



PYTHON 3.10: FUNCIONALIDADES E NOVIDADES DA VERSÃO

Nesse ebook vamos ver o que há de novo na tão aguardada versão 3.10 do Python!

Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA 

Olá Pythonista!

Hoje você vai ficar por dentro da tão aguardada **versão 3.10 do Python!**

Vamos falar sobre as novas funcionalidades, as novidades, as inovações e o que há de **mais interessante** nessa nova versão do Python.

Está ansioso para testar o poderoso **Pattern Matching** (“Correspondência ou Casamento de Padrões”)?

Estão dizendo por aí que é o Switch/Case com esteróides que o Python nunca teve 😊

Ou está ansioso para ver como as mensagens de erros vão ser incrementadas na versão 3.10?

Neste post, você aprenderá sobre:

- Debugar código com **mensagens de erro mais úteis e precisas**
- Usar *Pattern Matching*
- Iterar sobre estruturas de forma mais segura com `zip()` utilizando o parâmetro `strict`

Então vem com a gente que vamos te explicar **TUDO!**

Versão 3.10 do Python



Python 3.10 foi lançado!

O grupo de desenvolvedores têm trabalhado na nova versão desde Maio de 2020 para trazer a você uma melhor, mais rápida e mais segura versão do Python.

Em 4 de outubro de 2021, a primeira versão 3.10 oficial **foi disponibilizada!**

Cada nova versão do Python traz uma série de mudanças que estão disponíveis na documentação.

Aqui, você aprenderá sobre os novos **recursos mais legais e empolgantes.** 😊

Para experimentar os novos recursos, você precisa executar o Python 3.10!

Você pode obtê-lo na [página inicial do Python](#). Como alternativa, você pode usar o Docker com a [imagem Python mais recente](#).

E agora vamos às **NOVIDADES**!

Melhores Mensagens de Erro

Apesar de Python ser uma linguagem conhecida por ser amigável ao Desenvolvedor, algumas partes deixam a desejar.

Uma das maiores reclamações quanto à isso eram as mensagens de erro!

O Python 3.10 vem com uma série de mensagens de erro mais precisas e construtivas para tentar contornar isso.

Veja o exemplo clássico do seu primeiro “Hello World”. Mas vamos supor que você tenha se esquecido de fechar a string com aspas.

```
print("Hello World)
```

Veja como o erro não ajuda muito:

```
File "<stdin>", line 1
    print("Hello World)
                  ^
SyntaxError: EOL while scanning string literal
```

Havia um `SyntaxError` no código!

`EOL`, o que isso realmente significa?

Você volta ao seu código e, depois de olhar e pesquisar um pouco, percebe que está faltando aspas no final da String.

Uma das melhorias mais impactantes no Python 3.10 são mensagens de erro melhores e mais precisas para muitos problemas comuns!

Ufa!

Veja agora a mensagem de erro no Python 3.10:

```
File "<stdin>", line 1
    print("Hello World!")
               ^
SyntaxError: unterminated string literal (detected at line 3)
```

A mensagem de erro ainda é um pouco técnica, mas o misterioso **EOL se foi** e ainda adicionaram **a linha onde o erro ocorreu!**

Agora, a mensagem informa que você precisa encerrar sua string!

Existem melhorias semelhantes para muitas mensagens de erro diferentes. Vamos ver outro exemplo!

Imagine o seguinte cenário, em que temos um **dict** que não foi fechado!

Ainda não é um EXPERT nos Dicionários do Python? Então já clique no link para acessar nosso Post COMPLETO sobre Dicionários do Python em seguida!

Este é o código (com erro):

```
vendas = {  
    'João': 12,  
    'Clara': 10,  
    'Ana': 21,  
    'Carlos': 14  
  
for nome, vendas in vendas.items():  
    print(f'{nome} vendeu {vendas} items este mês')
```

Observação: Nesse post teremos diversos exemplos utilizando f-strings. Se você ainda não domina esse conceito, já clica aqui e dá uma lida nesse [post completo sobre F-Strings no Python](#)

E agora perceba que a mensagem de erro não nos diz **ABSOLUTAMENTE NADA**:

```
File "<stdin>", line 7  
    for nome, vendas in vendas.items():  
        ^  
SyntaxError: invalid syntax
```

Pô, time Guido!

Agora veja como as coisas mudam na versão 3.10 do Python:

```
File "<stdin>", line 7  
    vendas = {  
        ^  
SyntaxError: '{' was never closed
```

Tradução do Erro:

SyntaxError : '{' não foi fechado

Aí sim, #TeamPython! 

