Pydantic aur FastAPI Notes (DACA Project)

Pydantic Kya Hai?

Pydantic ek Python library hai jo **data validation** aur **settings management** ke liye use hoti hai. Yeh ek security guard ki tarah kaam karta hai jo data ke type, format, aur rules ko check karta hai. FastAPI ke saath iska bohot zyaada istemaal hota hai kyunki APIs mein data validation bohot zaroori hai.

Main Points

- **Purpose**: Data ko validate aur serialize (JSON mein convert) karna.
- **Popular Use**: FastAPI ke saath APIs banane mein.
- **Ease of Use**: Python type hints ke saath kaam karta hai, jo code ko simple banata hai.

Highlights

- **Type Safety**: Galat data type se app ko bachata hai.
- **Error Handling**: Detailed errors debugging ko asaan banate hain.
- **Flexibility**: Custom rules aur complex data structures support karta hai.

Pydantic Ke Features

Pydantic ke yeh features isko powerful banate hain:

1. **Type-Safe Validation**
    - Python type hints (jaise `int`, `str`, `List[str]`) ke saath data check karta hai.
    - Example: Agar `id` integer hona chahiye aur string diya, toh error dega.
2. **Automatic Conversion**
    - Data ko sahi type mein convert karta hai.
    - Example: `"123"` string ko `123` integer mein badal deta hai.

3. **Error Handling**
    - Detailed error messages deta hai jo debugging ko asaan banate hain.
    - Example: "Field `id` integer hona chahiye, lekin string diya."
4. **Nested Models**
    - Complex data structures (jaise user ke multiple addresses) ko handle karta hai.
    - Example: Ek user ke saath uske addresses ka list store kar sakta hai.
5. **Serialization**
    - Data ko JSON ya dusre formats mein convert karta hai, jo APIs ke liye zaroori hai.
    - Example: User object ko JSON dictionary mein badalta hai.
6. **Custom Validators**
    - Apne khud ke validation rules bana sakte hain.
    - Example: "Naam kam se kam 2 characters ka hona chahiye."
7. **Default aur Optional Fields**
    - Fields ke liye default values ya optional setting kar sakte hain.
    - Example: `age: int | None = None` (optional field).

Highlight

- Pydantic ke features data integrity, automation, aur flexibility ko ensure karte hain, jo modern apps ke liye critical hain.

Pydantic Kyun Zaroori Hai (DACA Ke Liye)?

DACA shayad ek AI-driven ya chatbot-based project hai. Pydantic isme important hai kyunki:

1. **Data Integrity**
    - Galat data (jaise invalid email ya string jahan number chahiye) se app ko bachata hai.
2. **Complex Data Handling**
    - Nested models ke through user messages aur metadata (jaise timestamp, session ID) ko handle karta hai.
3. **API Integration**
    - FastAPI ke saath JSON conversion aur validation ko seamless banata hai.
4. **Debugging**
    - Clear error messages developers ko galti samajhne mein madad dete hain.

Highlight

- DACA jaise projects mein, jahan complex data aur API calls hote hain, Pydantic ka use app ko reliable aur maintainable banata hai.

Step 1: Getting Started with Pydantic

Project Setup

Pydantic aur FastAPI ke saath kaam shuru karne ke liye project setup:

Naya Project Banayein

```
 uv init fastdca_p1
cd fastdca_p1
```

1.

Virtual Environment Banayein

```
 uv venv
source .venv/bin/activate
```

2.

FastAPI aur Pydantic Install Karein

```
 uv add "fastapi[standard]"
```

3.

Main Points

- Virtual environment isolated environment banata hai taake dependencies conflict na karein.
- `fastapi[standard]` Pydantic aur FastAPI ke saath zaroori libraries install karta hai.

Example 1: Basic Pydantic Model

File: `pydantic_example_1.py`
Yeh ek simple `User` model banata hai jo id, name, email, aur age ko validate karta hai.

Code:

```
from pydantic import BaseModel, ValidationError
```

```
class User(BaseModel):
    id: int
    name: str
    email: str
    age: int | None = None

# Valid data
user_data = {"id": 1, "name": "Alice", "email": "alice@example.com", "age": 25}
user = User(**user_data)
print(user)
print(user.model_dump())

# Invalid data
try:
    invalid_user = User(id="not_an_int", name="Bob", email="bob@example.com")
except ValidationError as e:
    print(e)
```

Run:

```
uv run python pydantic_example_1.py
```

Output:

```
Valid data:
id=1 name='Alice' email='alice@example.com' age=25
{'id': 1, 'name': 'Alice', 'email': 'alice@example.com', 'age': 25}
```

-

```
Invalid data:
1 validation error for User
id
  value is not a valid integer (type=type_error.integer)
```

-

Main Points

- `BaseModel`: Pydantic ka base class hai jo validation ke liye use hota hai.
- `model_dump()`: Model ko JSON-compatible dictionary mein convert karta hai.
- **Use Case**: User data validation APIs ke liye.

Highlight

- Yeh example basic validation dikhata hai, jo galat data se app ko bachata hai.

Example 2: Nested Models

File: `pydantic_example_2.py`
Yeh complex data structures ke liye hai, jahan ek user ke multiple addresses store hote hain.

Code:

```
from pydantic import BaseModel, EmailStr

class Address(BaseModel):
    street: str
    city: str
    zip_code: str

class UserWithAddress(BaseModel):
    id: int
    name: str
    email: EmailStr
    addresses: list[Address]

user_data = {
    "id": 2,
    "name": "Bob",
    "email": "bob@example.com",
    "addresses": [
        {"street": "123 Main St", "city": "New York", "zip_code": "10001"},
        {"street": "456 Oak Ave", "city": "Los Angeles", "zip_code": "90001"},
    ],
}
user = UserWithAddress.model_validate(user_data)
print(user.model_dump())
```

Run:

```
uv run python pydantic_example_2.py
```

Output:

```
{
  "id": 2,
  "name": "Bob",
  "email": "bob@example.com",
  "addresses": [
    { "street": "123 Main St", "city": "New York", "zip_code": "10001" },
    { "street": "456 Oak Ave", "city": "Los Angeles", "zip_code": "90001" }
```

```
  ]
}
```

Main Points

- `EmailStr`: Valid email format check karta hai.
- `list[Address]`: Multiple addresses ko store karta hai.
- **Use Case**: Complex user profiles ke liye.

Highlight

- Nested models complex data ko organize karte hain, jo APIs mein common hai.

Example 3: Custom Validators

File: `pydantic_example_3.py`
Yeh custom validation rules ko dikhata hai, jaise naam 2 characters se chota nahi hona chahiye.

Code:

```
from pydantic import BaseModel, EmailStr, validator, ValidationError
from typing import List

class Address(BaseModel):
    street: str
    city: str
    zip_code: str

class UserWithAddress(BaseModel):
    id: int
    name: str
    email: EmailStr
    addresses: List[Address]

    @validator("name")
    def name_must_be_at_least_two_chars(cls, v):
        if len(v) < 2:
            raise ValueError("Name must be at least 2 characters long")
        return v

try:
    invalid_user = UserWithAddress(
        id=3,
        name="A",
```

```
        email="charlie@example.com",
        addresses=[{"street": "789 Pine Rd", "city": "Chicago", "zip_code": "60601"}],
    )
except ValidationError as e:
    print(e)
```

Run:

uv run python pydantic_example_3.py

Output:

```
1 validation error for UserWithAddress
name
  ValueError: Name must be at least 2 characters long
```

Main Points

- `@validator`: Custom rules define karta hai.
- **Use Case**: Project-specific validation ke liye.

Highlight

- Custom validators flexibility dete hain taake project ke unique needs ko handle kiya ja sake.

Step 2: FastAPI Application with Complex Pydantic Models

File: `main.py`

Ek FastAPI app jo chatbot ke liye kaam karta hai, complex Pydantic models ke saath.

Code:

```
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel, Field
from datetime import datetime, UTC
from uuid import uuid4

app = FastAPI(
    title="DACA Chatbot API",
```

```
    description="A FastAPI-based API for a chatbot in the DACA tutorial series",
    version="0.1.0",
)

class Metadata(BaseModel):
    timestamp: datetime = Field(default_factory=lambda: datetime.now(tz=UTC))
    session_id: str = Field(default_factory=lambda: str(uuid4()))

class Message(BaseModel):
    user_id: str
    text: str
    metadata: Metadata
    tags: list[str] | None = None

class Response(BaseModel):
    user_id: str
    reply: str
    metadata: Metadata

@app.get("/")
async def root():
    return {"message": "Welcome to the DACA Chatbot API! Access /docs for the API
documentation."}

@app.get("/users/{user_id}")
async def get_user(user_id: str, role: str | None = None):
    user_info = {"user_id": user_id, "role": role if role else "guest"}
    return user_info

@app.post("/chat/", response_model=Response)
async def chat(message: Message):
    if not message.text.strip():
        raise HTTPException(
            status_code=400, detail="Message text cannot be empty")
    reply_text = f"Hello, {message.user_id}! You said: '{message.text}'. How can I assist you
today?"
    return Response(
        user_id=message.user_id,
        reply=reply_text,
        metadata=Metadata()
    )
```

Run:

```
fastapi dev main.py
```

Test: Open `http://localhost:8000/docs` in browser.

Main Points

- **Models**:
  - `Metadata`: Timestamp aur session_id store karta hai.
  - `Message`: User ka message aur metadata.
  - `Response`: Server ka reply.
- **Endpoints**:
  - `/`: Welcome message.
  - `/users/{user_id}`: User info return karta hai.
  - `/chat/`: Message accept karta hai aur reply deta hai.
- **Use Case**: Chatbot APIs ke liye.

Highlight

- FastAPI aur Pydantic ka combination API development ko fast aur reliable banata hai.

Presenting by Humaiza naz