Tratamento de Exceção

Você já deve ter notado que algumas operações podem dar errado em certas circunstâncias, e esses erros provocam o tratamento do nosso programa.

Por exemplo, quando solicitamos que o usuário digite um número inteiro e ele digita qualquer outra coisa. O erro ocorre especificamente na conversão da entrada para `int`. Veja o exemplo abaixo:

```
entrada = 'olá'
inteiro = int(entrada)
```

Erro mostrado na tela:

```
ValueError: invalid literal for int() with base 10: 'olá'
```

Note que o erro possui um nome, `ValueError`, e uma mensagem explicando o que ocorreu.

Vejamos outro exemplo bastante famoso: a divisão por zero.

```
x = 1/0
```

Erro mostrado na tela:

```
ZeroDivisionError: division by zero
```

Observe a mesma estrutura do erro anterior: temos um nome (`ZeroDivisionError`) e uma mensagem explicando o que ocorreu.

Esses erros, que não são erros de lógica nem de sintaxe, são o que chamamos de **exceções**. São pequenos problemas que o programa pode encontrar durante sua execução, como não encontrar um arquivo ou uma função receber um valor de tipo inesperado.

Vamos começar aprendendo como lidar com códigos que podem provocar erros, evitando o travamento do programa, e em seguida iremos aprender a criar as nossas próprias exceções para alertar outros programadores sobre problemas que possam ter ocorrido em nossas classes e funções.

> A documentação oficial do Python traz uma lista completa de exceções que já vem prontas e a relação de hierarquia entre elas.

1. Tratando uma exceção

1.1. try/except

Tratar uma exceção significa que quando surgir um dos erros mencionados, nós iremos assumir responsabilidade sobre ele e iremos providenciar algum código alternativo. Dessa maneira, o Python não irá mais travar o nosso programa, e sim desviar seu fluxo para o código fornecido.

O bloco mais básico para lidarmos com exceção é o `try`/`except`.

Dentro do `try` vamos colocar o pedaço de código com potencial para dar erro. Estamos pedindo que o Python **tente** executar aquele código, cientes de que pode não dar certo.

Dentro do `except`, colocamos o código que deverá ser executado **somente** se algo de errado ocorrer no `try`. Caso ocorra exceção em alguma linha do `try`, a execução irá **imediatamente** para o `except`, ignorando o restante do código dentro do `try`. Vejamos um exemplo:

```
numerador = 1

for denominador in range(3, -1, -1):
    try:
        divisao = numerador/denominador
        print('Deu certo!') # roda APENAS se a linha acima não gerar exceção

    except:
        divisao = 'infinito'

    print(f'{numerador}/{denominador} = {divisao}')
```

Saída na tela:

```
Deu certo!
1/3 = 0.3333333333333333
Deu certo!
1/2 = 0.5
Deu certo!
1/1 = 1.0
1/0 = infinito
```

O bloco acima já resolve a grande maioria dos problemas. Mas vamos estudar mais algumas possibilidades para deixar nosso tratamento ainda mais sofisticado e especializado.

Você deve ter notado que enfatizamos o fato de exceções poderem ter um nome. Esse nome pode nos ajudar a identificar com sucesso qual dos erros possíveis ocorreu e tratá-lo com sucesso.

Vamos considerar a função abaixo:

```
def divisao(a, b):
    return a/b
```

Um erro óbvio que pode ocorrer nessa função seria o `ZeroDivisionError`, que é obtido quando o zero é passado como segundo parâmetro da função. Porém, ele não é o único erro possível.

O que acontece se passarmos um parâmetro que não seja numérico? `TypeError`, pois utilizamos tipos inválidos para o operador de divisão `/`.

