 25.
Também podemos importar funções específicas de um módulo utilizando a sintaxe

```
from módulo import função.
from math import sqrt
resultado = sqrt(25)
print(resultado)  # Imprime 5.0
```

Neste caso, importa-se apenas a função sqrt() do módulo math, o que nos permite utilizá-la diretamente sem ter que precedê-la com o nome do módulo.

Funções e classes de módulos padrão

MATH
Fornece funções matemáticas, como sqrt() (raiz quadrada), sin() (seno), cos() (cosseno), entre outras.

RANDOM
Oferece funções para gerar números aleatórios, como random() (número aleatório entre 0 e 1), randint() (número inteiro aleatório em um intervalo), entre outras.

DATETIME
Permite trabalhar com datas e horas, como datetime.now() (data e hora atual), datetime.date() (data), datetime.time() (hora), entre outras.

```
import random
import datetime
numero_aleatorio = random.randint(1, 10)
print(numero_aleatório)  # Imprime um número inteiro aleatório entre 1 e 10
data_atual = datetime.datetime.now()
print(data_atual)  # Imprime a data e hora atual
```

Estes são apenas alguns exemplos dos muitos módulos disponíveis na biblioteca padrão de Python. Você pode consultar a documentação oficial de Python para obter mais informações sobre os módulos e suas funcionalidades.

Criação de módulos próprios

Além de utilizar os módulos padrão do Python, também podemos criar nossos próprios módulos para organizar e reutilizar nosso código.

Criar e utilizar módulos personalizados
Para criar um módulo personalizado, simplesmente criamos um novo arquivo Python com o nome desejado e definimos as funções, classes e variáveis que queremos incluir no módulo. Por exemplo, criamos um arquivo (no mesmo diretório onde estamos executando Python) chamado meu_modulo.py com o seguinte conteúdo:

```
#meu_modulo.py
def saudar(nome):
  print(f"Olá, {nome}!")
def calcular_soma(a, b):
  return a + b
```

Depois, podemos importar e utilizar as funções definidas em meu_modulo.py em outro arquivo Python.

```
import meu_modulo
meu_modulo.saudar("João")  # Imprime "Olá, João!"
resultado = meu_modulo.calcular_soma(5, 3)
print(resultado)  # Imprime 8
```

Neste exemplo, importa-se o módulo meu_modulo e utilizam-se as funções saudar() e calcular_soma() definidas nele.

Organização do código em módulos

À medida que nossos programas crescem em tamanho e complexidade, é uma boa prática organizar nosso código em módulos separados segundo sua funcionalidade. Isso nos permite manter um código mais legível, agrupado em módulos e fácil de manter.
Por exemplo, podemos ter um módulo operacoes.py que contenha funções relacionadas com operações matemáticas, e outro módulo utilidades.py que contenha funções de uso geral.

```
# operacoes.py
def somar(a, b):
  return a + b
def subtrair(a, b):
  return a - b

# utilidades.py
def imprimir_mensagem(mensagem):
  print(mensagem)

def obter_nome_usuario():
  return input("Digite seu nome: ")
```

Depois, podemos importar e utilizar essas funções em nosso programa principal.

```
import operacoes
import utilidades

resultado = operacoes.somar(5, 3)
utilidades.imprimir_mensagem(f"O resultado da soma é: {resultado}")

nome = utilidades.obter_nome_usuario()
utilidades.imprimir_mensagem(f"Olá, {nome}!")
```

Ao organizar nosso código em módulos, podemos reutilizar funções e manter um código mais estruturado e agrupado em módulos.

Pacotes

Um pacote é uma forma de organizar módulos relacionados em uma estrutura hierárquica de diretórios. Os pacotes nos permitem agrupar módulos relacionados e evitar conflitos de nomes entre módulos.

Criar e utilizar pacotes
Para criar um pacote, criamos um diretório com o nome desejado e adicionamos um arquivo especial chamado __init__.py dentro do diretório. Este arquivo pode estar vazio ou conter código de inicialização do pacote.

Por exemplo, criamos um diretório chamado meu_pacote com a seguinte estrutura:

```
meu_pacote/
  __init__.py
  modulo1.py
  modulo2.py
```

Depois, podemos importar e utilizar os módulos do pacote em nosso programa.

```
from meu_pacote import modulo1, modulo2
modulo1.funcao1()
modulo2.funcao2()
```

Neste exemplo, são importados os módulos modulo1 e modulo2 do pacote meu_pacote e são utilizadas as funções definidas neles.

A importação e criação de módulos e pacotes em Python nos permite organizar e reutilizar nosso código de maneira eficiente. Ao modularizar nosso código, podemos manter um código mais legível, estruturado e fácil de manter.

Lembre-se de explorar a biblioteca padrão de Python e aproveitar os módulos existentes, que podem facilitar muitas tarefas comuns. Além disso, não hesite em criar seus próprios módulos e pacotes para organizar e reutilizar seu código de maneira eficaz.