Pydantic
Complete Guide

Data Validation and Type Safety in Python

A Comprehensive Guide to Pydantic,
Data Models, Validation, and Production-Grade Python Code

Learning Notes

Based on Comprehensive Tutorial

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to Pydantic

1.1 What is Pydantic?

Definition

Pydantic is a powerful Python library for data validation and settings management using Python type annotations.

Pydantic enables you to:

- Perform type validation automatically
- Validate complex data structures
- Build robust data models
- Structure complex data easily
- Write production-grade code with confidence

1.2 Why Pydantic Matters

Important Note

Python is a **dynamically typed language**, which means it does not have static typing like Java or C++. While this flexibility is great for beginners, it becomes a significant challenge when writing production-grade code.

The Challenge: Without proper validation, you cannot ensure:

- Variables contain the correct type
- Data meets business requirements
- Invalid data doesn't enter your system

1.3 Where Pydantic is Used

Pydantic is extensively used across various domains:

1. **FastAPI**: Building REST APIs (Pydantic is core to FastAPI)
2. **Configuration Management**: YAML/JSON config file validation
3. **Data Science**: ML pipeline data validation
4. **Data Engineering**: ETL pipeline data structures
5. **Any Production Code**: Where data integrity is critical

Industry Importance

If you're entering the data science or software engineering industry, having solid knowledge of Pydantic is **essential**.

2 Understanding the Problems Pydantic Solves

Before diving into solutions, let's understand exactly what problems Pydantic addresses.

2.1 Problem Setup: Patient Management System

Let's consider a real-world scenario where we're building a function to insert patient data into a database.

2.1.1 The Basic Function

```python
def insert_patient_data(patient_name, patient_age):
    """
    Insert patient data into database
    """
    print(patient_name)
    print(patient_age)

    # Imagine database insertion code here
    return "Patient data inserted successfully"
```

Listing 1: Initial Patient Insertion Function

The Scenario:

- A senior programmer writes this function
- A junior programmer will use it to insert data
- The junior programmer only sees the function signature
- They don't see the implementation details

2.2 Problem 1: No Type Validation

2.2.1 What Can Go Wrong

The junior programmer might do this:

```python
insert_patient_data("John Doe", "30")  # Age as string!
```

Listing 2: Problematic Usage

The Issue:

- We expected `patient_age` to be an integer
- But the junior programmer sent it as a string
- **The code will still work!**
- Wrong data type gets inserted into the database

Warning

This is a serious failure!

In production databases, you want to enforce a strict schema:

- All patient names should be strings

- All patient ages should be integers
- No exceptions!

But our current code doesn't enforce this at all.

2.2.2 Attempt 1: Type Hints

Let's try Python's type hinting:

```python
def insert_patient_data(patient_name: str, patient_age: int):
    print(patient_name)
    print(patient_age)
    return "Patient data inserted successfully"

# Usage
insert_patient_data("John Doe", 30)  # Correct
```

Listing 3: Adding Type Hints

Benefits:

- IDE shows suggested types
- Documentation is clearer
- Other developers see expected types

Problem:

- Type hints are just *suggestions*
- They don't enforce anything
- Wrong types still work!

```python
# This still works even though age is wrong type!
insert_patient_data("John Doe", "30")  # No error!
```

Listing 4: Type Hints Don't Stop This

Important Note

Key Understanding: Python's type hints provide information but do NOT enforce types. The code will execute even with wrong types.

2.2.3 Attempt 2: Manual Validation

Let's enforce types manually:

```python
def insert_patient_data(patient_name: str, patient_age: int):

    if type(patient_name) == str and type(patient_age) == int:
        print(patient_name)
        print(patient_age)
        return "Patient data inserted successfully"
    else:
        raise ValueError("Invalid input types")
```

```python

# Now this will raise an error
insert_patient_data("John Doe", "30")  # ValueError!
```

Listing 5: Manual Type Checking

This Works! But is it a good solution?

Note: In real Python code, `isinstance()` is preferred over `type() ==` because it correctly handles inheritance and subclasses.

2.2.4 The Scalability Problem

Consider if you need multiple functions:

```python
def insert_patient_data(patient_name: str, patient_age: int):
    if type(patient_name) == str and type(patient_age) == int:
        # Insert logic
        pass
    else:
        raise ValueError("Invalid input")

def update_patient_data(patient_name: str, patient_age: int):
    if type(patient_name) == str and type(patient_age) == int:
        # Update logic
        pass
    else:
        raise ValueError("Invalid input")
```

Listing 6: Multiple Functions Need Same Validation

Problems:

- Repeating the same validation code
- Just 2 fields and 2 functions - already messy
- What if you have 10 fields? 20 functions?
- What if you add a new field later?
- Need to update ALL functions!

Warning

This approach doesn't scale!

Writing manual validation code for every function is:

- Time-consuming
- Error-prone
- Difficult to maintain
- Not DRY (Don't Repeat Yourself)

2.3 Problem 2: Data Validation

Type validation is just the beginning. We also need to validate **data constraints**.

2.3.1 Business Rules

Consider these realistic requirements:

- **Age**: Cannot be negative
- **Email**: Must follow email format
- **Phone**: Must be valid phone format
- **Many more constraints...**

2.3.2 Manual Data Validation

```python
def insert_patient_data(patient_name: str, patient_age: int):

    # Type validation
    if type(patient_name) == str and type(patient_age) == int:

        # Data validation
        if patient_age < 0:
            raise ValueError("Age cannot be negative")
        else:
            print(patient_name)
            print(patient_age)
            return "Patient data inserted successfully"
    else:
        raise ValueError("Invalid input types")

# This will fail
insert_patient_data("John Doe", -1)  # ValueError: Age cannot be
    negative
```

Listing 7: Adding Data Validation

Now we have:

- Type validation (str, int)
- Data validation (age > 0)

2.3.3 The Complexity Explosion

Imagine adding more fields:

```python
def insert_patient_data(patient_name: str,
                        patient_age: int,
                        patient_email: str,
                        patient_phone: str):

    # Type validation for all fields
    if (type(patient_name) == str and
        type(patient_age) == int and
        type(patient_email) == str and
        type(patient_phone) == str):
```

```

        # Data validation for age
        if patient_age < 0:
            raise ValueError("Age cannot be negative")

        # Data validation for email
        # Need regex for email format!

        # Data validation for phone
        # Need validation for phone format!

        # ... more validations ...

        # Finally, insert data
        pass
```

Listing 8: Complexity Grows Quickly

The Nightmare:

- 10 fields = 10 type checks + 10 data validations
- Tons of boilerplate code
- Must repeat in EVERY function
- Adding a new field = updating ALL functions
- Email/phone validation requires regex patterns

Warning

This is exactly the problem Pydantic solves!

Pydantic ensures:

- Automatic type validation
- Automatic data validation
- No repetitive boilerplate code
- Centralized validation logic
- Easy to maintain and extend

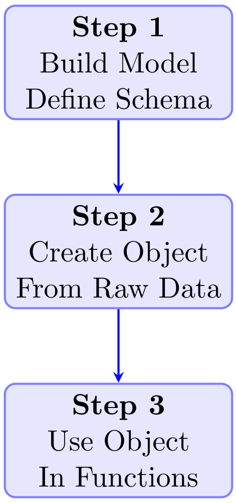
2.4 Summary: The Two Core Problems

Problem 1: Type Validation	**Problem 2: Data Validation**
Python is dynamically typed	Data must meet business rules
Type hints don't enforce types	Age cannot be negative
Manual type checking is tedious	Email must be valid format
Not scalable for production code	Phone must be valid format
Pydantic solves both problems elegantly	

3 The Pydantic Solution

3.1 How Pydantic Works: 3-Step Process

Pydantic follows a clean, three-step workflow:



3.1.1 Step 1: Build a Pydantic Model

A model is a class that defines your data schema:

- Define which fields you need
- Specify the data type for each field
- Add validation constraints

Example: For patient data, you define:

- Field: `name`, Type: `str`
- Field: `age`, Type: `int`
- Constraint: age should always be > 0

3.1.2 Step 2: Instantiate the Model

You create an object from the model using raw input data:

- Prepare data as a dictionary
- Pass it to the Pydantic model
- **Validation happens automatically here!**

During this step:

- Type validation is performed
- Data validation is performed
- If all checks pass $\rightarrow$ you get a validated object
- If any check fails $\rightarrow$ automatic error is raised

3.1.3 Step 3: Use the Validated Object

Pass the validated Pydantic object to your functions:

- Functions receive validated data
- No need for manual validation inside functions
- Clean, maintainable code

Important Note

Key Insight: Validation happens in Step 2, not in your business logic functions. This separation of concerns makes code much cleaner.

3.2 Practical Implementation

3.2.1 Installation

First, install Pydantic:

```
pip install pydantic
```

Pydantic Versions

Use Pydantic V2

Pydantic has two major versions:

- V1: Older version
- V2: Current version (recommended)

Why V2?

- Written in Rust (much faster)
- Better features and improvements
- Industry standard going forward

Always ensure you're using Pydantic V2 for new projects.

3.2.2 Step 1: Import and Create Model

```python
from pydantic import BaseModel

class Patient(BaseModel):
    patient_name: str
    patient_age: int
```

Listing 9: Creating Your First Pydantic Model

What's happening:

- Import `BaseModel` from pydantic
- Create a class that inherits from `BaseModel`
- Define fields with type annotations

Important Note

Important: Your model class MUST inherit from `BaseModel`. This is what makes it a Pydantic model.

