



DATACLASSES NO PYTHON

Guia completo de Dataclasses: menos boilerplate, `__init__` automático, `field()`, `frozen`, `order`, `asdict()`, casos práticos (configuração, DTO), `dataclass` vs classe normal vs `NamedTuple`.

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 

 **Atualizado para Python 3.13** (Dezembro 2025)

Conteúdo enriquecido com comparações, `field()` avançado e casos práticos.

Salve salve Pythonista!

Dataclasses eliminam boilerplate de classes de dados! Gera automático: `__init__`, `__repr__`, `__eq__` e mais.

Neste guia, você vai aprender: -  **Menos código** - `__init__` automático -  **field()** - Valores padrão e factory -  **frozen** - Imutabilidade -  **Comparações** - Dataclass vs classe normal vs NamedTuple

Neste artigo, vamos falar sobre as **dataclasses** no Python, uma funcionalidade introduzida na versão 3.7 da linguagem.

As dataclasses são uma forma simplificada de criar classes que armazenam dados, proporcionando uma maneira fácil e rápida de implementar classes com atributos e métodos específicos para manipulação de dados.

Vamos ver sua utilidade e como podemos definí-las no Python.

Então vamos nessa!

```
<div class="suggestions-header">
    <h2>Leia também</h2>
</div>
<div class="suggestions-body">
    <ul>
        <li>
            <a href="https://pythonacademy.com.br/blog/introducao-a-programacao-orientada-a-objetos-no-python" target="_blank">
                Introdução à Programação Orientada a Objetos no Python
            </a>
        </li>
        <li>
            <a href="https://pythonacademy.com.br/blog/classes-e-objetos-no-python" target="_blank">
                Classes e Objetos no Python
            </a>
        </li>
    </ul>
</div>
```

Por que utilizar **dataclasses**?

É comum em Python utilizarmos classes para representar estruturas de dados, como por exemplo, um usuário com nome, idade e email.

Para criar essa classe, precisamos definir os atributos, o método construtor, métodos e outras funcionalidades específicas.

Isso pode ser um pouco trabalhoso e propenso a erros.

As **dataclasses** surgem como uma solução para tornar esse processo de criação de classes para armazenamento de dados mais simples, reduzindo a quantidade de código necessário.

Com as dataclasses, podemos criar essas classes de forma mais declarativa e legível, concentrando apenas na definição dos atributos e deixando o Python gerar o código necessário automaticamente.

Como utilizar `dataclasses`

Para usar as dataclasses, precisamos importar o módulo `dataclasses`.

Vamos começar criando uma classe simples que representa um usuário:

```
import dataclasses

@dataclasses.dataclass
class Usuario:
    nome: str
    idade: int
    email: str
```

Essa classe é decorada com o `@dataclasses.dataclass`, indicando que queremos utilizar a funcionalidade de dataclass do Python.

Em seguida, declaramos os atributos da classe, junto com os seus tipos.

A partir desse momento, o Python já faz automaticamente algumas coisas para nós:

- Cria um método construtor que recebe os valores dos atributos e atribui esses valores aos atributos correspondentes;
- Cria métodos *getters* e *setters* para cada atributo;
- Implementa os métodos especiais `__repr__` e `__eq__` para exibição amigável do objeto e comparação de igualdade, respectivamente.

Agora, podemos criar objetos dessa classe e acessar seus atributos da seguinte forma, assim como fazemos com classes comuns:

```
usuario1 = Usuario("João", 25, "joao@exemplo.com")

print(usuario1.nome)
print(usuario1.idade)
print(usuario1.email)
```

Ao executar esse código, teremos a seguinte saída:

```
João
25
joao@exemplo.com
```

Podemos também modificar os valores dos atributos:

```
usuario1.nome = "Maria"
usuario1.idade = 30
usuario1.email = "maria@exemplo.com"

print(usuario1.nome)
print(usuario1.idade)
print(usuario1.email)
```

Agora teremos a seguinte saída:

```
Maria
30
maria@exemplo.com
```

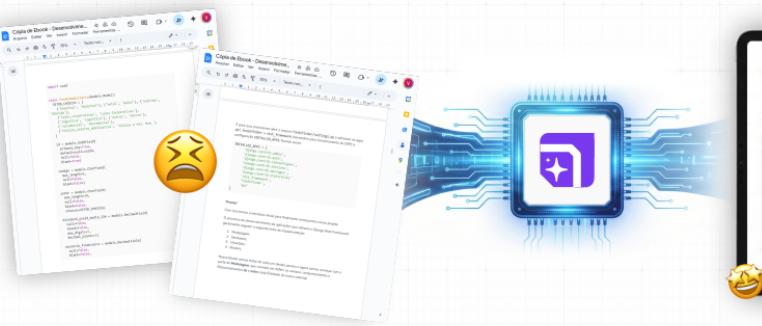
Como podemos perceber, o Python nos permite trabalhar facilmente com as data-classes, tanto na criação quanto na modificação dos valores dos atributos.

