Pydantic

python

1. Introducción a Pydantic

¿Qué es Pydantic?
Pydantic es una librería de Python que permite validar datos usando anotaciones de tipo. Su principal objetivo es garantizar que los datos que maneja tu aplicación cumplan con la estructura y tipos esperados, detectando errores antes de que causen problemas.

¿Para qué sirve?
Imagina que recibes datos de un formulario web, una API externa o un archivo JSON. Sin validación, podrías encontrarte con:

```
# Sin Pydantic – datos sin validar
usuario = {
    "nombre": "Juan",
    "edad": "veinticinco",  # debería ser int
    "email": "juan@"         # email inválido
}

# El código falla más adelante cuando intentas usar edad
edad_en_5_anos = usuario["edad"] + 5  # TypeError
```

Con Pydantic, la validación ocurre inmediatamente:

```
from pydantic import BaseModel, EmailStr

class Usuario(BaseModel):
    nombre: str
    edad: int
    email: EmailStr

# Esto lanza ValidationError de inmediato
usuario = Usuario(
    nombre="Juan",
    edad="veinticinco",  # Error detectado aquí
    email="juan@"
)
```

Si la importación de Pydantic te da fallo, tienes que instalar el paquete activando previamente el entorno virtual o creándolo si no lo has hecho:
python -m venv .venv
source .venv/bin/activate
pip install "pydantic[email]"

pydantic es la versión básica, si queremos soporte para mails y URL, usar pydantic[email].

Ejemplos comparativos
Sin Pydantic (código frágil)

```
def crear_producto(data):
    # Validación manual, propensa a errores
    if "nombre" not in data:
```

```
        raise ValueError("nombre es obligatorio")
    if not isinstance(data["precio"], (int, float)):
        raise ValueError("precio debe ser numérico")
    if data["precio"] < 0:
        raise ValueError("precio no puede ser negativo")

    return data

# Fácil olvidar validaciones o hacerlas inconsistentes
producto = crear_producto({"nombre": "Laptop", "precio": -100})
```

Con Pydantic (código robusto):

```
from pydantic import BaseModel, Field

class Producto(BaseModel):
    nombre: str
    precio: float = Field(gt=0)  # greater than 0
    stock: int = Field(default=0, ge=0)  # greater or equal 0

# Validación automática y consistente
producto = Producto(nombre="Laptop", precio=999.99)
# ValidationError si precio < 0 automáticamente
```

¿Qué es Field?

Field es una función de Pydantic que permite añadir validaciones y metadatos adicionales a los campos de un modelo. Mientras que las anotaciones de tipo (str, int, etc.) definen QUÉ tipo de dato es, Field define CÓMO debe comportarse ese campo.

Parámetros más comunes de Field

```
class Producto(BaseModel):
    # Valor por defecto
    nombre: str = Field(default="Sin nombre")

    # Valor por defecto usando factory (para listas, etc.)
    etiquetas: list = Field(default_factory=list)

    # Validaciones numéricas
    precio: float = Field(gt=0)              # greater than (mayor que)
    descuento: float = Field(ge=0, le=100)   # entre 0 y 100 (inclusive)
    stock: int = Field(ge=0)                 # greater or equal (mayor o
igual)

    # Validaciones de strings
    codigo: str = Field(min_length=3, max_length=10)
    descripcion: str = Field(max_length=500)

    # Documentación (útil para APIs)
    sku: str = Field(description="Código único del producto")

    # Alias (nombre diferente en JSON)
    precio_venta: float = Field(alias="precioVenta")
```

Ejemplo completo

```
from pydantic import BaseModel, Field
from typing import Optional

class Empleado(BaseModel):
    nombre: str = Field(min_length=2, max_length=100)
    edad: int = Field(ge=18, le=65, description="Edad del empleado")
    salario: float = Field(gt=0, description="Salario mensual en euros")
    departamento: str = Field(default="General")
    activo: bool = Field(default=True)
    email: Optional[str] = Field(default=None, pattern=r'^[\w\.-]+@[\w\.-
]+\.\w+$')

# Uso
empleado = Empleado(
    nombre="Ana García",
    edad=30,
    salario=2500.50,
    email="ana@empresa.com"
)

print(empleado.nombre)        # Ana García
print(empleado.departamento)  # General (valor por defecto)
```

2. Modelos básicos

Creación de tu primer modelo

Un modelo en Pydantic es una clase que hereda de BaseModel. Cada atributo de la clase representa un campo con su tipo de dato.

```
from pydantic import BaseModel

class Persona(BaseModel):
    nombre: str
    edad: int
    activo: bool

# Crear una instancia
persona = Persona(nombre="Carlos", edad=25, activo=True)

print(persona.nombre)  # Carlos
print(persona.edad)    # 25
print(persona.activo)  # True
```

Tipos de datos básicos

Pydantic soporta todos los tipos estándar de Python:

```
from pydantic import BaseModel
from typing import List, Dict, Optional
from datetime import datetime, date

class EjemploTipos(BaseModel):
```

```
    # Tipos simples
    texto: str
    numero_entero: int
    numero_decimal: float
    verdadero_falso: bool

    # Tipos de colecciones
    lista_numeros: List[int]
    diccionario: Dict[str, str]

    # Tipos de fecha y hora
    fecha_nacimiento: date
    fecha_registro: datetime

    # Tipos opcionales (pueden ser None)
    telefono: Optional[str] = None
    direccion: Optional[str] = None

# Ejemplo de uso
ejemplo = EjemploTipos(
    texto="Hola",
    numero_entero=42,
    numero_decimal=3.14,
    verdadero_falso=True,
    lista_numeros=[1, 2, 3, 4, 5],
    diccionario={"clave1": "valor1", "clave2": "valor2"},
    fecha_nacimiento="1990-05-15",      # Se convierte automáticamente
    fecha_registro="2025-01-17T10:30:00"  # Se convierte automáticamente
)

print(ejemplo.lista_numeros)      # [1, 2, 3, 4, 5]
print(ejemplo.fecha_nacimiento)   # 1990-05-15
print(ejemplo.telefono)           # None
```

3. Conversión de Datos (Serialización) en Pydantic

En programación, la serialización es el proceso de convertir un objeto complejo (como una instancia de una clase) en un formato que se pueda almacenar o transmitir fácilmente, como un diccionario o un archivo JSON.

Pydantic facilita esto enormemente mediante tres métodos principales:
- De Objeto a Diccionario: .model_dump()
Este es el método más utilizado. Transforma tu objeto de Pydantic en un diccionario estándar de Python (dict).

- **¿Para qué sirve?** Para manipular los datos como una colección de clave-valor antes de guardarlos en una base de datos o enviarlos a otra función.
- **Ejemplo:**

```
from pydantic import BaseModel

class Planta(BaseModel):
```

```
    nombre: str
    cantidad: int

# Creamos el objeto (instancia)
mi_planta = Planta(nombre="Monstera", cantidad=3)

# Convertimos a diccionario
datos_dict = mi_planta.model_dump()

print(datos_dict)
# Resultado: {'nombre': 'Monstera', 'cantidad': 3}
print(type(datos_dict))
# Resultado: <class 'dict'>
```

- De Objeto a JSON: .model_dump_json()

A diferencia del anterior, este método devuelve una **cadena de texto (string)** en formato JSON.

- **¿Para qué sirve?** Es el formato estándar para enviar datos a través de internet (APIs) o para escribir directamente en archivos .json.
- **Ejemplo:**

```
# Siguiendo el ejemplo anterior
datos_json = mi_planta.model_dump_json()

print(datos_json)
# Resultado: '{"nombre":"Monstera","cantidad":3}'
print(type(datos_json))
# Resultado: <class 'str'>
```

- De Diccionario a Objeto: .model_validate()

A veces el proceso es al revés: recibes datos de un archivo o de una web (un diccionario) y quieres convertirlos en un objeto de Pydantic para aprovechar su **validación automática**.

- **¿Para qué sirve?** Para asegurar que los datos que vienen de "fuera" cumplen con las reglas de tu modelo.
- **Ejemplo:**

```
# Imaginemos que leemos esto de un archivo JSON
datos_externos = {"nombre": "Cactus", "cantidad": 10}

# Lo convertimos en un objeto seguro de Pydantic
nueva_planta = Planta.model_validate(datos_externos)

print(nueva_planta.nombre) # Resultado: Cactus
print(type(nueva_planta))  # Resultado: <class '__main__.Planta'>
```