02

Tratamento de exceções

DOWNLOAD

Segue o link (https://s3.amazonaws.com/caelum-online-public/python/13-python.zip) com os arquivos desta aula.

Built in Errors

O que acontece se tentamos ler um arquivo que não existe? Se você ainda não passou por essa experiência, passará agora! Vamos mudar o nome do arquivo:

```
>>> from models import *
>>> perfis = Perfil.gerar_perfis('nao-existe.csv')
...
IOError: [Errno 2] No such file or directory: 'nao-existe.csv'
```

Recebemos um `IOError` que é um dos erros, ou melhor uma das exceções, que já vem embutida no Python. Existem várias outras exceções como o `TypeError`, que também já vimos durante o treinamento. Por exemplo, recebemos um `TypeError` quando tentamos concatenar um tuple com uma lista:

```
>>> [1,2] + (3,4)
TypeError: can only concatenate list (not "tuple") to list
```

Além das exceções que acontecem durante a execução da aplicação, existem erros que indicam algum problema na sintaxe. Por exemplo, se fizermos uma indentação errada ou declararmos uma variável sem inicializá-la, recebemos um erro:

```
>>> def imprime():
... print('indentação errada')
...
IndentationError: expected an indented block
```

Nesse caso a aplicação nem foi executada. O interpretador do Python já parou antes, pois há um problema na sintaxe que impede sua execução.

Resumindo, existem dois tipos de erros/exceções. Erros do interpretador pois há algum erro na sintaxe (o programa nem executa), e erros de execução quando acontece algum problema durante da execução do programa (não existe problema de sintaxe).

Tratamento de exceções

Como vimos, a nossa aplicação está vulnerável, bastando abrir um arquivo que não existe. Porém, isso não é incomum de acontecer, não é mesmo? Alguém pode tê-lo apagado e por aí vai. O ideal é que nossa aplicação trate o problema e exiba uma mensagem amigável para o usuário. Não é uma boa prática exibirmos para ele códigos ou mensagens de erros como `IOError` ou `TypeError`. Essas mensagens só devem existir para o desenvolvedor e não podem vazar para o usuário.

O Python tem uma forma de avisar ao ambiente de execução que algum código é potencialmente perigoso. Existe um bloco do tipo `try-except` (tente-exceto) para capturar erros durante a execução. Por exemplo, vamos carregar um arquivo inexistente dentro de um bloco `try` e no bloco `except` definimos o tipo de exceção que pode acontecer e qual é o tratamento. Crie o arquivo `erros.py`, nesse exemplo vamos imprimir apenas uma mensagem na tela:

```
# -*- coding: UTF-8 -*-

#erros.py
try:
        open('nao_existe.txt','r')
        print('O arquivo foi aberto')
        arquivo.close()
except IOError as erro:
        print('Deu IOError')
```

Executando:

```
$ python erros.py
Deu IOError
```

Veja que exibimos nossa mensagem de erro, não a mensagem padrão do Python. Repare que não foi exibida a mensagem "O arquivo foi aberto". A linha com o comando `open` gerou o erro, e o fluxo foi para o bloco `except`. Lá foi tratado o `IOError` e impressa a mensagem para o usuário. Muito melhor do receber um `IOError`!

Trabalhando com a causa

Há vários motivos da não abertura de um arquivo. Por exemplo, sua inexistência ou podemos não ter permissão de leitura. É por isso que todo erro lançado pelo Python carrega consigo informações sobre a sua causa, inclusive podemos acessá-la através da palavra chave `as`:

```
# -*- coding: UTF-8 -*-

# erros.py
try:
        open('nao_existe.txt','r')
        print('arquivo foi aberto')
        arquivo.close()
except IOError as erro:
        print('Deu IOError: %s' % erro)
```

E chamando na linha de comando:

```
$ python erros.py
Deu IOError: [Errno 2] No such file or directory: 'nao_existe.txt'
```

Recebemos o código de erro e uma mensagem sobre a causa.

Tratando várias exceções

Dentro do bloco `try` pode acontecer mais de um problema. Por exemplo, vimos o uso de `*valores` na hora de criar um `Perfil`. A variável `valores` nada mais é do que uma lista, e cada elemento é passado no construtor da classe `Perfil`. Se o construtor tem 3 parâmetros, a lista deve ter a mesma quantidade de elementos.

