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

Crie ebooks profissionais incríveis em minutos com IA

Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!

- Capas gerados por IA
- Infográficos feitos por IA
- Edite em Markdown em Tempo Real
- Adicione Banners Promocionais

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS

> ✅ ***Atualizado para Pydantic V2*** *(Fevereiro 2025) Tipos avançados: Decimal, UUID, DateTime, validators customizados, Field metadata.*

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.

```
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 **Ebookr.ai** usa Pydantic extensivamente para validar e estruturar conteúdo gerado por IA. É uma plataforma que criei para gerar ebooks profissionais sobre qualquer tema — com infográficos automáticos, capa gerada por IA e exportação em PDF. Conhece lá!*

Utilizando UUID

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

```python
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="Identificador único do produto")
    nome: str = Field(..., min_length=3, max_length=50,
        example="Caneca")
    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.

Casos Práticos com Tipos Avançados

1. Sistema de Pagamentos

```python
from pydantic import BaseModel, Field, validator
from decimal import Decimal
from datetime import datetime
from uuid import UUID, uuid4
from enum import Enum

class StatusPagamento(str, Enum):
    PENDENTE = "pendente"
    PROCESSANDO = "processando"
    APROVADO = "aprovado"
    REJEITADO = "rejeitado"

class Pagamento(BaseModel):
    id: UUID = Field(default_factory=uuid4)
    valor: Decimal = Field(..., gt=0, decimal_places=2)
    moeda: str = Field(default="BRL", regex="^[A-Z]{3}$")
    status: StatusPagamento = StatusPagamento.PENDENTE
    created_at: datetime = Field(default_factory=datetime.now)
    processado_em: Optional[datetime] = None

    @validator('valor')
    def validar_valor_maximo(cls, v):
        if v > Decimal('999999.99'):
            raise ValueError('Valor excede limite de R$ 999.999,99')
        return v

    class Config:
        use_enum_values = True

# Uso
pagamento = Pagamento(valor=Decimal('150.50'))
print(pagamento.model_dump_json(indent=2))
```

2. Agendamento de Eventos

```python
from pydantic import BaseModel, Field, validator, root_validator
from datetime import datetime, timedelta
from typing import Optional
from zoneinfo import ZoneInfo

class Evento(BaseModel):
    titulo: str = Field(..., min_length=5, max_length=100)
    inicio: datetime
    fim: datetime
    timezone: str = Field(default="America/Sao_Paulo")
    participantes_min: int = Field(default=1, ge=1)
    participantes_max: int = Field(..., ge=1, le=1000)

    @validator('fim')
    def fim_depois_inicio(cls, v, values):
        if 'inicio' in values and v <= values['inicio']:
            raise ValueError('Fim deve ser após início')
        return v

    @root_validator
    def validar_duracao(cls, values):
        inicio = values.get('inicio')
        fim = values.get('fim')

        if inicio and fim:
            duracao = fim - inicio
            if duracao > timedelta(hours=24):
                raise ValueError('Evento não pode durar mais de 24h')

        return values

    @validator('participantes_max')
    def max_maior_que_min(cls, v, values):
        if 'participantes_min' in values and v <
        values['participantes_min']:
            raise ValueError('Máximo deve ser >= mínimo')
        return v
```

3. Documentação de API com Field

```python
from pydantic import BaseModel, Field, EmailStr, HttpUrl
from typing import List, Optional
from datetime import date

class UsuarioAPI(BaseModel):
    """Schema para criação de usuário via API"""

    nome: str = Field(
        ...,
        min_length=2,
        max_length=100,
        title="Nome Completo",
        description="Nome completo do usuário",
        example="João Silva"
    )

    email: EmailStr = Field(
        ...,
        title="E-mail",
        description="E-mail válido do usuário",
        example="joao@example.com"
    )

    idade: int = Field(
        ...,
        ge=18,
        le=120,
        title="Idade",
        description="Idade do usuário (18-120)",
        example=25
    )

    site: Optional[HttpUrl] = Field(
        None,
        title="Website",
        description="URL do site pessoal",
        example="https://joao.com"
    )

    tags: List[str] = Field(
        default_factory=list,
```

```
        max_items=10,
        title="Tags",
        description="Lista de tags (máx 10)",
        example=["python", "dev"]
    )

    class Config:
        schema_extra = {
            "example": {
                "nome": "João Silva",
                "email": "joao@example.com",
                "idade": 25,
                "site": "https://joao.com",
                "tags": ["python", "dev"]
            }
        }

# Gerar schema JSON para documentação OpenAPI
print(UsuarioAPI.schema_json(indent=2))
```