3.2.3 Step 2: Create Validated Object

```python
# Raw data as dictionary
patient_info = {
    'patient_name': "Nitish",
    'patient_age': 30
}
```

```

# Create Pydantic object (validation happens here!)
patient_1 = Patient(**patient_info)
```

Listing 10: Instantiating the Model

Behind the scenes:

- Pydantic checks if `patient_name` is a string ✓
- Pydantic checks if `patient_age` is an integer ✓
- All validation passes → object created successfully

3.2.4 Step 3: Use in Functions

```
def insert_patient_data(patient: Patient):
    """
    Function now receives validated Patient object
    No manual validation needed!
    """
    print(patient.patient_name)
    print(patient.patient_age)
    return "Patient data inserted successfully"

# Call function with validated object
insert_patient_data(patient_1)
```

Listing 11: Using Validated Objects in Functions

Key observations:

- Function signature specifies `Patient` type
- No manual validation code inside function
- Access fields using dot notation: `patient.patient_name`
- Clean and readable code

3.3 Complete Working Example

```
from pydantic import BaseModel

# Step 1: Define Model
class Patient(BaseModel):
    patient_name: str
    patient_age: int

# Step 2: Create Validated Object
patient_info = {'patient_name': "Nitish", 'patient_age': 30}
patient_1 = Patient(**patient_info)

# Step 3: Use in Function
def insert_patient_data(patient: Patient):
    print(patient.patient_name)
    print(patient.patient_age)
    return "Patient data inserted successfully"

```

```
insert_patient_data(patient_1)
```

Listing 12: Full Pydantic Example

Output:

```
Nitish
30
```

3.4 Automatic Validation in Action

3.4.1 What Happens with Wrong Type

```python
# Wrong type for age (string instead of int)
patient_info = {'patient_name': "Nitish", 'patient_age': "thirty"}
patient_1 = Patient(**patient_info)  # This will fail!
```

Listing 13: Invalid Data Example

Result:

```
ValidationError:
  patient_age
    Input should be a valid integer
```

Automatic Error Handling

Pydantic automatically:

- Detects the type mismatch
- Raises a descriptive `ValidationError`
- Tells you exactly what's wrong
- Prevents invalid data from entering your system

All without you writing any validation code!

3.5 Benefits Achieved

Before Pydantic	**With Pydantic**
Manual type checking in every function	Define schema once
Repetitive validation code	No repetition
Easy to miss validations	Automatic validation
Hard to maintain	Easy to maintain
Difficult to extend	Easy to add new fields
Boilerplate everywhere	Clean, readable code

3.6 Type Coercion: Pydantic's Smart Feature

Pydantic doesn't just validate - it can also intelligently convert types.

3.6.1 Automatic Type Conversion

```python
# Age as string "30" instead of integer 30
patient_info = {'patient_name': "Nitish", 'patient_age': "30"}
patient_1 = Patient(**patient_info)

# Pydantic converts "30" to 30 automatically!
print(patient_1.patient_age)  # Output: 30 (integer)
```

Listing 14: Pydantic Type Coercion

What happened:

- You provided age as string `"30"`
- Pydantic detected it should be integer
- Pydantic successfully converted it to `30`
- No error raised - smart conversion!

3.6.2 When Coercion Works

Pydantic performs type coercion when conversion makes sense:

Input Type	Expected Type	Result
`"30"`	`int`	✓ Converted to 30
`"3.14"`	`float`	✓ Converted to 3.14
`"true"`	`bool`	✓ Converted to True
`"thirty"`	`int`	× Cannot convert

Important Note

Smart, Not Magic

Pydantic coercion is intelligent:

- `"30"` to `int`: Works ✓
- `"thirty"` to `int`: Fails ×
- Only valid conversions are performed

3.7 Multiple Functions: The Real Benefit

3.7.1 Before Pydantic

```python
def insert_patient_data(patient_name: str, patient_age: int):
    if type(patient_name) == str and type(patient_age) == int:
        # Logic here
        pass
    else:
        raise ValueError("Invalid input")

```

```python
def update_patient_data(patient_name: str, patient_age: int):
    if type(patient_name) == str and type(patient_age) == int:
        # Logic here
        pass
    else:
        raise ValueError("Invalid input")

# Repeated validation in every function!
```

Listing 15: Without Pydantic - Repeated Code

3.7.2 With Pydantic

```python
class Patient(BaseModel):
    patient_name: str
    patient_age: int

def insert_patient_data(patient: Patient):
    # No validation code needed
    pass

def update_patient_data(patient: Patient):
    # No validation code needed
    pass

# Validation defined once, used everywhere!
```

Listing 16: With Pydantic - Clean Code

Key Advantage

With Pydantic:

- Define validation once in the model
- All functions automatically benefit
- Adding a new field? Update only the model
- Changes propagate to all functions automatically

4 Building Complex Pydantic Models

4.1 Working with Multiple Data Types

Let's build a more realistic patient model with various data types.

