***TÓPICOS***

1. [**Variáveis compostas**](#_Variáveis_Compostas)
   * 1. [**Criação de variáveis compostas**](#_Criação_de_uma)
     2. [**Variável composta como condição de parada**](#_Variável_composta_como)
     3. [**Comportamento de uma variável composta**](#_Comportamento_de_uma)
2. [**Tuplas**](#_Tuplas)
   * 1. [**Particularidades das tuplas**](#_Particularidades_das_tuplas)
     2. [**Soma das tuplas**](#_Soma_de_tuplas)
     3. [**Métodos para a classe tuple( )**](#_Métodos_para_a)
3. [**Listas**](#_Listas)
   * 1. [**Particularidades das listas**](#_Particularidades_das_listas)
     2. [**Métodos para a classe list( )**](#_Métodos_para_a_1)
     3. [**Compreensão de listas**](#_Compreensão_de_listas)
4. [**Dicionários**](#_Dicionários)
   * 1. [**Particularidades dos dicionários**](#_Particularidades_dos_dicionários)
     2. [**Métodos para a classe dict( )**](#_Métodos_da_classe)
     3. [**Compreensão de dicionários**](#_Compreensão_de_dicionários)
5. [**Conjuntos**](#_Conjuntos)
   1. [**Particularidades dos conjuntos**](#_Particularidades_dos_conjuntos)
   2. [**Métodos para a classe set( )**](#_Métodos_para_a_2)
6. [**Aplicações de variáveis compostas**](#_Aplicações_das_variáveis)
7. [**Funções**](#_Funções)
   * 1. [**Parâmetros nomeados e posicionais**](#_Parâmetros_nomeados_e)
     2. [**Argumentos arbitrários**](#_Argumentos_arbitrários)
     3. [**Parâmetros opcionais**](#_Parâmetros_opcionais)
     4. [**Escopo de variáveis**](#_Escopo_de_variáveis)
     5. [**Retorno de valores**](#_Retorno_de_valores)
     6. [**Função lambda**](#_Função_lambda)
8. [**Interactive help**](#_Interactive_help)
   * 1. [**Docstrings**](#_Docstrings)
9. [**Módulos e Pacotes**](#_Módulos_e_Pacotes)
   * 1. [**Módulos**](#_Módulos)
     2. [**Pacotes**](#_Pacotes)
10. [**Tratamento de erros**](#_Tratamento_de_erros)
    * 1. [**Lista com as principais exceções do Python**](#_Lista_com_as)

**PYTHON 3 - MÓDULO 3**

# **Variáveis Compostas**

## **Criação de uma variável composta**

Uma variável composta pode ser criada de três maneiras. Ex.:

1. **var = ( )** - Ao utilizarmos **parênteses**, criaremos uma tupla.
2. **var = [ ]** - Ao utilizarmos **colchetes**, criaremos uma lista.
3. **var = { }** - Ao utilizarmos **chaves**, criaremos um dicionário.

Para referenciarmos nossos objetos que estão armazenadas dentro de tuplas, listas e dicionários, utilizaremos os **colchetes**, independente de qual tipo seja a variável composta,

Primeiro, iremos focar nas tuplas.

Obs.: No Python 3, podemos criar tuplas sem a utilização de parênteses, porém, para melhor identificação, utilizaremos os parênteses em nossos códigos.

## **Variável composta como condição de parada**

**for c in var**

**print(c)**

O comando acima irá exibir cada um dos elementos da variável.

Ao utilizar a variável composta no comando var, temos certas facilidade e limitações. Veja no exemplo abaixo:

**for pos, cont in enumerate(tupla):  
    print(f'Posição {pos} - {cont}')**

No comando acima, só conseguimos mostras as posições dos objetos da tupla somente coma a ajuda da função **enumerate**, utilizando uma variável (pos) como au xílio.

## **Comportamento de uma variável composta**

O índice vai de 0 até um certo valor definido pelo usuário.

