tratamento_de_excecoes

March 22, 2022

1 Tratamento de exceções

Pedro Cardoso

(ISE/UAlg - pcardoso@ualg.pt)

Até agora mensagens de erro foram apenas mencionadas mas já tivemos alguns exemplo. Existem pelo menos dois tipos distintos de erros: erros de sintaxe e exceções.

1.1 Erros de sintaxe

Erros de sintaxe, correpondem a uma codicação que não segue as regras de sintax do Python:

[ ]:
```
while True
    print('Hello world')
```

O *parser* repete a linha inválida e apresenta uma pequena 'seta' apontando para o ponto da linha em que o erro foi detectado. Mas, possivelemente mais importante, o nome do ficheiro e número de linha são exibidos para que possa rastrear o erro no texto do script.

1.2 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**: já veremos como tratá-las em programas Python.

A maioria das exceções não são tratadas pelos programas e acabam resultando em mensagens de erro

[ ]:
```
1/0
```

[ ]:
```
spam + 3
```

[ ]:
```
"spam" + 3
```

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`.

[ ]:
```
def f1():
    print(x+y)
```

```
def f0():
    f1()

f0()
```

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). No exemplo, podemos concluir que a chamada foi feita inicialmente o método `f0`, que depois chamou o método `f1`, onde ocorreu a erro e foi levanta a exceção.

1.3 Tratamento de exceções

É possível escrever programas que tratam exceções específicas.

No exemplo seguinte pede-se dados ao utilizador um inteiro válido. Tente escrever o seu nome… o que acontece?

[ ]:
```
while True:
   x = int(input("Qual a sua idade: "))
```

Para prevenir podemos então criar um bloco que proteja o código

[ ]:
```
while True:
    try:
        x = int(input("Qual a sua idade: "))
        break
    except ValueError:
        print("Oops!  Isso não é um número inteiro.  Tente de novo...")

print(x)
```

A instrução `try` funciona da seguinte maneira: 1. se tudo correr bem, o bloco de instruções entre as palavras reservadas `try` e `except` é executada. 1. Se nenhuma exceção ocorrer, o bloco do `except` é ignorado e a execução do bloco do `try` é finalizada. 1. Se ocorrer uma exceção durante a execução da cláusula `try`, as instruções remanescentes na cláusula são ignoradas. E.g., uma exceção na linha `x = int(input("Qual a sua idade: "))` faz com que não seja executado comando `break` 1. Se o tipo da exceção ocorrida tiver sido previsto em algum `except`, então essa cláusula será executada. Depois disso, a execução continua após a instrução `try... except...`. 1. Se a exceção levantada não corresponder a nenhuma exceção listada na cláusula de exceção, então 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. 1. 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. Um tratador pode ser sensível a múltiplas exceções, desde que sejam especificadas num tuplo.

1.4 Exemplo

o código que se segue não está protegido para a ocorrência de exceções

[ ]:
```
f = open('myfile.txt') # o nome do ficheiro é my_file.txt
s = f.readline()
i = int(s.strip())
```

Neste caso podemos fazer o seguinte: 1. sem que o ficheiro myfile.txt exista 2. Crie o ficheiro `my_file.txt` 1. preencha-o com a sua idade e corra a célula 1. preencha-o com o seu nome e corra a célula 1. mude o nome do ficheiro para `meu_fich.txt` e corra a célula

[ ]:
```
import sys

try:
    f = open('my_file.txt')
    s = f.readline()
    i = int(s.strip())
    print("inserido o número intereiro", i)
except FileNotFoundError as err:
    print("FileNotFoundError : Erro!: {0}".format(err))
except ValueError:
    print(f'ValueError: Não consigo converter "{s}" para inteiro.')
except:
    print("Outro erro: Unexpected error:", sys.exc_info()[0])
```

1.5 Clausula `else`

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()
```

1.6 Argumentos da exceção

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 `except` pode especificar uma variável depois do nome (ou da tupla de nomes) da exceção, `e`. A variável é associada à instância de exceção capturada, com os argumentos armazenados em 'e.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.

[ ]:
```
try:
    raise Exception('spam', 'eggs')
except Exception as e:
    print(type(e))    # a instancia da excexao
    print(e.args)     # argumentos armazenados em ".args"
    print(e)          # __str__ está implementado...

