Developing a RESTful API with FastAPI [Part 2]

> **Objective: In this lecture, we will explore Pydantic's core functionalities for input validation in FastAPI. While we won’t delve into advanced features, we will focus on the essential concepts needed to effectively validate inputs for machine learning applications.**

Understanding Pydantic Models and Their Field Types

What is Pydantic?

Imagine you're organizing a club and need everyone to submit their details: name, age, favorite hobbies, etc. You want to make sure the information they provide is correct—no missing names, no weird characters in age, and hobbies listed properly. This is where **Pydantic** comes in!

Pydantic is a Python library that helps us define and validate data using Python classes and **type hints**. It acts like a super-strict teacher checking that every form is filled out correctly before letting anyone in. Think of it like a spell-checker for data—it makes sure the information you’re working with is correct and matches the rules you set. In FastAPI, Pydantic plays a big role. Documentation

Why Use Pydantic?

- **Type Checking:** Pydantic ensures that the data fits the types you expect (e.g., names are strings, ages are integers).
- **Custom Logic:** You can add your own rules, like making sure ages are realistic or that someone can’t list "sleeping" as their only hobby (though, who doesn’t love naps?).
- **Inheritance:** You can create a "template" for models and reuse parts of it for similar forms.
- **Works Well with FastAPI**: FastAPI uses Pydantic to create rules for data, check incoming information, and even generate automatic documentation for your API.
- **Saves Time**: Instead of writing lots of extra code to handle data, you just create simple rules using Pydantic. FastAPI handles the rest.

- **Automatic Errors**: If the data is wrong, Pydantic creates clear error messages. These messages help users (and developers) understand what went wrong.

Why is Pydantic Important?

- **Keeps Data Organized**: You can use Pydantic to define what your data should look like. This ensures everything stays consistent and makes your code easier to read.
- **Handles Errors for You**: If someone sends bad data, Pydantic catches the mistake and explains what's wrong instead of letting your app crash.
- **Simple to Use**: Pydantic lets you create models using normal Python code. You don't need anything fancy to start using it.

The Basics of Defining a Model

Defining a model in Pydantic is straightforward:

1. **Inherit from `BaseModel`:** This is like saying, "I'm making a special kind of form."
2. **List Fields:** Specify each field in the form as a class property, with its type next to it.

Here's a simple example of a model to capture basic details about a person:

script_name: example1.py

```
from pydantic import BaseModel

class Person(BaseModel):
    first_name: str
    last_name: str
    age: int
```

When you define `first_name`, `last_name`, and `age`, Pydantic ensures:

- `first_name` and `last_name` must be strings.
- `age` must be an integer.

If someone tries to enter `"John"` for `age`, Pydantic will raise its hand and say, **"Nope, not allowed!"**

Going Beyond Simple Fields

Sometimes, you need to gather more complex information. What if you wanted to include someone's birthday, their gender, and a list of hobbies? Let's see how to handle that:

script_name: example2.py

```
from datetime import date
from enum import Enum
from typing import List
from pydantic import BaseModel

# Gender options using Enum
class Gender(str, Enum):
    MALE = "MALE"
    FEMALE = "FEMALE"
    NON_BINARY = "NON_BINARY"

# Updated model with more fields
class Person(BaseModel):
    first_name: str
    last_name: str
    gender: Gender
    birthdate: date
    interests: List[str]
```

Here’s what’s happening:

1. **Enums for Gender:** Instead of letting people enter anything, we define valid options (MALE, FEMALE, NON_BINARY). If someone enters something weird like `"Attack Helicopter"`, Pydantic will throw an error.
2. **Dates for Birthdate:** Using the `date` type ensures only valid dates like `2005-05-15` are accepted.
3. **Lists for Interests:** The `List[str]` type means users can give a list of their hobbies, like `["reading", "gaming", "sports"]`.

What Happens When Things Go Wrong?

If someone tries to enter invalid data, Pydantic doesn’t quietly let it slide—it tells you exactly what went wrong.