4.1.1 Extended Patient Model

```python
from pydantic import BaseModel
from typing import List, Dict, Optional

class Patient(BaseModel):
    name: str
    age: int
    weight: float
    height: float
    bmi: float
    married: bool
    allergies: List[str]
    contact_details: Dict[str, str]
```

Listing 17: Complex Patient Model

Data types covered:

- `str`: Text fields (name)
- `int`: Whole numbers (age)
- `float`: Decimal numbers (weight, height, bmi)
- `bool`: True/False values (married)
- `List[str]`: List of strings (allergies)
- `Dict[str, str]`: Dictionary with string keys/values (contact details)

4.2 Why Import from typing Module

4.2.1 The Question

Why do we write `List[str]` instead of just `list`?

```python
# Wrong way
allergies: list  # Only validates it's a list

# Right way
allergies: List[str]  # Validates it's a list AND items are strings
```

Listing 18: Understanding the Difference

The difference:

- `list`: Only checks if variable is a list
- `List[str]`: Checks if it's a list AND every item is a string

4.2.2 Two-Level Validation

```
from typing import List

class Patient(BaseModel):
    allergies: List[str]

# Valid
patient_1 = Patient(allergies=["Pollen", "Dust"])

# Invalid - not a list
patient_2 = Patient(allergies="Pollen")

# Invalid - list but contains integer
patient_3 = Patient(allergies=["Pollen", 123])
```

Listing 19: List Validation Example

What Pydantic validates:

1. Is `allergies` a list? (First level)
2. Is every item in the list a string? (Second level)

Important Note

This two-level validation is why we use `List[str]` from the typing module instead of just `list`.

4.2.3 Dictionary Validation

Same concept applies to dictionaries:

```
from typing import Dict

class Patient(BaseModel):
    contact_details: Dict[str, str]

# Valid - keys and values are strings
patient = Patient(
    contact_details={'email': 'abc@gmail.com', 'phone': '9876543210'
   }
)

# Invalid - value is integer
patient = Patient(
    contact_details={'phone': 9876543210}
)
```

Listing 20: Dictionary Validation

Validation ensures:

1. `contact_details` is a dictionary
2. Every key is a string
3. Every value is a string

4.3 Complete Complex Model Example

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

class Patient(BaseModel):
    name: str
    age: int
    weight: float
    height: float
    bmi: float
    married: bool
    allergies: List[str]
    contact_details: Dict[str, str]

# Create patient with all fields
patient_info = {
    'name': "Sujil S",
    'age': 25,
    'weight': 75.2,
    'height': 175.5,
    'bmi': 24.4,
    'married': True,
    'allergies': ["Pollen", "Dust"],
    'contact_details': {
        'email': 'abc@gmail.com',
        'phone': '9876543210'
    }
}

patient_1 = Patient(**patient_info)

# Access any field
print(patient_1.name)            # Sujil S
print(patient_1.allergies)       # ['Pollen', 'Dust']
print(patient_1.contact_details) # {'email': '...', 'phone': '...'}
```

Listing 21: Full Complex Patient Model with Data

5 Required and Optional Fields

5.1 Default Behavior: All Fields Required

By default, every field in a Pydantic model is **required**.

```
class Patient(BaseModel):
    name: str
    age: int
    weight: float

# Missing 'weight' - will fail!
patient = Patient(name="John", age=30)  # ValidationError!
```

Listing 22: All Fields Required by Default

What happens:

- Pydantic expects ALL fields
- If any field is missing → `ValidationError`
- This ensures data completeness

5.2 Making Fields Optional

Sometimes you want fields that may or may not be provided.

5.2.1 Using Optional

```
from typing import Optional

class Patient(BaseModel):
    name: str                    # Required
    age: int                     # Required
    allergies: Optional[List[str]] = None  # Optional
```

Listing 23: Optional Fields

What this means:

- `name` and `age` are required
- `allergies` is optional
- If `allergies` not provided → defaults to `None`

Important Note

Important Rule: When you make a field optional, you MUST provide a default value (typically `None`).

5.2.2 Optional Field Behavior

```
class Patient(BaseModel):
    name: str
    age: int
    allergies: Optional[List[str]] = None

```

```python
# Without allergies - works!
patient_1 = Patient(name="John", age=30)
print(patient_1.allergies)  # Output: None

# With allergies - also works!
patient_2 = Patient(
    name="John",
    age=30,
    allergies=["Pollen"]
)
print(patient_2.allergies)  # Output: ['Pollen']
```

Listing 24: Optional Field Examples

5.3 Default Values

You can set default values for any field, not just optional ones.

5.3.1 Setting Defaults

```python
class Patient(BaseModel):
    name: str                    # Required
    age: int                     # Required
    married: bool = False        # Optional with default
    country: str = "India"       # Optional with default
```

Listing 25: Fields with Default Values

Behavior:

- If `married` not provided → defaults to `False`
- If `country` not provided → defaults to `"India"`
- User can still override defaults

5.3.2 Default Values in Action

```python
class Patient(BaseModel):
    name: str
    age: int
    married: bool = False

# Not providing 'married'
patient = Patient(name="John", age=30)
print(patient.married)  # Output: False (default value used)

