PYTHON ACADEMY

TRABALHANDO COM ESTRUTURAS COMPLEXAS NO PYDANTIC

Neste ebook, veremos como o Pydantic lida com modelos aninhados, permitindo representar relações entre dados em Python. Vamos explorar como o Pydantic lida com listas, dicionários e estruturas JSON complexas.

PYTHONACADEMY.COM.BR

> ✅ ***Atualizado para Pydantic V2*** *(Janeiro 2025)*
> *Modelos aninhados, listas, dicionários, JSON complexo, validação recursiva.*

Salve salve Pythonista 👋

No desenvolvimento de aplicações Python, a manipulação de estruturas de dados complexas pode ser desafiadora.

Com o **Pydantic v2**, a tarefa de validação e modelagem torna-se mais intuitiva.

Este artigo foca nos modelos aninhados, que permitem representar relações entre dados em Python.

Vamos explorar como o Pydantic lida com listas, dicionários e estruturas JSON complexas.

Além disso, veremos como converter modelos aninhados em dicionários e JSON, essencial para transferências de dados.

Representando relações entre dados

O **Pydantic** permite a construção de modelos aninhados, ou seja, modelos que contêm outros modelos como campos.

Esta função é útil para representar estruturas hierárquicas.

Veja como criar um modelo aninhado simples:

```python
from pydantic import BaseModel

class Endereco(BaseModel):
    rua: str
    cidade: str

class Usuario(BaseModel):
    nome: str
    email: str
    endereco: Endereco
```

Neste exemplo, `Usuario` possui um campo `endereco` do tipo `Endereco`.

Isso estabelece uma relação entre `Usuario` e seu endereço.

Validação de listas e dicionários

O **Pydantic** também lida bem com coleções de dados.

Podemos validar listas e dicionários de modelos aninhados facilmente.

Veja um exemplo com listas:

```python
class Pedido(BaseModel):
    id_pedido: int
    descricao: str

class Cliente(BaseModel):
    nome: str
    pedidos: list[Pedido]
```

Aqui, `Cliente` tem uma lista de `pedidos`, cada um representado pelo modelo `Pedido`.

Vamos ver como instanciar e validar:

```
pedido1 = Pedido(id_pedido=1, descricao="Pedido 1")
pedido2 = Pedido(id_pedido=2, descricao="Pedido 2")
cliente = Cliente(nome="Alice", pedidos=[pedido1, pedido2])
print(cliente)
```

E a saída será:

```
nome='Alice' pedidos=[Pedido(id_pedido=1, descricao='Pedido 1'), Pedi-
do(id_pedido=2, descricao='Pedido 2')]
```

Essa abordagem promove **organização e clareza** no código, especialmente com **estruturas JSON complexas** em APIs.

> ***Modelos complexos na prática:*** *Estou usando Pydantic para estruturar dados no* ***DevBook****, uma plataforma que gera ebooks técnicos com IA. Syntax highlighting perfeito, infográficos automáticos e exportação em PDF profissional. Vale conferir!*

Validando estruturas JSON complexas

Estruturas JSON complexas são comuns ao lidar com APIs.

O Pydantic facilita a **validação** e manipulação dessas estruturas.

Veja como usar modelos para validar um JSON complexo:

```python
import json

dados_json = '''
{
    "nome": "Carlos",
    "email": "carlos@example.com",
    "endereco": {
        "rua": "Rua das Flores",
        "cidade": "São Paulo"
    }
}
'''

dados = json.loads(dados_json)
usuario = Usuario(**dados)
print(usuario)
```

E a saída será:

```
nome='Carlos' email='carlos@example.com' endereco=Endereco(rua='Rua das Flores', cidade='São Paulo')
```

Neste exemplo, o JSON é convertido em um modelo `Usuario`, que é validado automaticamente.

Conversão de modelos em dicionários e JSON

O **Pydantic** permite converter modelos em dicionários e JSON facilmente.

Isso é útil para exportar dados de forma estruturada.

Conversão para dicionários

Usando o método `model_dump`, é possível converter para dicionário:

```
usuario_dict = usuario.model_dump()
print(usuario_dict)
```

A saída será:

```
{'nome': 'Carlos', 'email': 'carlos@example.com', 'endereco': {'rua':
'Rua das Flores', 'cidade': 'São Paulo'}}
```

Conversão para JSON

Com `model_dump_json`, transformamos para JSON:

```
usuario_json = usuario.model_dump_json()
print(usuario_json)
```

A saída será:

```
{"nome": "Carlos", "email": "carlos@example.com", "endereco": {"rua":
"Rua das Flores", "cidade": "São Paulo"}}
```

Esses métodos são fundamentais para transferir dados em aplicações web.

Casos Práticos Reais

1. Sistema de E-commerce Completo

```python
from pydantic import BaseModel, EmailStr, Field
from typing import List, Optional
from decimal import Decimal

class ItemCarrinho(BaseModel):
    produto_id: int
    nome: str
    quantidade: int = Field(gt=0)
    preco_unitario: Decimal

    @property
    def subtotal(self) -> Decimal:
        return self.preco_unitario * self.quantidade

class Endereco(BaseModel):
    rua: str
    numero: str
    complemento: Optional[str] = None
    bairro: str
    cidade: str
    estado: str
    cep: str

class Cliente(BaseModel):
    id: int
    nome: str
    email: EmailStr
    cpf: str
    enderecos: List[Endereco]

class Pedido(BaseModel):
    id: int
    cliente: Cliente
    itens: List[ItemCarrinho]
    endereco_entrega: Endereco

    @property
    def total(self) -> Decimal:
        return sum(item.subtotal for item in self.itens)

# Uso
```

```
pedido_data = {
    "id": 1001,
    "cliente": {
        "id": 5,
        "nome": "Maria Silva",
        "email": "maria@example.com",
        "cpf": "12345678900",
        "enderecos": [
            {"rua": "Av. Paulista", "numero": "1000", "bairro": "Bela
        Vista",
             "cidade": "São Paulo", "estado": "SP", "cep": "01310100"}
        ]
    },
    "itens": [
        {"produto_id": 10, "nome": "Notebook", "quantidade": 1, "pre-
        co_unitario": "3500.00"},
        {"produto_id": 20, "nome": "Mouse", "quantidade": 2, "pre-
        co_unitario": "50.00"}
    ],
    "endereco_entrega": {"rua": "Av. Paulista", "numero": "1000",
        "bairro": "Bela Vista",
                         "cidade": "São Paulo", "estado": "SP", "cep":
        "01310100"}
}

pedido = Pedido(**pedido_data)
print(f"Total do pedido: R$ {pedido.total}")  # R$ 3600.00
```

2. API de Rede Social

```python
from datetime import datetime
from typing import List, Optional

class Usuario(BaseModel):
    id: int
    username: str
    nome_completo: str
    bio: Optional[str] = None

class Comentario(BaseModel):
    id: int
    autor: Usuario
    texto: str
    created_at: datetime
    likes: int = 0

class Post(BaseModel):
    id: int
    autor: Usuario
    titulo: str
    conteudo: str
    tags: List[str]
    comentarios: List[Comentario]
    created_at: datetime

    @property
    def total_comentarios(self) -> int:
        return len(self.comentarios)

    @property
    def total_likes_comentarios(self) -> int:
        return sum(c.likes for c in self.comentarios)
```

3. Configuração de Aplicação Multi-Ambiente

```python
from pydantic import BaseModel, Field
from typing import Dict, Any

class DatabaseConfig(BaseModel):
    host: str
    port: int = 5432
    database: str
    username: str
    password: str
    pool_size: int = Field(default=10, ge=1, le=100)

class RedisConfig(BaseModel):
    host: str
    port: int = 6379
    db: int = 0

class AppConfig(BaseModel):
    app_name: str
    debug: bool = False
    database: DatabaseConfig
    redis: RedisConfig
    api_keys: Dict[str, str]
    features: Dict[str, bool]

config_data = {
    "app_name": "MyApp",
    "debug": True,
    "database": {
        "host": "localhost",
        "database": "myapp_db",
        "username": "user",
        "password": "secret"
    },
    "redis": {"host": "localhost"},
    "api_keys": {"openai": "sk-xxx", "stripe": "pk-xxx"},
    "features": {"new_ui": True, "beta_feature": False}
}

config = AppConfig(**config_data)
```

Quando Usar Modelos Complexos

✅ **APIs REST com payloads aninhados**

Perfeito para request/response bodies complexos

✅ **Configurações de aplicação**

Validar configs YAML/JSON com múltiplos níveis

✅ **Dados de domínio complexos**

E-commerce, redes sociais, sistemas financeiros

✅ **Integração com ORMs**

Mapear relacionamentos SQL (1:N, N:N)

