

# Tratamento de Exceções (try / except)

Conceitos, estruturas e aplicação prática.

**Alunos:** Eduardo Dutra e Igor Gonçalves  
**Professor:** Luiz Antonio Schalata

# 1 – Introdução

## O que vamos ver?

- Os conceitos básicos do que são **Erros** e **Exceções**;
- Conceitos e estrutura dos blocos:
  - **try/except;**
  - **try/except/finally;**
  - **múltiplos blocos except.**
- Vamos ver o que é o raise, pra que serve e por que devemos usar;
- Exemplo prático aplicado no projeto de uma mini calculadora.

# 2.1 – Conceitos (Erros)

## O que são erros?

- Os erros em programação são situações em que o código não consegue nem ser interpretado ou executado, geralmente porque foi escrito de forma incorreta. Esses erros são detectados pelo interpretador antes do programa rodar.
- Existem também os erros lógicos, eles não impedem o programa de rodar, mas podem alterar o resultado e/ou o fluxo do programa em algum determinado ponto.

## 2.1 – Erro

Está ligado diretamente com erros de sintaxe e erros lógicos presentes dentro do código, vejamos:

- **Erro de sintaxe:** Quando o código é mal escrito. O interpretador do Python não começa a rodar o programa.
- **Erro de lógica:** O programa consegue rodar sem travar, porém, o resultado lógico está errado.

Exemplo: `print("x")`

Exemplo: `print("1" + "3")`

## 2.2 – Conceitos (exceções)

### O que são exceções?

- **Diferente** dos erros, elas **não impedem** o funcionamento do código, porém em algum momento algo inesperado pode alterar o funcionamento do programa, são erros lógicos e não de sintaxe.
- Existem as exceções nativas do **Python** que normalmente retornam uma mensagem de erro no terminal e outras gerais que captam qualquer erro que o sistema não tenha identificado.

## 2.2 – Exceção

Sinaliza um evento ocorrido durante a execução de um programa que altera o fluxo normal do código e ocorre mesmo que a sintaxe esteja correta. Vejamos três exceções padrões do Python:

- **ValueError**: Surge quando uma função pega um argumento de tipo correto, porém de valor impróprio;
- **ZeroDivisionError**: Surge quando a segunda divisão ou módulo é zero
- **Exception** : Classe para a maioria da exceções e captura qualquer tipo de exceção que não seja uma exceção de nível mais baixo

## 3.1 – try/except

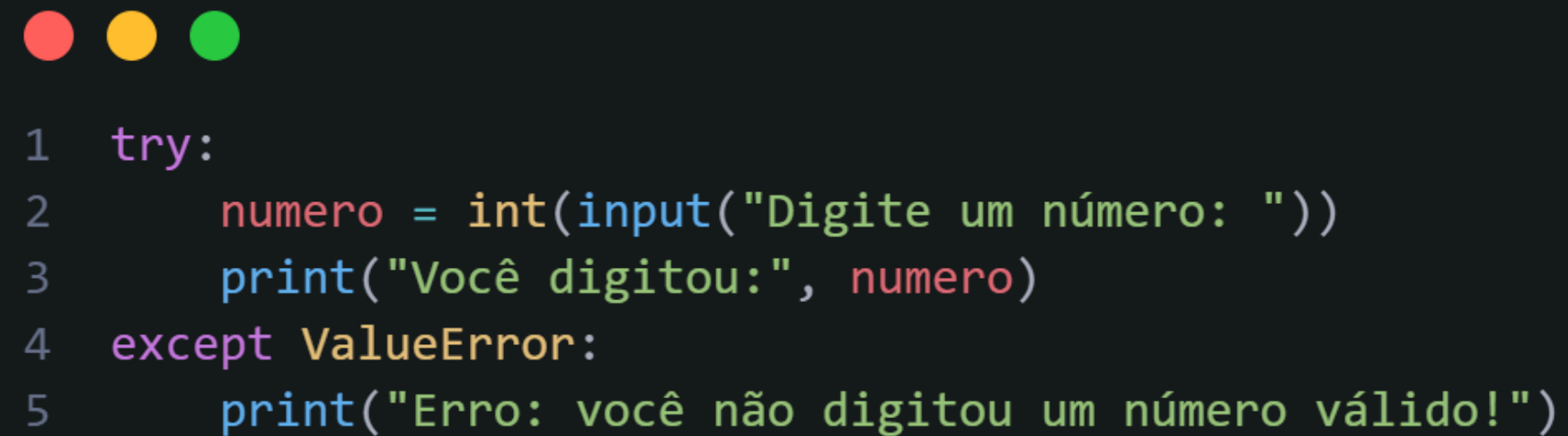
Existem diversas formas de fazermos sua estrutura, sendo o modo mais simples com o bloco **try/except**.