 Estou desenvolvendo o **DevBook**, uma plataforma que usa IA para gerar ebooks técnicos profissionais. Depois de ler, dá uma passada no site!

 DevBook

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! PRIMEIRO CAPÍTULO 100% GRÁTIS 

Atributos opcionais e valores padrão

Em algumas situações, pode ser necessário definir atributos opcionais em nossas dataclasses.

Para isso, basta definir o valor padrão como `None` ou algum outro valor desejado:

```
@dataclasses.dataclass  
class Usuario:  
    nome: str  
    idade: int = 0  
    email: str = None
```

Agora podemos criar objetos da classe `Usuario` sem fornecer todos os atributos:

```
usuario1 = Usuario("João")  
usuario2 = Usuario("Maria", 30)  
  
print(usuario1.nome)  
print(usuario1.idade)  
print(usuario1.email)  
  
print(usuario2.nome)  
print(usuario2.idade)  
print(usuario2.email)
```

A saída será:

```
João  
0  
None  
Maria  
30  
None
```

Dessa forma, podemos criar objetos da classe `Usuario` com diferentes combinações de atributos obrigatórios e opcionais.

Métodos personalizados

Além de gerar automaticamente os métodos getters e setters, as dataclasses nos permitem **implementar os nossos próprios métodos personalizados**.

Esses métodos podem ser implementados normalmente, como em qualquer outra classe.

Vamos adicionar um método `saudacao` à nossa classe `Usuario`, que retorna uma saudação personalizada com nome e idade do usuário:

```
@dataclasses.dataclass
class Usuario:
    nome: str
    idade: int = 0
    email: str = None

    def saudacao(self):
        return f"Olá, meu nome é {self.nome} e tenho {self.idade} anos."
```

Agora podemos invocar esse método em um objeto da classe `Usuario`:

```
usuario1 = Usuario("João", 25)
usuario2 = Usuario("Maria", 30)

print(usuario1.saudacao())
print(usuario2.saudacao())
```

A saída será:

```
Olá, meu nome é João e tenho 25 anos.
Olá, meu nome é Maria e tenho 30 anos.
```

Podemos adicionar quantos métodos desejarmos às nossas dataclasses, tornando-as mais ricas em funcionalidades de acordo com as necessidades da nossa aplicação.

Comparação de objetos

Ao criar uma dataclass, o Python automaticamente implementa o método especial `__eq__` para realizar a comparação de igualdade entre objetos.

Esse método compara os valores dos atributos das instâncias, garantindo que dois objetos sejam considerados iguais se seus atributos tiverem os mesmos valores.

```
usuario1 = Usuario("João", 25, "joao@exemplo.com")
usuario2 = Usuario("Maria", 30, "maria@exemplo.com")
usuario3 = Usuario("João", 25, "joao@exemplo.com")

print(usuario1 == usuario2)
print(usuario1 == usuario3)
```

A saída será:

```
False
True
```

Como podemos observar, a comparação `usuario1 == usuario2` retorna `False`, pois os valores dos atributos são diferentes, enquanto a comparação `usuario1 == usuario3` retorna `True`, uma vez que os objetos têm os mesmos valores para os atributos.

Dataclass vs Classe Normal vs NamedTuple

```
from dataclasses import dataclass
from typing import NamedTuple

# Classe Normal: Muito boilerplate
class PessoaNormal:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def __repr__(self):
        return f"PessoaNormal(nome='{self.nome}', idade={self.idade})"

    def __eq__(self, other):
        return self.nome == other.nome and self.idade == other.idade

# Dataclass: Automático!
@dataclass
class PessoaData:
    nome: str
    idade: int

# NamedTuple: Imutável
class PessoaTuple(NamedTuple):
    nome: str
    idade: int
```

Comparação

Feature	Classe Normal	Dataclass	NamedTuple
Boilerplate	✗ Alto	✓ Mínimo	✓ Mínimo
Mutável	✓ Sim	✓ Sim (default)	✗ Não
Type hints	✓ Opcional	✓ Obrigatório	✓ Obrigatório
repr	✗ Manual	✓ Auto	✓ Auto
eq	✗ Manual	✓ Auto	✓ Auto
Valores padrão	✓ Sim	✓ Sim	✓ Sim
Herança	✓ Sim	✓ Sim	✗ Não

Quando Usar Cada Um?