Ao manusear uma variável composta, o último índice é sempre **subtraído** por um. Ex.:

**string = 'teste'**

**string[1: ] = 'este'**

**string[0:4] = 'test'**

**string[-1] = 'e' (Última letra da string)**

**string[-5] = 't' (Primeira letra da string)**

# **Tuplas**

Variável com mais de um índice de armazenamento na memória do computador. Indo de 0 até um valor de sua preferência. Uma string é um exemplo de uma tupla, e várias palavras como objetos de uma tupla são exemplos de listas de letras.

## **Particularidades das tuplas**

As tuplas podem armazenas objetos de diferentes tipos, seja string, int, float ou boolean.

Tuplas são **imutáveis**. Não conseguimos alterar o valor de uma tupla durante a execução do programa.

## **Soma de tuplas**

Podemos realizar a soma de duas tuplas criando uma terceira para receber tal soma. Ex.:

**a = (1, 2, 3)  
b = (3, 2, 1)  
c = a + b  
print(c)**

O resultado será: (1, 2, 3, 3, 2, 1). Lembrando que 'a + b' é diferente de 'b + a'.

## **Métodos para a classe tuple( )**

Temos algumas funções que podemos estar utilizando para nos auxiliar no manuseio das tuplas. Abaixo, citaremos algumas delas.

* **enumerate('tupla')** - A função enumerate tem o papel de enumerar os objetos de uma tupla, lista ou dicionário, como foi visto anteriormente.
* **index('x', início, fim )** - Muito parecida com a função find, que só funciona ao ser utilizado em variáveis simples, a função index tem como objetivo identificar o índice mais próximo da informação passada por parâmetro. Funciona em todas as variáveis compostas.
* **del('var')** - A função é responsável por deletar uma variável (simples ou composta) da memória do computador.
* **max('var')** - A função max é responsável por identifica o maior valor numérico dentro da variável composta.
* **min('var')** - Com a função contrária ao max, a min exibe o menor valor numérico da variável composta.

Obs.: As tuplas são imutáveis, porém, conseguimos deletá-las no decorrer do nosso programa.

# **Listas**

Muito semelhantes às tuplas, as listas possuem uma única e grande diferença. São mutáveis.

Nas listas, podemos adicionar e remover informações, seja substituindo ou acrescentando informações.

Podemos criar uma lista das seguintes formas: Como o uso dos colchetes: **'lista' = [ ]** / Atribuindo a própria classe list: **'lista' = list( )**

## **Particularidades das listas**

Como já informado anteriormente, as listas são **mutáveis**, podendo ser alteradas ao longo do nosso programa.

Ao realizarmos a atribuição de uma lista para outra, nós também criamos um vínculo entre elas, onde qualquer alteração feita em uma delas, será aplicada na outra. Ex.:

**x = [1, 2, 3]  
y = x  
y[1] = 4  
print(x, y)**

No exemplo acima, tanto a lista x quanto a lista y, terão em seu armazenamento a sequência 1, 4, 3.

Para armazenarmos uma cópia de x em y, onde x não será alterado caso y seja, é utilizando o fatiamento da lista a ser atribuída. Ex.:

**x = [1, 2, 3]  
y = x[:]  
y[1] = 4  
print(x, y)**

No código acima, atribuímos na lista y, uma cópia de toda a lista x.

## **Métodos para a classe list()**

1. **.append('objeto')** - Adiciona um objeto passado por parâmetro ao final da lista.
2. **insert(i, 'objeto')** - Adiciona um objeto em uma posição específica, passada por parâmetro, dentro da lista.
3. **del 'lista'[i]** - Remove o objeto de uma lista na posição passada por parâmetro.
4. **.pop( )** - Elimina o último objeto da lista. Também podemos passar uma posição por parâmetro, mas não é tão utilizado dessa forma.
5. **.remove['objeto']** - Remove da lista, todos os objetos iguais ao que foi passado por parâmetro. Diferente das anteriores, não há como especificar uma posição.
6. **.sort( )** - Ordena uma lista em ordem crescente ou alfabética, dependendo dos objetos contidos nela. Também podemos utilizar da seguinte maneira: **.sort(reverse=True)**. Assim, iremos mostrar a lista de forma decrescente.
7. **sorted('tupla')** - Parecida com a função sort( ), a função sorted funciona em todas as variáveis compostas, porém não altera a variável em si. Muito utilizada para armazenar uma variável em outra variável.
8. **len('lista')** - Mostra o comprimento de uma variável composta.

