Dataclasses

Live de Python #150

Roteiro

1. O problema
Uma introdução sobre o por quê e a utilidade das dataclasses

2. PO
Obsessão por primitivos

3. Tipagem
Uma revisão da live #84

4. Dataclasses
O que não foi nomeado ainda

5. fields
Trabalhando os argumentos

6. Alternativas não builtin
Um breve resumo do que não falamos

picpay.me/dunossauro

apoia.se/livedepython

PIX

Ajude o projeto

Ademar, Alex Menezes, Alexandre Fernandes, Alexandre Harano, Alexandre Souza, Alexandre Tsuno, Alysson Oliveira, Amaziles Jose, Andre Rodrigues, André Almeida, Antonio Cassiano, Bianca Rosa, Bruno Fernandes, Bruno Rocha, Caio Vinicius, Carlos Alberto, César Moreira, César Túlio, Davi Alves, David Kwast, Diego Moreira, Dilenon Stefan, Douglas Bastos, Edgard Sampaio, Edivaldo Venancio, Edson Braga, Eduardo Marcos, Eduardo Sidney, Elias Da, Eugenio Mazzini, Everton Alves, Fabio Castro, Fabrício Vilela, Faricio Lima, Fernando Lanfranchi, Flavkaze, Franklin Sousa, Fábio Serrão, Gabriel Simonetto, Gabriela Santiago, Geandreson Costa, Gladson Araujo, Guilherme Felitti, Guilherme Marson, Guilherme Ostrock, Henrique Machado, Hélio De, Isaac Ferreira, Israel Azevedo, Italo Bruno, Jeison Sanches, Johnny Tardin, Jonatas Baldin, Jonatas Leon, Jones Ferreira, Jorge Luiz, José Willia, Jovan Costa, João Lugão, João Paulo, Juan Ernesto, Jônatas Silva, Júlia Kastrup, Kaneson Alves, Leonardo Cordeiro, Leonardo Galani, Leonardo Ribeiro, Lorena Carla, Lucas Barreto, Lucas Barros, Lucas Ferreira, Lucas Sartor, Luiz Lima, Maiquel Leonel, Maiquel Leonel, Marcela M, Marcelo Pontes, Maria Clara, Natan Cervinski, Nicolas Teodosio, Otavio Carneiro, Patric Lacouth, Patrick Henrique, Paulo Henrique, Paulo Henrique, Pedro Baesse, Pedro Martins, Peterson W, Rafael De, Reinaldo Chaves, Renan Gomes, Renne Rocha, Rodrigo Ferreira, Rodrigo Vaccari, Ronaldo Fraga, Rubens Gianfaldoni, Sandro Roberto, Silvio Xm, Thiago Dias, Thiago Martins, Tyrone Damasceno, Valdir Junior, Victor Matheus, Vinícius Bastos, Vinícius Borba, Wesley Mendes, Willian Lopes, Willian Lopes, Willian Vieira, Wilson Beirigo

Obrigado você

Uma introdução sobre o porquê e as utilidades

O problema

Quantas vezes você já passou por esse cenário?

```python
dados = [
   {
     'nome': 'Eduardo',
     'sobrenome': 'Mendes',
     'telefone': {'residencial': '1111-111', 'móvel': '999-999-999'},
     'ddd', 19
   },
  {
    'nome': 'Fausto',
    'sobrenome': 'mago',
    'telefone': {'residencial': '2222-2222', 'móvel': '888-888-888'},
    'ddd', 51
  },
]
```

Quantas vezes você já passou por esse cenário?

```python
dados = [
    {
        'nome': 'Eduardo',
        'sobrenome': 'Mendes',
        'telefone': {'residencial': '1111-111', 'móvel': '999-999-999'},
        'ddd': 19,
    },
    ...
]

nomes_completos = [f"{dado['nome']} {dado['sobrenome']}" for dado in dados]

eduardo_telefone_residencial = [
    dado['telefone']['residencial'] for dado in dados if dado['nome'] == 'Eduardo'
][0]
```

PO

Obsessão por Primitivos

Obsessão por primitivos?

A obsessão por primitivos é caracterizada por estruturas que podem ser representadas por objetos concretos, mas ela é representada por um tipo “primitivo” da linguagem. Como o nosso dicionário:

```
1    {
2        'nome': 'Eduardo',
3        'sobrenome': 'Mendes',
4        'telefone': {'residencial': '1111-111', 'móvel': '999-999-999'},
5        'ddd': 19,
6    }
```

Obsessão por primitivos?

Quando não respeitamos uma regra de domínios, acabamos fazendo funções ou rotinas “soltas” para conseguir obter o resultado esperado.

```python
dados = [
    {
        'nome': 'Eduardo',
        'sobrenome': 'Mendes',
        'telefone': {'residencial': '1111-111', 'móvel': '999-999-999'},
        'ddd': 19,
    },
    ...
]

nomes_completos = [f"{dado['nome']} {dado['sobrenome']}" for dado in dados]
```

Para tentar fugir dessa “dificuldade” de acesso, temos as nametuples, que são “construtoras” de objetos

NamedTuples

Um exemplo com namedtuple

Fugindo da complexidade do dicionário

```python
from collections import namedtuple

pessoa = namedtuple('Pessoa', 'nome sobrenome telefone ddd')

dados = [
    pessoa('Eduardo', 'Mendes', {'residencial': '1111-111', 'móvel': '999-999-999'}, 19),
    pessoa('Fausto', 'mago', {'residencial': '2222-2222', 'móvel': '888-888-888'}, 51),
]

eduardo = dados[0] # OU
eduardo = pessoa('Eduardo', 'Mendes', {'residencial': '1111-111', 'móvel': '999-999-999'}, 19)

eduardo.nome # 'Eduardo'
eduardo.sobrenome # 'Mendes'
```

Explorando as namedtuples

- Imutabilidade
  - Não recebe novas atribuições para valores
  - Tem o mesmo estado até o final da execução
- Representação
  - Já implementa __repr__ e __str__
- É comparável
  - Implementa __eq__
- É montada de maneira programática
  - namedtuple(‘nome’, *args)

Tipagem

Adicionando checagem estática em namedtuples (PEP-526)

typing.NamedTuple

Caso você queira uma checagem estática de tipos, você pode implementar uma NamedTuple tipada.

```python
from typing import NamedTuple, Dict


class Pessoa(NamedTuple):
    nome: str
    sobrenome: str
    telefone: Dict[str, str]
    ddd: int


eduardo = Pessoa('Eduardo', 8, {'residencial': '1111-111', 'móvel': '999-999-999'}, 19)
```

Rodando o mypy para checagem

```python
from typing import NamedTuple, Dict


class Pessoa(NamedTuple):
    nome: str
    sobrenome: str
    telefone: Dict[str, str]
    ddd: int


eduardo = Pessoa('Eduardo', 8, {'residencial': '1111-111', 'móvel': '999-999-999'}, 19)
```

```
$ mypy exemplo_03.py
exemplo_03.py:11: error: Argument 2 to "Pessoa" has incompatible type "int"; expected "str"
Found 1 error in 1 file (checked 1 source file)
```

Problemas dessa abordagem

- Imutabilidade
  - Pode ser que você não queira
- Extensão
  - Não é possível estender tuplas (herança)
- A representação o objeto é sempre completa
- Não é possível criar métodos

Como fugir da obsessão por primitivos?

Existe um pattern chamado ‘Value Object’, que tem como ideia criar uma classe para representar o domínio. Ou seja, devemos criar um objeto para representar uma abstração.

```python
from typing import Dict


class Pessoa:
    def __init__(self, nome: str, sobrenome: str, telefone: Dict[str, str], ddd: int):
        self.nome = nome
        self.sobrenome = sobrenome
        self.telefone = telefone
        self.ddd = ddd


eduardo_1 = Pessoa('Eduardo', 8, {'residencial': '1111-111', 'móvel': '999-999-999'}, 19)
eduardo_2 = Pessoa('Eduardo', 8, {'residencial': '1111-111', 'móvel': '999-999-999'}, 19)
```

Deu certo?

