



# Pattern Matching

Live de Python #171



## 1. Comparações

Uma olhada em casos estranhos

## 2. Pattern Matching

Agora sim, vamos começar

## 3. Não literais

O que difere de um switch case

## 4. Dando nomes

Usando os dados dos cases

## 5. Os dicionários

Cases interessantes

## 6. Objetos

O gerente ficou maluco



[picpay.me/dunossauro](https://picpay.me/dunossauro)



[apoia.se/livedepython](https://apoia.se/livedepython)



PIX



Ajude o projeto



Ademar Peixoto, Alex Lima, Alex Lopes, Alexandre Harano, Alexandre Santos, Alexandre Tsuno, Alexandre Villares, Alynne Ferreira, Alysson Oliveira, Amaziles Carvalho, André Rocha, Arnaldo Turque, Bruno Batista, Bruno Oliveira, Caio Nascimento, Carlos Chiarelli, Cleber Santos, César Almeida, Davi Ramos, David Kwast, Diego Guimarães, Dilenon Delfino, Elias Soares, Eugenio Mazzini, Everton Alves, Fabiano Gomes, Fabio Barros, Fabio Castro, Fabrícia Diniz, Fabrício Coelho, Flavkaze, Francisco Alencar, Fábio Serrão, Gabriel Simonetto, Gabriel Soares, Gabriela Santiago, Geandreson Costa, Guilherme Castro, Guilherme Felitti, Guilherme Ostrock, Gustavo Chacon, Henrique Machado, Israel Fabiano, Italo Silva, Johnny Tardin, Jonatas Leon, Jonatas Oliveira, Jorge Plautz, Jose Mazolini, José Prado, João Lugão, João Schiavon, Juan Gutierrez, Julio Silva, Jônatas Silva, Júlia Kastrup, Kaneson Alves, Leonardo Cruz, Leonardo Galani, Leonardo Mello, Lidiane Monteiro, Lorena Ribeiro, Lucas Barros, Lucas Mello, Lucas Mendes, Lucas Teixeira, Lucas Valino, Luciano Ratamero, Maiquel Leonel, Marcela Campos, Marcelo Rodrigues, Maria Clara, Marina Passos, Matheus Vian, Melissa Mendonça, Natan Cervinski, Nicolas Teodosio, Osvaldo Neto, Patric Lacouth, Patricia Minamizawa, Patrick Gomes, Paulo Tadei, Pedro Pereira, Peterson Santos, Rafael Lino, Reinaldo Silva, Renan Moura, Revton Silva, Rodrigo Ferreira, Rodrigo Mende, Rodrigo Vaccari, Ronaldo Silva, Sandro Mio, Silvio Xm, Thiago Araujo, Thiago Borges, Thiago Bueno, Tyrone Damasceno, Victor Geraldo, Vinícius Bastos, Vinícius Ferreira, Vítor Gomes, Wendel Rios, Wesley Mendes, Willian Lopes, Willian Rosa, Wilson Duarte, Érico Andrei



Obrigado você



Uma olhada para  
o python < 3.10

Compa  
rações

# Um if padrão para representar um switch



```
1  valor = 1
2
3  if valor == 1:
4      print('O valor é 1')
5  elif valor == 2:
6      print('O valor é 2')
7  else:
8      print('O valor não é 1 nem 2')
```

# Uma solução para o "switch"



```
1  valor = 1
2
3  valores = {
4      1: 'O valor é 1',
5      2: 'O valor é 2'
6  }
7
8  if valor in valores:
9      print(valores[valor])
10 else:
11     print('O valor não é 1 nem 2')
```

# Uma nova solução



```
1  valor = 1
2
3  match valor:
4      case 1:
5          print('O valor é 1')
6      case 2:
7          print('O valor é 2')
8      case _:
9          print('O valor não é 1 nem 2')
```



Agora sim, vamos  
começar

Pattern  
Matching

# Pattern Matching [what's new]



## PEP 634: Structural Pattern Matching

Structural pattern matching has been added in the form of a *match statement* and *case statements* of patterns with associated actions. Patterns consist of sequences, mappings, primitive data types as well as class instances. Pattern matching enables programs to extract information from complex data types, branch on the structure of data, and apply specific actions based on different forms of data.

# Pattern Matching [what's new]



## PEP 634: Structural Pattern Matching

O Pattern Matching foi adicionada na forma de um instrução `Match` e instruções de `Case` de padrões com ações associadas.

Os padrões consistem em sequências, mapeamentos, tipos de dados primitivos, bem como instâncias de classe.

O Pattern Matching permite que os programas extraiam informações de tipos de dados complexos, ramifiquem na estrutura de dados e apliquem ações específicas com base em diferentes formas de dados.

# Basicamente, isso



```
1 match subject:
2     case <padrão_1>:
3         <ação_1>
4     case <padrão_2>:
5         <ação_2>
6     case <padrão_3>:
7         <ação_3>
8     case _:
9         <ação_coringa>
```

# Basicamente, isso



```
1 match subject:
2     case <padrão_1>:
3         <ação_1>
4     case <padrão_2>:
5         <ação_2>
6     case <padrão_3>:
7         <ação_3>
8     case _:
9         <ação_coringa>
```



```
match valor:
    case 1:
        print(1)
    case 2:
        print(2)
    case 3:
        print(3)
    case _:
        print('Diferente de 1,2 3')
```

```
1 match subject:
2     case <padrão_1>:
3         <ação_1>
4     case <padrão_2>:
5         <ação_2>
6     case <padrão_3>:
7         <ação_3>
8     case _:
9         <ação_coringa>
```



```
match valor:
    case 1:
        print(1)
    case 2:
        print(2)
    case 3:
        print(3)
    case _:
        print('Diferente de 1,2 3')
```

O Pattern Matching foi adicionada na forma de um instrução **Match** e instruções de **Case** de padrões com **ações associadas**.

```
1 match subject:
2     case <padrão_1>:
3         <ação_1>
4     case <padrão_2>:
5         <ação_2>
6     case <padrão_3>:
7         <ação_3>
8     case _:
9         <ação_coringa>
```



```
match valor:
    case 1:
        print(1)
    case 2:
        print(2)
    case 3:
        print(3)
    case _:
        print('Diferente de 1,2 3')
```

O Pattern Matching foi adicionada na forma de um instrução **Match** e instruções de **Case** de padrões com **ações associadas**.

Isso então é um switch case do  
C, do Java, do Javascript?



Não! É mais poderoso





```
1  # Exemplo em C
2  switch(grade) {
3      case 'A' :
4          printf("A!");
5          break;
6      case 'B' :
7      case 'C' :
8          printf("B ou C");
9          break;
10     default :
11         printf("Dados inválidos");
12 }
```

```
# Exemplo em Java
switch(num){
    case 'A':
        System.out.println("A!");
    case 'B':
    case 'C':
        System.out.println("B ou C");
    default:
        System.out.println("Dados inválidos");
}
```

# Qual a diferença?



Ac comparações em um switch case são "**literais**", ou seja, o dado que está no switch é um tipo "não composto". Ou seja, um número, uma string e etc...

# Exemplo em Java

```
switch(num){  
    case 'A':  
        System.out.println("A!");  
    case 'B':  
    case 'C':  
        System.out.println("B ou C");  
    default:  
        System.out.println("Dados inválidos");  
}
```

# Qual a diferença?



Ainda vamos chegar nesse caso em Python.

Para pessoas com ansiedade: é um operador `|`

```
# Exemplo em Java
switch(num){
    case 'A':
        System.out.println("A!");
    case 'B':
    case 'C':
        System.out.println("B ou C");
    default:
        System.out.println("Dados inválidos");
}
```

O que difere de  
um switch case

Não  
literais

# Pattern matching



Pattern matching não é uma novidade do python, é uma construção sintática presente em diversas linguagens, como F#, Rust, Scala, Haskell e etc...

A ideia principal é fazer "match" com algum caso específico, porém, não somente literais.

# Vamos para um tipo "composto"



Vamos imaginar uma lista. Vamos criar um caso para uma lista com 3 elementos, onde esses elementos são exatamente [1, 2, 3].

```
1 match lista:
2     case [1, 2, 3]:
3         print('Lista da regra')
```

# Vamos para um tipo "composto"



Vamos imaginar uma lista. Vamos criar um caso para uma lista com 3 elementos, onde esses elementos são exatamente [1, 2, 3].

```
1 match lista:  
2     case [1, 2, 3]:  
3         print('Lista da regra')
```

Eu sei, ainda parece um match literal. Mas calma.

# Um match não tão literal



E se quisermos ao mesmo ponto, saber somente se o primeiro valor é 1.  
Como isso ficaria?

```
1 match lista:
2     case [1, 2, 3]:
3         print('Lista da regra')
4     case [1, _, _]:
5         print('1 é o primeiro')
```



# Um match não tão literal



E se quisermos ao mesmo ponto, saber somente se o primeiro valor é 1.  
Como isso ficaria?

```
1 match lista:
2     case [1, 2, 3]:
3         print('Lista da regra 1')
4     case [1, _, _]:
5         print('1 é o primeiro')
```

Aqui já não importam o segundo e o terceiro valor.

O bloco será executado se a lista tiver exatamente 3 valores e o primeiro for 1.



# Um match não tão literal

E se quisermos ao mesmo ponto, saber somente se o primeiro valor é 1.  
Como isso ficaria?

```
1 match lista:
2     case [1, 2, 3]:
3         print('Lista da regra 1')
4     case [1, _, _]:
5         print('1 é o primeiro')
```

A comparação seria

`lista[0] == 1 and len(lista) == 3`

# Um match não tão literal



```
1 match lista:
2     case [1, 2, 3]:
3         print('Lista da regra')
4     case [1, _, _]:
5         print('1 é o primeiro')
6     case [_, 2, _]:
7         print('2 é o segundo')
```

# Um match não tão literal



```
1 match lista:
2     case [1, 2, 3]:
3         print('Lista da r
4     case [1, _, _]:
5         print('1 é o primetro
6     case [_, 2, _]:
7         print('2 é o segundo')
```

Aqui já não importam o primeiro e o terceiro valor.

O bloco será executado se a lista tiver exatamente 3 valores e o segundo for 2.

# Um caso mais legal



```
1  match lista:
2      case [] | [_]:
3          print('Um ou nenhum elemento')
4      case [1, 2]:
5          print('Lista = [1, 2]')
6      case [1, *_]:
7          print('Um é o primeiro de uma lista > 1')
```

# Operador OR |



```
1 match lista:
2     case [] | [_]:
3         print('Um ou nenhum elemento')
4     case [1, 2]:
5         print('Lista = [1, 2]')
6     case [1, *_]:
7         print('Um é o primeiro de uma lista > 1')
```

Caso onde a lista é vazia, ou contém apenas um elemento

# Match exato



```
1 match lista:
2     case [] | [_]:
3         print('Um ou n')
4     case [1, 2]:
5         print('Lista = [1, 2]')
6     case [1, *_]:
7         print('Um é o primeiro de uma lista > 1')
```

Caso onde a lista é exatamente [1, 2]

# Empacotamento



```
1 match lista:
2     case [] | [_]:
3         print('Um ou nenh
4     case [1, 2]:
5         print('Lista =
6     case [1, *_]:
7         print('Um é o primeiro de uma lista > 1')
```

Caso onde a lista começa com 1 e tem uma quantidade qualquer de elementos



# Quem pode ser comparado?



```
1 match lista:
2     case [] | [_]:
3         print('Um ou nenh
4     case [1, 2]:
5         print('Lista = [1
6     case [1, *_]:
7         print('Um é o primeiro de uma lista > 1')
```

Nessa comparação, podemos  
usar qualquer tipo de objeto  
que implemente o protocolo de  
Sequência  
(collections.abc.Sequence):

`__getitem__` e `__len__`

# Fluxo de execução




```
1  def http_error(status):  
2      match status:  
3          case 400:  
4              return "Bad request"  
5          case 404:  
6              return "Not found"  
7          case 418:  
8              return "I'm a teapot"  
9          case _:  
10             return "Something's wrong with the Internet"
```



# Fluxo de execução

```
1 def http_error(status):  
2     match status:  
3         case 400:  
4             return "Bad request"  
5         case 404:  
6             return "Not found"  
7         case 418:  
8             return "I'm a teapot"  
9         case _:  
10            return "Something's wrong with the Internet"
```

A red arrow originates from the 'match status:' line (line 2) and points downwards to the 'case \_:' line (line 9), indicating the flow of execution when no specific status code matches.

Como o fluxo é executado de cima para baixo, as regras mais genéricas devem ser colocadas no final

# Fluxo de execução

```
1  def http_error(status):  
2      match status:  
3          case 400:  
4              return "Bad request"  
5          case 404:  
6              return "Not found"  
7          case 418:  
8              return "I'm a teapot"  
9          case _:  
10             return "Something's wrong with the Internet"
```



A comparação usada para literais é ==

```
status == 400  
status == 404  
status == 418
```

# Tipos de comparações

Comparações usando ==

value == 'literal'  
value == 1

```
1 match value:
2     case 'literal':
3         return 'sou comparado com =='
4     case 1:
5         return 'sou comparado com =='
6
7     case False:
8         return 'sou comparado com is'
9     case True:
10        return 'sou comparado com is'
11    case None:
12        return 'sou comparado com is'
```

# Tipos de comparações

Comparações usando is

value is False  
value is True  
value is None

```
1  match value:
2      case 'literal':
3          return 'sou comparado com =='
4      case 1:
5          return 'sou comparado com =='
6
7      case False:
8          return 'sou comparado com is'
9      case True:
10         return 'sou comparado com is'
11     case None:
12         return 'sou comparado com is'
```

# Quem pode ser comparado?



```
1 match lista:
2     case [] | [_]:
3         print('Um ou nenh
4     case [1, 2]:
5         print('Lista = [1
6     case [1, *_]:
7         print('Um é o primeiro de uma lista > 1')
```

Comparações do protocolo de  
sequência

`len(lista) == 0 or len(lista) == 1`

## Quem pode ser comparado

Comparações do protocolo de sequência

`len(lista) == 2 and lista[0] == 1 and lista[1] == 2`

```
1 match lista:
2     case [] | [_]:
3         print('Um ou nenhum elemento')
4     case [1, 2]:
5         print('Lista = [1, 2]')
6     case [1, *_]:
7         print('Um é o primeiro de uma lista > 1')
```



# Quem pode ser co

Comparações do protocolo de sequência

`lista[0] == 1 and len(lista) >= 1`

```
1 match lista
2     case [] | [_]:
3         print('Um elemento')
4     case [1, 2]:
5         print('Lista = [1, 2]')
6     case [1, *_]:
7         print('Um é o primeiro de uma lista > 1')
```

# Aprofundando o OR



O operador de or `|` pode ser usado em dois casos diferentes.

- Um padrão ou Outro Padrão
- Sub padrões

```
1 match value:
2     case [] | [_]:
3         return 'Ou vazio ou 1 elemento'
4     case [1, *_]:
5         return 'Primeiro elemento é 1'
6     case [2|3, *_]:
7         return 'Primeiro elemento é 2 ou 3'
8     case [_, 5|6, *_]:
9         return 'Segundo elemento é 5 ou 6'
```

# Aprofundando o OR



O operador de or `|` pode ser usado em dois casos diferentes.

- Um padrão ou Outro Padrão
- Sub padrões

```
1 match value:
2     case [] | [_]:
3         return 'Ou vazio ou 1 elemento'
4     case [1, *_]:
5         return 'Primeiro elemento é 1'
6     case [2|3, *_]:
7         return 'Primeiro elemento é 2 ou 3'
8     case [_, 5|6, *_]:
9         return 'Segundo elemento é 5 ou 6'
```

Usando os dados  
dos cases

Dando  
Nomes

# Usando os nomes

Até o momento usamos `_` para os dados que "enchiam vazios" para os matches, porém, podemos usar esses valores dentro dos blocos e dar nomes a eles.

```
1  match lista:
2      case [x]:
3          print(f'[{x=}]')
4      case 1, x:
5          print(f'1 e {x}')
6      case _, x:
7          print(f'_ e {x}')
8      case 1, *grupo:
9          print(f'1 e {grupo}')
```

# Usando os nomes

Até o momento usamos `_` para os dados que "enchiam vazios" para os matches, porém, podemos usar esses valores dentro dos blocos para dar nomes a eles.

Lista com um único elemento e esse elemento terá o nome "x" dentro desse bloco

```
1  match lista:
2      case [x]:
3          print(f'[{x=}]')
4      case 1, x:
5          print(f'1 e {x}')
6      case _, x:
7          print(f'_ e {x}')
8      case 1, *grupo:
9          print(f'1 e {grupo}')
```

# Usando os nomes

Até o momento usamos `_` para os dados que "enchiam vazios" para os matches, porém, podemos usar esses valores dentro dos blocos e dar nomes a eles.

Match de `[1, ??]` e qualquer valor. Esse caso está aqui só pra mostrar que os `[ ]` são opcionais

```
1  match lista:
2      case [x]:
3          print(f'[{x=}]')
4      case 1, x:
5          print(f'1 e {x}')
6      case _, x:
7          print(f'_ e {x}')
8      case 1, *grupo:
9          print(f'1 e {grupo}')
```

# Usando os nomes

Até o momento usamos `_` para os dados que "enchiam vazios" para os matches, porém, podemos usar esses valores dentro dos blocos e dar nomes a eles.

É possível nomear e não nomear no mesmo case

```
1  match lista:
2      case [x]:
3          print(f'[{x=}]')
4      case 1, x:
5          print(f'1 e {x}')
6      case _, x:
7          print(f'_ e {x}')
8      case 1, *grupo:
9          print(f'1 e {grupo}')
```



# Usando os nomes

Até o momento usamos `_` para os dados que "enchiam vazios" para os matches, porém, podemos usar esses valores dentro dos blocos e dar nomes a eles.

Os empacotamentos também podem ser nomeados

```
1  match lista:
2      case [x]:
3          print(f'[{x=}]')
4      case 1, x:
5          print(f'1 e {x}')
6      case _, x:
7          print(f'_ e {x}')
8      case 1, *grupo:
9          print(f'1 e {grupo}')
```

Utilizando os  
nomes no case

Guards

# Guards

Os guards são maneiras de adicionar uma camada maior de validação aos nossos cases.

```
def chato_das_cores(cor):  
    match cor:  
        case r, g, b:  
            return 'Cadê o alpha?'  
        case r, g, b, a if a == 255:  
            return 'Tudo transparente? é Sérió?'  
        case r, g, b, a if r == 255:  
            return 'Muito vermelho, para!'  
        case r, g, b, a if g == 255:  
            return 'Não, muito verde :('  
        case r, g, b, a if b == 255:  
            return 'Azul? MESMO????'  
        case r, g, b, a:  
            return 'Agora sim <3'
```

Sequência de  
literais

Opera  
dor As

# O operador AS, dando nome as escolhas



```
def movimento(comando: str) -> str:
    match comando.split():
        case ['pular']:
            return 'Pulando'
        case ['mover']:
            return 'Pra onde?'
        case 'mover', 'direita' | 'esquerda' as direção:
            return f'Movendo lateralmente para {direção}'
        case 'mover', 'cima' | 'baixo' as direção:
            return f'Movendo horizontalmente para {direção}'
```

# O operador AS, dando nome as escolhas



```
def movimento(comando: str) -> str:
    match comando.split():
        case ['pular']:
            return 'Pulando'
        case ['mover']:
            return 'Pra onde?'
        case 'mover', 'direita' | 'esquerda' as direção:
            return f'Movendo lateralmente para {direção}'
        case 'mover', 'cima' | 'baixo' as direção:
            return f'Movendo horizontalmente para {direção}'
```

Cases  
interessantes

Dicion  
ários

# O caso dos dicionários

Os dicionários, a meu ver, são os objetos mais flexíveis a respeito do pattern matching. Eles podem dar match em chaves e valores

```
1 match dicionario:
2     case {'a': 1, 'b': 2}:
3         print('match literal')
4     case {'a': _, 'b': 2}:
5         print('match na chave b')
6     case {'a': 1, 'b': _}:
7         print('match na chave a')
8     case {'a': _, 'b': _}:
9         print('nenhum match')
10    case {'error': _}:
11        print('deu erro')
```



# O caso dos dicionários

Os dicionários, ao meu ver, são os objetos mais flexíveis a respeito do pattern matching. Eles podem dar match em chaves e valores

Chaves e valores

```
1 match dicionario:
2     case {'a': 1, 'b': 2}:
3         print('match literal')
4     case {'a': _, 'b': 2}:
5         print('match na chave b')
6     case {'a': 1, 'b': _}:
7         print('match na chave a')
8     case {'a': _, 'b': _}:
9         print('nenhum match')
10    case {'error': _}:
11        print('deu erro')
```

# O caso dos dicionários

Os dicionários, ao meu ver, são os objetos mais flexíveis a respeito do pattern matching. Eles podem dar match em chaves e valores

Match somente de chaves

```
1 match dicionario:
2     case {'a': 1, 'b': 2}:
3         print('match literal')
4     case {'a': _, 'b': 2}:
5         print('match na chave b')
6     case {'a': 1, 'b': _}:
7         print('match na chave a')
8     case {'a': _, 'b': _}:
9         print('nenhum match')
10    case {'error': _}:
11        print('deu erro')
```

# O caso dos dicionários

Os dicionários, ao meu ver, são os objetos mais flexíveis a respeito do pattern matching. Eles podem dar match em chaves e valores

Qualquer chave pode ser usada.

```
1 match dicionario:
2     case {'a': 1, 'b': 2}:
3         print('match literal')
4     case {'a': _, 'b': 2}:
5         print('match na chave b')
6     case {'a': 1, 'b': _}:
7         print('match na chave a')
8     case {'a': _, 'b': _}:
9         print('nenhum match')
10    case {'error': _}:
11        print('deu erro')
```

# Empacotamento



```
— □ ×  
  
d = {'chave': 'valor', 'outra chave': 'outro valor'}  
  
match d:  
    case {'chave': 'valor', **kwargs} if kwargs:  
        print(kwargs)  
  
# {'outra chave': 'outro valor'}
```

# Objetos

0 gerente ficou  
maluco

# Match com objetos



A ideia principal agora é fazer com que os atributos dos objetos obtenham as correspondências.

```
from dataclasses import dataclass

@dataclass
class Pessoa:
    nome: str
    idade: int
    funcionário: bool = False
```

# Nosso match posicional



```
def entrada_no_cinema(pessoa: Pessoa, preço: int):  
    match pessoa:  
        case Pessoa('Eduardo'):  
            return f'Live de Python na telona!'  
        case Pessoa(nome, idade) if idade >= 65:  
            return f'{nome.capitalize()} você paga {preço/2}'  
        case Pessoa(nome, idade, True):  
            return f'{nome.capitalize()} você paga {preço/3}'  
        case Pessoa(nome, idade):  
            return f'{nome.capitalize()} você paga {preço}'
```

# Uma classe tradicional



Vamos mudar nossa dataclass para uma classe "tradicional"

```
class Pessoa:
    def __init__(self, nome, idade, funcionário=False):
        self.nome = nome
        self.idade = idade
        self.funcionário = funcionário
```



# Resultado da execução



```
In [52]: entrada_no_cinema(Pessoa('Eduardo', 18), 10)
```

```
-----  
TypeError                                Traceback (most recent call last)
```

```
<ipython-input-52-bdab8bba05f3> in <module>
```

```
----> 1 entrada_no_cinema(Pessoa('Eduardo', 18), 10)
```

```
<ipython-input-44-91708eaddac5> in entrada_no_cinema(pessoa, preço)
```

```
    1 def entrada_no_cinema(pessoa: Pessoa, preço: int):
```

```
    2     match pessoa:
```

```
----> 3         case Pessoa(nome, idade) if idade >= 65:
```

```
    4             return f'{nome.capitalize()} você paga {preço/2}'
```

```
    5         case Pessoa(nome, idade, 'funcionário'):
```

```
TypeError: Pessoa() accepts 0 positional sub-patterns (2 given)
```

# A alteração necessária



```
def entrada_no_cinema(pessoa: Pessoa, preço: int):  
    match pessoa:  
        case Pessoa(nome=nome, idade=idade) if idade >= 65:  
            return f'{nome.capitalize()} você paga {preço/2}'  
        case Pessoa(nome=nome, idade=idade, funcionário=True):  
            return f'{nome.capitalize()} você paga {preço/2}'  
        case Pessoa(nome='Eduardo'):  
            return f'Live de Python na telona!'
```

# A alteração necessária



```
def entrada_no_cinema(pessoa: Pessoa, preço: int):  
    match pessoa:  
        case Pessoa(nome=nome, idade=idade) if idade >= 65:  
            return f'{nome.capitalize()} você paga {preço/2}'  
        case Pessoa(nome=nome, idade=idade, funcionário=True):  
            return f'{nome.capitalize()} você paga {preço/2}'  
        case Pessoa(nome='Eduardo'):  
            return f'Live de Python na telona!'
```

## \_\_match\_args\_\_

Para que seja possível a avaliação posicional, devemos adicionar um atributo da classe `\_\_match\_args\_\_` com uma tupla que diz qual a posição dos argumentos:

```
class Pessoa:
    __match_args__ = ('nome', 'idade', 'funcionário')

    def __init__(self, nome, idade, funcionário=False):
        self.nome = nome
        self.idade = idade
        self.funcionário = funcionário
```



[picpay.me/dunossauro](https://picpay.me/dunossauro)



[apoia.se/livedepython](https://apoia.se/livedepython)



PIX



Ajude o projeto