# **Listas compostas**

As listas compostas são **listas** que possuem **outras listas** em seu conteúdo. Ex.:

**dadosPessoais = [['Matheus', 21], ['Daniel', 18], ['Leonardo', 17]]**

Também podemos atribuir uma lista diretamente a outra:

**pessoas = []  
dados = ['Matheus', 21]  
pessoas.append(dados[:])**

Da forma acima, atribuímos uma **cópia** da lista 'dados' à lista 'pessoas', onde 'pessoas' teria **['Matheus', 21]** como **índice 0**. Caso atribuíssemos a própria lista dados, toda a alteração feita ela seria **refletida** no **respectivo índice** da lista pessoas. Ex.:

**pessoas = []  
dados = ['Matheus', 21]  
pessoas.append(dados)  
dados.append('Davi')  
dados.append(16)  
print(pessoas)**

No comando acima, após **adicionarmos** informações na lista 'dados', o índice 0 da lista 'pessoas' **também é alterado**, ficando da seguinte forma: ['Matheus', 21, 'Davi', 16]

## **Exibição de listas compostas**

Seguindo a lógica acima, ao utilizarmos o comando:

**print(pessoas[0])**, será exibida a lista **['Matheus', 21]**.

Da mesma forma, com o comando:

**print(pessoas[0][0])**, irá exibir o índice 0 da lista de índice 0, no caso **'Matheus'**.

Com esse raciocínio, caso tenhamos mais listas, bastar selecionarmos o índice da que queremos exibir, e caso necessário, o índice do objeto dentro dessa lista.

## **Compreensão de listas**

Podemos preencher listas através da estrutura de repetição for, junto das estruturas condicionais:

**# Exemplo 1:**

**‘lista’ = list(range(50))**

**‘parOuImpar’ = ['Par' if c % 2 == 0 else 'Não é par' for c in l]**

**# Exemplo 2:**

**‘lista’ = [1, 2, 3, 4, 5]**

**‘listaInvertida’ = [c for c in reversed.(‘lista’)]**

# **Dicionários**

Os dicionários são semelhantes às listas, porém ao invés de somente índices numéricos, podemos também utilizar índices literais.

Podemos criar um dicionário das seguintes formas: Como o uso das chaves: **'lista' = { }** ou atribuindo a própria classe dict: **'dicionário' = dict( )**

## **Particularidades dos dicionários**

Para adicionarmos um elemento a nosso dicionário, ao contrário das listas, que utilizavam o método append( ), é necessário realizarmos a atribuição do objeto ao dicionário em questão.

Ex.:

**‘dicionário’ = { }**

**dicionário[‘novo índice’] = ‘valor’**

Para removermos um elemento, é utilizada a mesma função que vimos anteriormente: **del**

Ex.: **del ‘dicionário’[‘índice’]** - Junto do método, temos que passar por parâmetro o índice do dicionário que desejamos deletar.

Na atribuição dos dicionários, assim como nas listas, será necessário atribuirmos uma cópia do dicionário em questão. Caso contrário, relacionaremos o dicionário ao objeto que o recebe, onde toda alteração feita nele, será refletida no valor contido no objeto.