Sim, deu certo. Conseguimos não usar nenhuma forma. Agora temos como vantagem:

- Mutabilidade / Imutabilidade
  - Vai ao seu gosto
- Expansão
  - Podemos usar herança
- Métodos
  - Podemos criar métodos

Qual o problema então?

```
$ mypy exemplo_05.py
exemplo_05.py:12: error: Argument 2 to "Pessoa" has incompatible type "int"; expected "str"
exemplo_05.py:13: error: Argument 2 to "Pessoa" has incompatible type "int"; expected "str"
Found 2 errors in 1 file (checked 1 source file)
```

```
>>> eduardo_1 = Pessoa('Eduardo', 'Mendes', {'residencial': 1, 'móvel': 2}, 19)
>>> eduardo_2 = Pessoa('Eduardo', 'Mendes', {'residencial': 1, 'móvel': 2}, 19)
>>> eduardo_1 == eduardo_2
False
>>> eduardo_1
<__main__.Pessoa object at 0x7f2393d9c6d0>
```

Como ter o mesmo efeito das namedtuples?

A implementação simplória é cara.

Para ter a mesma funcionalidade tivemos que implementar:

- __eq__
- __repr__

```python
from typing import Dict


class Pessoa:
    def __init__(self, nome: str, sobrenome: str, telefone: Dict[str, str], ddd: int):
        self.nome = nome
        self.sobrenome = sobrenome
        self.telefone = telefone
        self.ddd = ddd

    def __eq__(self, other):
        return all([
            self.nome == other.nome,
            self.sobrenome == other.sobrenome,
            self.telefone == other.telefone,
            self.ddd == other.ddd
        ])

    def __repr__(self):
        return f'Pessoa({self.nome}, {self.sobrenome}, {self.telefone}, {self.ddd})'
```

dataclasses

O que não nomeamos até agora (PEP-557)

Dataclasses

Bom, você deve estar se perguntando agora, precisávamos ver TUDO isso? Acredito que sim, na PEP-557 (Data Classes), Eric Smith, define uma dataclass como:

> “mutable namedtuples with defaults (...) which inspects a class definition for variables with type annotations”
>
> “namedtuples mutáveis com valores default que inspecionam a definição da classe para variáveis com anotações de tipo”

https://www.python.org/dev/peps/pep-0557/

Dataclasses

As dataclasses foram feitas para facilitar objetos de domínio específico. Elas na implementam a representação, a mutabilidade ou imutabilidade, a comparação e o hash das classes. Assim, sendo mais simples de implementar a preço de um decorador. Por serem classes, podem herdar outras classes e também criar métodos customizados.

Qual a cara de uma dataclass?

```python
from typing import Dict
from dataclasses import dataclass

@dataclass
class Pessoa:
    nome: str
    sobrenome: str
    telefone: Dict[str, str]
    ddd: int
```

Ela é tudo isso mesmo?

```
1 >>> eduardo_1 = Pessoa('Eduardo', 'Mendes', {'residencial': 1, 'móvel': 2}, 19)
2 >>> eduardo_2 = Pessoa('Eduardo', 'Mendes', {'residencial': 1, 'móvel': 2}, 19)
3 >>> eduardo_1 == eduardo_2
4 True
5 >>> eduardo_1.nome
6 'Eduardo'
7 >>> eduardo_1
8 Pessoa(nome='Eduardo', sobrenome='Mendes', telefone={'residencial': 1, 'móvel': 2}, ddd=19)
```

O método post__init

```python
@dataclass
class Pessoa:
    nome: str
    sobrenome: str
    telefone: Dict[str, str]
    ddd: int
    nome_completo: str = None

    def __post_init__(self):
        self.nome_completo = f'{self.nome} {self.sobrenome}'
```

A ideia do post_init é ser executado após a inicialização da classe.

