CAPÍTULO 08

8. Erros e exceções

Até agora mensagens de erro foram apenas mencionadas, mas se você testou os exemplos, talvez tenha esbarrado em algumas. Existem pelo menos dois tipos distintos de erros: *erros de sintaxe* e *exceções*.

8.1. Erros de sintaxe

Erros de sintaxe, também conhecidos como erros de parse, são provavelmente os mais frequentes entre aqueles que ainda estão aprendendo Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

O parser repete a linha inválida e apresenta uma pequena 'seta' apontando para o ponto da linha em que o erro foi detectado. O erro é causado (ou ao menos detectado) pelo símbolo que *precede* a seta: no exemplo, o erro foi detectado na função `print()`, uma vez que um dois-pontos (`':'`) está faltando antes dela. O nome de arquivo e número de linha são exibidos para que você possa rastrear o erro no texto do script.

8.2. Exceções

Mesmo que um comando ou expressão estejam sintaticamente corretos, talvez ocorra um erro na hora de sua execução. Erros detectados durante a execução são chamados *exceções* e não são necessariamente fatais: logo veremos como tratá-las em programas Python. A maioria das exceções não são tratadas pelos programas e acabam resultando em mensagens de erro:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

A última linha da mensagem de erro indica o que aconteceu. Exceções surgem com diferentes tipos, e o tipo é exibido como parte da mensagem: os tipos no exemplo são `ZeroDivisionError`, `NameError` e `TypeError`. A string exibida como sendo o tipo da exceção é o nome da exceção embutida que ocorreu. Isso é verdade para todas exceções pré-definidas em Python, mas não é necessariamente verdade para exceções definidas pelo usuário (embora seja uma convenção útil). Os nomes das exceções padrões são identificadores embutidos (não palavras reservadas).

O resto da linha é um detalhamento que depende do tipo da exceção ocorrida e sua causa.

A parte anterior da mensagem de erro apresenta o contexto onde ocorreu a exceção. Essa informação é denominada *stack traceback* (situação da pilha de execução). Em geral, contém uma lista de linhas do código-fonte, sem apresentar, no entanto, linhas lidas da entrada padrão.

Exceções embutidas lista as exceções pré-definidas e seus significados.

