

Gerenciamento de Erros

Luiza
<CODE>

Ementa:

1. Debugging (VS Code)
2. O que são exceções
3. Exceções checadas e não-checadas
4. Tratamento de exceções (BD, HTTP)
5. Log da aplicação (arquivo)

1. Debugging

1.1 Erros

Quando estamos desenvolvendo nosso software, estamos sujeitos alguns tipos de **erros**. Dentre eles, podemos destacar três tipos que podem ocorrer:

- Erros de **sintaxe**
- Erros de **execução**
- Erros de **semântica**

1.1.1 Erros de sintaxe

Sintaxe se refere à **estrutura** de um programa e as **regras** que regem essa estrutura. Na língua portuguesa, por exemplo, uma frase deve começar com uma letra maiúscula e terminar com um ponto final.

1. Debugging

1.1.2 Erros de execução

Outro tipo de erro é chamado erro de **execução** (*runtime error*). Ele possui esse nome porque só percebemos o erro após executarmos o programa. Geralmente também são chamados de **exceções** porque esses erros sugerem que alguma coisa excepcional aconteceu.

1.1.3 Erros de semântica

O último tipo de erro é o de **semântica**. Nessas situações, o software irá executar sem nenhuma mensagem de erro. Porém, o seu programa não irá retornar o resultado esperado. Ele não vai fazer a coisa correta (especificamente o que você programou).

1. Debugging

1.2 Debugging

Erros de programação são denominados bugs e o processo de encontrar e corrigir bugs é chamado de depuração ou debugging.

Colocar algumas chamadas **print()** no seu código geralmente é uma estratégia para obter informações adicionais sobre o que o programa está fazendo.

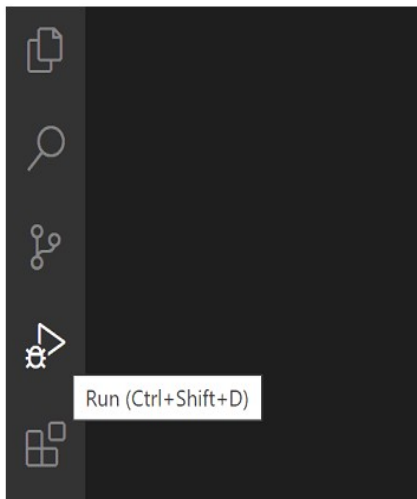
Podemos ir além e utilizar de depuradores, onde podemos definir pontos de interrupção em uma linha específica do código. O editor **VS Code** fornece uma experiência de depuração abrangente para Python.

Nas próximas etapas vamos entender como utilizar o VS Code para depurar.

1. Debugging

1.2.1 Executar visualização

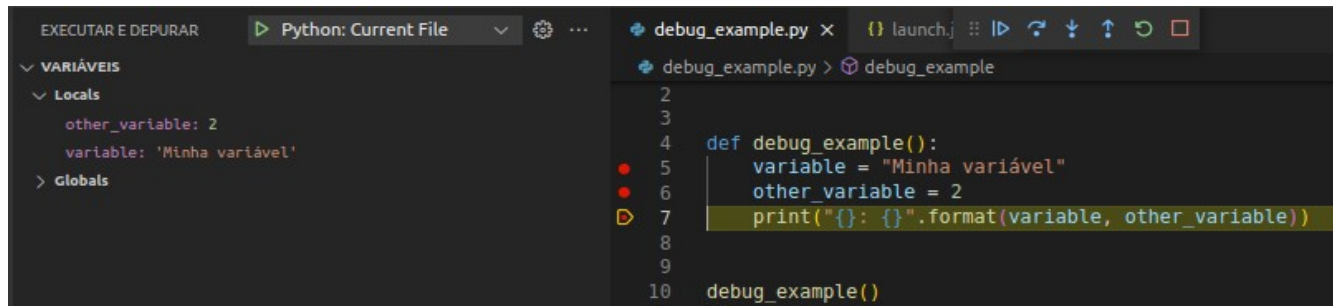
Para abrir a visualização “Executar”, selecione o ícone “Executar” na barra de atividades ao lado do VS Code.



1. Debugging

1.2.2 Pontos de interrupção

Podemos adicionar **pontos de interrupção** para verificar os locais que desejamos investigar. Eles podem ser inseridos clicando na margem do editor ou usando F9 na linha atual. O controle mais preciso do ponto de interrupção (ativar/desativar/reaplicar) pode ser feito na seção **BREAKPOINTS** da visualização Executar.



```

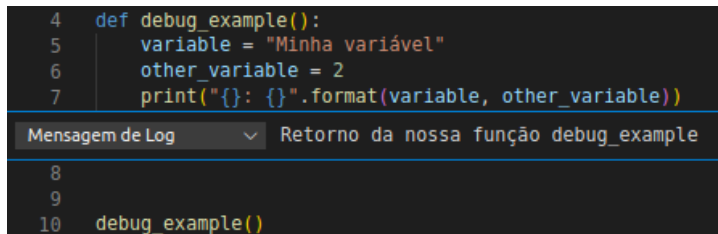
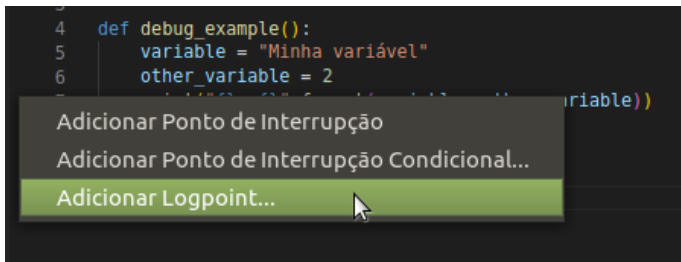
EXECUTAR E DEPURAR Python: Current File
VARIÁVEIS
  Locals
    other_variable: 2
    variable: 'Minha variável'
  Globals

debug_example.py x {} launch.j :: |D ↺ ↻ ↶ ↷
debug_example.py > debug_example
2
3
4 def debug_example():
5     variable = "Minha variável"
6     other_variable = 2
7     print("{}: {}".format(variable, other_variable))
8
9
10 debug_example()
  
```

1. Debugging

1.2.3 Logpoints

Um **logpoint** é uma variante de um ponto de interrupção que não "interrompe" no depurador, mas registra uma mensagem no **Console de Depuração**. Podemos inserir uma mensagem personalizada para o log:

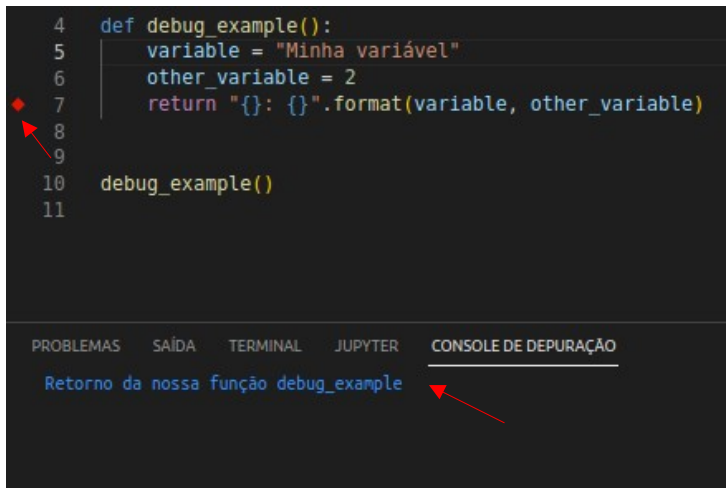


1. Debugging

1.2.3 Logpoints

Os **logpoints** são especialmente úteis para injetar logs durante a depuração de servidores de produção que não podem ser pausados ou interrompidos.

Eles são representados por um ícone em forma de "**diamante**".



```
4 def debug_example():
5     variable = "Minha variável"
6     other_variable = 2
7     return "{}: {}".format(variable, other_variable)
8
9
10 debug_example()
11
```

The screenshot shows a Jupyter Notebook interface. The top part displays a Python function `debug_example()` with four lines of code. A red diamond icon, representing a logpoint, is placed on the left margin next to line 7. A red arrow points from this icon to the bottom of the notebook. The bottom part of the notebook shows a tab labeled "CONSOLE DE DEPURAÇÃO" (Debug Console). Below this tab, the text "Retorno da nossa função debug_example" is displayed, with a red arrow pointing to it.

1. Debugging

1.2.4 Iniciar a depuração

Para iniciar uma sessão de depuração, primeiro selecione a configuração (sugerida de Python) usando o menu suspenso “**Configuração**” na visualização “Executar”. Depois de definir sua configuração de inicialização, inicie sua sessão de depuração com F5.

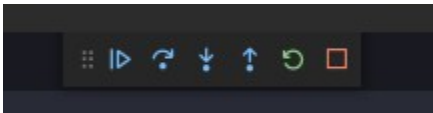
Assim que uma sessão de depuração é iniciada, o painel **DEBUG CONSOLE** é exibido e mostra a saída de depuração e a barra de status muda de cor (laranja para temas de cores padrão):



1. Debugging

1.2.5 Ações de depuração

Assim que uma sessão de depuração for iniciada, a barra de ferramentas de depuração aparecerá na parte superior do editor.

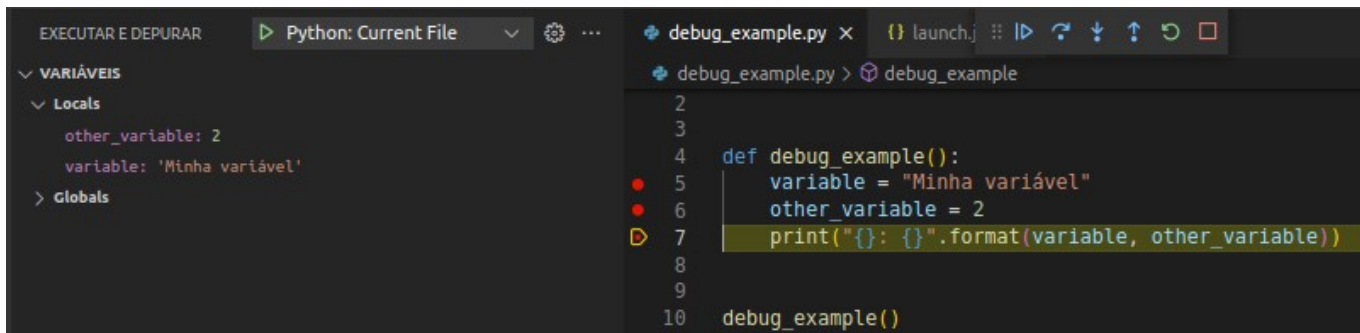


- Continuar / Pausar **F5**
- Passar por cima de **F10**
- Passo para **F11**
- Sair **Shift+F11**
- Reinicie **Ctrl+Shift+F5**
- Parar **Shift+F5**

1. Debugging

1.2.6 Inspeção de dados

As **variáveis** podem ser inspecionadas na seção **VARIÁVEIS** da visualização “Executar” ou passando o mouse sobre sua origem no editor. Os valores das variáveis e a avaliação da expressão são relativos ao quadro de pilha selecionado na seção **CALL STACK**.



```
EXECUTAR E DEPURAR Python: Current File debug_example.py x launch.j :: |> ? ↕ ↑ ↻ □
VARIÁVEIS
  Locals
    other_variable: 2
    variable: 'Minha variável'
  Globals
    >
debug_example.py > debug_example
2
3
4 def debug_example():
5     variable = "Minha variável"
6     other_variable = 2
7     print("{}: {}".format(variable, other_variable))
8
9
10 debug_example()
```

2. O que são exceções

2.1 Exceções

Quando ocorre um comportamento inesperado devido a uma função que não pode seguir normalmente, o que desejamos é basicamente **parar** o que está sendo executado e avançar diretamente de volta para o lugar onde podemos **lidar com a falha**. Conseguimos fazer isso com o tratamento de exceção.

Elas são uma forma que nos torna possível **interromper** o que está sendo executado com problema, lançando uma exceção que basicamente é um simples valor. Isso geralmente assemelha a um retorno super carregado a partir de uma função. Nesse momento acontece o **desenrolamento do stack**, onde esse valor salta para fora não apenas da função atual mas também para fora de todo o caminho até a primeira chamada que iniciou a execução atual.

2. O que são exceções

Utilizamos a palavra-chave **raise** para gerar uma exceção. Para capturá-la, envolvemos o trecho de código em um bloco **try**, seguido pela palavra-chave **except**. Quando ocorre um comportamento inesperado no bloco **try**, uma exceção é lançada e o bloco **except** é executado. O nome da variável (entre parênteses) após captura será vinculado ao valor de exceção. Após o término do bloco **except** ou do bloco **try** o fluxo de execução retoma sob toda a instrução **try/except**.

```
10 def exception_example():
11     condition_error = True
12
13     try:
14         if condition_error:
15             raise TypeError("Nossa condição com erro!")
16     except Exception as err:
17         print(f"Erro capturado: {err}")
18
19
20 exception_example()
21
22
```

PROBLEMAS SAÍDA TERMINAL JUPYTER CONSOLE DE DEPURAÇÃO

```
michelly@michelly-Latitude-3490:~/Documentos/Luiza Code$ python3 debug_example.py
Erro capturado: Nossa condição com erro!
```

2. O que são exceções

Infelizmente **erros** e entrada de **dados incorretos** acontecem. Devemos sempre encontrá-los e corrigi-los. As suites de testes automatizadas e asserções são extremamente importantes para que esses problemas sejam percebidos mais facilmente e tratados antes de serem executados em produção.

Na maioria das vezes, estes problemas provocados por fatores externos do programas devem ser tratados normalmente. Geralmente, quando o problema pode ser tratado de forma local, **valores de retorno especiais** é um caminho sensato para monitorá-los. Caso contrário as **exceções** são preferíveis.



3. Exceções checadas e não-checadas

A diferença entre exceções checadas e não checadas é a forma como a aplicação está preparada para lidar com o erro.

Nos dois casos podemos usar as declarações **try...except**.

3.1 Exceções não checadas

Na exceção não checada o erro ocorre porque houve um comportamento fora do fluxo normal, como, por exemplo uma string json mal formatada.

```
import json

def exception_example():
    string_json = '{ "message": "Olá mundo!" }'

    try:
        object = json.loads(string_json)
        print(json.dumps(object))
    except Exception as err:
        print(f"Erro capturado: {err}")

exception_example()
```

3.1 Exceções não checadas

Neste exemplo nós tentamos converter uma string para objeto, mas como a string não está no formato de um json válido por ter uma aspa a mais, o erro acontece e vai parar no bloco **except** onde nós podemos tratar o erro da forma que acharmos melhor, mas como ele não é checado, ou seja, não sabemos nenhum detalhe do que pode ter ocorrido, o tratamento fica bem genérico.

```
import json

def exception_example():
    string_json = '{ "message": "Olá mundo!" }'

    try:
        object = json.loads(string_json)
        print(json.dumps(object))
    except Exception as err:
        print(f"Erro capturado: {err}")

exception_example()
```

3.2 Exceções checadas

Na exceção checada nós preparamos o fluxo da aplicação para "checar" se aconteceu algo que precisa ser tratado como erro.

```
class CustomerError(Exception):
    def __init__(self, name, message, code):
        self.name = name
        self.message = message
        self.code = code
        super().__init__(self.name, self.message, self.code)

def exception_check():
    customer_list = []

    try:
        if len(customer_list) < 1:
            raise CustomerError(
                name="CustomerListError",
                message="A lista de clientes não pode ser vazia",
                code=9000
            )
    except Exception as err:
        print(err)

exception_check()
```

3.2 Exceções checadas

No Python, assim como em outras linguagens, existem classes internas específicas para tipos de erros.

Segue exemplo de uma exceção checada ao testar converter uma string para inteiro.

```
def exception_example():  
    string_example = 'abcd'  
  
    try:  
        int(string_example)  
    except ValueError as err:  
        print(f"Ocorreu o seguinte erro de sintaxe: {err}")  
  
exception_example()
```

3.2 Exceções checadas

Nesta alteração nós estamos checando qual o tipo de erro que aconteceu para podermos tratar de forma diferente.

E também podemos criar erros totalmente personalizados para as necessidades da aplicação. Foi o que fizemos com o **CustomerError**.

```
class CustomerError(Exception):
    def __init__(self, name, message, code):
        self.name = name
        self.message = message
        self.code = code
        super().__init__(self.name, self.message, self.code)

def exception_check():
    customer_list = []

    try:
        if len(customer_list) < 1:
            raise CustomerError(
                name="CustomerListError",
                message="A lista de clientes não pode ser vazia",
                code=9000
            )
    except Exception as err:
        print(err)

exception_check()
```

3.3 Exceções de Banco de Dados

Quando estamos desenvolvendo aplicações que persistem dados, é importante preparar o fluxo para lidar com erros que podem ocorrer ao tentar alterar ou buscar dados.

No bloco de código abaixo estamos fazendo uma conexão e verificando se está disponível.

```
from pymongo import MongoClient
from pymongo.errors import ConnectionFailure

client = MongoClient()
try:
    client.admin.command('ismaster')
except ConnectionFailure:
    print("O servidor não está disponível")
```

3.3 Exceções de Banco de Dados

A forma de tratar os erros é bem similar a que já vimos. Porém, em se tratando de banco de dados, é necessário tomar todo cuidado para não gravar dados inconsistentes. Então toda vez que fizermos alguma alteração é muito importante tratar os erros que possam acontecer.

