## Documentação de Pyhton

A melhor referência em documentação para a linguagem Python pode ser acessada no link: [3.12.4 Documentation (python.org)](https://docs.python.org/3/)

Aqui pode-se obter toda a referência da linguagem, lembrando que a opção em inglês é a melhor pois não provoca entendimentos equivocados. Pode ser acessada também com a opção em *“Brazilian Portuguese”.*

Também é acessível na versão que o desenvolvedor ou estudante esteja utilizando.

As seções Tutorial e referência da Biblioteca bem como a Referência da linguagem são os caminhos mais buscados e importantes para entender bem o Python e resolver diversas dúvidas.

## Palavras reservadas

As seguintes palavras são reservadas e devem ser evitadas no uso de identificadores, variáveis, classes, comandos e funções.

False await else import pass

None break except in raise

True class finally is return

and continue for lambda try

as def from nonlocal while

assert del global not with

async elif if or yield

Você sempre pode checar quais são. Na sua IDE (Exemplo: Google Colab), você pode rodar o código help("keywords") que retornará uma lista de palavras-chave do python da versão que está usando.

Além destas palavras há a palavra “type” que é uma palavra reservada contextual, isto é, para alguns contextos específicos, *type, match e case* são palavras assim e por isso devemos evitá-las.

|  |  |
| --- | --- |
| False | Tipo de dado booleano que indica que uma condição é Falsa. |
| None | Palavra reservada para definir um valor nulo (null) para uma variável.  É similar a atribuir 0, False ou empty (vazio) a uma variável |
| True | Tipo de dado booleano que indica que uma condição é Verdadeira |
| and | Operador logico para uso em condições  A > B and A > C; A = C and B = C |
| as | Cria um alias (apelido) para uma definição qualquer, muito usado em referências a outros módulos.  Import calendar as calend |
| assert | Utilizado para depuração de códigos até em tempo de execução.  X = “hello”  Se uma condição retornar True então nada acontecerá:  assert x == hello”  Se uma condição retornar False então uma AssertionError será retornada  assert x == “goodbye” |
| async |  |
| await |  |
| break | Serve para quebrar uma execução de um loop ou um bloco de programa.  *for i in range(9):*  *if i > 3:*  *break*  *print(i)* |
| class | Para definir uma classe em Python. Ver OOP Criando uma classe do tipo pessoa  *class Pessoa:*  *nome = “João”*  *idade = 18*  *est\_civil = “Casado”* |
| continue | Para continuar com a próxima iteração dentro de um loop  *for i in range(9):   if i == 3:     continue   print(i)* |
| def | É utilizado para definir uma função  *def my\_function():   print("Hello from a function")  my\_function()* |
| del | Deleta um objeto, á casos raros em que se cria uma classe em tempo de execução e após algum processamento deleta a mesma para economia de memória ou evitar sobras de memória expostas a ataques de hackers.  Pode ser usado para deletar também variáveis, listas, ou partes de listas.  *class MyClass:   name = "John"  del MyClass  print(MyClass)*  *##Deletar o primeiro item de uma list :*  *x = ["apple", "banana", "cherry"]  del x[0]  print(x)* |
| elif | Utilizado dentro de condições como um novo teste se o if não for satisfeito.  *for i in range(-5, 5):   if i > 0:     print("YES")   elif i == 0:     print("WHATEVER") #nenhum e nem outro!   else:     print("NO")* |
| else | Se uma condição no if não passar, então executa-se o que estiver no bloco do else.  *x = 2 if x > 3:   print("YES") else:   print("NO")* |
| except | Utilizado quando uma exceção ocorre durante a execução de um programa. Muitas vezes o desenvolvedor identifica que uma “exception” pode ocorrer então com um “except” pode contornar a situação ou identificar o que pode estar acontecendo.  *try:   x > 3 except:   print("Something went wrong")* |
| finally | Usada com exceções ( exceptions), um bloco de código que irá executar se ocorrer uma exceção ou não! Um bloco “finally“ será executado sempre que uma exceção ocorrer ou não, mas precisa ser implementada: o bloco será executado não importando se o bloco try gerou ou não um erro. É útil para fechar objetos e limpar recursos que possam ficar na memória!  *try:   x > 3 except:   print("Something went wrong") else:   print("Nothing went wrong") finally:   print("The try...except block is finished")* |
| for | Para criar um loop. Estrutura de repetição.  *fruits = ["apple", "banana", "cherry"]*  *for x in fruits:   print(x)*  e também:  *for x in range(1, 9):   print(x)* |
| from | Para importar partes específicas de um modulo.  Importando apenas a seção time do modulo datetime, e imprimir a hora assim como ela está  *From datetime import time  x = time(hour=15)  print(x)*  *Irá imprimir 15:00:00* |