Diferentemente das listas, os dicionários possuem uma função responsável por tal feito. Essa função é o **.copy( ).**

## **Métodos para a classe dict( )**

1. **del ‘dicionário’[‘índice’]** - Como visto antes, é responsável por deletar um objeto de dentro de nosso dicionário.
2. **.update(‘dicionário’)** - Adiciona ou atualiza informações contidas em um dicionário. Caso a informação não exista, cria um índice com um valor, caso contrário, atualiza o valor do índice existente. Também funciona ao utilizarmos iteráveis com conjuntos de palavra-chave. Ex: lista[(índice, valor)]
3. **.values( )** - Seleciona somete os objetos existentes dentro de nosso dicionário. Ex.: **print(‘dicionário’.values( ))** irá exibir na tela somente os objetos de determinado dicionário.
4. **.keys( )** - Seleciona somente os índices dos objetos existentes dentro do dicionário referido. Ex.: **print(‘dicionário’.keys( ))** irá exibir na tela somente os índices de determinado dicionário.
5. **.items( )** - Seleciona todas as informações contidas dentro de um dicionário, tanto seus valores quanto seus índices. Ex.: **print(‘dicionário’.items( ))** irá exibir na tela ambas as informações contidas no determinado dicionário.
6. **.copy( )** irá fazer uma cópia do dicionário em questão, mantendo todas as informações contidas nele.

Podemos utilizar esses métodos em laços e repetição, assim como visto anteriormente. Ex.:

**for k, v in ‘dicionário’.items()**

**print(‘O {k} é {v}’)**

No exemplo acima, o primeiro termo serão as chaves, e o segundo os valores.

## **Compreensão de dicionários**

A compreensão de dicionários utiliza a mesma base da compreensão de listas. A única diferença é que devemos especificar a chave e o índice. Ex.:

**‘lista’ = [(‘chave1’, ‘índice1’), (‘chave2’, ‘índice2’), (‘chave3’, ‘índice3’)]**

**‘dicionário’ = [c: i for c: i in ‘lista’]**

Dessa forma, criaremos um dicionário com as tuplas presentes na lista, com as respectivas chaves e índices.

# **Conjuntos**

Os conjuntos são estruturas de dados capazes de armazenar todo o tipo de dado que seja imutável, como números, strings e tuplas.

Os conjuntos são criados atribuindo-se a própria classe set: **‘conjunto’** = **set()** ou através de colchetes: **‘conjunto’ = { }**, assim como os dicionários.

## **Particularidades dos conjuntos**

Os conjuntos possuem uma particularidade muito útil em certas situações: Não aceitam objetos repetidos. Caso tentarmos atribuir um objeto já existente no conjunto em questão, o objeto não será atribuído.

Também podemos criar um conjunto através de uma lista, ou converter uma lista em um conjunto. Ex.:

**#Criando conjunto a partir de uma lista:**

**‘lista’ = [1, 2, 3, 4, 5]**

**‘conjunto’ = set(‘lista’)**

**#Convertendo uma lista em um conjuto:**

**‘lista\_conjunto’ = [1, 2, 3, 4, 5]**

**‘lista\_conjunto’ = set(‘lista\_conjunto’)**

Obs: Todos os elementos duplicados a mais serão ignorados.

Ou até mesmo através de uma string:

**‘string’ = ‘Python’**

**‘conjunto’ = set(‘string’)**

Cada elemento da string será passado para o conjunto em questão.

