Tratar exceções

5 minutos

Quando você se depara pela primeira vez com exceções que mostram rastreamentos grandes como saída, pode ficar tentado a capturar todos os erros a fim de impedir que isso aconteça.

Se você estiver em uma missão para Marte, o que poderá fazer se uma mensagem de texto no sistema de navegação informar "*ocorreu um erro*"? Imagine que não há nenhuma outra informação ou contexto, apenas uma luz vermelha piscando com essa mensagem de erro. Como desenvolvedor, é útil colocar-se do outro lado do programa: o que o usuário pode fazer quando ocorre um erro?

Embora este módulo aborde como tratar exceções capturando-as, não é necessário capturar exceções o tempo todo. Às vezes, é útil permitir que exceções sejam geradas para que os outros chamadores possam lidar com os erros.

Blocos try e except

Vamos usar o exemplo de navegação para criar código que abre arquivos de configuração para a missão de Marte. Os arquivos de configuração podem ter todos os tipos de problemas, portanto, é essencial relatar os problemas com precisão quando eles surgirem. Sabemos que, se um arquivo ou diretório não existir, será gerado `FileNotFoundError`. Se quisermos lidar com essa exceção, podemos fazer isso com um bloco `try` e `except`:

Python

```
try:
     open('config.txt')
except FileNotFoundError:
     print("Couldn't find the config.txt file!")
```

Output

```
Couldn't find the config.txt file!
```

Após a palavra-chave `try`, você adiciona o código que tem o potencial de causar uma exceção. Em seguida, adicione a palavra-chave `except` junto com a possível exceção, seguida

por qualquer código que precise ser executado quando essa condição ocorrer. Como *config.txt* não existe no sistema, o Python imprime que o arquivo de configuração não está lá. O bloco `try` e `except`, juntamente com uma mensagem útil, impede a geração de um rastreamento e, mesmo assim, não deixa de informar ao usuário sobre o problema.

Embora um arquivo que não exista seja comum, esse não é o único erro que você pode encontrar. Permissões de arquivo inválidas podem impedir a leitura de um arquivo, mesmo que ele exista. Vamos criar um arquivo Python chamado *config.py* no Visual Studio Code. Adicione o seguinte código ao arquivo que localiza e lê o arquivo de configuração do sistema de navegação:

Python

```
def main():
    try:
        configuration = open('config.txt')
    except FileNotFoundError:
        print("Couldn't find the config.txt file!")


if __name__ == '__main__':
    main()
```

Em seguida, crie um *diretório* chamado *config.txt*. Tente chamar o arquivo *config.py* para ver um novo erro que deve ser semelhante a este:

Bash

```
python3 config.py
```

Output

```
Traceback (most recent call last):
  File "/tmp/config.py", line 9, in <module>
    main()
  File "/tmp/config.py", line 3, in main
    configuration = open('config.txt')
IsADirectoryError: [Errno 21] Is a directory: 'config.txt'
```

Uma forma pouco útil de lidar com esse erro seria capturar todas as exceções possíveis para evitar um rastreamento. Para entender por que a captura de todas as exceções é problemática, experimente atualizar a função `main()` no arquivo recém-criado *config.py*:

Python

```
def main():
    try:
        configuration = open('config.txt')
    except Exception:
        print("Couldn't find the config.txt file!")
```

Agora, execute o código novamente no mesmo local em que o arquivo *config.txt* existe com permissões incorretas:

Bash

```
python3 config.py
```

Output

```
Couldn't find the config.txt file!
```

O problema agora é que a mensagem de erro está incorreta. O diretório existe, mas tem permissões diferentes e o Python não pode lê-lo. Quando você está lidando com erros de software, pode ser muito frustrante ter erros que:

- Não indicam o real problema.
- Forneçam saídas que não correspondam ao real problema.
- Não indiquem o que pode ser feito para corrigir o problema.

Vamos corrigir esse trecho de código para abordar todas essas frustrações. Reverta para a captura `FileNotFoundError` e adicione outro bloco `except` para capturar `PermissionError`:

Python

```
def main():
    try:
        configuration = open('config.txt')
    except FileNotFoundError:
        print("Couldn't find the config.txt file!")
    except IsADirectoryError:
        print("Found config.txt but it is a directory, couldn't read it")
```

Agora execute-o novamente, no mesmo local em que o diretório *config.txt* está:

Bash

```
python3 config.py
```

Output

```
Found config.txt but couldn't read it
```

Agora, exclua o arquivo *config.txt* para garantir que o primeiro bloco `except` seja alcançado:

Bash

```
rm -f config.txt
python3 config.py
```

Output

```
Couldn't find the config.txt file!
```

Quando os erros são de natureza semelhante e não há necessidade de tratá-los individualmente, você pode agrupar as exceções usando parênteses na linha `except`. Por exemplo, se o sistema de navegação estiver sob cargas pesadas e o sistema de arquivos ficar muito ocupado, faz sentido capturar `BlockingIOError` e `TimeOutError` juntas:

Python

```
def main():
    try:
        configuration = open('config.txt')
    except FileNotFoundError:
        print("Couldn't find the config.txt file!")
    except IsADirectoryError:
        print("Found config.txt but it is a directory, couldn't read it")
    except (BlockingIOError, TimeoutError):
        print("Filesystem under heavy load, can't complete reading configuration
file")
```

Dica

Mesmo que você possa agrupar exceções, faça isso somente quando não houver necessidade de tratá-las individualmente. Evite agrupar muitas exceções e fornecer mensagens de erro muito generalizadas.

Se você precisar acessar o erro associado à exceção, deverá atualizar a linha `except` para incluir a palavra-chave `as`. Essa técnica será útil se uma exceção for muito genérica e a mensagem de erro puder ser útil:

Python

```
try:
    open("mars.jpg")
except FileNotFoundError as err:
     print("Got a problem trying to read the file:", err)
```

Output

```
Got a problem trying to read the file: [Errno 2] No such file or directory:
'mars.jpg'
```

Nesse caso, `as err` significa que `err` se torna uma variável tendo o objeto da exceção como valor. Em seguida, ele usa esse valor para imprimir a mensagem de erro associada à exceção. Outro motivo para usar essa técnica é acessar os atributos do erro diretamente. Por exemplo, se estiver capturando uma exceção `OSError` mais genérica, que é a *exceção pai* de `FilenotFoundError` e `PermissionError`, você poderá diferenciá-las pelo atributo `.errno`:

Python

```
try:
    open("config.txt")
except OSError as err:
     if err.errno == 2:
         print("Couldn't find the config.txt file!")
     elif err.errno == 13:
        print("Found config.txt but couldn't read it")
```

Output

```
Couldn't find the config.txt file!
```

Sempre tente usar a técnica que fornece a melhor legibilidade possível para o código e ajuda na manutenção futura dele. Às vezes, é necessário usar código menos legível para oferecer uma experiência melhor ao usuário em caso de erro.

Unidade seguinte: Exercício – Tratar exceções

Continuar >