E se a quantidade não bate? Nesse caso recebemos um `TypeError`, de novo. Também precisamos nos previnir desse tipo de problema. Para isso podemos repetir o bloco `except`:

```
# -*- coding: UTF-8 -*-

#erros.py

#Não podemos nos esquecer de importar o módulo models
from models import *
try:
        arquivo = open('perfis.csv','r')
        valores = arquivo.readline().split(':') #deve ser virgula, para simular o problema
        Perfil(*valores)
        arquivo.close()
except IOError as erro:
        print('Deu IOError %s' % erro)
except  TypeError as erro:
        print('Deu TypeError %s' % erro)
`
```

E chamando na linha de comando:

```
$ python erros.py
Deu TypeError __init__() takes exactly 4 arguments (2 given)
```

Como alternativa podemos listar os tipos de erro em um único bloco `except`:

```
# -*- coding: UTF-8 -*-

#erros.py
from models import *
try:
        arquivo = open('perfis.csv','r')
        valores = arquivo.readline().split(':')
        Perfil(*valores)
        arquivo.close()
except (IOError,TypeError) as erro:
        print('Deu Error %s' % erro)
```

Testanto novamente:

```
$ python erros.py
Deu Error __init__() takes exactly 4 arguments (2 given)
```

NÃO ESQUEÇA DE ALTERAR A FUNÇÃO SPLIT PARA VOLTAR A TRABALHAR COM A VÍRGULA

Bloco finally

Analisando bem o nosso código, pode acontecer que o arquivo seja aberto com a função `open`, mas um erro é lançado pela função `readline()`. Nesse caso não estamos fechando o arquivo, algo grave que não permitirá que outros usuários ou aplicações consigam interagir com o arquivo.

Quando há algum código que deve ser sempre executado com ou sem exceção, devemos usar um bloco `finally`. Então o fechamento do arquivo deve ficar dentro do bloco `finally`. O problema ainda é que não temos acesso a variável `arquivo` pois ela está visível somente no bloco `try`. Por isso vamos definir a variável antes do bloco `try`:

```
# -*- coding: UTF-8 -*-

#erros.py
from models import *
arquivo = None #inicializar a variável arquivo
try:
        arquivo = open('perfis.csv','r')
        valores = arquivo.readline().split(',')
        Perfil(*valores)
except (IOError,TypeError) as erro:
        print('Deu Error %s' % erro)
finally:
        if(arquivo is not None):
                print('fechando arquivo')
                arquivo.close()
```

Sintaxe with-as

Há também casos que queremos abrir um arquivo sem um bloco `except`, pois vamos testar e executar o código sem pensar no tratamento da exceção, algo assim:

```
# -*- coding: UTF-8 -*-

#erros.py
from models import *
arquivo = None
try:
        arquivo = open('perfis.csv','r')
        valores = arquivo.readline().split(',')
        Perfil(*valores)
finally:
        if(arquivo is not None):
                print('fechando arquivo')
                arquivo.close()
```

Repare que o código continua complexo por causa do `finally` e prejudica a legibilidade. Quando trabalhamos com arquivos, IO em geral, existe uma atalho para a abertura de arquivos:

```
>>> with open("perfis.csv") as arquivo:
...     for linha in arquivo:
...         print linha
```

```
...
Ana Paula Gonçalves, 21-34345432, Amigas Ltda

Camila Almeida, 21-21215643, Auron Ltda

Alexandrina Pessoa, 11-23416531, Primo Serviços

João da Silva, 45-21249834, Preço Ótimo
```

Não precisamos nos preocupar em inicializar a variável `arquivo` nem fazer o bloco `finally`. O Python assume essa responsabilidade permitindo que foquemos apenas em nosso código, agora muito mais legível.

Lançando exceções

Na nossa aplicação estamos lendo dados do arquivo `csv`, e para cada linha, criamos um novo perfil. Como os dados vêm de uma fonte externa é preciso ter cautela no uso deles. O uso dos dados de um perfil deve ser verificado pela aplicação.