## **Métodos para a classe set( )**

1. **add( )** - Adiciona um único elemento ao conjunto. Caso esse elemento já esteja contido no conjunto, a ação é ignorada.
2. **update( )** - Adiciona um iterável (Listas, Tuplas, Dicionários e Conjuntos) ao conjunto. Somente os elementos não existentes no conjunto em questão serão adicionados.
3. **remove( )** - Remove todo elemento igual ao passado por parâmetro. Caso não encontre nenhum elemento, retorna uma mensagem de erro.
4. **discard( )** - Remove o elemento contido no índice passado por parâmetro.
5. **pop( )** - Remove o primeiro elemento contido no conjunto. Caso o conjunto esteja vazio, retornará uma mensagem de erro.
6. **clear( )** - Limpa o conteúdo do conjunto, deixando-o vazio.
7. **‘cojunto1’.intersection(‘conjunto2’)** - Realizará a interseção de dois conjuntos, criando um terceiro apenas com os elementos existentes nos outros dois. Também podemos realizar a operação com o operador |. Ex.: ‘conjunto1’ | ‘conjunto2’.
8. **‘conjunto1’.difference(‘conjunto2’)** - Criará um terceiro conjunto contendo a diferença entre dois outros conjuntos. O primeiro conjunto será a base de comparação. Também podemos realizar a operação com o operador -. Ex.: ‘cojunto1’ - ‘conjunto2’.
9. **‘conjunto1’.symmetric\_difference.(‘conjunto2’)** - Criará um terceiro conjunto contendo apenas os valores exclusivos entre dois outros conjuntos. Também podemos realizar a operação com o operador ^. Ex.: ‘conjunto1’ ^ ‘conjunto2’.
10. **‘conjunto1’.symmetric\_difference\_update.(‘conjunto2’)** - Possui a mesma função do método anterior, porém ao invés de criar um terceiro conjunto, irá alterar o primeiro. Também podemos realizar a operação com o operador ^=. Ex.: ‘conjunto1’ ^ ‘conjunto2’.

# **Aplicações das variáveis compostas**

Tendo em mente toda as variáveis compostas apresentadas anteriormente, podemos utilizá-las separadamente ou em conjunto.

Podemos criar uma lista com vários dicionários como elementos ou um dicionário com lista em cada um dos índices. As possibilidades são inúmeras e vão depender da ocasião e necessidade de seu programa.

**Funções**

As funções o papel de eliminar rotinas em nosso código. Em Python, podemos criar funções com funções específicas de acordo com nossa necessidade. Para isso, utilizamos a palavra-chave **def**:

**def exibir(x):**

**print(x)**

**Programa principal**

O x passado por parâmetro na função ‘**exibir( )**’ (Parâmetro formal), será o objeto passado por parâmetro ao chamar a função no programa principal (Parâmetro real). No PyCharm, é aconselhado adicionarmos duas linhas entre o def e o programa principal. Ex.:

**#Programa principal**

**frase = ‘Uma frase qualquer’**

**exibir(frase)**

Como resultado, a frase passada por parâmetro será exibida na tela através da função exibir.

## **Parâmetros nomeados e posicionais**

É possível passarmos quantos parâmetros forem necessários, desde que a quantidade respectiva seja criada junto a função. Podemos passar os parâmetros de duas maneiras: Nomeados e Posicionais.

Parâmetros nomeados

**def dividir(x, y):**

**divisao = x / y**

**print(divisao)**

**#Programa principal**

**num1 = 0**

**num2 = 1**

**divisao(y = num1, x =num2)**

Obs.: Após informarmos o primeiro parâmetro nomeado, todos adiante também devem ser nomeados

Parâmetros posicionais

**def dividir(x, y):**

**divisao = x / y**

**print(divisao)**

**#Programa principal**

**num1 = 0**

**num2 = 1**

**divisao(num1, num2)**

Ao contrário do exemplo acima, o x será o num1 e y num2.

**Argumentos arbitrários**

Quando desejamos passar uma quantidade incerta de valores para a nossa função, devemos utilizar o símbolo de **\*** no parâmetro de nossa função criada. Com isso, será criada uma tupla com os valores informados, onde todas as regras de uma tupla serão aplicados ao parâmetro. Ex.:

**def funcao(\*numeros):**

**for num in números:**

**print(num)**

No exemplo acima, serão recebidos quantos elementos forem necessários, e logo em seguida, exibidos na tela com o auxílio de uma estrutura de repetição.

Caso passemos uma lista, devemos utilizar \* ao passar a lista por parâmetro, para que essa seja desempacotada e por fim, recebida pela função. Ex.:

**def funcao(\*numeros):**

**for num in números:**

**print(num)**

**lista\_numeros = [1, 2, 3, 4, 5]**

**funcao(\*lista\_numeros)**

## **Parâmetros opcionais**

Ao criarmos uma função, temos a opção de estabelecermos parâmetros opcionais. Com eles, a função pode ou não receber tal parâmetro quando chamada na função principal. Ex.:

**def função(a, b, c=0):**

Na função acima, o parâmetro ‘c’ é um parâmetro opcional. Caso não seja informado um terceiro valor no programa principal, ‘c’ receberá 0 e a função será executada normalmente. Lembrando que o valor do parâmetro opcional pode ser escolhido livremente de acordo com sua necessidade.

