**Para saber mais: escopo de uma variável**

Em Python, o escopo de uma variável é definido pela região do código onde ela pode ser acessada. No caso de uma função, o escopo pode ser dividido em duas categorias: **escopo global** e **escopo local**.

O escopo global é o espaço no qual uma variável pode ser acessada por qualquer função ou código que esteja sendo executado no programa. Já o escopo local é o espaço no qual a variável pode ser acessada apenas pela função em que foi definida.

O problema de escopo ocorre quando uma variável é definida dentro do escopo de uma função e, em seguida, é referenciada fora do escopo da função. Nesse caso, o Python gera uma mensagem de erro, indicando que a variável não foi definida (NameError).

Abaixo segue um exemplo que ilustra esse comportamento. Inicialmente, vamos criar uma variável x externa a função soma(), na qual definimos uma outra variável y e, por fim, imprimimos a soma das duas variáveis.

x = 7

def **soma**():

y = 9

print(x + y)

Copiar código

Note que o x é a nossa variável definida no escopo global e o y a variável definida no escopo local da função soma(). Quando tentamos executar a nossa função, a soma é realizada normalmente:

soma()

Copiar código

**Saída:**

16

Copiar código

No entanto, o Python gera um erro quando tentamos imprimir a soma de x e y fora do escopo da função, pois a variável y existe apenas dentro da função soma().

print(x + y)

Copiar código

**Saída:**

---------------------------------------------------------------------------

NameError Traceback (most recent **call** **last**)

<ipython-input-4-f09a7b03ddbf> **in** <**module**>

----> 1 print(x + y)

NameError: name 'y' **is** **not** defined

Copiar código

Para corrigir esse erro, podemos tornar y uma variável global ou retornar seu valor na função e atribuí-lo a uma variável externa. No próximo vídeo, vamos aprender como proceder nesse tipo de situação.

**Para saber mais: trabalhando com tuplas**

As **tuplas** são estruturas de dados imutáveis da linguagem Python que são utilizadas para armazenar conjuntos de múltiplos itens e frequentemente são aplicadas para agrupar dados que não devem ser modificados. Ou seja, não é possível adicionar, alterar ou remover seus elementos depois de criadas. Vamos explorar um pouco mais desse tipo de estrutura voltada à aplicação em ciência de dados.

Tuplas são especialmente úteis em situações nas quais precisamos garantir que os dados não sejam alterados acidental ou intencionalmente. Por exemplo, em um conjunto de dados que representa o cadastro de estudantes, podemos utilizar uma tupla para representar aquele(a) estudante em específico e manter no banco de dados de uma instituição de ensino. Dessa forma, garantimos que as informações de cada estudante não sejam alteradas inadvertidamente.

Para criar uma tupla, basta separar seus elementos por vírgulas. Por exemplo, podemos criar uma tupla com um registro de uma estudante da seguinte maneira:

cadastro = ("Júlia", 23, "São Paulo", "SP", "Python para DS 1")

Para acessar os elementos de uma tupla, podemos usar o índice entre colchetes. Por exemplo:

print(cadastro[0]) # imprime Júlia

print(cadastro[-1]) # imprime Python para DS 1

Além disso, por também ser um iterável, podemos desempacotar os dados de uma tupla passando cada valor para uma variável. Por exemplo:

nome, idade, cidade, estado, turma = cadastro

E exibir os dados cadastrais da estudante:

print(f'A estudante {nome} tem {idade} anos e mora em {cidade}-{estado}. Ela está matriculada na turma de {turma}.')

**Saída: A** estudante Júlia tem 23 anos e mora **em** São Paulo-SP. Ela está matriculada na turma de Python para DS 1.

**Para saber mais: função zip**

A zip() é uma função embutida do Python que recebe um ou mais iteráveis (lista, string, dict, etc.) e retorna-os como um iterador de tuplas onde cada elemento dos iteráveis são pareados. Ela é útil para fazer iterações simultâneas em várias listas.