Definimos um valor default (None) para ‘nome_completo’ e preenchemos após a inicialização

O decorador de dataclass

O decorador diz o que esperamos dessa classe, o que faz ela ser totalmente customizável:

- init=False
  - se quisermos que a classe não inicie
- repl=False
  - Se não quisermos a representação da classe
- eq=False
  - Se não quisermos a comparação
- frozen=False
  - Se não quisermos que seja mutável

O decorador dataclasse

```python
@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False, frozen=False)
class Pessoa:
    nome: str
    sobrenome: str
    telefone: Dict[str, str]
    ddd: int
```

field

Trabalhando os argumentos

A função field

A função field nos auxilia nas escolhas e o que fazer com os atributos. Vai ser carregado depois? Tem um valor padrão? Se for vazio, posso fazer um factory? Devo exibir esse dado na representação?

```python
from typing import Dict
from dataclasses import dataclass, field

@dataclass(frozen=True)
class Pessoa:
    nome: str
    sobrenome: str
    ddd: int = field(repr=False)
    telefone: Dict[str, str] = field(default_factory=dict)
    nome_completo: str = field(init=False)
```

A função field

```python
@dataclass(frozen=True)
class Pessoa:
    nome: str
    sobrenome: str
    ddd: int = field(repr=False)
    telefone: Dict[str, str] = field(default_factory=dict)
    nome_completo: str = field(init=False)
```

```
>>> p = Pessoa()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 3 required positional arguments: 'nome', 'sobrenome', and 'ddd'
```

O estado da arte

```python
from typing import Dict
from dataclasses import dataclass, field

@dataclass
class Pessoa:
    nome: str
    sobrenome: str
    ddd: int = field(repr=False)
    telefone: Dict[str, str] = field(default_factory=dict)
    nome_completo: str = field(init=False)

    def __post_init__(self):
        self.nome_completo = f'{self.nome} {self.sobrenome}'
```

```
>>> p = Pessoa('Fausto', 'Mago', 13)
>>> p.telefone |= {'móvel': '99'}
>>> p
Pessoa(nome='Fausto', sobrenome='Mago', telefone={'móvel': '99'}, nome_completo='Fausto Mago')
```

Conversões

```
>>> from dataclasses import asdict, astuple
>>> p = Pessoa('Fausto', 'Mago', 13)
>>> asdict(p)
# {'nome': 'Fausto', 'sobrenome': 'Mago', 'ddd': 13, 'telefone': {}, 'nome_completo': 'Fausto Mago'}
>>> astuple(p)
# ('Fausto', 'Mago', 13, {}, 'Fausto Mago')
>>> p
Pessoa(nome='Fausto', sobrenome='Mago', telefone={}, nome_completo='Fausto Mago')
```

Conversões

```
>>> p = Pessoa(nome='Fausto', sobrenome='Mago', 13)

>>> pt = astuple(p)
>>> Pessoa(*pt[:-1])
# Pessoa(nome='Fausto', sobrenome='Mago', telefone={}, nome_completo='Fausto Mago')

>>> pd = asdict(p)
>>> del pd['nome_completo']>>> Pessoa(**pd)
# Pessoa(nome='Fausto', sobrenome='Mago', telefone={}, nome_completo='Fausto Mago')
```

Alternativas

Não builtin

Alternativas legais

- Attrs
  - Biblioteca mais antiga e com mais funções
  - Funciona com typing e sem typing
  - “Classes sem boilerplate”
  - Possui validadores
- Pydantic
  - Biblioteca mais “nova”
  - Só aceita typing
  - Padrão no fastAPI
  - Possui validadores

Comparação rápida

Features	dataclass	Attrs	Pydantic
Frozen	x	x	x
Defaults	x	x	x
To tuple	x	x	x
To dict	x	x	x
Validators		x	x
Converters		x	x
slots		x	
Criação programática	x	x	

picpay.me/dunossauro

apoia.se/livedepython

PIX

Ajude o projeto