| global | Para declarar uma variável global  Declarar uma variável global dentro de uma função e usá-la fora da função  *#create a function: def myfunction():   global x   x = "hello"  #execute the function: myfunction()  #x should now be global, and accessible in the global scope. print(x)*  *Um cuidado deve se ter aqui! Criar novas variáveis globais numa função não é problema, a atenção é não manipular esta variável desorganizadamente fora do escopo da função, pois podem ter impactos se a mesma variável foi definida em outro escopo!* |
| if | Para criar uma declaração condicional (se...)  Imprimir “Yes” se x for maior que 3:  *x = 5 if x > 3:   print("YES")*  ***#outro exemplo:***  *x = 5 if x > 6:   print("YES") else:   print("NO")* |
| import | Para importar um módulo por inteiro.  Importar o módulo datetime e exibir a data e hora corrente.  *import datetime  x = datetime.datetime.now() print(x)* |
| in | Para verificar se um valor está presente numa lista, numa tupla, etc.  Checar se “banana” está presente em uma lista:  *fruits = ["apple", "banana", "cherry"]  if "banana" in fruits:   print("yes")*  ***\*\*\* A palavra reservada “in” tem dois propósitos:***  ***Utilizada para checar se um valor está presente num sequencia (list, range, string, etc)***  ***Utilizada também para iterações através de um sequência com o uso de um loop for:***  *fruits = ["apple", "banana", "cherry"]  for x in fruits:   print(x)* |
| is | Para testar se duas variáveis são iguais  *x = ["apple", "banana", "cherry"]  y = x  print(x is y) # retorna True*  *!!! a palavra reservada “is” é utilizada para testar se 2 variáveis referem-se ao mesmo objeto. O teste retorna True se os 2 objetos são o mesmo objeto.*  *E o teste retorna False se eles não são os mesmos objetos, mesmo que os objetos sejam 1005 iguais.*  *x = ["apple", "banana", "cherry"]*  *y = ["apple", "banana", "cherry"]*  *print(x is y) # retorna False* |
| lambda | Para criar uma função anônima. Esta palavra-chave é usada para criar pequenas funções anônimas.  Uma função com lambda pode assumir qualquer número de argumentos, mas só pode ter uma expressão.  A expressão é avaliada e o resultado é retornado.  Exemplos:  *x = lambda a : a + 10  print(x(5)) #irá imprimir 15*  *--------outro exemplo:-------*  *x = lambda a, b, c : a + b + c  print(x(5, 6,3)) #irá imprimir 14* |
| nonlocal | Para declarar uma variável não local.  A palavra-chave é usada para trabalhar com variáveis dentro de funções aninhadas, onde a variável não deve pertencer à função interna.  Use a palavra-chave para declarar que a variável não é local  Exemplos: Criar uma função dentro de uma função, que usa a variável x como uma variável não local;  *def myfunc1():   x = "John"   def myfunc2():     nonlocal x     x = "hello"   myfunc2()   return x  print(myfunc1()) #iráimprimir hello*  *---- a mesma função mas sem o nonlocal*  *def myfunc1():*  *x = "John"*  *def myfunc2():*  *x = "hello"*  *myfunc2()*  *return x*  *print(myfunc1()) #retorna John*  \*\*\*\* nota \*\*\*\*  A declaração nonlocal no Python é usada para referenciar variáveis em escopos externos ao escopo local, mas ainda dentro do escopo da função. Isso é útil quando você precisa modificar uma variável que está em um escopo superior ao da função atual. Por exemplo:  x = 0  def outer():  x = 1    def inner():  nonlocal x  x = 2  print("inner:", x)    inner()  print("outer:", x)  outer()  print("global:", x)  # Saída: inner: 2  # outer: 2  # global: 0  **Sem o uso de nonlocal, a variável x dentro de inner() seria uma nova variável local, não afetando a variável x em outer().**  **Já com nonlocal, a variável x em inner() é a mesma que a de outer().** |
| not | Operador lógico de negação. Retorna True se a declaração não for verdadeira.  *x = False print(not x) #retorna True*  A palavra reservada not, é um operador lógico. O valor de retorno será **True** se a declaração **não for True**, caso contrário retornará **False**. |
| or | Operador lógico para combinar 2 ou mais condições!  Retorna True se uma das condições ou declarações forem True.  *x = (5 > 3 or 5 > 10)  print(x) # Retorna True. Uuma condição sendo verdadeira faz com que a condição inteira seja verdadeira!!!* |
| pass | Uma instrução nula, uma declaração que não fará nada.  Pode ser utilizada para reservar um espaço para um fragmento de código que ainda não foi definido totalmente, mas precisa ser referenciado por enquanto. Exemplo:  *For x in [0, 1, 2]:   pass*  *\*\*\* Quando uma instrução”pass” é executada, nada acontece, mas você pode receber um erro quando o código vazio não é permitido. Códigos vazios não são permitidos em loops, definições de função ou em declarações “if”.*  *Ou seja sem a instrução pass teríamos um looping no exemplo que resultaria em erro (código vazio). Outro exemplo:*  *def myfunction():*  *pass*  *#reservou-se um definição de função mas ainda não tem sua implementação definida, contudo ela pode estar referenciada em outro fragmento de código, neste caso o” pass” garante que não será devolvido um erro!*  *Outro exemplo:*  *Usando a palavra reservada pass numa declaração “if”:*  *a = 33 b = 200  if b > a:   pass*  *#retorna nada !* |
| raise | Utilizada para abrir uma exceção. Voce pode definir que tipo de erro de exceção irá gerar e o texto para imprimir ao usuário.  *X = -1  if x < 0:   raise Exception(“Sorry, no numbers below zero”)*  *Outro exemplo:*  *#Raise a TypeError if x is not an integer:*  *x = "hello"*  *if not type(x) is int:*  *raise TypeError("Only integers are allowed")* |
| return | Palavra reservada para sair de uma função e retornar um valor.  *def myfunction():   return 3+3  print(myfunction())*  *\*\*\* Atenção!!! Instruções ainda que no bloco da função, mas colocadas após a palavra reservada “return”; não serão executadas:*  *def myfunction():*  *return 3+3*  *print("soma esta correta")*  *print(myfunction())* |
| try | Para criar uma declaração try ... excepetion (tentar...exceção) muito utilizada para tratamento de error de exceção!  Exemplo: pegar um bloco de código e decidir o que fazer se ele gerar um erro  try:   x > 3 except:   print("Something went wrong") #imprime “algo deu errado” e sai do bloco com mensagem para auxiliar o tratamento, paralisando a execução.  Você pode definir blocos diferentes para diferentes situações onde tipos de erros e blocos sejam executados se nada der errado. Ver outros exemplos mostrados em “exception” e “finnaly”  Try gera um bloco de exceção, para manipularmos o tratamento do erro de exceção:  *try:   print(x) except NameError:   print("Variable x is not defined") except:   print("Something else went wrong")*  #agora um exemplo com “finally” :  *try:   print(x) except:   print("Something went wrong") finally:   print("The 'try except' is finished")*  # Aqui um outro exemplo quando tratamos de abertura de arquivos: para open e para write em arquivos.  *try:   f = open("demofile.txt")   try:     f.write("Lorum Ipsum")   except:     print("Something went wrong when writing to the file")   finally:     f.close() except:   print("Something went wrong when opening the file")* |
| while | Para criar um loop (estrutura de condição) while. Um loop do tipo while será executado até que a sua condição seja falsa, ou seja atingida!  Exemplo:  *x = 0  while x < 9: “ executado enquanto x for menor que 9   print(x)   x = x + 1 #lembre-se de sempre incrementar o contador para garantir que o loop não rode pelo infinito!*  *Podemos usar um break, se for o caso, para interromper a execução do while:*  *i = 1 while i < 6:   print(i)   if i == 3:     break   i += 1*  *A variável que compõe a condição para o looping while precisa ser definida antes ou é preciso conhecê-la, do contrário o while fica inaplicável. E é preciso garantir que a variável mude de estado ou valor durante o looping, para não ficar executando ao infinito.* |
| with | A palavra-chave with no Python é usada em conjunto com context manager para garantir que recursos sejam automaticamente fechados ou liberados após sua utilização, de forma adequada, mesmo se ocorrerem erros durante a execução do código.  Aqui estão algumas situações em que você pode usar o with:  \*\*\* Leitura e escrita de arquivos:  Ao abrir um arquivo com with open('seuarquivo.txt', 'r') as arquivo:  **o arquivo é automaticamente fechado quando o bloco with é concluído**.  Isso evita que você precise explicitamente chamar file.close() e garante que o arquivo seja fechado corretamente, mesmo em caso de exceções.    \*\*\* Conexões de banco de dados:  Quando você se conecta a um banco de dados, o with pode ser usado para gerenciar a conexão.  Por exemplo:  with sqlite3.connect('meubanco.db') as conexao:    \*\*\* Trabalhando com sockets:  O with também é útil para gerenciar conexões de rede, como sockets.  Exemplo:  with socket.create\_connection(('localhost', 8080)) as s:    \*\*\* Manipulação de arquivos temporários:  Se você estiver criando arquivos temporários, o with pode garantir que eles sejam excluídos após o uso.  Por exemplo:  with tempfile.NamedTemporaryFile(delete=True) as temp\_file:.  Em resumo, sempre que você precisar adquirir e liberar recursos automaticamente, o with deve ser usado para garantir a liberação dos recursos que foram e estão utilizados. |
| yeld | Para retornar uma lista de vlores de um gerador.  Exemplo: Retornar 3 valores de uma função:  *def myFunc():   yield "Hello"   yield 51   yield "Good Bye"  x = myFunc()  for z in x:   print(z) # retorna os valores definidos pelo yeld* |