Tipos Avançados: Comparação

Tipo	Quando Usar	Validação	Exemplo
Decimal	Valores monetários, precisão	Precisão decimal exata	Preços, salários
UUID	IDs únicos, chaves primárias	Formato UUID válido	IDs de banco
DateTime	Timestamps, agendamentos	Formato ISO 8601	created_at
EmailStr	E-mails	Formato de e-mail	Cadastros
HttpUrl	URLs	URL válida	Links externos
IPvAnyAddress	Endereços IP	IPv4/IPv6	Logs, segurança
Json	JSON em string	JSON válido	Configs dinâmicas
SecretStr	Senhas, tokens	-	Credenciais

Validators: Níveis de Validação

Field-level validator

```
@validator('campo')
def validar_campo(cls, v):
    # Valida um campo específico
    return v
```

Root validator (validar múltiplos campos)

```
@root_validator
def validar_modelo_completo(cls, values):
    # Valida relações entre campos
    return values
```

Validator com pre=True (antes da validação de tipo)

```
@validator('campo', pre=True)
def processar_antes(cls, v):
    # Processa antes da conversão de tipo
    return v
```

Quando Usar Validators Customizados

✅ **Regras de negócio complexas** Ex: CEP deve ser válido para o estado

✅ **Validações com dependências** Ex: data_fim > data_inicio

✅ **Normalização de dados** Ex: converter telefone para formato padrão

✅ **Integração com APIs externas** Ex: validar CPF consultando serviço

Quando NÃO Usar

❌ **Validações simples de tipo** Type hints já resolvem

❌ **Performance crítica** Validators adicionam overhead

❌ **Lógica de negócio pesada** Melhor em camada de serviço

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.

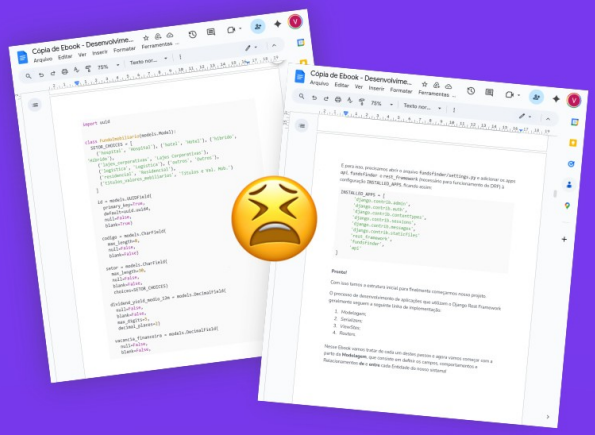
Essas ferramentas tornam o gerenciamento de dados em Python mais eficiente e seguro, fortalecendo a integridade das suas aplicações.

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

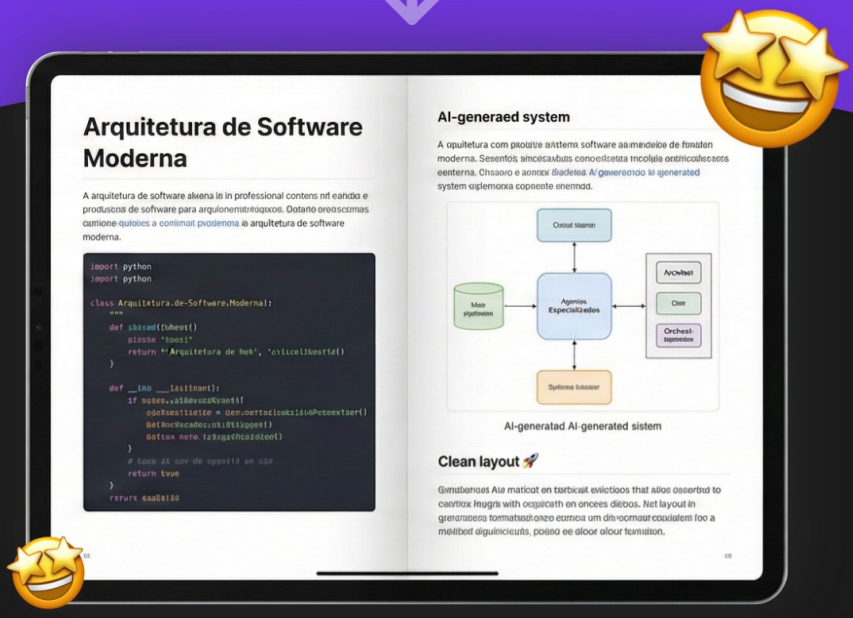
Não se esqueça de conferir!

Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA

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