Example:

```
from pydantic import ValidationError

# Invalid gender value
try:
    person = Person(
        first_name="John",
        last_name="Doe",
        gender="INVALID",
        birthdate="1991-01-01",
```

```
        interests=["travel", "sports"]
    )
except ValidationError as e:
    print(e)
```

Output:

```
1 validation error for Person
gender
  value is not a valid enumeration member; permitted:
  'MALE', 'FEMALE', 'NON_BINARY' (type=type_error.enum;
enum_values=['MALE', 'FEMALE', 'NON_BINARY'])
```

Actually, this is exactly what we already did in Part 1, Developing a RESTful API with FastAPI, to limit the allowed values of the path parameter.

There are more examples in **example2.py**

Why This Matters

By defining models like this:

- **You reduce errors**: Your code ensures only valid data gets through.
- **It's easier to work with data**: You can trust the data structure without checking it every time.
- **Custom Rules = Cleaner Code**: Want to limit ages to between 0 and 120? Or check that interests aren't empty? Pydantic makes it easy!

More About Fields in Pydantic Models

Now that we know how to define simple fields like strings, numbers, and lists, let's dive into more advanced features of Pydantic fields.

Working with Dates

Dates can sometimes be tricky. For example, what happens if someone writes `"13/42/1991"` as their birthday? That's not a real date, but without checks, your program might accept it!

Pydantic uses Python's `date` class to handle this. It can:

1. Parse valid dates in ISO format (`YYYY-MM-DD`), such as `"1991-01-01"`.
2. Reject invalid dates like `"1991-13-42"` or `"Hello!"`.

script_name: example3.py

Example 3:

```
from datetime import date
from pydantic import BaseModel, ValidationError

class Person(BaseModel):
    first_name: str
    last_name: str
    birthdate: date

try:
    person = Person(
        first_name="John",
        last_name="Doe",
        birthdate="1991-13-42"  # Invalid date
    )
except ValidationError as e:
    print(e)
```

Output:

```
1 validation error for Person
birthdate
  invalid date format (type=value_error.date)
```

What happens if the date is valid? Pydantic converts it into a Python `date` object, which is easier to work with.

Example of Valid Date:

```
person = Person(
    first_name="John",
    last_name="Doe",
    birthdate="1991-01-01"
)
print(person)
```

Output:

```
first_name='John' last_name='Doe'
birthdate=datetime.date(1991, 1, 1)
```

Notice how the `birthdate` field is now a `datetime.date` object.

Nested Models (Sub-Objects)

What if someone's address has multiple parts, like street, city, and postal code? Instead of adding all these fields to the main `Person` model, we can use **nested models**.

script_name: nested_models.py

Step 1: Define the Address Model

```
from pydantic import BaseModel

class Address(BaseModel):
    street_address: str
    postal_code: str
    city: str
    country: str
```

Step 2: Add Address to the Person Model

```
class Person(BaseModel):
    first_name: str
    last_name: str
    birthdate: date
    address: Address
```

How Nested Models Work

Nested models let you create a `Person` with an `Address`. You can provide the address either as:

1. **An existing `Address` object**, or
2. **A dictionary** (Pydantic will convert it into an `Address` for you).

Example:

```
person = Person(
    first_name="John",
    last_name="Doe",
    birthdate="1991-01-01",
    address={
        "street_address": "123 Elm Street",
        "postal_code": "12345",
        "city": "Springfield",
        "country": "US"
    }
)
print(person)
```

Output:

```
first_name='John' last_name='Doe'
birthdate=datetime.date(1991, 1, 1)
address=Address(street_address='123 Elm Street',
postal_code='12345', city='Springfield', country='US')
```

Validation in Nested Models

If any field in the nested model is missing or invalid, Pydantic will raise an error. Let’s try an example where the `country` field is missing:

Example:

script_name: nested_models.py

```
try:
    person = Person(
        first_name="John",
        last_name="Doe",
        birthdate="1991-01-01",
        address={
            "street_address": "123 Elm Street",
            "postal_code": "12345",
            "city": "Springfield"
            # Missing "country"
        }
    )
except ValidationError as e:
    print(e)
```

Output:

```
1 validation error for Person
address -> country
  field required (type=value_error.missing)
```

Real-World Use Case

Think of this in terms of filling out a college application:

- **Student Info**: Name, age, grade level.
- **Parent Info**: Name, phone, email (nested model).
- **Address Info**: Street, city, postal code, and country (nested model).

Using nested models makes your code cleaner and easier to maintain.

Advanced `Field` in Pydantic

In programming, when you create a form or a model, you might want to set **rules** about what kind of data can go into each field. For example:

- **Age** must be between 1 and 100.
- **Name** cannot be empty.
- A **list of hobbies** must have at least one hobby.

In Pydantic, we use **`Field`** to set these rules (called **validation constraints**) and add descriptions to make things easier to understand.

1. What is `Field`?

`Field` is a tool in Pydantic that:

1. **Sets rules** for validating the data.
2. **Adds metadata** like descriptions or examples to fields (useful for documentation).

You use `Field` inside a **Pydantic model** to control how data is handled for each field. Documentation

2. Basic Usage of `Field`

Here's an example where we use `Field` to define rules for a `Person` model:

script_name: fields.py

```
from pydantic import BaseModel, Field

class Person(BaseModel):
    name: str = Field(..., description="The full name of the person.")
    age: int = Field(..., ge=1, le=120, description="Age must be between 1 and 120.")

# Example usage
person = Person(name="John Doe", age=25)
print(person)
```

Explanation:

1. `name: str = Field(...)`:
   - The `...` means this field is **required** (it cannot be empty).
   - The `description` adds a helpful note about what this field is.
2. `age: int = Field(..., ge=1, le=120)`:
   - `ge=1`: The age must be **greater than or equal to 1**.
   - `le=120`: The age must be **less than or equal to 120**.

3. Adding Metadata

You can use `Field` to add more details, like examples or default values, to make it easier to use the model:

```
class Person(BaseModel):
    name: str = Field(..., description="The full name of the person.", example="Jane Doe")
    age: int = Field(..., ge=1, le=120, description="Age must be between 1 and 120.", example=30)

# Example usage
person = Person(name="Alice", age=22)
print(person)
```

Features Used:

- `description`: Explains what the field represents.
- `example`: Provides a sample value (helpful for documentation).

4. Advanced Field Features

a. Default Values

If a field is optional or has a default value, you can use `Field` to set it:

```
class Person(BaseModel):
    name: str = Field(..., description="The full name of the person.")
    age: int = Field(18, ge=1, le=120, description="Age must be between 1 and 120. Defaults to 18.")

# Example usage
person = Person(name="Charlie")
print(person)  # age defaults to 18
```

b. Setting Constraints for Strings

You can use `Field` to validate string fields:

```
class User(BaseModel):
    username: str = Field(..., min_length=3, max_length=20, description="Username must be between 3 and 20 characters.")
    email: str = Field(..., regex=r"[^@]+@[^@]+\.[^@]+", description="Email must be in a valid format.")

# Valid example
user = User(username="student123", email="student@example.com")

# Invalid example (raises a ValidationError)
invalid_user = User(username="a", email="invalid-email")
```

Features Used:

- `min_length` and `max_length`: Set minimum and maximum length for strings.
- `regex`: Ensures the string matches a specific pattern (like a valid email).

c. Constraints for Lists

You can also validate lists, ensuring they’re not empty or have specific lengths:

script_name: fields3.py

```
from typing import List

class HobbyList(BaseModel):
```

```
    hobbies: List[str] = Field(..., min_items=1, description="You
must list at least one hobby.")

# Valid example
valid_hobbies = HobbyList(hobbies=["reading", "gaming"])

# Invalid example (raises a ValidationError)
invalid_hobbies = HobbyList(hobbies=[])
```

5. Example: Comprehensive Student Model

1. Create a `Student` model.
    - Fields: `name` (string), `age` (integer), `grade_level` (string), `enrolled_courses` (list of strings).
2. Add validation for:
    - `age`: Must be between 5 and 18.
    - `grade_level`: Must be one of `"Freshman"`, `"Sophomore"`, `"Junior"`, or `"Senior"`.

Here's how you could use `Field` to create a full student model with detailed constraints:

script_name: fields4.py

```
from pydantic import BaseModel, Field
from typing import List
from enum import Enum


# Grade levels using Enum
class GradeLevel(str, Enum):
    FRESHMAN = "Freshman"
    SOPHOMORE = "Sophomore"
    JUNIOR = "Junior"
    SENIOR = "Senior"


# Student model
class Student(BaseModel):
    name: str = Field(..., min_length=1, description="The name of
the student.")
    age: int = Field(..., ge=5, le=18, description="Age must be
between 5 and 18.")
    grade_level: GradeLevel = Field(..., description="The grade
level of the student.")
    enrolled_courses: List[str] = Field(..., min_items=1,
description="At least one course must be enrolled.")

# Valid input
student = Student(
    name="Alice",
    age=15,
    grade_level="Sophomore",
    enrolled_courses=["Math", "Science"],
```

```
)
print(student)

# Invalid input (raises a ValidationError)
try:
    invalid_student = Student(
        name="",
        age=20,
        grade_level="Graduate",
        enrolled_courses=[],
    )
except Exception as e:
    print(e)
```

6. Why Use `Field`?

- **Ensures Data Quality**: Automatically validates the data so you don't have to check it yourself.
- **Readable Code**: Makes your code more organized and descriptive.
- **Reduces Bugs**: Prevents invalid data from causing errors later in the program.

7. When to Use `Field`?

Use `Field` when:

1. You're building a model that needs validation rules.
2. You want to make your code self-explanatory with metadata like `description` and `example`.

Key Features of `Field`

Feature	Usage Example	Purpose
`default`	`Field(18)`	Sets a default value if the field is not provided.
`description`	`Field(..., description="...")`	Adds a helpful explanation about the field.
`example`	`Field(..., example="Alice")`	Shows an example value (useful for documentation).
`ge` and `le`	`Field(..., ge=5, le=18)`	Sets a range for numbers (e.g., age between 5 and 18).
`min_length`	`Field(..., min_length=3)`	Requires a minimum number of characters for a string.
`regex`	`Field(..., regex="pattern")`	Ensures the value matches a specific pattern (like a valid email).
`min_items`	`Field(...,`	Ensures a list has at least a certain number of items.

Feature	`min_items=1)` ample	Purpose

Coding Challenge 1: Adventure Game Character Creation

Imagine you're creating an adventure game where players can design their own characters. Your task is to build a **Character** model using Pydantic that ensures only valid characters can be created.

Requirements:

1. Define a `Character` model with the following fields:
    - `name` (string): The character's name.
    - `age` (integer): The character's age.
    - `class_type` (string): The character's role in the game (e.g., Wizard, Warrior, Archer, etc.).
    - `abilities` (list of strings): A list of special abilities the character has.
2. Add validation rules:
    - `age`: Must be between 10 and 100.
    - `class_type`: Must be one of `"Wizard"`, `"Warrior"`, `"Archer"`, or `"Healer"`.
    - `abilities`: Must have at least one ability in the list.
3. Test your model with:
    - Valid inputs (e.g., a 25-year-old Wizard with abilities like "Fireball" and "Teleport").
    - Invalid inputs (e.g., an 8-year-old Warrior with no abilities).
4. Create a script called challenge1.py containing your solutions and push to the "fastapi-project/part_2" folder in your git branch.

Let's see who can design the most robust and creative character model! □□

1. Default Values and Optional Fields

Default values and optional fields are critical for building flexible and user-friendly APIs. Let's break down the concepts with deeper explanations and examples:

Default Values

A **default value** is assigned to a field in your model when the user does not provide a value for it. This is useful for fields

that have a commonly assumed or fallback value.

Why Use Default Values?

- Reduce the burden on users by pre-filling common information (e.g., setting `country` to "USA").
- Ensure the application doesn't break when certain non-critical fields are missing.

How Default Values Work in Pydantic

When a field has a default value, the model will use this value if the user omits it while creating an instance of the model.