✅ **Microserviços**

Contratos de dados entre serviços

Quando NÃO Usar

❌ **Dados muito profundamente aninhados (>5 níveis)**

Dificulta manutenção e debugging

❌ **Performance crítica com grandes volumes**

Validação tem custo - considere dataclasses

❌ **Estruturas completamente dinâmicas**

Se não há schema definido, Dict[str, Any] pode ser melhor

❌ **Dados muito simples**

Overhead desnecessário para estruturas planas

Pydantic vs Outras Abordagens

Critério	Pydantic	Dataclasses	Dict	NamedTuple
Validação	✅ Automática	❌ Não	❌ Não	❌ Não
Aninhamento	✅ Excelente	⚠️ Manual	✅ Sim	⚠️ Limitado
JSON	✅ Built-in	⚠️ Manual	✅ Nativo	❌ Não
Performance	⚠️ Média	✅ Rápido	✅ Rápido	✅ Rápido
Type hints	✅ Completo	✅ Completo	❌ Não	✅ Sim
Mutável	✅ Sim	✅ Sim	✅ Sim	❌ Não
Uso ideal	APIs/Validação	Data classes	Flexível	Imutáveis

Conclusão

Neste guia sobre **Modelos Complexos no Pydantic**, você aprendeu:

✅ **Modelos aninhados** - Models dentro de models
✅ **Listas tipadas** - List[Model] com validação
✅ **Dicionários** - Dict com tipos complexos
✅ **JSON complexo** - Estruturas profundamente aninhadas
✅ **Serialização** - model_dump() e model_dump_json()

Principais lições: - **Aninhamento** permite estruturas complexas e organizadas - **Validação recursiva** garante consistência em toda estrutura - **Listas e dicts** suportam tipos complexos - Perfeito para **APIs REST** com payloads complexos

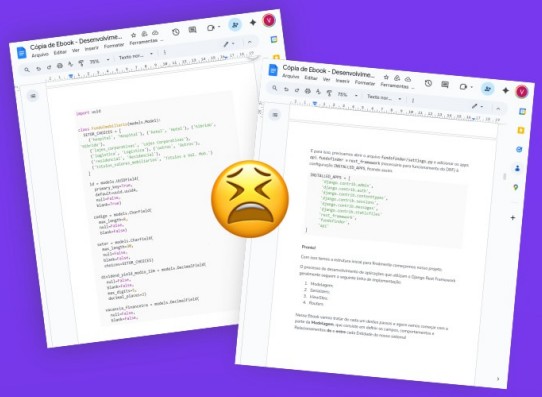
Próximos passos: - Aprenda Pydantic intro - Explore Tipos avançados

Não se esqueça de conferir!

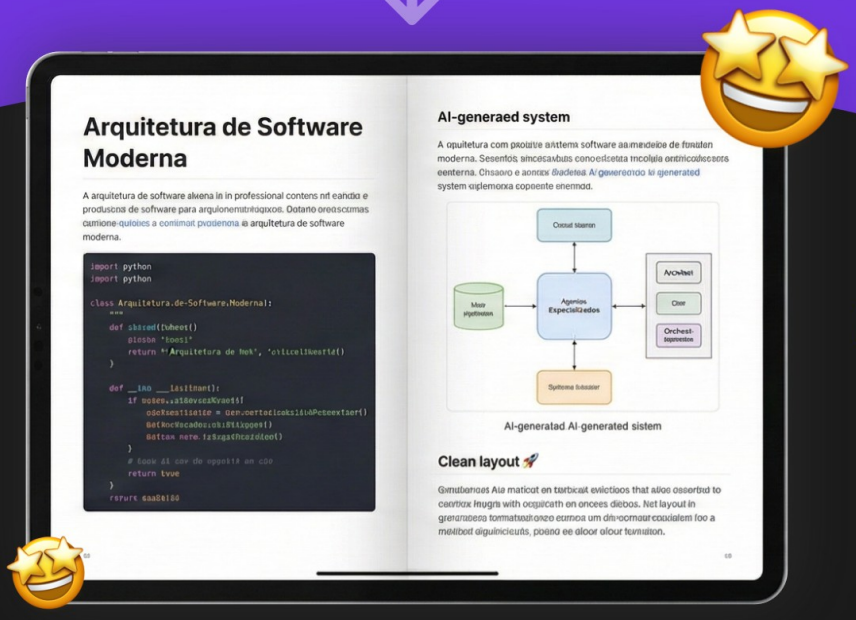
DevBook

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

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