## **Escopo de variáveis**

Trabalhando com funções, temos sempre que nos atentar ao escopo que nossas variáveis pertencem. A seleção correta dos escopos ao criarmos uma variável é fundamental para o funcionamento de nosso programa. Temos dois tipos de escopo:

1. Escopo local: No escopo local, toda variável contida dentro de uma função, poderá ser utilizada somente dentro da mesma.
2. Escopo global: Já no escopo global, se enquadram todas as variáveis criadas em nosso programa principal.

Ex 1:

**def função(b):**

**a = 2**

**b += 3**

**print(a)1**

**print(b)2**

**#Programa principal**

**a = 1**

**print(a)3**

Ao criarmos uma variável de mesmo nome tanto no escopo local quanto no global, a variável de escopo local será a utilizada dentro da função.

No código acima, o primeiro print irá exibir o **número 2**, o segundo o **número 4** e o terceiro o **número 1**.

Caso queira que a variável global seja a utilizada, deve utilizar a palavra-chave **global. Ex 2:**

**def função(b):**

**global a**

**a = 2**

**b += 3**

**print(a)1**

**print(b)2**

**#Programa principal**

**a = 1**

**print(a)3**

Utilizando a palavra-chave **global**, toda alteração realizada na variável **a**, dentro da função, será refletida no escopo global.

Desta vez, o primeiro print irá mostrar o número 2, o segundo print o número 5, e o terceiro print o número 2 novamente.

Obs.: Assim como as variáveis, importações de módulos e bibliotecas também serão limitados depende do escopo onde estão. Para as importações poderem ser utilizadas em todo o programa, devem ser postas no início dele.

## **Retorno de valores**

Uma função pode ou não retornar um valor. Ao retornarmos um valor de uma função, devemos armazená-la em uma variável em nosso programa principal. Conseguimos retornar um valor utilizando a função **return** ao final da função.

Ex.:

**def somar(a, b):**

**soma = a + b**

**return soma**

**#Programa principal**

**resp = soma**

**print(resp)**

Dessa forma, a variável **resp** irá receber o valor retornado (**soma**) pela função somar, exibindo-o na tela.

## **Função lambda**

Uma função lambda é uma função anônima, ou seja, não possui nome. Com a função lambda, atrelamos uma função a uma variável. Ela pode ser chamada pela expressão lambda ‘argumento(s)’: ‘expressão’. Ex.:

**funcLam = lambda x, y: x \* y**

**num1 = 2**

**num2 = 3**

**print(funcLam(num1, num2)**

Podemos também utilizá-la para ordenar listas, através da função sort( ). Ex.:

**‘lista’: [[‘a’, 1], [‘b’, 2], [‘c’, 3]]**

**‘lista’.sort(key=lambda i: i[0 ou 1])**

Com 0, o primeiro índice será levado em consideração ao ordenar (Ordem alfabética), já com 1, o segundo índice (Ordem numérica crescente).

# **Interactive help**

Em Python, temos o modo de ajuda interativa, onde conseguimos obter a documentação de diversos objetos em Python, como: Bibliotecas, Módulos, Classes e Funções. Para isso, utilizamos a função embutida **help( )**.

Ela pode ser utilizada de duas maneiras:

1. Console Python: Indo até o console Python e inserindo a função help( ), iremos entrar no modo de ajuda iterativa. Em seguida, basta informarmos o objeto que desejamos obter informações.
2. Via editor: Ao inserirmos a função help( ), passando um objeto por parâmetro, e rodarmos nosso código, serão exibidas as informações de tal objeto.

## **Docstrings**

As informações dos objetos que são transmitidas pela função help( ), são chamadas de **docstrings**, ou seja, **strings de documentação**.