- 1 - O código que pode gerar um exceção fica dentro do bloco try;
- 2 - Caso uma exceção aconteça, o fluxo é imediatamente redirecionado para o bloco except, onde está o tratamento.

Essa estrutura garante que o programa não pare de uma maneira tão brusca mesmo após erros.

# 3.1 – Estrutura

Exemplo de um bloco **try/except**:



```
1  try:
2      numero = int(input("Digite um número: "))
3      print("Você digitou:", numero)
4  except ValueError:
5      print("Erro: você não digitou um número válido!")
```



## 3.2 – try/except/finally

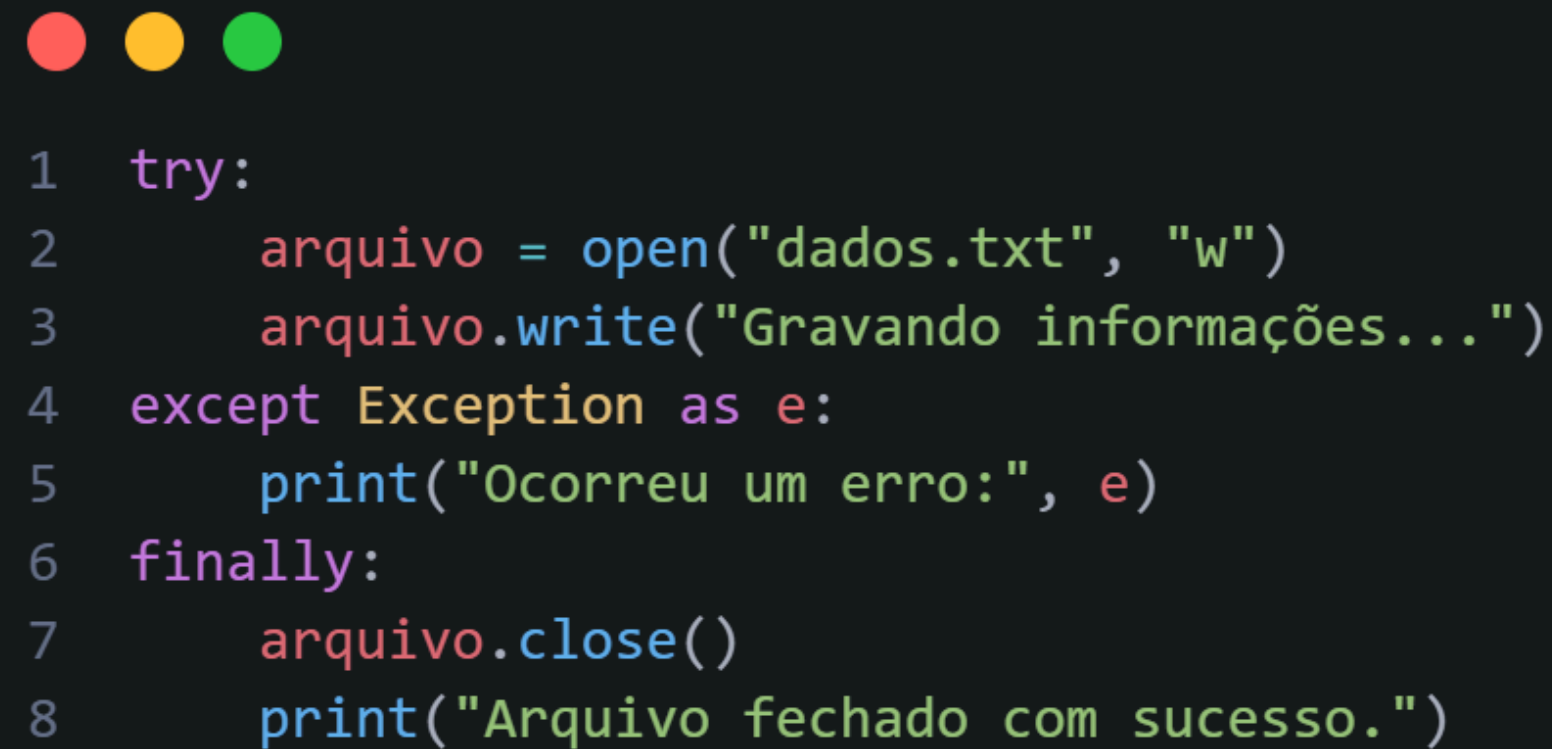
Podem existir casos onde nós precisamos garantir que o código seja executado mesmo após algum erro e para isso nós usamos o **try/except/finally**.

