Resumo do Python PEP 8

O PEP 8 é um guia de estilo para a escrita de código em Python. Ele fornece diretrizes e recomendações sobre como escrever código Python legível e consistente. Aqui está um resumo didático e objetivo do PEP 8:

Indentação: Use 4 espaços para cada nível de indentação. Não use tabulações para indentação.

Comprimento das Linhas: Limite as linhas a 79 caracteres para facilitar a leitura e a visualização em terminais comuns. Se a linha for muito longa, quebre-a de forma a manter a legibilidade.

Espaços em Branco:

- Use espaços em torno de operadores e após vírgulas.
- Evite espaços em branco extras no final das linhas.
- Use uma linha em branco para separar funções e classes e dentro de funções para separar partes lógicas do código.

Nomeação:

- Use nomes descritivos para variáveis, funções e classes.
- Use snake_case para nomes de funções, variáveis e métodos.
- Use CapWords para nomes de classes.

Comentários:

- Escreva comentários para explicar partes complicadas do código.
- Mantenha os comentários atualizados conforme o código é modificado.

Docstrings: Escreva docstrings para módulos, funções, classes e métodos públicos.

Importações:

- Agrupe as importações em três seções: importações de bibliotecas padrão, importações de bibliotecas de terceiros e importações de módulos locais.
- Evite importações absolutas.

Expressões e Instruções:

- Prefira **`if foo`** sobre **`if foo == True`**.
- Prefira **`if not foo`** sobre **`if foo == False`**.
- Use a forma de expressão if/else para atribuições curtas, ao invés de usar a declaração if/else.

Tratamento de Exceções: Utilize a cláusula **`try`** e **`except`** de forma apropriada para lidar com exceções.

Convenções:

- Siga as convenções locais quando não entrar em conflito com o PEP 8.
- Evite complexidade excessiva.
- Seja consistente.

Seguir as diretrizes do PEP 8 ajuda a garantir que o código Python seja claro, legível e consistente, facilitando a colaboração com outros desenvolvedores e a manutenção do código ao longo do tempo.

Observações:

1. Importação Absoluta:

Uma importação absoluta é uma declaração que importa um módulo usando o caminho absoluto do módulo em relação ao diretório raiz do projeto. Por exemplo, se você tiver um pacote chamado **`my_package`** e dentro dele um módulo chamado **`my_module`**, uma importação absoluta desse módulo seria:

```
from my_package import my_module
```

Isso significa que o interpretador Python irá procurar pelo pacote **`my_package`** e depois pelo módulo **`my_module`** dentro dele.

2. Docstring:

Uma docstring é uma string de documentação que aparece logo após a definição de uma função, classe, módulo ou método em Python. Ela fornece uma descrição do que a função, classe ou módulo faz, bem como informações sobre os parâmetros, valores de retorno e outros detalhes relevantes. As docstrings são acessíveis por meio do atributo especial **`__doc__`** do objeto ao qual estão associadas.

Veja este exemplo:

```python
def minha_funcao(parametro):
    """
    Esta é uma docstring.

    Descreve o que esta função faz e quais parâmetros ela espera.
    Retorna o resultado calculado.
    """
    # Corpo da função
    return resultado
```

A docstring deve ser incluída logo após a definição de uma função, método, classe ou módulo, e deve seguir algumas convenções de formatação para garantir que seja fácil de ler e entender.

3. Expressão if/else para atribuições curtas

Esta sugestão pode tornar o código mais conciso e legível, no uso da forma de expressão if/else em atribuições curtas:

```python
# Exemplo com a declaração if/else
x = 10
if x > 5:
    y = 'Maior que 5'
else:
    y = 'Menor ou igual a 5'

print(y)  # Saída: Maior que 5
```

Agora, o mesmo exemplo usando a forma de expressão if/else para atribuições curtas:

```python
# Exemplo com a forma de expressão if/else para atribuições curtas
x = 10
y = 'Maior que 5' if x > 5 else 'Menor ou igual a 5'

print(y)  # Saída: Maior que 5
```

Neste exemplo, a variável **y** é atribuída com o valor **'Maior que 5'** se **x** for maior que 5, caso contrário, é atribuído o valor **'Menor ou igual a 5'**. Essa atribuição é feita em uma única

linha, o que pode tornar o código mais conciso e legível, especialmente para atribuições simples.

4. Tratamento de exceções

Suponha que você tenha uma função que divide dois números e gostaria de lidar com a exceção **ZeroDivisionError** caso o divisor seja zero:

```
def divide_numeros(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        print("Resultado da divisão:", resultado)
    except ZeroDivisionError:
        print("Erro: Divisão por zero não é permitida.")

# Exemplo de uso
divide_numeros(10, 2)  # Saída: Resultado da divisão: 5.0
divide_numeros(10, 0)  # Saída: Erro: Divisão por zero não é permitida.
```

Neste exemplo, a função **divide_numeros()** tenta dividir o **dividendo** pelo **divisor** dentro de um bloco **try**. Se a divisão for bem-sucedida, o resultado é impresso. Se ocorrer uma exceção do tipo **ZeroDivisionError**, o bloco **except** será executado, e uma mensagem de erro será impressa. Isso permite que o programa lide com a exceção de forma adequada e continue executando sem ser interrompido pelo erro.