Sendo assim, quando criamos uma função, também é importante criarmos sua documentação, para instruirmos outros que forem utilizá-la. Para isso, é só abrir e fechar aspas duplas três vezes, digitando as informações entre elas:

**def funcao(x, y, z):**

**“““**

**informações da função**

**”””**

# **Módulos e Pacotes**

## **Módulos**

Além dos módulos já embutidos no PyCharm, temos como criar módulos contendo nossas próprias funções. Para isso, basta criarmos um arquivo .py com nossas funções e o importarmos em um outro arquivo onde se faz necessário o seu uso. Para importá-lo, utilizamos um comando já visto anteriormente:

**import modulo**

Também conseguimos importar uma função específica desse módulo com:

**from modulo import funcao**

Algumas vantagens que podemos citar ao utilizar modularização é:

1. Organização do código
2. Facilidade na manutenção
3. Ocultação de código detalhado
4. Reutilização em outros projetos

## **Pacotes**

Quando nosso programa se torna muito extenso, e somente os módulos não são suficientes, podemos utilizar os chamados pacotes. Neles, teremos módulos que irão separar as funções de acordo com suas utilidades. Ex.:

1. **Pacote**
   1. **Módulo A**
      1. **Função a**
      2. **Função b**
   2. **Módulo B**
      1. **Função c**
      2. **Função d**
      3. **Função e**

Para criarmos um pacote no PyCharm, devemos clicar com o botão direito em nosso diretório e selecionar a opção ‘New’ -> ‘Python Package’. Podemos também criar mais pacotes dentro de outros pacotes, tudo de acordo com a necessidade do nosso programa.

# **Tratamento de erros**

Ao tratarmos erros em Python, devemos utilizar a estrutura try-except:

**try:**

**bloco de comando**

**except:**

**print(‘Não foi possível executar o comando acima’)**

Caso não seja possível executar o bloco de comandos passados ao try, o except é executado. As palavras-chave **try** e **except** são obrigatórias ao utilizarmos a estrutura de tratamento de exceções, mas ainda temos **else** e **finally**, que podem ser utilizadas em conjunto das outras.

* **try**: Feita tentativa de execução do bloco de comando.
* **except**: Executado quando o try retorna um erro (exceção).
* **else**: Executado quando try funciona corretamente.
* **finally**: Executado independente do try retornar ou não um erro (exceção).

Todas as exceções contidas em Python fazem parte de uma classe chamada **Exception**. Podemos utilizar essa classe juntamente da palavra-chave **except**, para indicarmos qual o erro apresentado. Ex.:

**except Exception as ‘objeto’:**

**print(‘Não foi possível realizar o comando acima. O erro apresentado foi:’, objeto.\_ \_class\_ \_)**

Em nossa estrutura de repetição de dados, podemos ter vários **except**, cada um para uma exceção em específico. Ex.:

**Except TypeError**

**Except ValueError**

**Except etc**

## **Lista com as principais exceções do Python**

1. **SyntaxError**: Ocorre quando há um erro de sintaxe no código, indicando que o interpretador Python não pode entender a estrutura do programa.
2. **IndentationError**: Ocorre quando a indentação do código não está correta. A indentação em Python é crucial para definir blocos de código.
3. **TypeError**: Ocorre quando uma operação é realizada em um tipo de dado incompatível. Isso pode incluir operações entre tipos diferentes, como tentar somar uma string a um número.
4. **NameError**: Ocorre quando um nome de variável ou função não é encontrado no escopo atual.
5. **ValueError**: Ocorre quando uma função recebe um argumento do tipo correto, mas com um valor inapropriado. Por exemplo, tentar converter uma string não numérica para inteiro.
6. **IndexError**: Ocorre quando você tenta acessar um índice que está fora do alcance de uma sequência, como uma lista ou tupla.
7. **KeyError**: Ocorre quando você tenta acessar uma chave que não existe em um dicionário.
8. **FileNotFoundError**: Ocorre quando um arquivo que você está tentando acessar não é encontrado.