- 1 - Extremamente útil para liberar recursos importantes mesmo após um erro;
- 2 - Caso ocorra algum erro durante o try o finally ainda sim será executado.

Muito usado para fechar encerrar conexões com o banco de dados, liberar memória e fechar arquivos.

## 3.2 – Estrutura

Exemplo de um bloco **try/except/finally**:



```
1  try:
2      arquivo = open("dados.txt", "w")
3      arquivo.write("Gravando informações...")
4  except Exception as e:
5      print("Ocorreu um erro:", e)
6  finally:
7      arquivo.close()
8      print("Arquivo fechado com sucesso.")
```

## 3.3 – múltiplos **except**


Em programas complexos existem diversos erros que podem vim a acontecer e por isso se deve usar múltiplos blocos **except**, cada um tratando um tipo específico de exceção.

- 1 - Evita mensagens genéricas de erro e permite um tratamento mais preciso e estável;
- 2 - Pode também ser combinado com um **finally** após tratar diversos erros.

Muito comum ser visto vários **except** em códigos complexos por questões de boas práticas, segurança e robustez.

## 3.3 – Estrutura

Exemplo de um bloco com múltiplos **except**:



```
1  try:
2      x = int(input("Digite um número: "))
3      resultado = 10 / x
4  except ValueError:
5      print("Erro: entrada inválida. Digite apenas números.")
6  except ZeroDivisionError:
7      print("Erro: não é possível dividir por zero.")
8
```

# 4.1 – Uso do Raise

## O que é o raise?

- Ele é uma palavra chave usada para lançar exceções de uma forma manual. Diferente do **try/except** que captura erros já existentes, o **raise** provoca erros de propósito quando uma determinada condição não é atendida.
- Extremamente útil para impor regras lógicas e condicionais no programa, forçando com que um erro aconteça, permitindo assim, o desenvolvedor criar suas próprias regras de validação.

## 4.2 – Uso do Raise

### Por que usar o raise?

- Controle de regras e negócio para garantir que certas condições sejam devidamente atendidas;
- Validação de entradas impedindo com que não possam ser usados dados inválidos, lançando uma exceção personalizada;
- Deixa o código mais claro, ajudando a deixar explícito onde e por que uma operação não deve continuar.

## 4.3 – Estrutura

Exemplo de um bloco com **raise**:



```
1 def set_idade(idade):  
2     if idade < 0:  
3         raise ValueError("Idade não pode ser negativa!")  
4     print("Idade registrada:", idade)  
5  
6 set_idade(25)  
7 set_idade(-5)
```

# 5.1 – Miniaplicação: Calculadora

## O que é capaz de fazer

- Faz operações de um ou dois algoritmos, sendo eles:
  - Um algoritmo: raiz quadrada, logaritmo, seno, cosseno e tangente;
  - Dois algoritmos: adição, subtração, multiplicação, divisão e potencialização.



## 5.2 – Como ela funciona

### Vejamos alguns detalhes de seu funcionamento

- O usuário escolhe uma operação, alguns precisam de um algoritmo e outros de dois algoritmos;
- O programa executa o cálculo solicitado;
- Caso sejam fornecidos dados inválidos (como letras, divisão por zero) a exceção é capturada e uma mensagem de erro aparece;
- A calculadora continua rodando até o usuário decidir sair.

## 5.3 – Prática

### Funcionamento prático

- O código fonte pode ser encontrado através deste link:  
<https://github.com/igorgp06/seminario-POO-IFSC>
- Copie e cole o código, rode ele direto no seu editor de código;
- Alternativamente, pode-se rodar o código no **Google Colab**;
- A explicação completa do código pode ser encontrada no **README** do projeto presente no **GitHub**.

# 6.1 – Conclusão

## O que aprendemos

- A miniaaplicação da calculadora a aplicação prática dos conceitos do tratamento e a importância do tratamento das exceções além do uso do `raise`;
- Vimos como capturar erros comuns como a divisão por zero, garantindo segurança e estabilidade do programa;
- O uso de funções matemáticas com o **`math`** para cálculos simples e avançados.

# 7.1 – Referências

## Onde encontrar mais conteúdo

- Site oficial do Python: <https://www.python.org/>
- Guia para iniciantes: <https://pythoniluminado.netlify.app/>
- Vídeo aulas completas: [Curso em Vídeo - Python Mundo 1](#)  
[Curso em Vídeo - Python Mundo 2](#)  
[Curso em Vídeo - Python Mundo 3](#)



**INSTITUTO  
FEDERAL**

Santa Catarina

Câmpus  
Garopaba

**Fim!**

**Obrigado pela Atenção.**