A função zip() pode ser usada em conjunto com outras funções do Python, como map() e filter(), para criar soluções elegantes e concisas para certos problemas. Vamos fazer um simples teste para verificar esse comportamento:

objeto\_zip = zip([1,2,3])

objeto\_zip

Copiar código

**Saída:**

<zip at 0x7f28fc5c0040>

Copiar código

Percebemos que o zip() criou um objeto zip na memória, que seria o nosso iterável. Vamos colocar o resultado em uma lista para verificar a saída:

list(objeto\_zip)

Copiar código

**Saída:**

[(1,), (2,), (3,)]

Copiar código

Note que, com apenas um iterável, uma lista de tuplas foi gerada com cada tupla possuindo, como um dos pares, os elementos vindo da lista [1, 2, 3] e a outra parte dos pares vazia. Como utilizamos apenas um iterável, cada tupla está vazia no 2º elemento, uma vez que o zip() age em criar pares de iteráveis.

Mas o mais interessante é trabalhar com dois ou mais iteráveis em que podemos pareá-los. Por exemplo, se quisermos criar uma lista de tuplas com o mapeamento das regiões do Brasil com seus respectivos ids:

id = [1, 2, 3, 4, 5]

regiao = ["Norte", "Nordeste", "Sudeste", "Centro-Oeste", "Sul"]

mapa = list(zip(id, regiao))

mapa

Copiar código

**Saída:**

[(1, 'Norte'), (2, 'Nordeste'), (3, 'Sudeste'), (4, 'Centro-Oeste'), (5, 'Sul')]

Para uma pessoa cientista de dados, essa função pode auxiliar a parear 2 listas distintas em um único objeto zip, podendo este ser transformado em uma lista de tuplas (formato ideal para gerar um índice de mais de um nível que será explorado em alguns dos cursos da formação) ou em um dicionário passando o objeto zip para a função dict().

Agora, se as listas de entrada têm comprimentos diferentes, a saída contém o mesmo número de tuplas que a lista de menor comprimento e os elementos restantes dos outros iteráveis serão ignorados. Por exemplo:

codigos = ["1000", "1001", "1002", "1003", "1004", "1005"]

frutas = ["maçã", "uva", "banana", "laranja"]

mercadorias = list(zip(codigos, frutas))

mercadorias

Copiar código

**Saída:**

[('1000', 'maçã'), ('1001', 'uva'), ('1002', 'banana'), ('1003', 'laranja')]

Copiar código

Para fazer o processo contrário, de transformar uma tupla iterável em listas, basta passar o operador asterisco (\*) ao lado esquerdo do nome da tupla iterável que quer extrair os dados, repassando cada tupla para uma variável.

tupla\_iteravel = [('J392', 'João'), ('M890', 'Maria'), ('J681', 'José'), ('C325', 'Claúdia'), ('A49', 'Ana')]

ids, nomes = zip(\*tupla\_iteravel)

ids = list(ids)

nomes = list(nomes)

print("IDs = ", ids)

print("Nomes = ", nomes)

**Saída:**

IDs = ['J392', 'M890', 'J681', 'C325', 'A49']

Nomes = ['João', 'Maria', 'José', 'Claúdia', 'Ana']

A ideia de fazer um “unzip” é bem-vinda quando queremos extrair chaves ou valores separadamente ou gerar uma lista de tuplas separadas, com o conjunto de chaves e valores cada uma representados em uma tupla.

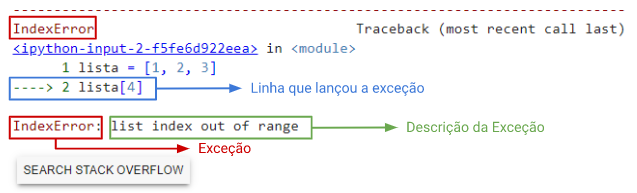
**Para saber mais: tipos de exceções**

No Python existem basicamente duas formas distintas de erros: os **de sintaxe** e as **exceções**. As exceções são uma forma de lidar com erros e situações inesperadas no código, garantindo um fluxo de execução mais controlado.

Como uma pessoa cientista de dados, você precisará ter atenção a situações como esta para evitar bugs ou problemas em seus códigos e análises que possam afetar a experiência tanto da pessoa usuária quanto a eficiência da sua análise.

### Como aparece uma exceção no Python:

Uma exceção aparece em nossa saída, como podemos notar na imagem abaixo:



Temos o **tipo da exceção**, seguido dos rastros que chamaram a exceção até a **linha exata que lançou a exceção** lista[4]. Logo abaixo, temos novamente o **tipo da exceção** e uma **breve descrição** sobre ela. Vamos agora observar alguns dos tipos mais comuns de exceções que podem ser lançadas no Python.

