tratamento_de_excecoes

March 3, 2020

1 Programação Orientada aos Objetos (POO) - parte I

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:

```
In [4]: while True print('Hello world')


          File "<ipython-input-4-2b688bc740d7>", 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. 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: logo veremos como tratá-las em programas Python.

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

```
In [5]: 1/0

---------------------------------------------------------------------------
```

```
        ZeroDivisionError                         Traceback (most recent call last)

        <ipython-input-5-9e1622b385b6> in <module>()
    ----> 1 1/0


        ZeroDivisionError: division by zero
```

```
In [6]: spam + 3
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-6-28973131a9bb> in <module>()
    ----> 1 spam + 3


        NameError: name 'spam' is not defined
```

```
In [7]: "spam" + 3
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-7-7457e73eb5f9> in <module>()
    ----> 1 "spam" + 3


        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 parte anterior da mensagem de erro apresenta o contexto onde ocorreu a exceção. Essa informação é denominada *stack traceback* (situação da pilha de execução).

1.3 Tratamento de exceções

É possível escrever programas que tratam exceções específicas.

No exemplo seguinte pede-se dados ao utilizador um inteiro válido.

```
In [8]: while True:
            try:
                x = int(input("Qual a sua idade: "))
                break
            except ValueError:
                print("Oops!  Isso não é um número.  Tente de novo...")

        print(x)
```

```
Qual a sua idade:
Oops!  Isso não é um número.  Tente de novo...
Qual a sua idade:
Oops!  Isso não é um número.  Tente de novo...
Qual a sua idade:
Oops!  Isso não é um número.  Tente de novo...
Qual a sua idade:
Oops!  Isso não é um número.  Tente de novo...
Qual a sua idade: 1
1
```

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 execução durante a execução da cláusula `try`, as instruções remanescentes na cláusula são ignoradas. 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`. 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 as especifique em uma tupla:

1.4 Exemplo

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

```
In [9]: f = open('myfile.txt')
        s = f.readline()
        i = int(s.strip())

        ---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call last)
```

```
    <ipython-input-9-190ad13a4376> in <module>()
----> 1 f = open('myfile.txt')
      2 s = f.readline()
      3 i = int(s.strip())


FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
```

Neste caso podemos fazer o seguinte Experimente a correr: 1. sem que o ficheiro myfile.txt exista 2. crie o ficheiro e preencha-o com o seu nome

```
In [10]: import sys

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

```
Não consigo converter "-@312źčźčğ!#!#!$##5"%23412č123" para inteiro.
```

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:

```
In [11]: 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()
```

```
cannot open -f
/run/user/1000/jupyter/kernel-2e2c4436-fea7-4509-8957-3b8ddac4a3cb.json has 12 lines
```

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. A variável é associada à instância de exceção capturada, com os argumentos armazenados em instancia.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.

```
In [12]: try:
             raise Exception('spam', 'eggs')
         except Exception as inst:
             print(type(inst))    # a instancia da excexao
             print(inst.args)     # argumentos armazenados em ".args"
             print(inst)          # __str__ está implementado...

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

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

1.7 Levantando exceções

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

```
In [13]: try:
             raise NameError('Ola...!')
         except NameError:
             print('Passou por aqui uma excecao!')
             raise
```

```
Passou por aqui uma excecao!
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-13-19d162633e48> in <module>()
  1 try:
----> 2     raise NameError('Ola...!')
```

```
 3 except NameError:
 4     print('Passou por aqui uma excecao!')
 5     raise


NameError: Ola...!
```

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`.

```
In [17]: def divide(x, y):
             try:
                 result = x / y
             except ZeroDivisionError:
                 print("erro: divisão por zero!")
             else:
                 print("o resultado é", result)
             finally:
                 print("... e agora, independentemente do que aconteça, cá vou eu!!")

In [18]: divide(1,2)

o resultado é 0.5
... e agora, independentemente do que aconteça, cá vou eu!!


In [19]: divide(1,0)

erro: divisão por zero!
... e agora, independentemente do que aconteça, cá vou eu!!
```

1.9 O exemplo dos ficheiros

não nos devemos esquecer de fechar os "open" que vamos fazendo

```
In [20]: 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)
```

```
Não consigo converter "-@312źčźčğ!#!#!$##5"%23412č123" para inteiro.
f está fechado?: False
f está fechado?: True
```

Como alternativa podemos garantir que o ficheiro foi fechado usando o `with`
Outro exemplo

```
In [21]: 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)
```

```
Não consigo converter "-@312źčźčğ!#!#!$##5"%23412č123" para inteiro.
f está fechado?: True
```

```
In [ ]:
```

1.10 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.

```
In [22]: 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
```

1.11 Exemplo

Implemente uma classe Data

```
In [23]: class Data:

            def __init__(self, a, m, d):
                if Data.valida(a, m, d):
                    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 é a/m/d valida. False caso contrário.
                :raises:
                    TypeError: xxxxxxxx
                    InvalidData: xxx excecao a ser implementada pelo aluno xxxx
                '''
                pass
```

```
In [24]: d = Data(2020, 2, 3)
         d
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

/usr/lib/python3/dist-packages/IPython/core/formatters.py in __call__(self, obj)
    697                 type_pprinters=self.type_printers,
    698                 deferred_pprinters=self.deferred_printers)
--> 699             printer.pretty(obj)
    700             printer.flush()
    701             return stream.getvalue()


/usr/lib/python3/dist-packages/IPython/lib/pretty.py in pretty(self, obj)
    401                         if cls is not object \
    402                                 and callable(cls.__dict__.get('__repr__')):
--> 403                             return _repr_pprint(obj, self, cycle)
    404
    405             return _default_pprint(obj, self, cycle)


/usr/lib/python3/dist-packages/IPython/lib/pretty.py in _repr_pprint(obj, p, cycle)
    701     """A pprint that just redirects to the normal repr function."""
    702     # Find newlines and replace them with p.break_()
--> 703     output = repr(obj)
    704     for idx,output_line in enumerate(output.splitlines()):
    705         if idx:


<ipython-input-23-c86c426934c7> in __repr__(self)
      6
      7     def __repr__(self):
----> 8         return f'{self.a}/{self.m}/{self.d}'
      9
     10     @staticmethod


AttributeError: 'Data' object has no attribute 'a'
```

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

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

1.13 A clusula `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.

```
In [25]: a = 1
         assert isinstance(a, int)
         print("ok")
```

```
ok
```

```
In [26]: a = 1.0
         assert isinstance(a, int)
         print("ok")
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-26-db35d6ea0513> in <module>()
          1 a = 1.0
    ----> 2 assert isinstance(a, int)
          3 print("ok")


        AssertionError:
```

```
In [27]: idade_filho = 18
         idade_mae = 4
         assert a > b, "erro: idades invalidas"
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-27-cec587f3da2b> in <module>()
          1 idade_filho = 18
          2 idade_mae = 4
    ----> 3 assert a > b, "erro: idades invalidas"


        NameError: name 'b' is not defined
```

```
In [28]: try:
             idade_filho = 18
```

```
        idade_mae = 4
        assert a > b
    except AssertionError as e:
        print("erro: idades invalidas")
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-28-3b22c4a6b28d> in <module>()
      2     idade_filho = 18
      3     idade_mae = 4
----> 4     assert a > b
      5 except AssertionError as e:
      6     print("erro: idades invalidas")


NameError: name 'b' is not defined
```