Podemos colocar diversos `except` após o `try`, cada um testando um tipo diferente de erro. Um último `except` genérico englobará todos os casos que não se encaixarem nos específicos. Veja o exemplo:

```
def divisao(a, b):
    return a/b

denominadores = [0, 2, 3, 'a', 5]

for d in denominadores:
    try:
        div = divisao(1, d)

    except ZeroDivisionError:
        div = 'infinito'

    except TypeError:
        div = f'1/{d}'

    except:
        div = 'erro desconhecido'

    print(f'1/{d} = {div}')
```

Saída na tela:

```
1/0 = infinito
1/2 = 0.5
1/3 = 0.3333333333333333
1/a = 1/a
1/5 = 0.2
```

1.2. else

Nosso bom e velho `else`, tipicamente usado em expressões condicionais acompanhando um `if`, também pode aparecer em blocos `try`/`except`. Seu efeito é o oposto do `except`: enquanto o `except` é executado quando algo dá errado, o `else` só é executado se absolutamente nada der errado. Por exemplo, poderíamos atualizar nosso exemplo anterior utilizando um `else`:

```
def divisao(a, b):
    return a/b

denominadores = [0, 2, 3, 'a', 5]

for d in denominadores:
    try:
        div = divisao(1, d)

    except ZeroDivisionError:
        print('infinito')

    except TypeError:
        print(f'1/{d}')

    except:
        print('erro desconhecido')

    else:
        print(f'1/{d} = {div}')
```

Saída na tela:

```
infinito
1/2 = 0.5
1/3 = 0.3333333333333333
1/a
1/5 = 0.2
```

Note que, no exemplo acima, não tem problema estarmos atribuindo valor pra `div` apenas no bloco `try`. Ela só será usada no `else`, ou seja, só será usada se tudo deu certo.

1.4. finally

Muitas vezes um erro pode ocorrer quando já realizamos diversas operações. Dentre essas operações, podemos ter solicitado recursos, como por exemplo abrir um arquivo, estabelecer uma conexão com a internet ou alocar uma grande faixa de memória.

O que aconteceria, por exemplo, se um comando como `return` aparecesse durante o tratamento deste erro após termos solicitado tantos recursos diferentes? O arquivo ficaria aberto, a conexão ficaria aberta, memória seria desperdiçada, etc.

O `finally` garante um local seguro para colocarmos código de limpeza - ou seja, devolver recursos que não serão mais utilizados: fechar arquivos, fechar conexões com servidor etc.

Ele **sempre** será executado após um bloco `try`/`except`, mesmo que haja um `return` no caminho.

Veja o exemplo abaixo para entender o que queremos dizer:

```
def teste(den):
    try:
        x = 1/den
        return x
    except:
        return 'infinito'
    finally:
        print('Opa')

print(teste(1))
print(teste(0))
```

Saída na tela:

```
Opa
1.0
Opa
infinito
```

Note que o conteúdo do bloco `finally` foi executado em ambas as chamadas, mesmo havendo um `return` dentro do `try` e outro dentro do `except`. Antes de sair da função e retornar o valor, o Python é obrigado a desviar a execução para o bloco `finally` e executar seu conteúdo.

Vejamos um exemplo mais completo: um bloco `try`/`except` tentará criar um arquivo (não se preocupe com detalhes de como arquivos funcionam - estudaremos isso muito em breve!). Dentro do `try`, teremos um bloco `try`/`except`/`finally`. O `try` tentará escrever algumas operações matemáticas no arquivo, o `except` exibirá uma mensagem caso uma operação seja inválida, e o `finally` garantirá que o arquivo será fechado **independentemente de um erro ter ou não ocorrido**.

```
def escreve_arquivo(nome_do_arquivo, denominador):
    try:
        arq = open(nome_do_arquivo, 'w') #abre o arquivo

        try:
            div = 1/denominador
            arq.write(str(div)) #escreve no arquivo
            return f'O número {div} foi escrito no arquivo.'

        except ZeroDivisionError:
            return 'Divisão por zero, não escrevemos no arquivo.'

        except TypeError:
            return 'Tipo inválido, não escreveremos no arquivo.'
```

```
        except:
            return 'Erro desconhecido, não escreveremos no arquivo.'

        finally:
            print(f'Fechando o arquivo {nome_do_arquivo}')
            arq.close() # o arquivo SEMPRE será fechado, mesmo que ocorra erro!


    except:
        return 'Não foi possível abrir o arquivo'


print(escreve_arquivo('teste1.txt', 1))
print(escreve_arquivo('teste2.txt', 0))
```

Saída na tela:

```
Fechando o arquivo teste1.txt
O número 1.0 foi escrito no arquivo.
Fechando o arquivo teste2.txt
Divisão por zero, não escrevemos no arquivo.
```

2. Levantando exceções

Quando estamos criando nossos próprios módulos, classes ou funções, muitas vezes vamos nos deparar com situações inválidas. Imprimir uma mensagem de erro não é uma boa ideia, pois o programa pode estar rodando em um servidor, pode ter uma interface gráfica, etc.

Logo, o ideal seria lançarmos exceções para sinalizar essas situações. Desta forma, se elas forem ignoradas, o programa irá parar, sinalizando para o programador que existe alguma situação que deveria ser tratada. Adicionalmente, podemos criar nossa própria mensagem de erro, sinalizando para o programador que ele deveria fazer algo a respeito.

Podemos utilizar a palavra `raise` seguida de `Exception()`, passando entre parênteses a mensagem personalizada de erro. Veja o exemplo:

```
salarios = []

def cadastrar_salario(salario):
    if salario <= 0:
        raise Exception('Salário inválido! Salários devem ser positivos!')

    salarios.append(salario)
```

```
cadastrar_salario(10)
cadastrar_salario(0)
```

Note que na primeira chamada, onde não ocorreu exceção, o salário foi cadastrado na lista. Já na segunda chamada, nossa função lançou a exceção e parou sua execução.

Idealmente, quem pretende utilizar a função deveria fazê-lo agora utilizando `try`, para manter o programa funcionando e tratar adequadamente o problema.

```
salarios = []

def cadastrar_salario(salario):
    if salario <= 0:
        raise Exception('Salário inválido! Salários devem ser positivos!')

    salarios.append(salario)

for i in range(3):
    salario = float(input('Digite o salário do funcionário: '))

    try:
        cadastrar_salario(salario)
    except:
        print('Opa, salário inválido!')

print(salarios)
```

Exemplo de execução:

```
Digite o salário do funcionário: 1000
Digite o salário do funcionário: -500
Opa, salário inválido!
Digite o salário do funcionário: 1500
[1000.0, 1500.0]
```

O `raise` também pode ser utilizado para lançar exceções que já existem, não necessariamente exceções "novas". Basta trocar `Exception()` pelo nome da exceção desejada. De fato, quando utilizamos `raise Exception()` estamos apenas lançando a exceção mais genérica, da qual outras são derivadas, apenas especificando sua mensagem de erro.

3. Criando exceções novas

Nota: este tópico utiliza conceitos de **programação orientada a objeto**. Ele está aqui para tornar esse capítulo mais completo. Caso você curse um módulo de programação orientada a objeto futuramente, é recomendável reler este material. Em todo caso, é possível utilizar os exemplos deste tópico como modelo para criar exceções mesmo sem compreender os detalhes do que está ocorrendo.

3.1. Herdando de Exception

Muitos problemas simples podem ser resolvidos através do `raise Exception(mensagem)`. Porém, você deve ter notado nos exemplos anteriores que o nome da nossa mensagem de erro foi `Exception`.

Exceções geralmente são implementadas através de classes. O "nome" do erro é o nome da classe de cada exceção. Existe uma exceção genérica chamada de `Exception`. Quando usamos `raise Exception(mensagem)`, estamos lançando essa exceção genérica junto de uma mensagem de erro personalizada.

O problema da nossa abordagem é que por utilizarmos uma exceção genérica não teremos como adicionar um `except` específico para nossa mensagem. Vamos criar nossa própria classe para escolher o nome do nosso erro. Exceções personalizadas geralmente **herdam** da classe `Exception`. Fazemos isso adicionando `(Exception)` após o nome de nossa classe.

Vamos colocar um construtor que recebe uma mensagem. Podemos definir uma mensagem padrão, caso ninguém passe a mensagem. Em seguida, chamaremos o construtor da superclasse `(Exception)`.

```
class SalarioInvalido(Exception):
    def __init__(self, message = 'Salários devem ser positivos!'):
        self.message = message
        super().__init__(self.message)

salarios = []

def cadastrar_salario(salario):
    if salario <= 0:
        raise SalarioInvalido()

    salarios.append(salario)

cadastrar_salario(0)
```

Mensagem de erro mostrada na tela:

```
SalarioInvalido: Salários devem ser positivos!
```

Agora sim temos um erro com seu próprio nome e uma mensagem padrão. Mas note que quem está usando a nossa exceção pode personalizar a mensagem se quiser, basta passar uma mensagem diferente entre parênteses. O tipo do erro ainda será o mesmo e ambos deverão ser identificados como `SalarioInvalido` no `Except`.

```
class SalarioInvalido(Exception):
    def __init__(self, message = 'Salários devem ser positivos!'):
        self.message = message
        super().__init__(self.message)
```

```
salarios = []

def cadastrar_salario(salario):
    if salario <= 0:
        raise SalarioInvalido('Deixa de ser mão-de-vaca e pague seus funcionários!')

    salarios.append(salario)

cadastrar_salario(0)
```

Mensagem de erro mostrada na tela:

```
SalarioInvalido: Deixa de ser mão-de-vaca e pague seus funcionários!
```

Para finalizar, vale sempre lembrar que podemos tratar essa exceção específica:

```
class SalarioInvalido(Exception):
    def __init__(self, message = 'Salários devem ser positivos!'):
        self.message = message
        super().__init__(self.message)

salarios = []

def cadastrar_salario(salario):
    if salario <= 0:
        raise SalarioInvalido()

    salarios.append(salario)

for i in range(3):
    salario = float(input('Digite o salário do funcionário: '))

    try:
        cadastrar_salario(salario)
    except SalarioInvalido:
        print('Nosso RH é uma vergonha :(')
    except:
        print('Exceção genérica')

print(salarios)
```

3.2. Adicionando atributos à exceção

É possível uma exceção trazer consigo informações sobre o valor que provocou o erro. Por exemplo, seria útil que a classe `SalarioInvalido` pudesse informar qual foi o salário inválido. Isso é útil, por exemplo, em *logs* que registram tudo o que ocorreu no programa, além de trazer informações importantes para o *debugging* do código.

Para isso, basta ajustar o construtor da classe de sua exceção:

```
class SalarioInvalido(Exception):
    def __init__(self, salario, mensagem='Salários devem ser positivos!'):
        self.salario = salario
        self.message = mensagem
        super().__init__(self.message)
```

Agora, ao lançar a exceção, devemos passar o salário:

```
salarios = []

def cadastrar_salario(salario):
    if salario <= 0:
        raise SalarioInvalido(salario)

    salarios.append(salario)
```

Por fim, ao tratar a exceção, podemos dar um *alias* (um "apelido") para ela utilizando a palavra `as`. Através desse apelido, podemos acessar seus atributos.

Note que imprimir o objeto faz com que sua mensagem seja impressa.

```
for i in range(3):
    salario = float(input('Digite o salário do funcionário: '))

    try:
        cadastrar_salario(salario)
    except SalarioInvalido as excecao:
        print(excecao) # "Salários devem ser positivos!"
        print(f'O salário problemático foi: {excecao.salario}')
    except:
        print('Exceção genérica')

print(salarios)
```

Exemplo de execução:

```
Digite o salário do funcionário: 1000
Digite o salário do funcionário: -500
Salários devem ser positivos!
```

```
O salário problemático foi: -500.0
Digite o salário do funcionário: 1500
[1000.0, 1500.0]
```