# Explicitly providing 'married'
patient = Patient(name="Jane", age=25, married=True)
print(patient.married)  # Output: True (user value used)
```

Listing 26: Using Default Values

5.4 Required vs Optional vs Default: Summary

Case	**Example**	**Input Required?**	**Can be None?**
Required	`name: str`	✓ Yes	× No
Default	`married: bool = False`	× No	× No
Required but Nullable	`x: Optional[int]`	✓ Yes	✓ Yes
Optional	`x: Optional[int] = None`	× No	✓ Yes

Important Note

Key Differences (Pydantic v2):

- **Required**: Must provide a value, **cannot be None**
- **Required but Nullable**: Must provide a value, **can be None** (`Optional[T]` without default)
- **Default**: Value is optional; if not provided, default is used (cannot be None unless specified)
- **Optional + Default**: Value is optional and **can be None** (`Optional[T] = None`)

6 Data Validation Techniques

We've covered type validation. Now let's explore data validation - ensuring values meet business rules.

6.1 Method 1: Custom Data Types

Pydantic provides specialized types for common validation scenarios.

6.1.1 EmailStr - Email Validation

```python
from pydantic import BaseModel, EmailStr

class Patient(BaseModel):
    name: str
    email: EmailStr  # Special type for emails

# Valid email
patient = Patient(name="John", email="john@gmail.com")

# Invalid email (missing @)
patient = Patient(name="John", email="johngmail.com")
```

Listing 27: Email Validation

What `EmailStr` does:

- Validates email format automatically
- Checks for `@` symbol
- Ensures proper email structure
- No manual validation code needed!

6.1.2 AnyUrl - URL Validation

```python
from pydantic import BaseModel, AnyUrl

class Patient(BaseModel):
    name: str
    linkedin_url: AnyUrl  # Special type for URLs

# Valid URL
patient = Patient(
    name="John",
    linkedin_url="http://linkedin.com/in/john"
)

# Invalid URL (missing protocol)
patient = Patient(
    name="John",
    linkedin_url="linkedin.com/in/john"
)
```

Listing 28: URL Validation

What `AnyUrl` validates:

- Presence of protocol (http://, https://, etc.)
- Valid URL structure
- Domain format

Important Note

Custom Types Save Time

Manual email/URL validation requires:

- Complex regex patterns
- Multiple validation checks
- Error-prone implementation

Pydantic's custom types handle all of this automatically!

6.2 Method 2: Field Function

For custom business logic validation, use the `Field` function.

6.2.1 Importing Field

```python
from pydantic import BaseModel, Field
from typing import Annotated
```

Listing 29: Importing Field and Annotated

6.2.2 Numeric Constraints

```python
class Patient(BaseModel):
    name: str
    age: Annotated[int, Field(gt=0, lt=120)]
    weight: Annotated[float, Field(gt=0)]
```

Listing 30: Validating Numeric Ranges

Available constraints for numbers:

- `gt`: Greater than (exclusive)
- `ge`: Greater than or equal
- `lt`: Less than (exclusive)
- `le`: Less than or equal

```python
# Valid age
patient = Patient(name="John", age=25, weight=75.5)

# Invalid age (negative)
patient = Patient(name="John", age=-5, weight=75.5)

# Invalid age (too high)
patient = Patient(name="John", age=150, weight=75.5)
```

Listing 31: Testing Numeric Validation

6.2.3 String Constraints

```
class Patient(BaseModel):
    name: Annotated[str, Field(min_length=3, max_length=50)]
    description: Annotated[str, Field(max_length=200)]
```

Listing 32: Validating String Length

String constraints:

- `min_length`: Minimum number of characters
- `max_length`: Maximum number of characters

```
# Valid name
patient = Patient(name="John Doe")

# Invalid - too short
patient = Patient(name="Jo")

# Invalid - too long (>50 chars)
patient = Patient(
    name="A" * 60
)
```

Listing 33: Testing String Validation

6.2.4 List Constraints

```
class Patient(BaseModel):
    allergies: Annotated[List[str], Field(max_length=5)]
```

Listing 34: Validating List Length

This ensures:

- Maximum 5 items in the list
- Each item must be a string

6.3 Field Function: Multiple Purposes

The `Field` function serves three main purposes:

1. **Validation Constraints**: `gt`, `lt`, `min_length`, etc.
2. **Metadata**: Title, description, examples
3. **Default Values**: Set defaults directly
4. **Type Coercion Control**: Strict mode settings

6.3.1 Adding Metadata

```
from typing import Annotated

class Patient(BaseModel):
    name: Annotated[
        str,
        Field(
            min_length=3,
            max_length=50,
            title="Patient Name",
            description="Full name of the patient",
            examples=["John Doe", "Jane Smith"]
        )
    ]
```

Listing 35: Field with Metadata

Metadata benefits:

- Appears in API documentation (FastAPI)
- Helps other developers understand fields
- Provides examples for API consumers
- Auto-generated documentation uses this

6.3.2 Controlling Type Coercion with Strict Mode

By default, Pydantic performs type coercion. You can disable this with `strict=True`.

