

# Erros e Exceções

- Aula 15 -  
Pensamento Computacional

Prof. Me. Lucas R. C. Pessutto



# Na aula de hoje...

01

## Escopo de Variáveis

Variáveis Locais x Globais

03

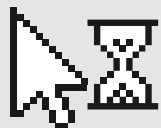
## Exceções

Comando try-except

02

## Erros

Tipos de Erros  
Debugging



# Variáveis Locais

- \* São instanciadas **dentro do escopo** de uma função
- \* Só podem ser referenciadas por comandos que estão dentro do mesmo escopo no qual elas foram instanciadas
- \* Existem **apenas enquanto o bloco** de código em que foram instanciadas está sendo executado



# Variáveis Locais

**Use variáveis locais, pois...**

- ... evitam confusão ao controlar os escopos
- ... economizam recursos do computador

# Variáveis Locais

**Lembre sempre que...**

... a troca de dados entre subprogramas deve ser feita somente por meio de **argumentos** e **retorno** de funções

# Variáveis Locais

Programa principal

```
a = 4
k = 9
...
X() # chama X
...
X() # chama X
...
```

4	9
a	k

principal

Função X

```
def X():
    a = 0 #local
    k = 5 #local
    ...
```

0	5
a	k

Função X

Vai para

Volta para

Uma função define **somente** variáveis locais

Variáveis locais não afetam os valores umas das outras, **mesmo que tenham o mesmo nome**

Quando a função acaba, suas variáveis locais **não são mais acessíveis**

# Variáveis Globais

- ★ São declaradas **fora do escopo** de uma função
- ★ Podem ser referenciadas **dentro do escopo de qualquer função**
- ★ Existem **durante toda a execução** do programa
- ★ Quando existe uma variável global e uma local com mesmo nome ativas no mesmo escopo, a variável local tem **prioridade** sobre a global
- ★ A primitiva **global** pode ser utilizada para modificar variáveis globais dentro do escopo de funções



# Variáveis Globais

**Evite utilizar variáveis globais!**



# Exemplo 1

```
def funcao1():  
    for i in range(1, cont):  
        print(".", end="")  
    print(f"\ncont funcao1 = {cont}")
```

*# Programa Principal*

*cont = 5 # cont pertence ao escopo principal*

*# Pode ser acessado pelo bloco interno de qualquer função*

```
print(f"cont principal = {cont}")  
funcao1()  
print(f"cont principal após funcao1 = {cont}")  
funcao1()  
print(f"cont principal após funcao1 = {cont}")
```

```
cont principal = 5  
....  
cont funcao1 = 5  
cont principal após funcao1 = 5  
....  
cont funcao1 = 5  
cont principal após funcao1 = 5
```

Variável cont do programa principal pode ser “acidentalmente” acessada na funcao1. Este tipo de utilização de variáveis NÃO segue o paradigma da programação estruturada e deve ser **EVITADO!**

## Exemplo 2

```
def funcao1():  
    cont = 10    # cont agora pertence ao escopo da funcao1  
    for i in range(1, cont):  
        print(".", end="")  
    print(f"\ncont funcao1 = {cont}")
```

```
# Programa Principal  
cont = 5    # cont pertence ao escopo principal  
print(f"cont principal = {cont}")  
funcao1()  
print(f"cont principal após funcao1 = {cont}")  
funcao1()  
print(f"cont principal após funcao1 = {cont}")
```

```
cont principal = 5  
.....  
cont funcao1 = 10  
cont principal após funcao1 = 5  
.....  
cont funcao1 = 10  
cont principal após funcao1 = 5
```

## Exemplo 3

```
def funcao1():
    global cont
    cont += 10 # cont agora é global
    for i in range(1, cont):
        print(".", end="")
    print(f"\ncont funcao1 = {cont}")

# Programa Principal
cont = 5 # cont pertence ao escopo principal
print(f"cont principal = {cont}")
funcao1()
print(f"cont principal após funcao1 = {cont}")
funcao1()
print(f"cont principal após funcao1 = {cont}")
```

```
cont principal = 5
.....
cont funcao1 = 15
cont principal após funcao1 = 15
.....
cont funcao1 = 25
cont principal após funcao1 = 25
```