Example: Setting a Default Value for `country`

script_name: optional_fields.py

```python
from pydantic import BaseModel

class Person(BaseModel):
    first_name: str
    last_name: str
    country: str = "USA"  # Default value

# Creating a person without specifying the country
person = Person(first_name="Jane", last_name="Doe")
print(person)
```

Output:

```
first_name='Jane' last_name='Doe' country='USA'
```

Explanation:

- The user only provided `first_name` and `last_name`.
- Pydantic automatically filled in `country` with its default value (`"USA"`).

Optional Fields

An **optional field** is one that the user doesn't need to provide. If omitted, it defaults to `None`. This is especially useful for fields where the data might not always be available (e.g., `age` in a user profile).

Why Use Optional Fields?

- Allow flexibility when certain information is not always required.
- Provide a way to indicate missing data explicitly using `None`.

How to Declare Optional Fields

Use `Optional` from Python's `typing` module and set a default value of `None`.

Example: Optional `age` Field

```
from typing import Optional
from pydantic import BaseModel

class Person(BaseModel):
    first_name: str
    last_name: str
    age: Optional[int] = None  # Optional field with default None

# Creating a person without specifying the age
person = Person(first_name="John", last_name="Doe")
print(person)
```

Output:

```
first_name='John' last_name='Doe' age=None
```

Explanation:

- Since `age` is optional, it's okay to leave it out.
- Pydantic sets `age` to `None` by default.

Combining Default and Optional

You can combine default values with optional fields to provide flexibility and meaningful defaults.

script_name: optional_fields.py

Example: Optional Status with Default Value

```
class Task(BaseModel):
    title: str
    completed: Optional[bool] = False  # Optional, defaults to False

# Creating a task without specifying 'completed'
task = Task(title="Submit assignment")
print(task)
```

Output:

```
title='Submit assignment' completed=False
```

Explanation:

- The `completed` field is optional, so the user isn't required to provide it.
- If not provided, it defaults to `False`.

Coding Challenge 2: Build Your Own Nested Model

Create a script called challenge2.py containing your solutions and push to the "fastapi-project/part_2" folder in your git branch.

Design a `School` model that includes:

1. **School Name**: A string.
2. **Location**: A nested model with `city`, `state`, and `zip_code`.
3. **Students**: A list of `Student` models.

Define the `Student` model as:

- Name (string)
- Age (integer)
- Grade Level (string, valid values: "Freshman", "Sophomore", "Junior", "Senior").

Write test cases with:

1. A valid school instance.
2. A missing or invalid field (e.g., no `zip_code` in `Location`).

Let’s see if you can build a nested model with perfect validation!

2. Validation Using Validators

Validators allow you to enforce custom rules and transformations on the data. These rules ensure that the data is not just valid but also logical for your application.

What Are Validators?

A validator is a method that:

1. Runs after the field value is assigned.
2. Checks or transforms the field value.
3. Raises an error if the value doesn’t meet your custom conditions.

Why Use Validators?

- Enforce business logic (e.g., age must be positive, names must not be empty).
- Transform raw input data into a standardized format (e.g., capitalize names).

Defining Validators

Use the `@validator` decorator on a class method. The method:

- Must accept the value to be validated as an argument.
- Can raise a `ValueError` or `TypeError` for invalid values.

Examples of Validators

Example 1: Ensuring Positive Age

script_name: validators1.py

```
from pydantic import BaseModel, validator

class Person(BaseModel):
    first_name: str
    last_name: str
    age: int

    @validator("age")
    def validate_age(cls, value):
        if value <= 0:
            raise ValueError("Age must be a positive number")
        return value

# Valid age
person = Person(first_name="Alice", last_name="Smith", age=25)
print(person)

# Invalid age
try:
    invalid_person = Person(first_name="Alice",
last_name="Smith", age=-5)
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
first_name='Alice' last_name='Smith' age=25
Error: Age must be a positive number
```

Explanation:

- The validator checks that `age` is greater than 0.
- If `age` is invalid, it raises a `ValueError`.

Example 2: Capitalizing Names

script_name: validators2.py Validators can also transform data into a consistent format.

```
class Person(BaseModel):
    first_name: str
    last_name: str

    @validator("first_name", "last_name")
    def capitalize_name(cls, value):
        return value.title()  # Capitalize the first letter of
each word
```

```python
# Creating a person with lowercase names
person = Person(first_name="john", last_name="doe")
print(person)
```

Output:

```
first_name='John' last_name='Doe'
```

Explanation:

- The validator automatically capitalizes both `first_name` and `last_name`.
- This ensures consistent formatting for names.