```
class Patient(BaseModel):
    age: Annotated[int, Field(strict=True)]  # No coercion
    weight: Annotated[float, Field(strict=True)]

# This will fail now
patient = Patient(age="30", weight="75.5")  # ValidationError!

# This works
patient = Patient(age=30, weight=75.5)  # Success
```

Listing 36: Strict Mode Example

When to use strict mode:

- When exact type matching is critical
- In financial applications (no implicit conversions)
- When debugging type-related issues
- For APIs with strict input requirements

Warning

Strict Mode Trade-offs

Pros:

- Explicit type requirements

- Prevents unexpected conversions
- Better error detection

Cons:

- Less forgiving for users
- Requires exact type matching
- May break existing code

Strict mode applies per field, not globally (unless using model_config).

```python
from pydantic import ConfigDict

class Patient(BaseModel):
    model_config = ConfigDict(strict=True)
```

7 Field Validators: Custom Validation Logic

7.1 When to Use Field Validators

Custom data types and Field constraints cover common cases. But what about business-specific validation?

Example Scenario: Hospital tied to specific banks (HDFC, ICICI). Need to validate employee emails from these domains only.

7.2 Creating Field Validators

```python
from pydantic import BaseModel, field_validator

class Patient(BaseModel):
    name: str
    email: str

    @field_validator('email', mode='before')
    @classmethod
    def validate_email_domain(cls, email):
        valid_domains = ['hdfc.com', 'icici.com']
        if '@' not in email:
            raise ValueError("Invalid email format")

        domain = email.split('@')[1]

        if domain not in valid_domains:
            raise ValueError('Invalid domain')

        return email
```

Listing 37: Email Domain Validation

7.3 Field Validator Components

- `@field_validator('email')`: Specify which field
- `mode='before'/'after'`: When to run
- `@classmethod`: Must be class method
- `cls`: Class reference
- `email`: Field value to validate

7.4 Mode: Before vs After

Mode	When Runs	Use Case
`before`	Before type coercion	Work with raw input
`after`	After type coercion	Work with converted types

7.5 Transformation with Validators

```python
class Patient(BaseModel):
    name: str

    @field_validator('name', mode='before')
    @classmethod
    def transform_name(cls, name):
        return name.title()  # Capitalize properly
```

Listing 38: Transform Name to Title Case

8 Model Validators: Cross-Field Validation

8.1 The Need for Model Validators

Field validators work on single fields. What if validation depends on multiple fields?

Example: If patient age > 60, emergency contact is required.

8.2 Creating Model Validators

```
from pydantic import model_validator

class Patient(BaseModel):
    age: int
    contact_details: Dict[str, str]

    @model_validator(mode='after')
    def validate_emergency_contact(self):
        if self.age > 60:
            if 'emergency' not in self.contact_details:
                raise ValueError(
                    'Patients over 60 need emergency contact'
                )
        return self
```

Listing 39: Multi-Field Validation

8.3 Key Differences

Aspect	Field Validator	Model Validator
Scope	Single field	Entire model
Input	Field value	Whole model (self)
Access	One field only	All fields
Return	Field value	Model instance

8.4 Model Validator Modes

Model validators support three modes: `before`, `after`, and `wrap`.

8.4.1 Before Mode

```
class Patient(BaseModel):
    name: str
    age: int

    @model_validator(mode='before')
    @classmethod
    def check_data_format(cls, data):
        # data is raw dictionary before any parsing
        if 'age' in data and data['age'] < 0:
            raise ValueError('Age cannot be negative')
        return data
```

Listing 40: Before Mode Validator

When it runs: Before Pydantic processes any fields

8.4.2 After Mode

```
class Patient(BaseModel):
    age: int
    weight: float

    @model_validator(mode='after')
    def validate_health_metrics(self):
        # self has all validated fields
        if self.age < 18 and self.weight > 100:
            raise ValueError('Unusual weight for age')
        return self
```

Listing 41: After Mode Validator

When it runs: After all fields are validated and parsed

8.4.3 Wrap Mode: Advanced Control

Wrap mode is the most powerful - it lets you control the entire validation process.