# Erros

- \* Ao escrever e rodar programas é comum que programadores gerem erros
- \* Em programas complexos, testes chegam a gastar 50% do tempo de desenvolvimento
- \* É importante conhecermos mecanismos para nos ajudar a descobrir e tratar erros feitos no código
- \* Erros podem se manifestar em programas de três formas:
  - Erros sintáticos
  - Exceções
  - Erros semânticos



# Erros Sintáticos

\* Ocorre ao se desrespeitar as **regras sintáticas** da linguagem

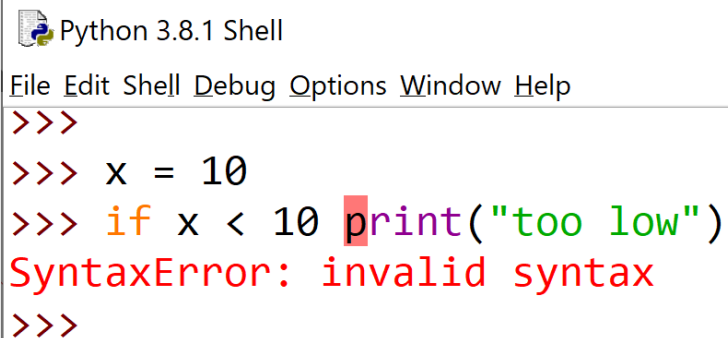
\* Exemplos:

→ Esquecer de delimitar um comando **if** com um ":"

→ Esquecer de fechar aspas ou parênteses abertos

\* Erro muito comum entre os programadores principiantes

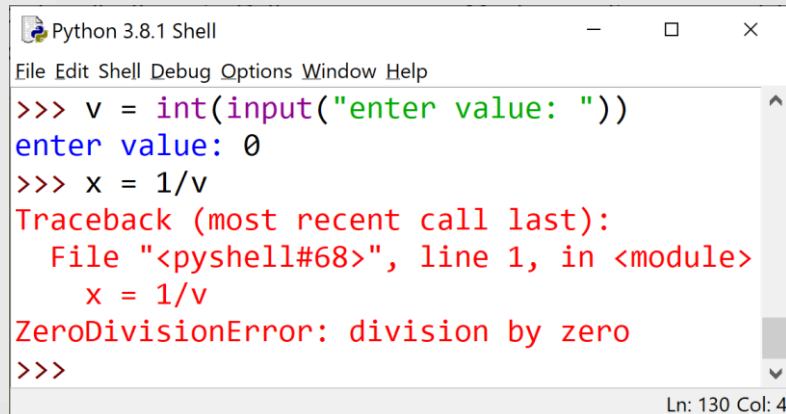
\* O interpretador Python indica um **SyntaxError** e destaca aproximadamente o local onde o erro ocorre



```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
>>>
>>> x = 10
>>> if x < 10 print("too low")
SyntaxError: invalid syntax
>>>
```

# Exceções

- \* Mesmo programas sintaticamente corretos podem apresentar erros durante a sua execução
- \* Exemplos:
  - Divisão por zero [**ZeroDivisionError**]
  - Problemas na conversão de tipos (converter um texto para int) [**TypeError**]
  - Errar a grafia do nome de uma variável ou função [**NameError**]
  - Falha ao acessar recursos externos (e.g., arquivos, urls) [**FileNotFoundError**]
- \* Python fornece mecanismos para detectar e tratar exceções em programas

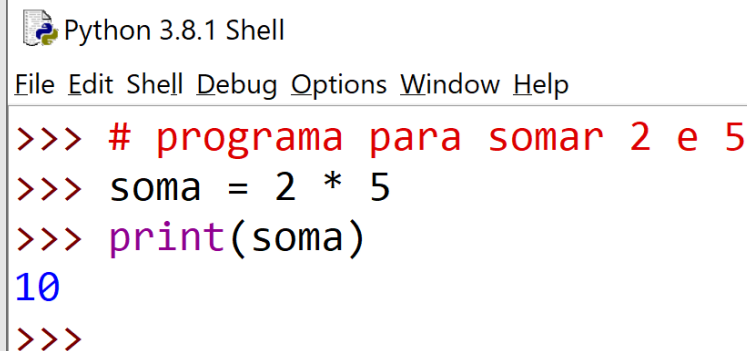


```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
>>> v = int(input("enter value: "))
enter value: 0
>>> x = 1/v
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    x = 1/v
ZeroDivisionError: division by zero
>>>
```

Ln: 130 Col: 4

# Erros Semânticos

- \* Programas “corretos” mas que não produzem o resultado esperado
- \* O programa executará normalmente
- \* Exemplos:
  - Erros na manipulação de dados
  - Expressões lógicas incorretas
  - Repetições infinitas
- \* São os erros mais difíceis de se detectar
- \* Existem ferramentas avançadas para teste de software que detectam esse tipo de erro

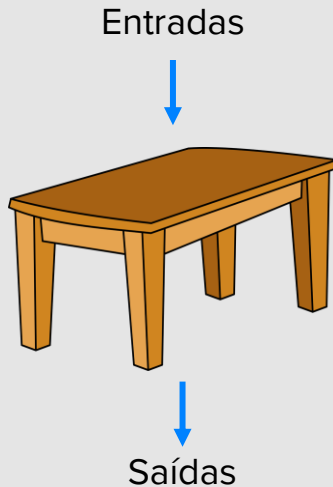
A screenshot of a Python 3.8.1 Shell window. The title bar says 'Python 3.8.1 Shell'. The menu bar includes 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The code entered is: >>> # programa para somar 2 e 5, >>> soma = 2 \* 5, >>> print(soma). The output is 10. This is a semantic error because the comment says the program is for adding 2 and 5, but the code multiplies them.