Example 3: Validating List Length

script_name: validators3.py You can validate more complex data types, like lists.

```python
from typing import List

class Hobby(BaseModel):
    hobbies: List[str]

    @validator("hobbies")
    def check_hobby_count(cls, value):
        if len(value) < 1:
            raise ValueError("At least one hobby is required")
        return value

# Valid hobbies
hobby = Hobby(hobbies=["reading", "sports"])
print(hobby)

# Invalid hobbies
try:
    invalid_hobby = Hobby(hobbies=[])
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
hobbies=['reading', 'sports']
Error: At least one hobby is required
```

Explanation:

- The validator ensures that the list of hobbies is not empty.

Chaining Validators

Validators can be chained to apply multiple rules to a single field.

Example: Validating and Formatting Email Addresses

script_name: validators4.py

```python
from pydantic import BaseModel, EmailStr, validator

class User(BaseModel):
    email: EmailStr

    @validator("email")
    def check_email_domain(cls, value):
        if not value.endswith("@example.com"):
            raise ValueError("Email must end with @example.com")
        return value.lower()  # Transform email to lowercase

# Valid email
user = User(email="John.Doe@EXAMPLE.com")
print(user)

# Invalid email
try:
    invalid_user = User(email="john.doe@gmail.com")
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
email='john.doe@example.com'
Error: Email must end with @example.com
```

Coding Challenge 3: Validating List Length

Create a script called challenge3.py containing your solutions and push to the "fastapi-project/part_2" folder in your git branch.

Context:

In many machine learning workflows, models require feature inputs, such as a list of hobbies, skills, or categories. These inputs must have sufficient data to produce meaningful results. For example, if the hobbies list is empty, the model cannot make any meaningful predictions.

Task:

You are building a Pydantic model to validate a list of hobbies for a machine learning model that recommends activities based on interests. Write a `Hobby` model that ensures:

- The `hobbies` list must contain at least one hobby.
- If the list is empty, the model should raise a `ValueError`.

Code Template:

```python
from typing import List
from pydantic import BaseModel, validator

class Hobby(BaseModel):
    hobbies: List[str]

    @validator("hobbies")
    def validate_hobbies(cls, value):
        # Validation logic here: ensure the list is not empty
        pass

# Test with valid and invalid hobbies
valid_hobby = Hobby(hobbies=["reading", "sports"])
print(valid_hobby)

invalid_hobby = Hobby(hobbies=[])  # Should raise an error
```

Expected Outcome:

- If the list contains at least one hobby (e.g., `["reading", "sports"]`), the model should instantiate successfully.
- If the list is empty, a `ValueError` should be raised with the message: **"At least one hobby is required."**

Coding Challenge 4: Validating Nested Models

Create a script called challenge4.py containing your solutions and push to the "fastapi-project/part_2" folder in your git branch.

Context:

When working with real-world data, it's common to have complex inputs that involve nested structures. For example, an ML model predicting the best city to live in based on personal preferences might need a user profile containing an address. In this case, the `city` field should have meaningful input (at least 3 characters long).

Task:

Create a `UserProfile` model that contains:

1. An `Address` model as a nested field.
2. The `city` field in `Address` must have at least 3 characters. If the `city` name is too short, the model should raise a `ValueError`.

Code Template:

```python
from pydantic import BaseModel, validator
```

```python
class Address(BaseModel):
    street: str
    city: str

    @validator("city")
    def validate_city_length(cls, value):
        # Validation logic here: ensure city has at least 3
characters
        pass

class UserProfile(BaseModel):
    username: str
    address: Address

# Test with valid and invalid addresses
valid_profile = UserProfile(username="jane_doe", address=
{"street": "123 Elm Street", "city": "Springfield"})
print(valid_profile)

invalid_profile = UserProfile(username="jane_doe", address=
{"street": "123 Elm Street", "city": "NY"})  # Should raise an
error
```

Expected Outcome:

- If `city="Springfield"`, the model should instantiate successfully.
- If `city="NY"`, a `ValueError` should be raised with the message: **"City name must be at least 3 characters long."**

Coding Challenge 5: Combining Multiple Validators

Create a script called challenge5.py containing your solutions and push to the "fastapi-project/part_2" folder in your git branch.