✓ Use Dataclass quando: - Precisa armazenar **dados estruturados** - Quer **menos boilerplate** - Dados são **mutáveis** - Precisa **métodos customizados**

✓ Use NamedTuple quando: - Dados são **imutáveis** - Precisa usar como **chave de dict** - Leve e **rápido** (performance) - Não precisa herança

✓ Use Classe Normal quando: - Lógica **complexa** - Não é apenas armazenamento - Precisa **controle total**

Casos Práticos

1. Configuração de App

```
from dataclasses import dataclass, field

@dataclass
class AppConfig:
    host: str = "localhost"
    port: int = 8000
    debug: bool = False
    allowed_hosts: list = field(default_factory=list)

config = AppConfig()
print(config) # AppConfig(host='localhost', port=8000, ...)

config_prod = AppConfig(host="0.0.0.0", port=80, allowed_hosts=['example.com'])
```

2. Data Transfer Object (DTO)

```
from dataclasses import dataclass, asdict
import json

@dataclass
class UserDTO:
    id: int
    username: str
    email: str
    active: bool = True

user = UserDTO(1, "alice", "alice@example.com")

# Converter para dict
user_dict = asdict(user)
print(user_dict)

# Serializar para JSON
json_data = json.dumps(asdict(user))
print(json_data) # {"id": 1, "username": "alice", ...}
```

3. Dados Imutáveis (frozen)

```
from dataclasses import dataclass

@dataclass(frozen=True) # Imutável!
class Coordenada:
    x: float
    y: float

coord = Coordenada(10.5, 20.3)
# coord.x = 15 # ✗ Erro! frozen=True

# Pode usar como chave de dict
cache = {coord: "valor"}
```

Conclusão

Neste guia de **Dataclasses**, você aprendeu:

- ✓ **Menos boilerplate** - `__init__`, `__repr__`, `__eq__` automáticos
- ✓ **field()** - Valores padrão com factory
- ✓ **frozen** - Imutabilidade
- ✓ **asdict()/astuple()** - Conversão
- ✓ **Comparações** - Dataclass vs Normal vs NamedTuple

Principais lições: - Dataclasses **eliminam boilerplate** de classes de dados - Use `field(default_factory=list)` para **valores mutáveis** - `frozen=True` torna instâncias **imutáveis** - `order=True` adiciona `__lt__`, `__le__`, etc. - **NamedTuple** para dados imutáveis simples

Próximos passos: - Pratique `@property` com dataclasses - Explore `field(init=False)` e `field(repr=False)` - Aprenda Pydantic para validação avançada - Estude attrs (alternativa a dataclasses)

As dataclasses são uma adição muito útil ao Python para simplificar a criação de classes de armazenamento de dados.

Neste artigo, vimos como utilizar dataclasses no Python, declarando os atributos das classes e aproveitando todas as funcionalidades geradas automaticamente.

Além disso, exploramos a criação de atributos opcionais, valores padrão, métodos personalizados e a comparação de igualdade entre objetos.

As dataclasses são uma ferramenta poderosa e muito útil para desenvolvimento em Python, tornando o código mais limpo, legível e produtivo.

Espero que este artigo tenha sido útil para você entender o que são as dataclasses no Python e como utilizá-las em suas aplicações.

Experimente utilizar dataclasses em seus projetos e desfrute de toda a praticidade e produtividade que elas oferecem.

Até a próxima! 

Nos vemos no próximo Artigo aqui do Blog!

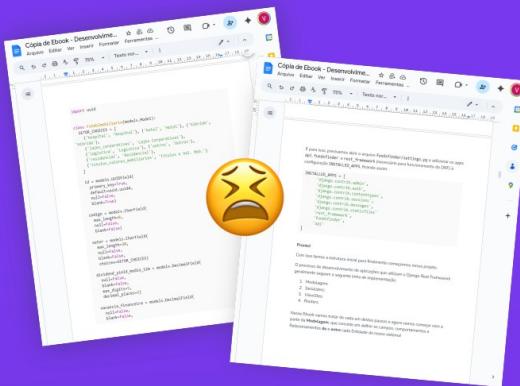
Não se esqueça de conferir!



DevBook

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



Adicione Banners Promocionais



• Infográficos feitos para...

Deixe que nossa IA faça o trabalho pesado



 Edite em Markdown em Tempo Real

TESTE AGORA



 PRIMEIRO CAPÍTULO 100% GRÁTIS