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
>>> # programa para somar 2 e 5
>>> soma = 2 * 5
>>> print(soma)
10
>>>
```

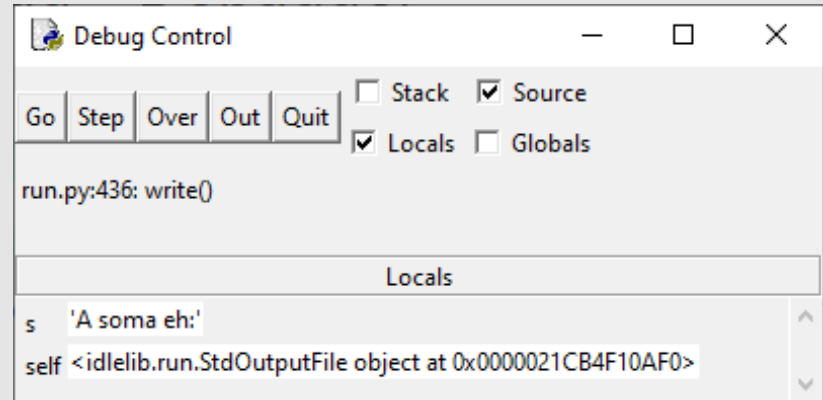
# Erros Semânticos

- \* Técnicas para ajudar na detecção de erros:

Teste de mesa



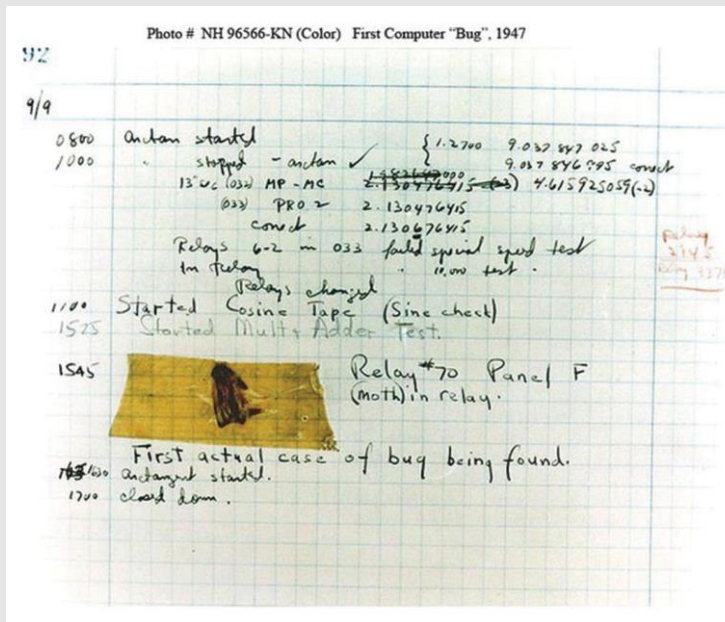
Debugging





# Debugging

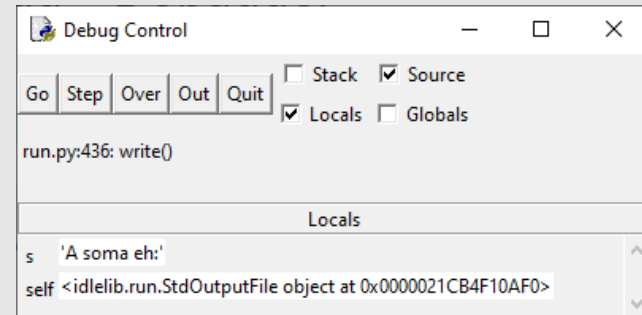
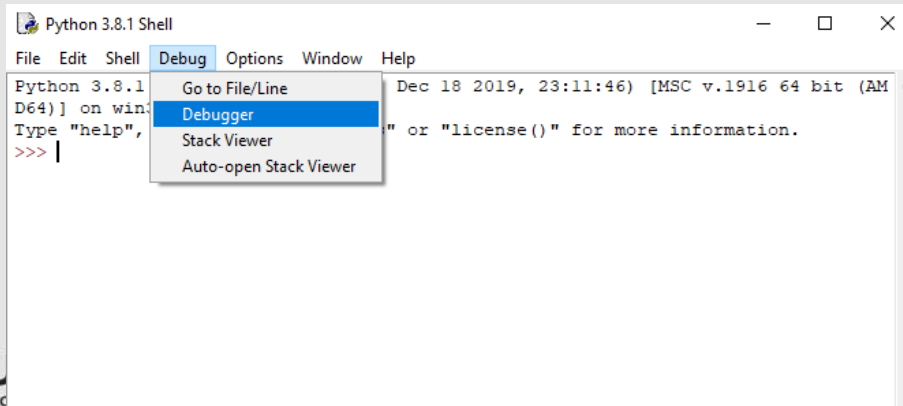
- \* Um problema não esperado em um programa é chamado de bug
- \* Debugging é o processo de encontrar e resolver bugs em um programa
- \* Um debugger fornece as seguintes ferramentas:
  - Controlar a velocidade da execução
  - Ajustar e resetar breakpoints (pontos de pausa) no programa
  - Visualizar os conteúdos das variáveis



Diário de William Burke, onde ele relata que encontrou uma mariposa presa entre os fios do computador Mark II

# Debugging

1. No Python IDLE Shell selecione Debug > Debugger
2. No Interpretador deverá aparecer a mensagem [DEBUG ON]
3. Uma janela chamada Debug Control deve aparecer:
  - A janela fornece os valores das variáveis quando o código está executando
  - Permite ver como os dados são manipulados em determinados trechos do programa



# Debugging

- \* Os seguintes botões ficam disponíveis ao executar um programa no debugger:
  - **Go**: avança a execução até o próximo breakpoint
  - **Step**: executa uma linha “passo-a-passo” incluindo chamadas de funções
  - **Over**: executa a linha atual e vai para a próxima “saltando” as chamadas de funções
  - **Out**: retorna da execução dos passos internos de uma função
  - **Quit**: interrompe a execução do programa
- \* Deixe selecionado nas checkboxes as opções “Locals” e “Source”
  - **Locals**: mostra os conteúdos das variáveis locais
  - **Source**: mostra qual trecho do código está sendo “debuggado” no momento
- \* Breakpoints são linhas de código escolhidas pelo programador que são pausadas durante a execução no modo debug
- \* São úteis para ver o estado das variáveis em determinados momentos escolhidos

# Debugging

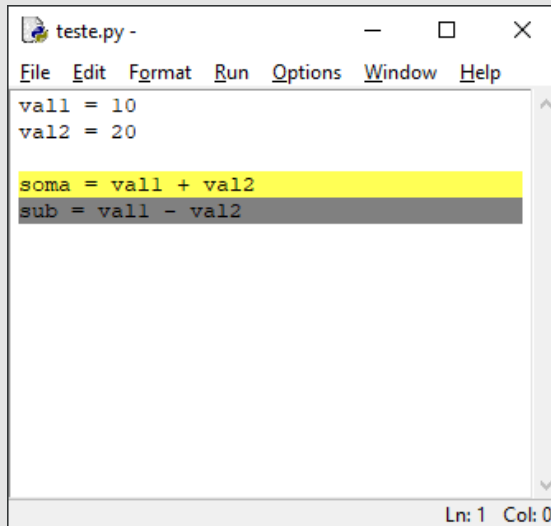
\* **Colocar** breakpoint no código:

→ Clique com o botão direito na linha desejada e selecione “Set Breakpoint”

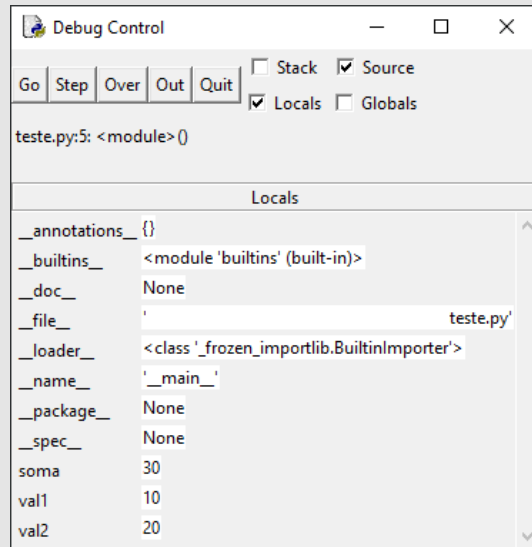
→ A linha deve ficar com cor amarela

\* **Remover** breakpoint do código:

→ Clique com o botão direito na linha desejada e selecione “Clear Breakpoint”