Context:

You are building an API for an inventory management system that interacts with a machine learning model to predict future stock requirements. The API needs to validate product data before storing it in the system.

Task:

Create a `Product` model that:

1. Validates `name`: Ensure it’s not an empty string.
2. Validates `price`: Ensure it’s a positive float.
3. Validates `stock`: Ensure it’s an integer greater than or equal to 0.

4. Raise appropriate errors for invalid fields.

Code Template:

```
from pydantic import BaseModel, validator

class Product(BaseModel):
    name: str
    price: float
    stock: int

    @validator("name")
    def validate_name(cls, value):
        # Validation logic here: ensure name is not empty
        pass

    @validator("price")
    def validate_price(cls, value):
        # Validation logic here: ensure price is positive
        pass

    @validator("stock")
    def validate_stock(cls, value):
        # Validation logic here: ensure stock is non-negative
        pass

# Test with valid and invalid products
valid_product = Product(name="Laptop", price=999.99, stock=10)
print(valid_product)

invalid_product = Product(name="", price=-499.99, stock=-5)  # Should raise multiple errors
```

Expected Outcome:

- For `name="Laptop", price=999.99, stock=10`, the model should instantiate successfully.
- For invalid inputs (`name=""`, `price=-499.99`, `stock=-5`), multiple `ValueError`s should be raised with the messages:
  - **"Name cannot be empty."**
  - **"Price must be positive."**
  - **"Stock cannot be negative."**

Coding Challenge 6: Combining Everything

Create a script called challenge6.py containing your solutions and push to the "fastapi-project/part_2" folder in your git branch.

Context:

You’re designing an API for a loan approval system powered by an ML model. The system must validate loan applications to ensure

only valid data is passed to the model. Invalid inputs should result in appropriate error messages.

Task:

Create a `LoanApplication` model that validates:

1. **Name:** Must not be empty.
2. **Income:** Must be greater than $1,000.
3. **Loan Amount:** Must not exceed 5 times the income.
4. **Loan Purpose:** Must be one of `"home"`, `"car"`, or `"education"`.
5. Raise errors with clear and specific messages for invalid inputs.

Code Template:

```
from pydantic import BaseModel, validator

class LoanApplication(BaseModel):
    name: str
    income: float
    loan_amount: float
    purpose: str

    @validator("name")
    def validate_name(cls, value):
        # Validation logic here: ensure name is not empty
        pass

    @validator("income")
    def validate_income(cls, value):
        # Validation logic here: ensure income is > 1000
        pass

    @validator("loan_amount")
    def validate_loan_amount(cls, values):
        # Validation logic here: ensure loan_amount <= 5 * income
        pass

    @validator("purpose")
    def validate_purpose(cls, value):
        # Validation logic here: ensure purpose is one of the
allowed options
        pass

# Test with valid and invalid applications
valid_application = LoanApplication(name="John Doe", income=5000,
loan_amount=15000, purpose="home")
print(valid_application)

invalid_application = LoanApplication(name="", income=900,
loan_amount=10000, purpose="vacation")  # Should raise multiple
errors
```

Expected Outcome:

- For a valid application: `LoanApplication(name='John Doe', income=5000, loan_amount=15000, purpose='home')`
- For invalid inputs (`name=""`, `income=900`, `loan_amount=10000`, `purpose="vacation"`), multiple errors should be raised:
  - **"Name cannot be empty."**
  - **"Income must be greater than $1,000."**
  - **"Loan amount cannot exceed 5 times the income."**
  - **"Purpose must be one of: home, car, education."**

Additional Notes for Coding Challenges

- Testing with both valid and invalid inputs for each exercise.
- Remember that validators can also transform inputs (e.g., capitalizing names or formatting numbers).