Pydantic and FastAPI Notes (DACA Project)

s. Each section includes headings, subheadings, highlights, and main points for clarity.

What is Pydantic?

Pydantic is a Python library used for **data validation** and **settings management**. Think of it as a gatekeeper that ensures your data is in the correct type, format, and adheres to defined rules. It's widely used with FastAPI because data validation is critical in APIs.

Main Points

- **Purpose**: Validates and serializes (converts to JSON) data.
- **Popular Use**: Building APIs with FastAPI.
- **Ease of Use**: Works with Python type hints, making code simple and readable.

Highlights

- **Type Safety**: Protects your app from incorrect data types.
- **Error Handling**: Provides detailed error messages for easier debugging.
- **Flexibility**: Supports custom rules and complex data structures.

Pydantic Features

Pydantic's features make it a powerful tool:

1. **Type-Safe Validation**
    - Uses Python type hints (e.g., `int`, `str`, `List[str]`) to validate data.
    - Example: If a field `id` requires an integer and a string is provided, Pydantic raises an error.
2. **Automatic Conversion**
    - Converts data to the correct type when possible.
    - Example: Converts the string `"123"` to the integer `123`.
3. **Error Handling**
    - Provides detailed error messages to simplify debugging.

    - Example: "Field `id` must be an integer, but a string was provided."
4. **Nested Models**
    - Handles complex data structures, like a user with multiple addresses.
    - Example: Stores a list of addresses for a user.
5. **Serialization**
    - Converts data to JSON or other formats, essential for APIs.
    - Example: Turns a user object into a JSON-compatible dictionary.
6. **Custom Validators**
    - Allows you to define custom validation rules.
    - Example: "Name must be at least 2 characters long."
7. **Default and Optional Fields**
    - Supports default values and optional fields.
    - Example: `age: int | None = None` (optional field with default `None`).

Highlight

- Pydantic's features ensure data integrity, automation, and flexibility, which are critical for modern applications.

Why is Pydantic Important for DACA?

DACA is likely an AI-driven or chatbot-based project. Pydantic is crucial because:

1. **Data Integrity**
    - Prevents incorrect data (e.g., invalid email or string where a number is expected) from breaking the app.
2. **Complex Data Handling**
    - Uses nested models to manage complex data like user messages and metadata (e.g., timestamp, session ID).
3. **API Integration**
    - Simplifies JSON conversion and validation with FastAPI.
4. **Debugging**
    - Clear error messages help developers identify issues quickly.

Highlight

- In projects like DACA, where complex data and API interactions are common, Pydantic makes the app reliable and maintainable.

Step 1: Getting Started with Pydantic

Project Setup

To start working with Pydantic and FastAPI, set up a project:

Create a New Project

```
 uv init fastdca_p1
cd fastdca_p1
```

1.

Create a Virtual Environment

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

2.

Install FastAPI and Pydantic

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

3.

Main Points

- A virtual environment creates an isolated space to avoid dependency conflicts.
- `fastapi[standard]` installs FastAPI, Pydantic, and necessary dependencies.

Example 1: Basic Pydantic Model

File: `pydantic_example_1.py`
This creates a simple `User` model to validate id, name, email, and age.

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 rech=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's base class for validation.
- `model_dump()`: Converts the model to a JSON-compatible dictionary.
- **Use Case**: Validates user data for APIs.

Highlight

- This example demonstrates basic validation, protecting the app from incorrect data.

Example 2: Nested Models

File: `pydantic_example_2.py`
This handles complex data structures, storing multiple addresses for a user.

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`: Ensures valid email format.
- `list[Address]`: Stores multiple addresses.
- **Use Case**: Manages complex user profiles.

Highlight

- Nested models organize complex data, common in API development.

“

Example 3: Custom Validators
File: `pydantic_example_3.py`
This demonstrates custom validation rules, ensuring a name is at least 2 characters long.

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`: Defines custom validation rules.
- **Use Case**: Project-specific validation requirements.

Highlight

- Custom validators provide flexibility to meet unique project needs.

Step 2: FastAPI Application with Complex Pydantic Models

File: `main.py`

A FastAPI app for a chatbot, using complex Pydantic models.

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 a browser.

Main Points

- **Models**:
    - `Metadata`: Stores timestamp and session ID.
    - `Message`: Captures user message and metadata.
    - `Response`: Server's reply.

- **Endpoints**:
  - `/`: Returns a welcome message.
  - `/users/{user_id}`: Returns user info.
  - `/chat/`: Accepts a message and responds.
- **Use Case**: Building chatbot APIs.

Highlight

- The combination of FastAPI and Pydantic makes API development fast, reliable, and scalable.

Presenting by Humaiza Naz