```
teste.py -
File Edit Format Run Options Window Help
val1 = 10
val2 = 20
soma = val1 + val2
sub = val1 - val2
Ln: 1 Col: 0
```



```
Debug Control
Go Step Over Out Quit
Stack Source
Locals Globals
teste.py:5: <module>()
Locals
__annotations__ {}
__builtins__ <module 'builtins' (built-in)>
__doc__ None
__file__ teste.py
__loader__ <class 'frozen_importlib.BuiltinImporter'>
__name__ '__main__'
__package__ None
__spec__ None
soma 30
val1 10
val2 20
Ln: 1 Col: 0
```

# Python Tutor

<https://pythontutor.com/python-debugger.html>

Online Python compiler and debugger - Python Tutor - Learn Python by visualizing code (also debug [JavaScript](#), [Java](#), [C](#), and [C++](#) code)

Write code in Python 3.6

```
1 valor = 25 # Trocar os comandos de input por atribuições diretas
2
3 div = 2
4 while valor > 1:
5     if valor % div == 0:
6         print(f"{valor:4} | {div}")
7         valor = valor // div
8         continue
9     div += 1
10
11 print(f"{valor:4}")
```

Visualize Execution

NEW: [subscribe](#) to our YouTube for weekly videos

hide exited frames [default]

inline primitives, don't nest objects [default]

draw pointers as arrows [default]

[Show code examples](#)

Generate permanent link

# Exceções (erros de execução)

```
val1 = int(input("Valor 1: "))
val2 = int(input("Valor 2: "))

result = val1 / val2
print(f"Resultado = {result}")
```

O que acontece quando se fornece essas entradas para o programa?