    x, y = e.args     # unpack args
    print('x =', x)
    print('y =', y)
```

1.7 Levantando exceções

A instrução raise permite ao programador forçar a ocorrência de um determinado tipo de exceção.

[ ]:
```
try:
    raise NameError('Ola...!')
except NameError:
    print('Passou por aqui uma excecao!')
```

1.8 Finalmente o `finally`!

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**.

Se uma cláusula `finally` estiver presente, esta será executada como a última tarefa antes da conclusão da instrução `try`.

[ ]:
```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("erro: divisão por zero!")
    else:
        print("o resultado é", result)
        return result
    finally:
        print("... e agora, independentemente do que aconteça, cá vou eu!!")
```

[ ]:
```
x = divide(1, 2)
x
```

[ ]:
```
divide(1, 0)
```

1.8.1 O exemplo dos ficheiros

Não nos devemos esquecer de fechar os "`open`" que vamos fazendo, logo podemos/devemos colocar o clode no bloco `finally`

[ ]:
```
try:
    f = open('my_file.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("Erro!: {0}".format(err))
except : # não e boa politica mas e melhor que nada ;)
    print(f'Não consigo converter "{s}" para inteiro.')
finally:
    # fecha o ficheiro, independentemente do que aconteça
    print("f está fechado?:", f.closed)
    f.close()

print("f está fechado?:", f.closed)
```

No caso dos ficheiros, como alternativa, podemos garantir que o ficheiro é fechado usando o `with`

[ ]:
```
with open('my_file.txt') as f:
    try:
        s = f.readline()
        i = int(s.strip())
    except OSError as err:
        print("Erro!: {0}".format(err))
    except : # não e boa politica mas e melhor que nada ;)
        print(f'Não consigo converter "{s}" para inteiro.')

print("f está fechado?:", f.closed)
```

1.9 Exceções definidas pelo programador

Programas podem definir novos tipos de exceções, através da criação de uma nova classe. Exceções devem ser derivadas da classe `Exception`, direta ou indiretamente (todas as exceções devem ser instâncias de uma classe derivada de BaseException).

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
```

1.10 Exercício

Implemente uma classe Data

[ ]:
```
class Data:
    """ classe que implementa o armazenamento de uma data.
    Se for instanciada com valores invalidos levanta uma excecao"""


    def __init__(self, a, m, d):
        Data.valida(a, m, d) # levanta excecao em caso de erro
        self.a, self.m, self.d = a, m, d

    def __repr__(self):
        return f'{self.a}/{self.m}/{self.d}'

    @staticmethod
    def valida(a, m, d):
        ''' devolve True se a data e a/m/d valida. Caso contrario levante uma␣
 ↪excecao adequada
```

```
        :raises:
            TypeError: se os valores a, m ou d nao forem inteiros, indicando␣
↪qual
            AnoInvalido: Nao existe ano 0 (xxx excecao a ser implementada pelo␣
↪aluno xxxx)
            MesInvalido: se m < 1 ou m > 12
            DiaInvalido: o mes m nao tem dia d
        '''
        raise Exception("To-do")

d = Data("2021", 4, 6)
```

[ ]:
```
d = Data(2021, 4, 6)
d
```

[ ]:
```
d = Data(2021, 2, 30)
d
```

1.11 conjunto do exceções pre-definidas (*builtin*)

https://docs.python.org/pt-br/3/library/exceptions.html#bltin-exceptions

1.12 A clausula `assert`

Em computação, asserção (em inglês: assertion) é um predicado que é inserido no programa para verificar uma condição que o desenvolvedor supõe que seja verdadeira em determinado ponto.

[ ]:
```
a = 1
assert isinstance(a, int)
print("ok")
```

[ ]:
```
a = 1.0
assert isinstance(a, int)
print("ok")
```

[ ]:
```
idade_filho = 18
idade_mae = 4
assert idade_filho < idade_mae, "erro: idades invalidas"
```

[ ]:
```
try:
    idade_filho = 18
    idade_mae = 4
    assert idade_mae > idade_filho
except AssertionError as e:
    print("erro: idades invalidas")
```

[ ]: