PYTHON ACADEMY

PYDANTIC E TIPOS AVANÇADOS: PERSONALIZANDO A VALIDAÇÃO DE DADOS

Neste ebook, abordaremos como trabalhar com tipos complexos como Decimal, UUID e DateTime, utilizar campos opcionais e valores padrão, implementar validações personalizadas com validator, e introduzir o uso de Field para configurar validações e metadados.

PYTHONACADEMY.COM.BR

Salve salve Pythonista 👋

A validação de dados é essencial no desenvolvimento de aplicações robustas em Python.

Com o **Pydantic**, é possível personalizar a validação de dados utilizando tipos avançados e mecanismos poderosos.

Neste artigo, abordaremos como trabalhar com tipos complexos como **Decimal**, **UUID** e **DateTime**, utilizar campos opcionais e valores padrão, implementar validações personalizadas com **validator**, e introduzir o uso de **Field** para configurar validações e metadados.

Dominar esses tópicos é crucial para garantir a integridade e a consistência dos dados em suas aplicações Python.

Trabalhando com Tipos Complexos: Decimal, UUID e DateTime

O **Pydantic** oferece suporte a diversos tipos complexos que facilitam a manipulação de dados específicos.

Vamos explorar como utilizar **Decimal**, **UUID** e **DateTime** em seus modelos.

Utilizando Decimal

O tipo **Decimal** é útil para representar números com precisão decimal, evitando problemas de arredondamento comuns com floats.

```python
from decimal import Decimal
from pydantic import BaseModel

class Produto(BaseModel):
    nome: str
    preco: Decimal
```

Explicação do código:

1. **Importação**: Importamos `Decimal` do módulo `decimal` e `BaseModel` do Pydantic.
2. **Definição da Classe**: Criamos a classe `Produto` herdando de `BaseModel`.
3. **Atributos**: Definimos `nome` como string e `preco` como `Decimal`.

> ***Falando em Pydantic:*** *O* ***DevBook*** *usa Pydantic extensivamente para validar e estruturar conteúdo gerado por IA. É uma ferramenta que criei para gerar ebooks técnicos profissionais — com syntax highlighting, infográficos e exportação em PDF. Conhece lá!*

Utilizando UUID

Para identificar objetos de forma única, o **UUID** (Universally Unique Identifier) é uma excelente escolha.

```
import uuid
from pydantic import BaseModel

class Usuario(BaseModel):
    id: uuid.UUID
    nome: str
```

Explicação do código:

1. **Importação**: Importamos `uuid` e `BaseModel`.

2. **Definição da Classe**: Criamos a classe `Usuario` com `id` do tipo `UUID` e `nome` como string.

Utilizando DateTime

O tipo **DateTime** permite trabalhar com data e hora de forma eficiente.

```
from datetime import datetime
from pydantic import BaseModel

class Evento(BaseModel):
    nome: str
    data_hora: datetime
```

Explicação do código:

1. **Importação**: Importamos `datetime` do módulo `datetime` e `BaseModel`.
2. **Definição da Classe**: Criamos a classe `Evento` com `nome` e `data_hora` como `DateTime`.

Utilizando Campos Opcionais (Optional) e Valores Padrão

Nem todos os campos dos modelos precisam ser obrigatórios. O **Pydantic** facilita a definição de campos opcionais e a atribuição de valores padrão.

Campos Opcionais

Usamos `Optional` para indicar que um campo pode ser ignorado ou ter valor `None`.

```
from typing import Optional
from pydantic import BaseModel

class Cliente(BaseModel):
    nome: str
    email: Optional[str] = None
```

Explicação do código:

1. **Importação**: Importamos `Optional` de `typing` e `BaseModel`.
2. **Definição da Classe**: Criamos `Cliente` com `nome` obrigatório e `email` opcional, com valor padrão `None`.

Valores Padrão

Atribuímos valores padrão diretamente nos campos para garantir que tenham um valor caso não sejam fornecidos.

```
from pydantic import BaseModel

class Configuracao(BaseModel):
    ativo: bool = True
    nivel_acesso: int = 1
```

Explicação do código:

1. **Definição da Classe**: Criamos `Configuracao` com `ativo` como `True` e `nivel_acesso` como `1` por padrão.

Validação Personalizada com Validator

Embora o **Pydantic** valide automaticamente muitos tipos, às vezes é necessário implementar validações específicas. O decorador `@validator` facilita essa customização.

```
from pydantic import BaseModel, validator
from decimal import Decimal

class Pedido(BaseModel):
    quantidade: int
    preco_unitario: Decimal

    @validator('quantidade')
    def quantidade_positiva(cls, v):
        if v <= 0:
            raise ValueError('A quantidade deve ser positiva')
        return v

    @validator('preco_unitario')
    def preco_valido(cls, v):
        if v <= Decimal('0.00'):
            raise ValueError('O preço unitário deve ser maior que
        zero')
        return v
```

Explicação do código:

1. **Definição da Classe**: Criamos `Pedido` com `quantidade` e `preco_unitario`.
2. **Validador de Quantidade**:
    - O método `quantidade_positiva` verifica se `quantidade` é maior que zero.

- Se não, lança um `ValueError`.

3. **Validador de Preço Unitário**:
   - O método `preco_valido` assegura que `preco_unitario` seja maior que zero.
   - Se não, lança um `ValueError`.

Esses validadores garantem que os dados atendam às regras de negócio definidas.

Introdução ao Field para Configurar Validações e Metadados

O **Field** do Pydantic permite configurar validações adicionais e adicionar metadados aos campos dos modelos, como descrições, títulos e exemplos.

```
from pydantic import BaseModel, Field
from uuid import UUID

class Produto(BaseModel):
    id: UUID = Field(..., title="ID do Produto", description="Identifi-
        cador único do produto")
    nome: str = Field(..., min_length=3, max_length=50, example="Cane-
        ca")
    preco: Decimal = Field(..., gt=0, description="Preço do produto em
        reais")
```

Explicação do código:

1. **Importação**: Importamos `Field` de `pydantic`, além de outros tipos necessários.
2. **Definição da Classe**: Criamos `Produto` com `id`, `nome` e `preco`.

3. **Uso de Field**:
    - `id`: Define título e descrição para documentação.
    - `nome`: Define comprimento mínimo e máximo, além de um exemplo.
    - `preco`: Define que o valor deve ser maior que zero e adiciona uma descrição.

O **Field** é especialmente útil para gerar documentação automática e garantir que os campos atendam a critérios específicos além da validação básica de tipos.

Conclusão

Neste artigo, exploramos como o **Pydantic** permite personalizar a validação de dados utilizando tipos avançados como **Decimal**, **UUID** e **DateTime**.

Aprendemos a definir campos opcionais e atribuir valores padrão, implementar validações personalizadas com o decorador `@validator`, e utilizar o **Field** para configurar validações adicionais e metadados.

Essas ferramentas tornam o gerenciamento de dados em Python mais eficiente e seguro, fortalecendo a integridade das suas aplicações.

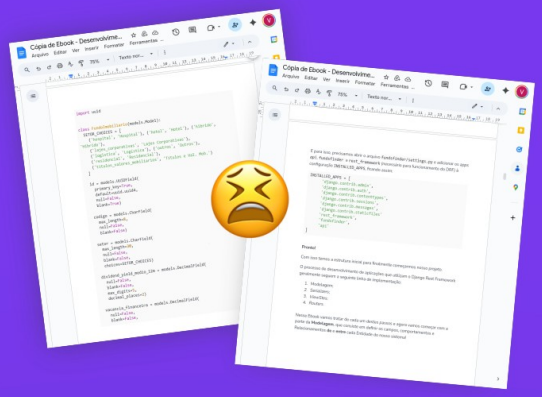
Com esses conhecimentos, você está pronto para criar modelos robustos e confiáveis em seus projetos Python.

Não se esqueça de conferir!

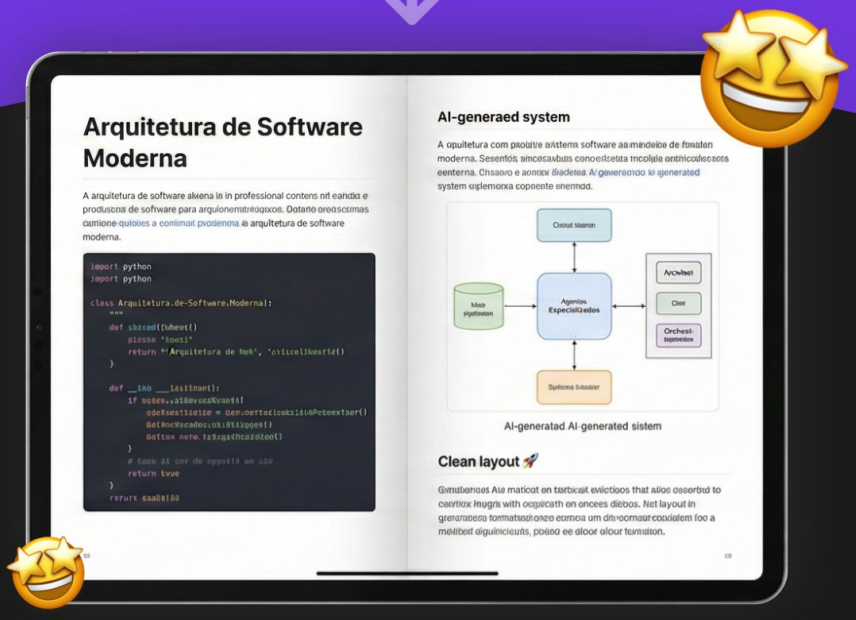
DevBook

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