Imagina que uma linha no arquivo `csv` não possui o nome da empresa. Podemos facilmente simular isso e apagar a empresa na primeira linha:

```
Ana Paula Gonçalves, 21-34345432
Camila Cruzes, 21-21215643, Auron Ltda
Alexandrina Pessoa, 11-23416531, Primo Serviços
João da Silva, 45-21249834, Preço Ótimo
```

E chamando o método estático para gerar os perfis como já vimos antes:

```
$ python nome_do_arquivo_que_chama_o_método_estatico_para_gerar_perfis.py
```

Recebemos a exceção seguinte, novamente um `TypeError`:

```
>>> Traceback (most recent call last):
...
  File "/Users/nico/Documents/dev/alura/python/python-code/models.py", line 23, in gerar_perfis
    perfis.append(classe(*valores))
TypeError: __init__() takes exactly 4 arguments (3 given)
```

Como criamos a aplicação e simulamos o problema, está claro para nós o que causou o `TypeError`. No entanto, nem sempre usamos o nosso código, e sim o código escrito por outros. Nesse caso seria ótimo receber uma exceção mais específica, algo que indica mais fácil porque o código não está funcionando.

Para isso o Python possibilita lançar uma exceção. Podemos interromper o fluxo da aplicação *jogando* uma exceção. Vamos abrir o arquivo `models.py` e alterar o método `gerar_perfis`. Dentro do laço vamos verificar o tamanho da lista de valores antes de chamar o construtor. Caso não tenha o tamanho de 3 criaremos uma exceção do tipo `ValueError`. A classe `ValueError` já existe e vem com o Python.

```
for linha in arquivo:
          valores = linha.split(',')
```

```
            if(len(valores) is not 3):
              raise ValueError('Uma linha no arquivo %s deve ter 3 valores' % nome_arquivo)

            perfis.append(classe(*valores))
```

A mensagem deve ser expressiva e ajudar a entender o problema. Até poderíamos imprimir o numero da linha para ajudar mais ainda.

Lançando mais exceções

Pensando mais ainda na robustez da nossa aplicação, faz sentido verificar cada parâmetro do `Perfil` ? Por exemplo, não deve ter um nome com um caracter só, nem empresa ou telefone. No entanto a nossa aplicação aceita.

Vamos verificar novamente os dados, só que dessa vez a verificação ficará dentro do construtor da classe `Perfil` . Encapsulamos esse código para deixar mais reutilizável.

Para testar o nome do perfil, usaremos um `if` que testa o tamanho da `string` , lançando uma exceção caso a mesma tenha menos de 3 caracteres:

```
class Perfil(object):
   'Classe padrão para perfis de usuários'

   def __init__(self, nome, telefone, empresa):

      if(len(nome) < 3):
        raise ValueError('Nome deve ter pelo menos 3 caracteres')

      #codigo omitido
```

Ao testar com o arquivo CSV com um nome de 2 caracteres, por exemplo:

```
An, 21-34345432, Amigas Ltda
Camila Cruzes, 21-21215643, Auron Ltda
Alexandrina Pessoa, 11-23416531, Primo Serviços
João da Silva, 45-21249834, Preço Ótimo
```

e chamar o código na linha de comando:

```
$ python nome_do_arquivo_que_chama_o_método_estatico_para_gerar_perfis.py
```

Recebemos, como esperado, um `ValueError` . A classe `ValueError` é bastante genérica e em alguns casos pode fazer mais sentido criar uma própria exceção, que indica melhor o que aconteceu. Assim o nome da classe já ajuda no entendimento do problema.

Vamos criar uma nova classe `ArgumentoInvalidoError` no arquivo `models.py` . A classe deve herdar da classe `Exception` padrão do Python.

```
class ArgumentoInvalidoError(Exception):
```

No construtor da classe recebemos, além da variável `self`, a mensagem da exceção que guardamos como atributo:

```
        def __init__(self, mensagem):
                self.mensagem = mensagem
```

Por fim o método `__str__`, para imprimir a mensagem:

```
        def __str__(self):
                return repr(self.mensagem)
```

Segue uma vez a classe completa:

```
class Argumento_Invalido_Error(Exception):

    def __init__(self, mensagem):
      self.mensagem = mensagem

    def __str__(self):
      return repr(self.mensagem)
```

Uma vez a classe criada, podemos lançar a exceção no construtor, e substituir `ValueError` com `ArgumentoInvalidoError`:

```
raise Argumento_Invalido_Error('Nome deve ter pelo menos 3 caracteres')
```