Entendendo o Unpack

Live de Python # 295

Nosso objetivo hoje é entender coisas desse tipo.

```
def func[*Ts](
   *args: *Ts, **kwargs: Unpack[Kwargs]
) -> tuple[tuple[*Ts], Kwargs]:
    return args, kwargs

(a, *b), y = func(*[1, 2, 3], **{})
```

Pra que tanto *?

Roteiro

1. Uma introdução

Afinal, para que serve o *?

2. Callables

Funções e métodos e os *'s

3. Anotações de tipo

Variadics e TypedDict

4. 3.15 o futuro vindouro...

** em comps e genx

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

Albano Maywitz, Alexandre Costa, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Alfredo Neto, Alynnefs, Alysson Oliveira, Andre Makoski, André Oliveira, Andre Paula, Antonio Filho, Apc 16, Apolo Ferreira, Artur_farias_, Athayr_, Aurelio Costa, Azmovi, Belisa Arnhold, Bernardo Fontes, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Canibasami, Caoptic, Carlos Gonçalves, Carlos Henrique, Cauã Oliveira, Celio Araujo, Christian Fischer, Claudemir Cruz, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Brito, Daniel Bruno, Daniel Souza, Daniel Wojcickoski, Danilo Boas, Danilo Silva, Darcioalberico_sp, David Couto, Dh44s, Diego Guimarães, Dilan Nery, Dunossauro, Edgar, Elias Soares, Emerson Rafael, Érico Andrei, Esdras, Everton Silva, Ewertonbello, Fábio Belotto, Fabio Faria, Fabiokleis, Fecar995, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Fernandocelmer, Ferrabras, Fichele Marias, Fightorfall, Francisco Aclima, Franklin Sousa, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Glauber Duma, Gleidson Costa, Gnomo Nimp, Grinaode, Guibeira, Guilherme Felitti, Guilherme Ostrock, Gustavo Pedrosa, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Idlelive, Ivan Santiago, Ivansantiagojr, Janael Pinheiro, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jerry Ubiratan, Jhonata Medeiros, João Pena, Joao Rocha, Joarez Wernke, Jonas Araujo, Jonatas Leon, Jose Barroso, Joseíto Júnior, José Predo), Josir Gomes, Jota_lugs, Jplay, Jrborba, Ju14x, Juan Felipe, Juli4xpy, Juliana Machado, Julio Cesar, Julio Franco, Julio Gazeta, Julio Silva, Kaio Peixoto, Kakaroto, Killfacept, Knaka, Krisquee, Laraalvv, Lara Nápoli, Leandro Pina, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lilian Pires, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucasgcode, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Polo, Lucas Schneider, Luciano_ratamero, Luciano Ratamero, Lúcia Silva, Luidduarte, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Marcelo Araujo, Marcelo Fonseca, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Marina Passos, Mate65br, Mateusamorim96, Matheus Mendez, Matheus Vian, Medalutadorespacialx, Michael Santos, Mlevi Lsantos, Murilo Carvalho, Nhambu, Oopaze, Otávio Carneiro, Patrick Felipe, Pytonyc, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Ramon Lobo, Renan, Renan Sebastião, Renato José, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Sousa, Rinaldo Magalhaes, Robsonpiere, Rodrigo Barretos, Rodrigo Santana, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samael Picoli, Samanta Cicilia, Santhiago Cristiano, Scrimf00x, Scrimfx, Sherlock Holmes, Shinodinho, Shirakawa, Sommellier_sr, Tarcísio Miranda, Tenorio, Téo Calvo, Teomewhy, Thamires Betin, Tharles Andrade, Thiago, Thiago Araujo, Thiago Lucca, Thiago Paiva, Thisiscleverson, Tiago, Tomás Tamantini, Valdir, Varlei Menconi, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vittu_dt, Vladimir Lemos, Vonrroker, Williamslews, Willian Lopes, Xxxxxxxxx, Zero! Studio

Obrigado você <3

Uma introdução aos *'S

Os *'s

O operador * em Python tem múltiplos usos, sendo fundamental para simplificar o código e aumentar a legibilidade.

Dependendo do contexto, o * representa coisas bastante diferentes. Como:

- Empacotamento de argumentos em funções
  - Posicionais e Nomeados
- Empacotamento de iteráveis
  - Criação de novos containers
  - Desenpacotamento em chamadas de invocáveis
- Marcação exclusiva de argumentos nomeados *(não veremos)*
- Matemática *(não veremos)*:
  - Multiplicação
  - Exponenciação

Antes dos *'s

Um dos exemplos clássicos durante o aprendizado de programação é a triangulação de variáveis.