```
from typing import Any

class Patient(BaseModel):
    name: str
    email: str
    age: int

    @model_validator(mode='wrap')
    @classmethod
    def audit_validation(cls, data: Any, handler):
        try:
            print("Validation starting...")
            # Let Pydantic do its validation
            result = handler(data)
            print("Validation successful!")
            return result
        except ValidationError as e:
            # Log failed validation attempts
            print(f"Validation failed: {e.errors()}")
            print(f"Raw data: {data}")
            raise
```

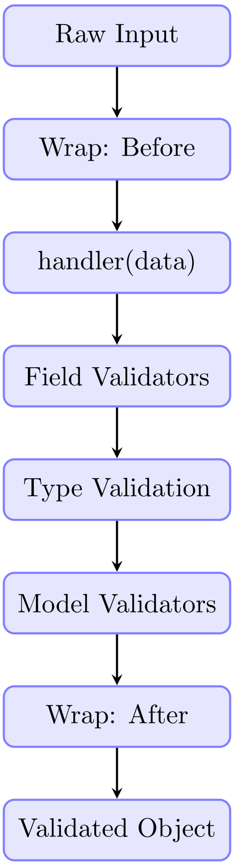
Listing 42: Wrap Mode for Logging

Wrap Mode Use Cases

When to use wrap validators:

- Logging all validation attempts (success/failure)
- Audit trails for compliance (HIPAA, GDPR)
- Custom retry logic on validation failure
- Performance monitoring
- Security logging (track invalid inputs)

8.5 Understanding Wrap Mode Execution Flow



8.6 Important Limitation: Core Type Parsing

Warning

Important Limitation of Wrap Validators

Wrap validators **do execute**, but Pydantic's core type validation (for types like `EmailStr` and `AnyUrl`) happens **inside** the `handler(data)` call.

This means:

- You **can catch and log** validation errors
- You **cannot intercept or bypass** core type validation
- Core type parsing always occurs when `handler(data)` is invoked

```python
class Patient(BaseModel):
    email: EmailStr

    @model_validator(mode='wrap')
    @classmethod
    def log_validation(cls, data, handler):
        try:
            return handler(data)  # Core validation happens here
        except Exception as e:
```

```
            print("Caught validation error:", e)
            raise

# Invalid email format (missing @)
patient = Patient(email="invalid")
# Error occurs inside handler(data), not before wrap runs
```

Listing 43: Wrap Validator Behavior

Important Note

What is `handler` in a Wrap Validator?

In a `@model_validator(mode="wrap")`, the `handler` is a **callable provided by Pydantic that executes the entire built-in validation pipeline**.

Calling `handler(data)` triggers:

- Core type parsing and coercion
- Validation of custom types (`EmailStr`, `AnyUrl`, etc.)
- Field validation and field validators
- Model validation (`before` and `after`)
- Construction of the final model instance

Important Rules:

- You **must call** `handler(data)` to continue validation
- If you do not call it, **no validation happens**
- Any exception raised inside `handler(data)` represents a validation failure

Key Insight: Wrap validators allow you to **observe, log, or wrap** Pydantic's validation process, but they **do not replace or bypass** core validation logic.

Final Conclusion: Why Use `wrap` Validators?

Using a `wrap` validator does **not change the validation result**. The same model is created or the same error is raised whether `wrap` is used or not.

The **only purpose** of `wrap` is to **guarantee observability** (logging, auditing, monitoring) of the validation process in a centralized way.

Without `wrap`:

- Validation still works correctly
- Errors can be printed using `try/except`
- Logging must be written manually at every call site
- Easy to forget or skip in large codebases

With `wrap`:

- Validation logic remains exactly the same
- Logging is executed automatically for every validation
- Observability is enforced at the model level
- Cannot be bypassed accidentally

Comparison: With vs Without `wrap`

Without `wrap`	**With `wrap`**
Validation works correctly	Validation works correctly
Errors can be printed using `try/except`	Errors can be logged inside the model
Logging must be written manually at every call site	Logging is executed automatically
Easy to forget or skip in large codebases	Enforced for every validation
Responsibility of individual developers	Centralized at model level
Same final result (model or error)	Same final result (model or error)

Correct Example Demonstrating This Behavior:

```python
from pydantic import BaseModel, ValidationError,
    model_validator

class Patient(BaseModel):
    age: int

    @model_validator(mode="wrap")
    @classmethod
    def audit_validation(cls, data, handler):
        print("AUDIT -> Incoming data:", data)
        try:
            return handler(data)  # Core validation (unchanged)
        except Exception as e:
            print("AUDIT -> Validation failed:", e)
            raise

# Result is the same with or without wrap
try:
    Patient(age="abc")
except ValidationError:
    pass
```

Key Insight: Printing errors outside the model and using `wrap` may produce the same output, but `wrap` guarantees that logging and auditing happen **everywhere and every time**, without relying on individual developers.

Final Takeaway: `wrap` validators exist for **enforced observability**, not for validation logic.

9 Computed Fields

9.1 What are Computed Fields?

Fields whose values are calculated from other fields, not provided by user.

Example: Calculate BMI from weight and height.

9.2 Implementation

```python
from pydantic import computed_field

class Patient(BaseModel):
    weight: float  # kg
    height: float  # cm

    @computed_field
    @property
    def bmi(self) -> float:
        return round(self.weight / (self.height/100)**2, 2)
```

Listing 44: Computed BMI Field

Key requirements:

- Use `@computed_field` decorator
- Use `@property` decorator
- Specify return type annotation
- Calculate from existing fields

9.3 Usage

```python
patient = Patient(weight=75, height=175)
print(patient.bmi)  # Output: 24.49 (automatically calculated)
```

Listing 45: Using Computed Field

Important Note

- Computed fields are read-only and cannot be set by user input.

10 Nested Models

10.1 Why Nested Models?

Complex data with hierarchical structure needs organization.

Problem: Address is complex (city, state, pincode) - storing as string is messy.

10.2 Creating Nested Models

```
class Address(BaseModel):
    city: str
    state: str
    pincode: int

class Patient(BaseModel):
    name: str
    age: int
    address: Address  # Nested model as field type
```

Listing 46: Address as Nested Model

10.3 Using Nested Models

```
# Create address object
address_info = {'city': 'Bangalore', 'state': 'Karnataka', 'pincode'
    : 560001}
address_1 = Address(**address_info)

# Create patient with nested address
patient_info = {
    'name': 'John',
    'age': 30,
    'address': address_1
}
patient_1 = Patient(**patient_info)

# Access nested fields
print(patient_1.address.city)     # Bangalore
print(patient_1.address.pincode)  # 560001
```

Listing 47: Creating Nested Objects

10.4 Benefits

1. **Organization**: Structured hierarchy
2. **Reusability**: Use Address in multiple models
3. **Validation**: Automatic validation of nested data
4. **Readability**: Clear data structure

11 Exporting Pydantic Models

11.1 Model to Dictionary

```
patient = Patient(name="John", age=30)

# Export to dict
data = patient.model_dump()
print(data)  # {'name': 'John', 'age': 30}
print(type(data))  # <class 'dict'>
```

Listing 48: Export as Dictionary

11.2 Model to JSON

```
json_data = patient.model_dump_json()
print(json_data)  # '{"name":"John","age":30}'
print(type(json_data))  # <class 'str'>
```

Listing 49: Export as JSON

11.3 Selective Export

```
# Include only specific fields
data = patient.model_dump(include={'name', 'age'})

# Exclude specific fields
data = patient.model_dump(exclude={'email'})

# Exclude nested fields
data = patient.model_dump(
    exclude={'address': ['state'], 'contact': ['phone']}
)

# Exclude fields with default values not explicitly set
data = patient.model_dump(exclude_unset=True)
```

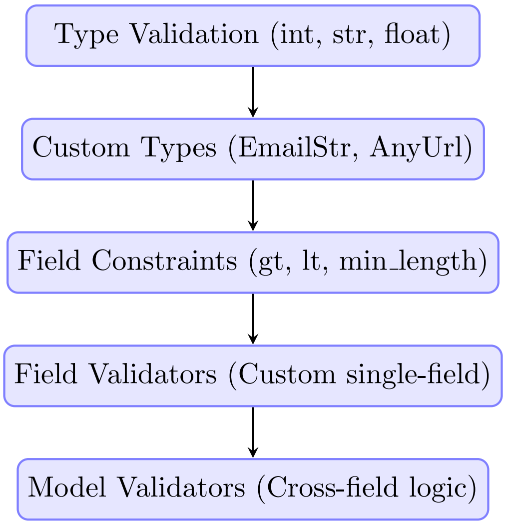
Listing 50: Include/Exclude Fields

12 Summary and Best Practices

12.1 Key Concepts Covered

1. **Problem Understanding**: Type and data validation in Python
2. **Basic Models**: Creating Pydantic models with BaseModel
3. **Data Types**: Working with complex types (List, Dict, Optional)
4. **Required/Optional**: Managing field requirements
5. **Custom Types**: EmailStr, AnyUrl for common validation
6. **Field Function**: Constraints and metadata
7. **Field Validators**: Custom single-field validation
8. **Model Validators**: Cross-field validation logic
9. **Computed Fields**: Calculated fields from other fields
10. **Nested Models**: Hierarchical data structures
11. **Export Options**: Dictionary and JSON conversion

12.2 Validation Hierarchy



12.3 Best Practices

- Use Pydantic V2 for new projects
- Define clear, descriptive field names
- Add metadata for API documentation
- Use computed fields for derived values
- Nest models for complex hierarchies
- Leverage custom types (EmailStr, AnyUrl)
- Write field validators for business logic
- Use model validators for cross-field rules

12.4 Common Use Cases

Use Case	Pydantic Feature
API request/response	BaseModel with Field metadata
Configuration files	Models with default values
Database models	Models with validation
Data pipelines	Computed fields, validators
Complex schemas	Nested models

12.5 Further Learning

This guide covers beginner to intermediate Pydantic usage. You now have enough knowledge to:

- Build FastAPI applications
- Validate configuration files
- Structure ML pipelines
- Write production-grade Python code

Next Steps

Practice by:

- Building a simple FastAPI application
- Creating models for your domain
- Experimenting with complex validators
- Exploring Pydantic's official documentation

End of Pydantic Complete Guide

"Data validation is not just about catching errors—
it's about building systems you can trust."

Happy coding with Pydantic!