8.3. Tratamento de exceções

É possível escrever programas que tratam exceções específicas. Observe o exemplo seguinte, que pede dados ao usuário até que um inteiro válido seja fornecido, ainda permitindo que o programa seja interrompido (utilizando `Control-C` ou seja lá o que for que o sistema operacional suporte); note que uma interrupção gerada pelo usuário será sinalizada pela exceção `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...             print("Oops!  That was no valid number.  Try
again...")
...
```

A instrução `try` funciona da seguinte maneira:

- Primeiramente, a *cláusula try* (o conjunto de instruções entre as palavras reservadas `try` e `except`) é executada.
- Se nenhuma exceção ocorrer, a *cláusula except* é ignorada e a execução da instrução `try` é finalizada.
- Se ocorrer uma exceção durante a execução de uma cláusura `try`, as instruções remanescentes na cláusula são ignoradas. Se o tipo da exceção ocorrida tiver sido previsto em algum `except`, essa *cláusura except* é executada, e então depois a execução continua após o bloco try/except.

- Se a exceção levantada não corresponder a nenhuma exceção listada na *cláusula de exceção*, então ela é entregue a uma instrução `try` mais externa. Se não existir nenhum tratador previsto para tal exceção, trata-se de uma *exceção não tratada* e a execução do programa termina com uma mensagem de erro.

A instrução `try` pode ter uma ou mais *cláusula de exceção*, para especificar múltiplos tratadores para diferentes exceções. No máximo um único tratador será executado. Tratadores só são sensíveis às exceções levantadas no interior da *cláusula de tentativa*, e não às que tenham ocorrido no interior de outro tratador numa mesma instrução `try`. Uma *cláusula de exceção* pode ser sensível a múltiplas exceções, desde que as especifique em uma tupla, por exemplo:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Uma classe em uma cláusula `except` é compatível com uma exceção se ela é da mesma classe ou de uma classe base desta (mas o contrário não é válido — uma *cláusula de exceção* listando uma classe derivada não é compatível com uma classe base). Por exemplo, o seguinte código irá mostrar B, C, D nesta ordem:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Se a ordem das *cláusulas de exceção* fosse invertida (`except B` no início), seria exibido B, B, B — somente a primeira *cláusula de exceção* compatível é ativada.

Todas as exceções herdam de `BaseException`, e por isso pode ser usado para servir como um curinga. Utilize esse recurso com extrema cautela, uma vez que

isso pode esconder erros do programador e do usuário! Também pode ser utilizado para exibir uma mensagem de erro e então levantar novamente a exceção (permitindo que o invocador da função atual também possa tratá-la):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except BaseException as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

Alternativamente, a última cláusula except pode omitir o(s) nome(s) de exceção, no entanto, o valor da exceção deve ser obtido de `sys.exc_info()[1]`.

A construção `try` … `except` possui uma *cláusula else* opcional, que quando presente, deve ser colocada depois de todas as outras cláusulas. É útil para um código que precisa ser executado se nenhuma exceção foi levantada. Por exemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

É melhor usar a cláusula `else` do que adicionar código adicional à cláusula `try` porque ela evita que acidentalmente seja tratada uma exceção que não foi levantada pelo código protegido pela construção com as instruções `try` … `except`.

Quando uma exceção ocorre, ela pode estar associada a um valor chamado *argumento* da exceção. A presença e o tipo do argumento dependem do tipo da exceção.

A *cláusula de exceção* pode especificar uma variável depois do nome da exceção. A variável é associada à instância de exceção capturada, com os argumentos armazenados em `instance.args`. Por conveniência, a instância

define o método `__str__()` para que os argumentos possam ser exibidos diretamente sem necessidade de acessar `.args`. Pode-se também instanciar uma exceção antes de levantá-la e adicionar qualquer atributo a ela, conforme desejado.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...    print(inst)          # __str__ allows args to be printed
directly,
...                         # but may be overridden in exception
subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Caso uma exceção tenha argumentos, os mesmos serão impressos como a última parte (‘detalhe’) da mensagem para as exceções não tratadas.

Além disso, tratadores de exceção são capazes de capturar exceções que tenham sido levantadas no interior de funções invocadas (mesmo que indiretamente) na *cláusula try*. Por exemplo:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4. Levantando exceções

A instrução `raise` permite ao programador forçar a ocorrência de um determinado tipo de exceção. Por exemplo:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

O argumento de `raise` indica a exceção a ser levantada. Esse argumento deve ser uma instância de exceção ou uma classe de exceção (uma classe que deriva de `Exception`). Se uma classe de exceção for passada, será implicitamente instanciada invocando o seu construtor sem argumentos:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

Caso você precise determinar se uma exceção foi levantada ou não, mas não quer manipular o erro, uma forma simples de instrução `raise` permite que você levante-a novamente:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5. Encadeamento de exceções

A instrução `raise` permite um `from` opcional que torna possível o encadear exceções. Por exemplo:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

Isso pode ser útil quando você está transformando exceções. Por exemplo:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
```

```
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

O encadeamento de exceções acontece automaticamente quando uma exceção é levantada dentro de uma seção `except` ou `finally`. O encadeamento de exceções pode ser desativado usando o instrução `from None`:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

Para mais informações sobre os mecanismos de encadeamento, veja Exceções embutidas.

8.6. Exceções definidas pelo usuário

Programas podem definir novos tipos de exceções, através da criação de uma nova classe (veja Classes para mais informações sobre classes Python). Exceções devem ser derivadas da classe `Exception`, direta ou indiretamente.

Classes de exceções podem ser definidas para fazer qualquer coisa que qualquer outra classe faz, mas em geral são bem simples, frequentemente oferecendo apenas alguns atributos que fornecem informações sobre o erro que ocorreu. Ao criar um módulo que pode gerar diversos erros, uma prática comum é criar uma classe base para as exceções definidas por aquele módulo, e as classes específicas para cada condição de erro como subclasses dela:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass
```

```
class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error
occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition
that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition
is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

É comum que novas exceções sejam definidas com nomes terminando em "Error", semelhante a muitas exceções embutidas.

Muitos módulos padrão definem novas exceções para reportar erros que ocorrem no interior das funções que definem. Mais informações sobre classes aparecem no capítulo Classes.

8.7. Definindo ações de limpeza

A instrução `try` possui outra cláusula opcional, cuja finalidade é permitir a implementação de ações de limpeza, que sempre devem ser executadas independentemente da ocorrência de exceções. Como no exemplo:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
```

```
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Se uma cláusula `finally` estiver presente, a cláusula `finally` será executada como a última tarefa antes da conclusão da instrução `try`. A cláusula `finally` executa se a instrução `try` produz uma exceção. Os pontos a seguir discutem casos mais complexos quando ocorre uma exceção:

- Se ocorrer uma exceção durante a execução da cláusula `try`, a exceção poderá ser tratada por uma cláusula `except`. Se a exceção não for tratada por uma cláusula `except`, a exceção será gerada novamente após a execução da cláusula: keyword:*!finally*.
- Uma exceção pode ocorrer durante a execução de uma cláusula `except` ou `else`. Novamente, a exceção é re-levantada depois que `finally` é executada.
- Se a cláusula `finally` executa uma instrução `break`, `continue` ou `return`, as exceções não são levantadas novamente.
- Se a instrução `try` atingir uma instrução `break`, `continue` ou `return`, a cláusula `finally` será executada imediatamente antes da execução da instrução `break`, `continue` ou `return`.
- Se uma cláusula `finally` incluir uma instrução `return`, o valor retornado será aquele da instrução `return` da cláusula `finally`, não o valor da instrução `return` da cláusula `try`.

Por exemplo:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Um exemplo mais complicado:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
```

```
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como você pode ver, a cláusula `finally` é executada em todos os casos. A exceção `TypeError` levantada pela divisão de duas strings não é tratada pela cláusula `except` e portanto é re-levantada depois que a cláusula `finally` é executada.

Em aplicação do mundo real, a cláusula `finally` é útil para liberar recursos externos (como arquivos ou conexões de rede), independentemente do uso do recurso ter sido bem sucedido ou não.

8.8. Ações de limpeza predefinidas

Alguns objetos definem ações de limpeza padrões para serem executadas quando o objeto não é mais necessário, independentemente da operação que estava usando o objeto ter sido ou não bem sucedida. Veja o exemplo a seguir, que tenta abrir um arquivo e exibir seu conteúdo na tela.

```
for line in open("myfile.txt"):
    print(line, end="")
```

O problema com esse código é que ele deixa o arquivo aberto um período indeterminado depois que o código é executado. Isso não chega a ser problema em scripts simples, mas pode ser um problema para grandes aplicações. A palavra reservada `with` permite que objetos como arquivos sejam utilizados com a certeza de que sempre serão prontamente e corretamente finalizados.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```