Relembre comigo :)

```
>>> a = 1
>>> b = 2
```

Se quisermos alterar os valores correspondentes entre **a** e **b**, normalmente criaríamos uma nova variável para "triangular" o valor:

```
>>> c = a
>>> a = b
>>> b = c
>>> a, b
(2, 1)
```

Um fenômeno interessante

Pythonicamente falando, podemos fazer algo como:

```
>>> a, b = 1, 2
```

Essa simples linha oculta um fenômeno bastante curioso da linguagem. A **Atribuição Paralela** [*destructuring assignment*]

Tanto a expressão `a, b`, quanto `1, 2` formam **tuplas**.

PS: o delimitador das tuplas são as `,`s e não os `()`.

Então, o que acontece é que estamos atribuindo paralelamente a tupla `1, 2`, na tupla `a, b`

A solução do triangulo

Logo, se podemos trocar valor entre duas tuplas, a triangulação pode ser resolvida com a atribuição paralela:

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> a, b
(2, 1)  # Olha a vírgula aí criando uma tupla :)
```

Se quiser reverter:

```
>>> b, a = a, b
>>> a, b
(1, 2)
```

Atribuição paralela

Esse comportamento pode ser usado com qualquer iterável com qualquer tamanho, se tivermos **o número certo de variáveis para o tamanho do iterável**:

```
>>> o, l, a, r = 'OLAR'
>>> o, l, a, r
('O', 'L', 'A', 'R')
```

Caso contrário, o interpretador levantará um erro:

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
  File "<python-input-15>", line 1, in <module>
    a, b = 1, 2, 3
    ^^^^
ValueError: too many values to unpack (expected 2)
```

`ValueError`: muitos valores para **UNPACK**

Unpack - Desempacotamento

As **atribuições paralelas** ganharam o nome de **Unpack*** no jargão da linguagem.

Esse comportamento pode ser manifestado de várias formas diferentes do que já vimos. Em um `for`, por exemplo:

```
for index, val in enumerate('pei!'):
   print(index, val)

0, 'p'
1, 'e'
2, 'i'
3, '!'
```

As tuplas internas são desempacotadas no laço `for`.

*Embora esse termo não apareça nem mesmo no glossário

Desempacotamento

Na versão 3.0 do Python (PEP 3132) *ganhamos* um operador que aprimora o desempacotamento. Fornecendo uma sintaxe para *pegar o "resto"*. O *****:

```
>>> primeiro, *resto = 1, 2, 3, 4
>>> resto
[2, 3, 4]
```

Todos os valores que não têm uma variável associada são "empacotados" em quem tem o *****. Algumas expressões para pensar nisso:

```
>>> primeiro, segundo, *resto_a = 1, 2, 3, 4
>>> primeiro, *resto_b, ultimo = 1, 2, 3, 4
>>> resto_a, resto_b
([3, 4], [2, 3])
```

Desempacotamento

Essas chamadas não funcionam com mais de * por expressão:

```
primeiro, *resto, *ultimo = 1, 2, 3, 4, 5
  File "<python-input-24>", line 1
    primeiro, *resto, *ultimo = 1, 2, 3, 4, 5
    ^^^^^^^^^^^^^^^^^^^^^^^^^
SyntaxError: multiple starred expressions in assignment
```

starred expression é o nome dado para o desempacotamento usando *. *Expressão estrela*.

Também não funcionam as expressões estrela sem atribuições:

```
>>> *resto
  File "<python-input-30>", line 1
    *resto
    ^^^^^^
SyntaxError: can't use starred expression here
```

Criação de novos containers

Na versão 3.5 (PEP 448), foram introduzidas novas formas de usar a expressão estrela, como o "desempacotamento" em um novo container. Algo como:

```
>>> la = [1, 2, 3]; lb = '456'

>>> [*la, *lb]
[1, 2, 3, 4, 5, 6]

>>> {*la}
{1, 2, 3}

>>> *la, 4
(1, 2, 3, 4)

>>> {*lb, '7'}
{'5', '4', '7', '6'}
```

Aninhamento de desempacotamento

Se o iterável a ser desempacotado for aninhado, o unpacking pode ser aninhado também.