Um tratamento de erro clássico é quando tentamos inserir dados em mais de uma tabela em sequência e, em algum momento, acontece um erro. Nesse caso usamos o bloco `except` para fazer o `rollback` da transação.

3.4 Exceções HTTP

Erros em requisições HTTP também são tratados com try...except, porém as informações do erro são diferentes. O objeto gerado pelo erro sempre tem um código de status HTTP e esse código tem um significado que indica qual erro ocorreu.

```
import requests

try:
    r = requests.get('http://www.google.com/nothere')
    r.raise_for_status()
except requests.exceptions.HTTPError as err:
    raise SystemExit(err)
```


3.4 Exceções HTTP

O trecho de código mostrado no slide anterior retornaria a seguinte resposta:

404 Client Error: Not Found for url: <http://www.google.com/nothere>

Para tratarmos os erros precisamos conhecê-los muito bem. Só assim podemos fazer um tratamento eficiente para que a aplicação seja resiliente.

3.5 Log da aplicação

Já vimos até agora como capturar e utilizamos muito o `print()` para visualização. Esta não é uma boa prática, apesar de ser comum durante o desenvolvimento. Quando uma aplicação está publicada em produção nós temos que implementar recursos que salvem os erros em algum lugar para que possamos voltar e analisá-los.

Uma das formas mais conhecidas é gravar os erros em arquivos.

No código de exemplo está sendo implementada uma função que recebe um objeto e grava um arquivo JSON com o erro usando a data e hora do erro como nome.

```
import json
from datetime import datetime

def save_error(error):
    filename = f"error_log_{datetime.now()}.json"

    with open(filename, "a") as f:
        f.write(json.dumps({"error": error}))
        f.write(json.dumps("\n"))
        f.close()

save_error("Erro de teste")
```

3.5 Log da aplicação

Agora quando acontecer um erro podemos usar a função para gravar o erro em arquivo.

Importante lembrar que este é um exemplo simplificado de como é possível manipular as informações do erro para que possamos salvar o que aconteceu para futura análise.

```
def exception_example():  
    string_json = '{ "message": "Olá mundo!" }'  
  
    try:  
        object = json.loads(string_json)  
        print(json.dumps(object))  
    except Exception as err:  
        print(f"Erro capturado: {err}")  
  
        save_error(f"{err}")  
  
exception_example()
```

3.5 Log da aplicação

Agora quando acontecer um erro podemos usar a função para gravar o erro em arquivo. Também podemos refatorar a função para salvar o erro de outra forma.

Importante lembrar que esse é um exemplo simplificado de como é possível manipular as informações e salvar o que aconteceu para futura análise.

Outras formas de logar erros podem depender do tamanho da aplicação e do ecossistema onde ela atua. Alguns exemplos:

- Logstash - <https://www.elastic.co/>
Um serviço usado para gravação de logs de qualquer tipo além de erros. Após a gravação dos logs é possível criar dashboard e buscas que ajudam a medir vários aspectos da aplicação.
- Slack - <https://slack.com/>
Existem muitas formas de interagir com um canal do slack. Uma delas é a de utilizar a API de envio de mensagens. Com esse recurso, podemos enviar mensagens a um canal específico com o conteúdo do erro. Ainda é possível marcar pessoas e times para que sejam notificados assim que o problema ocorre.

3.5 Log da aplicação

- Banco de dados

Apesar de existirem serviços profissionais dedicados a persistência de logs, uma solução própria pode ser implementada com a gravação de logs em banco de dados. Porém deve ser bem planejada para não inserir um volume muito alto e degradar a performance. Para isso existem rotinas de limpeza de tabelas que podem apagar dados antigos.

A persistência de informações da aplicação é essencial pois uma vez que o problema ocorre fica impossível saber o que aconteceu a menos que exista um registro.

Mais importante ainda é que o registro contínuo das informações torna possível o acompanhamento da diminuição ou aumento da quantidade de erros que acontecem e se acontecem e períodos ou horários diferentes.

Uma aplicação de qualquer tamanho ganha muitos benefícios quando tem recursos bem planejados e executados de gravação de logs.



Perguntas?

Magalu



#VemSerFeliz