### Tipos de Exceções

#### SyntaxError

Ocorre quando é detectado pelo ***parser*** (analisador) um erro na descrição do código. Normalmente uma seta aponta para a parte do código que gerou o erro, como uma espécie de dica onde o erro possa ter ocorrido.

print(10 / 2

**Saída:**

File "<ipython-input-16-2db3afa07d68>", line 1

print(10/2

^

SyntaxError: unexpected EOF **while** parsing

Note que esquecemos de fechar o parênteses e por isso foi apresentado um erro de sintaxe, ou seja, de escrita de código.

#### NameError

Exceção lançada quando tentamos utilizar um nome de algum elemento que não está presente em nosso código.

raiz = sqrt(100)

**Saída:**

---------------------------------------------------------------------------

NameError Traceback (most recent **call** **last**)

<ipython-input-17-2e14e900fb9f> **in** <**module**>

----> 1 raiz = sqrt(100)

NameError: name 'sqrt' **is** **not** defined

Neste caso, o interpretador não consegue aplicar o método da raiz quadrada por não ter sido importado junto a biblioteca math.

#### IndexError

Exceção lançada quando tentamos indexar alguma estrutura de dados como lista, tupla ou até string além de seus limites.

lista = [1, 2, 3]

lista[4]

**Saída:**

---------------------------------------------------------------------------

IndexError Traceback (most recent **call** **last**)

<ipython-input-18-f5fe6d922eea> **in** <**module**>

1 lista = [1, 2, 3]

----> 2 lista[4]

IndexError: list index **out** **of** **range**

Para esta situação, temos apenas 3 elementos na lista e tentamos ler o elemento da posição 4, que não existe. Recebendo a mensagem de que o index está fora da faixa.

#### TypeError

Exceção lançada quando um operador ou função são aplicados a um objeto cujo tipo é inapropriado.

"1" + 1

**Saída:**

---------------------------------------------------------------------------

TypeError Traceback (most recent **call** **last**)

<ipython-input-20-ec358fc6499a> **in** <**module**>

----> 1 "1" + 1

TypeError: can **only** concatenate str (**not** "int") **to** str

Note que tentamos “somar” uma string com um número inteiro e isso gerou uma exceção em nosso código. Isso ocorreu por 2 razões: o operador de soma foi considerado de concatenação por iniciarmos utilizando uma string (nesse caso, o sinal de adição serve para concatenar strings) e um valor do tipo inteiro não consegue ser concatenado nesse tipo de operação.

#### KeyError

Exceção lançada quando tentamos acessar uma chave que não está no dicionário presente em nosso código.

estados = {'Bahia': 1, 'São Paulo': 2, 'Goiás': 3}

estados["Amazonas"]

**Saída:**

---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

<ipython-input-22-45729db26889> **in** <module>

1 estados = {'Bahia': 1, 'São Paulo': 2, 'Goiás': 3}

----> 2 estados["Amazonas"]

KeyError: 'Amazonas'

Tentamos acessar os dados do Estado Amazonas que não está presente no dicionário, lançando assim a exceção.

#### Warning

Exceção lançada em situações que precisamos alertar à pessoa usuária sobre algumas condições do código. Essas condições não necessariamente interrompem a execução do programa, mas podem lançar avisos sobre uso de módulos obsoletos, ou que possam ser depreciados em atualizações futuras ou também para alterações que podem reverberar sobre alguma parte do código.

Lembrando que, no caso dos ***Warnings*** eles podem ser ignorados ou tratados como exceções.

**import** numpy as np

a = np.arange(5)

a / a # apresenta um **warning**

**Saída:**

<ipython-input-23-93a37b275923>:4: RuntimeWarning: invalid value encountered in true\_divide

a / a # apresenta um **warning**

array([nan, 1., 1., 1., 1.])

Tentamos fazer a divisão de zero por zero. Em um array Numpy, que é essa estrutura na saída do console, esse resultado gera um valor nan (Not a Number). Ou seja, você consegue seguir a execução do programa, mas é provável que precise tratar os dados para poder utilizar esse array em alguma operação mais a frente.