Um exemplo:

```
>>> tt = (1, (2, 3), 4)
>>> a, (b, c), d = tt
>>> b, c
2, 3
```

Ou então, de uma forma mais avançada:

```
>>> tt = (1, (2, 3, 4, 5, 6), 7)
>>> a, (b, *c), d = tt
>>> c
[3, 4, 5, 6]
```

Aninhamento de desempacotamento

Um exemplo um pouco mais radical:

```
>>> d = {'chave a': [1, 2, 3, 4], 'chave b': [5, 6, 7, 8]}

>>> for chave, (primeiro, *resto) in d.items():
...     print(chave, primeiro, resto)

chave a 1 [2, 3, 4]
chave b 5 [6, 7, 8]
```

Só pra ficar mais claro:

```
>>> d.items()
dict_items([('chave a', [1, 2, 3, 4]), ('chave b', [5, 6, 7, 8])])
```

Loucura total!

Ao infinito e além...

```
d = {'chave a': [1, (2, 3, 4), 5], 'chave b': [5, (6, 7, 8), 9]}

>>> for chave, (primeiro, (meio_a, *resto_meio), *resto) in d.items():
...     print(chave, primeiro, meio_a, resto_meio, resto)

chave a 1 2 [3, 4] [5]
chave b 5 6 [7, 8] [9]
```

Bom... Acho que vocês entenderam... Dá pra ir longe...

```
>>> a, (b, *meiota, c), *resto = [1, [2, 3, 4, 5], 6, 7, 8]

>>> a, b, meiota, c, resto
(1, 2, [3, 4], 5, [6, 7, 8])
```

Dicionários e Unpacking

Embora dicionários sejam iteráveis, eles têm uma sintaxe especial. A *estrela dupla*:

```
>>> da = {'a': 1, 'b': 2}; db = {'c': 3, 'd': 4}

# Somente as chaves
>>> [*da]
['a', 'b']
>>> {*da}
{'b', 'a'}

>>> {**da}
{'a': 1, 'b': 2}

>>> {**da, **db}
{'a': 1, 'b': 2, 'c': 3, 'd': 4}

>>> {'g': 42, **da}
{'g': 42, 'a': 1, 'b': 2}
```

Alguma dessas formas de uso do * são novas pra você? Aprendeu algo novo até aqui?

Comente

Funções e métodos

Callables

Callables

Uma das características interessantes do **unpacking** é poder usar ele em:

- **Chamadas** de funções e métodos (PEP 448, 3.5+)
  - Padrões, posicionais e nomeados
- **Definições** de funções e métodos
  - Variádicos posicionais e nomeados

O que nos permite expressar uma gama de novas operações diferentes usando os nossos amigos * e **

Argumentos posicionais!

Ao definir argumentos, eles têm posições :)

```
def xpto(a, b, c, d, e, /):
    return a, b, c, d, e
```

Vamos imaginar duas listas, `la` e `lb` e testar o poder do unpacking:

```
>>> la = [1, 2, 3, 4, 5]; lb = [1, 2, 3]

>>> xpto(*la)
(1, 2, 3, 4, 5)

>>> xpto(*lb, 4, 5)
(1, 2, 3, 4, 5)

>>> xpto(*lb, *(4, 5))
(1, 2, 3, 4, 5)
```

Argumentos nomeados!

Agora com os argumentos que tem um valor "default"

```
def xpto(*, a=1, b=2, c=3, d=4, e=5):
    return a, b, c, d, e
```

Os **argumentos nomeados** precisam ser chamados com o nome do parâmetro de forma **explícita**. Algo como:

```
>>> xpto(e=10, c=14)
(1, 2, 14, 4, 10)

>>> xpto(a=1)
(1, 2, 3, 4, 5)
```

Desempacotamento em nomeados

A estrutura que tem chaves e valores é o dicionário. Nesse momento, eles brilham ao serem desempacotados em funções, usando **:

```
>>> d = {'a': 42, 'e': 42}

>>> xpto(**d)
(42, 2, 3, 4, 42)

# Unidos com a chamada padrão
>>> xpto(**d, c=42)
(42, 2, 42, 4, 42)
```

As **chaves** serão passadas como o **nome** do parâmetro e os valores serão os parâmetros.

Bagulhos sinistros...

Os dicionários podem ser usados nos posicionais...

Embora esse comportamento seja "entendível", ele é BASTANTE estranho...

```
>>> def xpto(a, b):
...     return a, b

>>> d = {'a': 1, 'b': 2}

>>> xpto(**d)  # Ok...
(1, 2)

>>> xpto(*d)  # Ué...
('a', 'b')

>>> a, b = *d
    a, b = *d
           ^^
SyntaxError: can't use starred expression here
```

Argumentos padrões

Os padrões podem ser passados das duas formas, tanto com unpacking posicional ou unpacking nomeado:

```
def xpto(a, b, c, na=1, nb=2, nc=3):
    return a, b, c, na, nb, nc
```

Podem ser de qualquer formato de unpacking:

```
>>> xpto(*range(6))
(0, 1, 2, 3, 4, 5)
>>> xpto(*range(3))
(0, 1, 2, 1 , 2, 3)
>>> xpto(1, 2, 3, **{'na': 1, 'nb': 2, 'nc': 3})
(0, 1, 2, 1 , 2, 3)
>>> xpto(**{'a': 1, 'b': 2, 'c': 3, 'na': 1, 'nb': 2, 'nc': 3})
(0, 1, 2, 1 , 2, 3)
```

Definições vs Unpacking

Os `callables` podem *empacotar* argumentos posicionais e nomeados.

Os nossos conhecidos `*args`, para agrupar posicionais, e `**kwargs` para agrupar nomeados.

Eles atuam na direção oposta das chamadas. Pois **na chamada estamos desempacotando** `func(*iter)` e na **definição do parâmetro estamos os empacotando** em `def func(*args)`.

Comportamentos opostos, mas que usam os mesmos operadores :)

`*args` e `*kwargs` podem receber de 0..N argumentos, ou seja, a sua quantidade pode variar. O termo técnico para isso é **Variádico**. Quando a quantidade de argumentos *varia*.

*args

O `*args` é responsável por empacotar todos os parâmetros reais e posicionais não definidos em parâmetros formais. Os excedentes dos posicionais.

```
def xpto(a, *args):
    return a, args
```

Essa operação empacota todos os argumentos passados após o primeiro, **0..N**:

```
>>> xpto(1)
(1, ())
>>> xpto(1, 2)
(1, (2,))
>>> xpto(1, 2, 3, 4, 5)
(1, (2, 3, 4, 5))
>>> xpto(*[1, 2, 3, 4])
(1, (2, 3, 4))
```

**kwargs

Na mesma via de empacotar, os **`**kwargs`** são responsáveis por agrupar de **0..N** os argumentos nomeados não foram definidos formalmente.

```
def xpto(a=1, **kwargs):
    return a, kwargs
```

O que nos forneceria algo como:

```
>>> xpto(a=1, b=2)
(1, {'b': 2})
>>> xpto(**{'a': 1, 'b': 2})
(1, {'b': 2})
>>> xpto(**{'b': 2, 'c': 3})
(1, {'b': 2, 'c': 3})
```

Alguma dessas formas de uso do * são novas pra você? Aprendeu algo novo até aqui?

Comente

Anotações

De tipos para packing

Começando do começo

As **anotações de tipos** são **metadados**, assim como as **docstrings**, que foram inseridas na PEP 484 (3.5+), teorizadas na PEP 483 e expandidas na PEP 526 (3.6+) que traziam ao python, via ferramentas externas, verificação/checagem de tipos graduais.

A `484` tráz dois tipos importantes na nossa jornada:

- `typing.Tuple`: Tipo para anotar tuplas
- `typing.Dict`: Tipo para anotar dicionários

Que foram *atualizados* para os containers genéricos na PEP 585 (3.9+): `tuple` e `dict`

E também define os usos de anotações para `*args` e `**kwargs`.

`tuple` e `*args`

Como o empacotamento em `*args` é dado em tuplas, é importante entender a anotação das tuplas:

```python
# Define uma tupla com exatamente 3 valores onde todos são `int`
t0: tuple[int, int, int] = (1, 2, 3)

# Define uma tupla com N valores onde todos são `int`
t1: tuple[int, ...] = tuple(range(10))
```

A ideia da anotação de `*args` é usar somente um tipo e que todos os valores empacotados terão. Algo como:

```python
def xpto(*args: int) -> tuple[int, ...]: return args
```

Onde `*args: int` empacota os argumentos como `tuple[int, ...]`.

`tuple` e `*args`

Caso a tupla tenha mais de um tipo, é possível usar `Union` ou `|`:

```
def xpto(*args: int | str): ...
```

O que diz que a tupla "empacotada" pode ter o valor `int | str` em qualquer posição e o tamanho continha sendo *0..N*.

No caso, não é possível determinar o tamanho e em quais posições o tipo é `int` ou `str`. O que é possível nas tuplas:

```
t: tuple[int, float, str] = (42, .5, 'pei!')
```

Variadic Generics - TypeVarTuple (PEP 646)

A PEP 646 introduz, na versão 3.11+, uma condição *segura* de tipagem para anotar os variádicos posicionais.

Os **variádicos genéricos**:

```
from typing import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')

def xpto(*args: Unpack[Ts]) -> tuple[*Ts]:
   return args

x = xpto(1, 2, 3)
```

Onde dizemos ao sistema de tipos que "vários valores vão chegar" e ele vai inferir os tipos de toda a tupla empacotada.

TypeVarTuple

Essa adição permite uma segurança para quando os vadiádicos forem usados. Pois, dessa forma, eles são mantidos durante a execução do fluxo:

```
Ts = TypeVarTuple('Ts')

def xpto(*args: Unpack[Ts]) -> tuple[*Ts]:
   return args

x = xpto(1, 2, 3)
reveal_type(x)

---
mypy f.py
f.py:11: note: Revealed type is
    "tuple[builtins.int, builtins.int, builtins.int]"
```

O que resolve o tamanho e os tipos exatos da tupla passada a uma função.

TypeVarTuple

Outro ponto da PEP 646 é que foram inseridas duas formas para definir os variádicos posicionais.

A que já vimos, usando o tipo **`Unpack`**, que mantém retrocompatibilidade com versões anteriores via typing_extensions:

```
Ts = TypeVarTuple('Ts')

def xpto(*args: Unpack[Ts]) -> tuple[*Ts]: ...
```

E a nova sintaxe de *type star*:

```
def xpto(*args: *Ts) -> tuple[*Ts]: ...
```

Onde os tipos variádicos podem ser chamados diretamente como **`*`**.

Type Parameter Syntax – PEP 695

Já a PEP 695 (3.12+) adiciona os parâmetros de tipos, que também podem definir variádicos posicionais. Simplificando a definição:

```
def xpto[*Ts](*args: *Ts) -> tuple[*Ts]:
   return args

x = xpto(1, 2, 3)

reveal_type(x)
```

Obtendo o mesmo resultado:

```
mypy f.py
f.py:11: note: Revealed type is
    "tuple[builtins.int, builtins.int, builtins.int]"
```

Variádicos nomeados

Antes de adentrarmos no (`**kwargs`), temos que entender a sintaxe dada aos dicionários pela `484/585`:

```
d: dict[int, str] = {0: 'valor'}
```

Onde os tipos eram dados para `[chave, valor]`.

Para os `**kwargs`, a sintaxe usada definia somente o tipo usado pelas chaves:

```
def xpto(**kwargs: str) -> dict[str, str]:
    return kwargs
```

Fazendo todas as chaves serem representadas por `str`.

"""Variádicos""" nomeados

Também na 3.12, a PEP 692 introduz o conceito de genéricos variádicos para o `**kwargs` por meio dos `TypedDict` (PEP 589, 3.8+):

Onde as chaves são strings e os valores têm tipos anotados:

```python
from typing import TypedDict, Unpack


class Kwargs(TypedDict, total=False):
    na: int
    nb: float
    nc: str


def xpto(**kwargs: Unpack[Kwargs]) -> Kwargs:
    return kwargs
```

A ideia é que os `**kwargs` sejam previamente listados e tipados.

O que, no primeiro, momento faz exatamente o inverso do que é esperado do `**kwargs`. Não?

Como funciona?

Mas... vamos usar primeiro e criticar depois...

????

PEP 692

Pendido a revelação da função (`reveal_type(xpto)`), temos algo como:

```
mypy teste.py
teste.py:15: note: Revealed type is
    "def (
        **kwargs: Unpack[
            TypedDict('teste.Kwargs', {
                'na'?: builtins.int,
                'nb'?: builtins.float,
                'nc'?: builtins.str
            }
    )]) -> TypedDict('teste.Kwargs', {
                'na'?: builtins.int,
                'nb'?: builtins.float,
                'nc'?: builtins.str
            }"
```

PEP 692

Olhando o resultado da função:

```
x = xpto()
reveal_type(x)
```

Teríamos algo como:

```
mypy f.py
f.py:15: note: Revealed type is
    TypedDict('f.Kwargs',
        {
            'na'?: builtins.int,
            'nb'?: builtins.float,
            'nc'?: builtins.str
        }
    )
```

PEP 692

É interessante notar aqui que a anotação de tipo **limita** o uso da funcionalidade como ela foi pensada.

Pois, embora seja interessante agrupar os possíveis parâmetros, eles deixam de ser "variádicos" e se tornam somente coisas opcionais. Se você receber algo que não foi mapeado, vai ser uma infração de tipos.

Mas calma... A gente vai chegar lá

Sempre usar variádicos no typing?

Existe um caso especial onde anotar com variádicos não faz sentido. Em decoradores de forma geral. Pois elas envolvem a função, mas não a definem.

Para esses casos, o `ParamSpec` (PEP 612, 3.10+) deve ser usado:

```python
def decorator[T, **P](func: Callable[P, T]) -> Callable[P, T]:
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        return func(*args, **kwargs)
    return inner
```

Note que ****P não cria um unpacking de nomeados variádicos**. Ele cria um `ParamScpec`. O que gera uma confusão tremenda.

```python
from typing import ParamSpec
P = ParamSpec('P')  # isso pode evitar a confusão com o `**P`
```

Você já conhecia os tipos que vimos hoje? Qual você mais gostou?

Comente

O futuro vindouro...

3.15

TypedDict with Typed Extra Items (PEP 728)

A ideia da PEP 728 (3.15+) é flexibilizar os `TypedDicts`, em geral.

Mas isso é extremamente benéfico no caso dos `**kwargs`, pois eles poderão ser chamados com argumentos nomeados a mais que os especificados no `TypedDict`:

```
class Kwargs(TypedDict, total=False, extra_items=bool):
    na: int
    nb: float
    nc: str
```

Onde qualquer argumento a mais do que os mapeados pode ser passado e terá um tipo específico.

PS: Você pode usar união e etc...
PS2: Ainda não acho que seja a melhor solução, mas...

Unpacking in Comprehensions (PEP 798)

A PEP 798 **ainda está em draft**, pode não ser aceita

Aqui a ideia é tornar o resultado do que foi empacotado linear:

```
>>> [*x for x in [[1,2], [3, 4]]]
[1, 2, 3, 4]
```

O que atualmente pode ser feito assim:

```
>>> list(
    chain.from_iterable([x for x in [[1,2], [3, 4]]])
)
[1, 2, 3, 4]
```

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

Referências

- PSF. Instruções compostas. Disponível em: https://docs.python.org/3/reference/compound_stmts.html . Acesso em: 28 ago. 2025.
- PSF. typing — Support for type hints. Disponível em: https://docs.python.org/3/library/typing.html . Acesso em: 1 set. 2025.
- Callables — typing documentation. Disponível em: https://typing.python.org/en/latest/spec/callables.html#annotating-args-kwargs . Acesso em: 26 ago. 2025.
- PEP 448 – Additional Unpacking Generalizations | peps.python.org. Disponível em: https://peps.python.org/pep-0448/ . Acesso em: 26 ago. 2025.
- PEP 589 – TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys | peps.python.org. Disponível em: https://peps.python.org/pep-0589/ . Acesso em: 30 ago. 2025.
- PEP 612 – Parameter Specification Variables | peps.python.org. Disponível em: https://peps.python.org/pep-0612/ . Acesso em: 1 set. 2025.
- PEP 646 – Variadic Generics | peps.python.org. Disponível em: https://peps.python.org/pep-0646/ . Acesso em: 26 ago. 2025.
- PEP 692 – Using TypedDict for more precise **kwargs typing | peps.python.org. Disponível em: https://peps.python.org/pep-0692/ . Acesso em: 26 ago. 2025.
- PEP 695 – Type Parameter Syntax | peps.python.org. Disponível em: https://peps.python.org/pep-0695/ . Acesso em: 29 ago. 2025.
- PEP 728 – TypedDict with Typed Extra Items | peps.python.org. Disponível em: https://peps.python.org/pep-0728/ . Acesso em: 28 ago. 2025.
- PEP 798 – Unpacking in Comprehensions | peps.python.org. Disponível em: https://peps.python.org/pep-0798/ . Acesso em: 26 ago. 2025.
- PEP 3132 – Extended Iterable Unpacking | peps.python.org. Disponível em: https://peps.python.org/pep-3132/ . Acesso em: 26 ago. 2025.
- RAMALHO, Luciano. Python fluente. 1. ed.: Novatec Editora, 2022.
- RAMALHO, Luciano. Python Fluente, Segunda Edição (2023). Disponível em: https://pythonfluente.com/2/#iterable_unpacking_sec . Acesso em: 26 ago. 2025.