```
Valor 1: 8
Valor 2: 0

Traceback (most recent call last):
  File "teste.py", line 4, in <module>
    result = val1 / val2
ZeroDivisionError: division by zero
```

```
Valor 1: 1
Valor 2: dois

Traceback (most recent call last):
  File "teste.py", line 2, in <module>
    val2 = int(input("Valor 2: "))
ValueError: invalid literal for int()
with base 10: 'dois'
```

# Exceções (erros de execução)

- ★ Erros de execução típicos:
  - Divisão por zero
  - Fazer alguma operação com tipos incompatíveis
  - Utilizar um identificador que não foi definido
  - ...
- ★ Devem ser evitados ao máximo!
- ★ Poderiam ser evitados com o uso de `ifs`
  - São usados para tratar fluxos do programa
  - Piora a legibilidade do código
- ★ Python possui comandos especificamente feitos para tratar erros de execução

# Exceções (erros de execução)

```
import sys
```

```
val1 = input("Valor 1: ")  
val2 = input("Valor 2: ")
```

```
if val1.isdigit():  
    val1 = int(val1)  
else:  
    print("val1 é inválido!")  
    sys.exit()
```

```
if val2.isdigit():  
    val2 = int(val2)  
else:  
    print("val2 é inválido!")  
    sys.exit()
```

```
if val2 != 0:  
    result = val1 / val2  
    print(f"Resultado = {result}")  
else:  
    print("Denominador não pode ser zero!")
```

```
Valor 1: 8  
Valor 2: 0  
Denominador não pode ser zero!
```

```
Valor 1: 1  
Valor 2: dois  
Val2 é inválido!
```



# Comando try-except

- \* Bloco com o comando **try** testa o código procurando erros
  - Caso erros sejam encontrados, eles são tratados com o comando **except**
  - Após, o código segue normalmente
- \* Impede o Python de fechar o programa e exibir uma mensagem de erro padrão automaticamente
- \* Permite melhor **legibilidade** do código: erros são tratados separadamente

# Comando try-except

\* Sintaxe:

```
try:
    <comandos indentados>
except:
    <comandos indentados>
```

\* Exemplo:

```
try:
    print(x)
except:
    print("Algo deu errado")
```

Produz erro devido à variável x ser impressa sem ser previamente declarada

Saída é a mensagem customizada “Algo deu errado” em vez de erro

# Comando try-except

- \* Determinados tipos de erros podem ser tratados independentemente com comandos definidos
- \* Exemplo:

```
try:  
    print(x)  
except NameError:  
    print("Variável x não foi declarada")  
except:  
    print("Algo deu errado")
```

# Comando try-except

- \* Alguns dos principais tipos de erros:
  - **TypeError**: operação aplicada a um tipo incorreto
  - **ZeroDivisionError**: divisão por zero
  - **NameError**: variável não encontrada

# Comando try-except-else

- \* O comando else também pode ser utilizado após o except
  - O bloco apenas será executado se não houver erro
- \* Exemplo:

```
try:  
    print("Olá")  
except:  
    print("Algo deu errado")  
else:  
    print("Nada deu errado")
```

```
Olá  
Nada deu errado
```

# Comando try-except

\* Refazendo o exemplo com try-except:

```
import sys

try:
    val1 = int(input("Valor 1: "))
    val2 = int(input("Valor 2: "))
except ValueError:
    print("O valor informado é inválido!")
    sys.exit()

try:
    result = val1 / val2
    print(f"Resultado = {result}")
except ZeroDivisionError:
    print("Denominador não pode ser zero!")
```

# Comando try-except

- \* O programador pode causar uma exceção ao detectar que seu programa quando ocorre um erro.
- \* Para isso utilizamos o comando raise

```
x = "Olá"
```

```
if type(x) is not int:  
    raise TypeError("É permitido somente o uso de inteiros")
```

# Exemplo

```
# Garantindo que o número lido esteja no intervalo de 5 até 15
fim = False
while not fim:
    try:
        valor = int(input("Informe um número: "))
        if valor < 5 or valor > 15:
            raise ValueError("O número não está no intervalo correto!")
        fim = True
    except ValueError as excecao:
        print(f"Ocorreu um erro! {excecao}")
```