# Deep Dive into Pydantic Models

Say OL

say_ol@yahoo.com

September 28, 2024

# What is Pydantic?

- A data validation library using Python type annotations.
- Built for performance and usability.
- Automatically handles serialization and deserialization.

# Why Learn Pydantic?

- Simplifies data validation with clear and concise syntax.
- Encourages the use of type annotations, improving code clarity.
- Reduces boilerplate code for validation and serialization.

# Use Cases

- **Web APIs:** Validate incoming request data easily.
- **Configuration Management:** Manage application settings from environment variables.
- **Data Parsing:** Parse and validate complex data structures.

# Real Applications

- **FastAPI:** Utilizes Pydantic for request validation and response serialization.
- **Data Science Projects:** Ensures data integrity and type safety when processing datasets.
- **Microservices:** Validates data exchange between services, enhancing reliability.

# Basic Model Definition

## Example

```
from pydantic import BaseModel
class User(BaseModel):
    id: int
    name: str
    email: str
    age: int
if __name__ == "__main__":
    user = User(id=1, name="Michael",
                email="say_ol@yahoo.com", age=38)
    print(user)
    print(user.model_dump())
```

- Inherits from ‘BaseModel‘.
- Attributes use Python type hints.

# Creating Instances

## Example

```
user = User(id=1, name='Alice',
                email='alice@example.com', age=30)
print(user)
```

- Automatic validation occurs during instantiation.
- Outputs a User instance.

# Field Validation

## Example

```
from pydantic import EmailStr
    class User(BaseModel):
    email: EmailStr
if __name__=='__main__':
    user = User(email='alice@example.com') # Valid
    # Raises ValueError
    user_invalid = User(email='not-an-email')
```

- Use built-in validators for specific types (e.g., EmailStr).

# Default Values

## Example

```
class User(BaseModel):
    id: int
    name: str
    age: int = 30  # Default value
if __name__=='__main__':
    user = User(id=1, name='Alice')
    print(user.age)  # Outputs: 30
```

- Specify default values directly in the model.

# Custom Validation

## Example

```
from pydantic import field_validator, Field
class User(BaseModel):
    name: str
    age: int = Field(...,gt=0)
    @field_validator('age')
    def validate_age(cls, v):
        if v < 0:
            raise ValueError('Age must be positive')
        return v
if __name__=="__main__":
    user = User(name="Michael", age=38)
    print(user)
```

- Define custom validation logic with decorators.

# Nested Models

## Example

```
class Address(BaseModel):
    city: str
    state: str
class User(BaseModel):
    id: int
    name: str
    address: Address
if __name__ == "__main__":
    address = Address(city="Wonderland", state="Fantasy")
    user = User(id=1, name="Alice", address=address)
    print(user.address.city)  # Outputs: Wonderland
```

- Supports complex data structures.
- Validate nested attributes seamlessly.

# Model Serialization

## Example

```
user_dict = user.dict()
print(user_dict)  # Convert model to dictionary
```

- Easy conversion to dictionaries for JSON.
- Supports ‘exclude‘ and ‘include‘ options.

# Model Parsing

## Example

```
data = {'id': 1, 'name': 'Alice', 'age': 30}
user = User.parse_obj(data)  # Create model from dict
print(user)
```

- Use ‘parse_obj‘ to create instances from dictionaries.
- Automatically validates data.

# Settings Management

## Example

```
from pydantic_settings import BaseSettings
class Settings(BaseSettings):
    api_key: str
    db_url: str
if __name__=="__main__":
    settings = Settings(
        api_key="my_api_key",
        db_url="sqlite:///db.sqlite3",
    )
    print(settings)
```

- Extend ‘BaseSettings‘ for configuration management.
- Load values from environment variables.

# Settings Management

## Example

```
from pydantic_settings import BaseSettings
class Setting(BaseSettings):
    temp: str
    tmp: str
    path: str
if __name__=="__main__":
    setting = Setting()
    print(setting.temp)
    print(setting.tmp)
    print(setting.path)
```

# Conclusion

- Pydantic streamlines data validation and settings management.
- Encourages clean and maintainable code.
- Powerful features for modern Python applications.

# Questions?

**Thank you for your attention!**