Nesse mesmo código, imagine que tenhamos esquecido uma vírgula entre os itens do Dicionário, dessa forma:

```
vendas = {  
    'João': 12  
    'Clara': 10,  
    'Ana': 21,  
    'Carlos': 14  
  
for nome, vendas in vendas.items():  
    print(f'{nome} vendeu {vendas} items este mês')
```

Olha o erro, que nada a ver:

```
File "<stdin>", line 1  
    'Carlos': 14  
    ^  
IndentationError: unexpected indent
```

Aí não, galera!

```
File "<stdin>", line 1
  'João': 12
    ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Tradução do Erro:

SyntaxError : sintaxe inválida. Talvez você tenha esquecido uma vírgula?

Ponto pro time Python! 👍

Último exemplo, o uso incorreto de `=` e `==` em comparações:

```
numeros = [1, 5, 6, 8, 48, 55]

if len(numero) = 6: # Aqui deveria ser ==
    print(f'Número máximo de apostas')
```

O erro:

```
File "<stdin>", line 1
  if len(numero) = 6: # Aqui deveria ser ==
    ^
SyntaxError: invalid syntax
```

Até que vai, time!

A mensagem de erro até ajuda nesse caso, mas a melhoraram para:

```
File "<stdin>", line 1
    if len(numero) = 6: # Aqui deveria ser ==
                        ^
SyntaxError: cannot assign to attribute here. Maybe you meant '==' instead of '='?
```

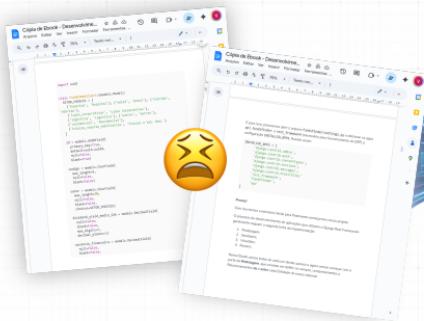
Tradução do Erro (com modificações):

*SyntaxError : Esse local não permite atribuição. Talvez você quis dizer
'==' em vez de '=' ?*

 *Estou construindo o **DevBook**, uma plataforma que usa IA para criar e-books técnicos — com código formatado e exportação em PDF. Depois de ler, dá uma passada lá!*

Crie Ebooks técnicos incríveis em minutos com IA

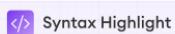
Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



Deixe que nossa IA faça o trabalho pesado



TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS 

Pattern Matching ou “Correspondência de Padrões”

Sua introdução à linguagem às vezes é chamada de `switch ... case` do Python, mas você verá que o conceito de *Pattern Matching* é **muito mais poderoso** que isso!

Esse recurso da linguagem pode ser usado para **detectar e desconstruir diferentes estruturas em seus dados** e também realizar a **correspondência literal de padrões**.

Confuso né? Mas calma que vamos destrinchar esse carinha

Primeiro, vamos falar sobre a sintaxe básica do `match/case` do Python:

```
match {elemento}:
    case {padrão 1}:
        {ação 1}
    case {padrão 2}:
        {ação 2}
    case {padrão 3}:
        {ação 3}
    # Outros padrões...
    case _:
        {ação padrão ou ação default}
```

Vamos entender:

- Primeiro, temos a adição das novas Palavras Reservadas (*Keywords*) `match` e `case`. A keyword `match` inicia um novo bloco para realizar a correspondência de padrões.
- Em seguida temos o `{elemento}` que pode ser um inteiro, um decimal, uma lista, um objeto e assim por diante: é ele quem vai ser “testado”.
- Depois, temos os blocos `case {padrão}: {ação}` que definem: dado que tal `{padrão}` foi correspondido, execute tal `{ação}`
- Caso não haja a correspondência de nenhum padrão acima (`case _:`), execute a `{ação padrão}`.

Agora vamos para parte legal: o exemplo!

```
parentesco = 'mãe'

match parentesco:
    case 'pai':
        print("O parentesco encontrado é 'Pai'")
    case 'mãe':
        print("O parentesco encontrado é 'Mãe'")
    case 'filho(a)':
        print("O parentesco encontrado é 'Filho(a)'")
    case _:
        print('Parentesco encontrado não mapeado')
```

Se colocarmos esse código em execução, veremos que haverá uma correspondência no `case 'mãe'` e a saída será:

```
O parentesco encontrado é 'Mãe'
```

Legal né?! 😍 Mas calma que esse é apenas o exemplo básico!

Vamos ver outro exemplo mais interessante!

Será que é possível utilizar esse tal de Correspondência de Padrões em Estruturas de Repetição “FOR”?

Com certeza!

Veja o exemplo a seguir:

```

notas_alunos = {
    'João': [0, 5, 10],
    'Clara': [7, 9, 8],
    'Maciel': "",
    'Jonas': [0, 1],
    'Maria': [5, 7, 10],
    'Marcelo': [],
    'Artur': [1]
}

# dict.items retorna uma tupla contendo a chave na primeira posição e o
# valor na segunda posição
for aluno, notas in notas_alunos.items():
    # Casamento de padrão com chave, valor
    match aluno, notas:
        case aluno, [x, y, z]:
            print(f'Aluno {aluno} fez três provas. Média = {(x + y + z)/3:.1f}')

        case aluno, [x, y]:
            print(f'Aluno {aluno} fez duas provas. Média = {(x + y)/2:.1f}')

        case aluno, [x]:
            print(f'Aluno {aluno} fez uma prova. Média = {x:.1f}')

        case aluno, []:
            print(f'Aluno {aluno} não fez nenhuma prova. REPROVADO')

        case _:
            print('>>> Formato inválido')

```

A saída será:

```
Aluno João fez três provas. Média = 5.0
Aluno Clara fez três provas. Média = 8.0
>>> Formato inválido
Aluno Jonas fez duas provas. Média = 0.5
Aluno Maria fez três provas. Média = 7.3
Aluno Marcelo não fez nenhuma prova. REPROVADO
Aluno Artur fez uma prova. Média = 1.0
```

E agora, vamos a **explicação**:

- Temos uma iteração simples de dicionário, utilizando o `dict.items()` que retorna uma tupla contendo `(chave, valor)`
- Em cada `case` temos a correspondência da `chave` que é `aluno` e `valor` que é uma Lista.
- Caso `notas` tenha uma correspondência com `[x, y, z]`, cairá no primeiro `case`.
- Caso `notas` tenha uma correspondência com `[x, y]`, cairá no segundo `case`.
- Caso `notas` tenha uma correspondência com `[x]`, cairá no terceiro `case`.
- Caso `notas` tenha uma correspondência com `[]` (lista vazia), cairá no quarto `case`.
- Caso não tenha correspondência com nenhum `case` acima, o `case _` será executado!

Ainda podemos refatorar o código da seguinte forma, ficando mais genérico e mais conciso.

Preste atenção no primeiro `case`! Veja que é possível especificar o tipo de objeto que queremos fazer a correspondência:

```

notas_alunos = {
    'João': [0, 5, 10],
    'Clara': [7, 9, 8],
    'Maciel': "",
    'Jonas': [0, 1],
    'Maria': [5, 7, 10],
    'Marcelo': [],
    'Artur': [1]
}

for aluno, notas in notas_alunos.items():
    match aluno, notas:
        case aluno, [int(0) | float(0.0), *resto]:
            print(f'Aluno {aluno} zerou a primeira prova. '
                  f'Suas outras notas foram {resto}')

        case aluno, list(notas_aluno):
            print(f'Aluno {aluno} fez {len(notas_aluno)} prova(s). '
                  f'Média = {sum(notas_aluno)/3:.1f}')

        case _:
            print('>>> Formato inválido')

```

E a saída, com pequenas alterações:

```

Aluno João fez 3 prova(s). Média = 5.0
Aluno Clara fez 3 prova(s). Média = 8.0
Aluno Maciel fez 0 prova(s). Média = 0.0
Aluno Jonas fez 2 prova(s). Média = 0.3
Aluno Maria fez 3 prova(s). Média = 7.3
Aluno Marcelo fez 0 prova(s). Média = 0.0
Aluno Artur fez 1 prova(s). Média = 0.3

```

Também podemos usar correspondência de padrões “**OR**”, utilizando o símbolo *pipe* **{% raw %}|{% endraw %}**.

Dessa forma, é possível passar dois (ou mais) possíveis padrões para testar.

Vamos supor que seja necessário verificar quem tirou `0` (Inteiro) ou `0.0` (Float) na primeira prova.

Podemos fazer isso da seguinte maneira:

```
notas_alunos = {  
    'João': [0, 5, 10],  
    'Clara': [7, 9, 8],  
    'Maciel': "",  
    'Jonas': [0, 1],  
    'Maria': [5, 7, 10],  
    'Marcelo': [],  
    'Artur': [1]  
}  
  
for aluno, notas in notas_alunos.items():  
    match aluno, notas:  
        case aluno, [int(0) | float(0.0), *resto]:  
            print(f'Aluno {aluno} zerou a primeira prova. '  
                  f'Suas outras notas foram {resto}')  
  
        case aluno, list(notas_aluno):  
            print(f'Aluno {aluno} fez {len(notas_aluno)} prova(s). '  
                  f'Média = {sum(notas_aluno)/3:.1f}')  
  
        case _:  
            print('>>> Formato inválido')
```

Veja a saída:

```
Aluno João zerou a primeira prova. Suas outras notas foram [5, 10]  
Aluno Clara fez 3 prova(s). Média = 8.0  
>>> Formato inválido  
Aluno Jonas zerou a primeira prova. Suas outras notas foram [1]  
Aluno Maria fez 3 prova(s). Média = 7.3  
Aluno Marcelo fez 0 prova(s). Média = 0.0  
Aluno Artur fez 1 prova(s). Média = 0.3
```

Vamos entender aquele primeiro `case` muito louco:

```
case aluno, [int(0) | float(0.0), *resto]:
```

- Primeiro fazemos a correspondência de `aluno` com a chave do dicionário, até aí nada de novo.
- Em seguida, fazemos uma correspondência com o primeiro elemento da lista que pode ser `int(0)` ou `float(0.0)`.
- Se o padrão acima tiver correspondência, pegamos o resto da lista com o padrão `*resto`

Bom, acho que já deu para perceber o poder da **Correspondência de Padrões** ou **Pattern Matching** do Python 3.10 né?!

Com certeza esse é um assunto que renderá um Post completo sobre *Pattern Matching*!

Adição do parâmetro `strict` ao método `zip()`

O método `zip()` percorre diversos iteráveis ao mesmo tempo, parando de iterar na menor sequência.

Isto é: se tivermos duas listas para iterar e uma tiver 4 elementos e outra tiver 3 elementos, `zip()` vai iterar apenas sobre os 3 primeiros elementos da primeira lista.

Vamos ver um exemplo:

```
usuarios = ['2021245', '2021859', '2021522', '2021636']
acessos = [5, 4, 8]

for u, a in zip(usuarios, acessos):
    print(f'Usuário {u} = {a} acessos')
```

Veja que a saída possui apenas 3 elementos, como esperado:

```
Usuário 2021245 = 5 acessos
Usuário 2021859 = 4 acessos
Usuário 2021522 = 8 acessos
```

A novidade trazida pela versão 3.10 do Python é a inclusão do parâmetro `strict`.

Caso esse parâmetro seja setado como `True` e a função `zip` detecte iteráveis de tamanhos diferentes, um erro `ValueError` será lançado.

Vamos ver o exemplo:

```
usuarios = ['2021245', '2021859', '2021522', '2021636']
acessos = [5, 4, 8]

for u, a in zip(usuarios, acessos, strict=True):
    print(f'Usuário {u} = {a} acessos')
```

Veja o erro:

```
Usuário 2021245 = 5 acessos
Usuário 2021859 = 4 acessos
Usuário 2021522 = 8 acessos
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is shorter than argument 1
```

Tradução do Erro: `ValueError` : argumento 2 da `zip()` é menor que o argumento 1

Adição de Parenteses à **Context Managers**

Context Managers permitem que possamos gerenciar melhor recursos computacionais como arquivos, *locks* ou interfaces de rede.

Eles são definidos com a palavra reservada `with`.

Para quebrar longas linhas do `with` antes do Python 3.10, era necessário utilizar a barra `\`, da seguinte forma:

```
with open('arquivo.txt', 'r', encoding='utf-8') as arquivo_leitura, \
    open('arquivo_saida.txt', 'w', encoding='utf-8') as arquivo_saida:
```

Agora, é possível utilizar *Context Managers* com parenteses, da seguinte maneira:

```
with (
    open('arquivo.txt', 'r', encoding='utf-8') as arquivo_leitura,
    open('arquivo_saida.txt', 'w', encoding='utf-8') as arquivo_saida
):
```

Uma pequena alteração, mas que melhora a **legibilidade do código**.

Conclusão

É sempre interessante ficarmos ligados nas novas funcionalidades que cada versão do Python nos traz, se quisermos nos manter relevantes no mercado de trabalho!

Neste tutorial, você viu novos recursos como:

- Mensagens de erro mais amigáveis
- O poder do *Pattern Matching* ou Correspondência de Padrões!
- Iterações mais seguras de sequências com `zip()` e o parâmetro `strict`

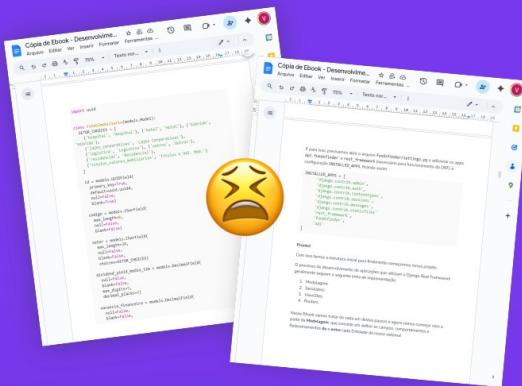
Divirta-se experimentando os novos recursos!

Se ficou com alguma dúvida, fique à vontade para deixar um comentário no box aqui embaixo! Será um prazer te responder! 😊



Crie Ebooks técnicos em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



Syntax Highlight

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS