Pydantic: Complete Guide to Data Validation in Python

Comprehensive Tutorial with Theory and Practice

September 27, 2025

Contents

1 Introduction to Pydantic

What is Pydantic?

Pydantic is a Python library that provides data validation and settings management using Python type hints. It's designed to be fast, extensible, and easy to use, making it perfect for API development, configuration management, and data parsing.

1.1 Core Concepts

- **BaseModel**: The foundation class for all Pydantic models
- **Type Hints**: Python's typing system for declaring expected data types
- **Validation**: Automatic data validation based on type hints
- **Serialization**: Converting Python objects to JSON/dict formats
- **Deserialization**: Creating Python objects from JSON/dict data

1.2 Why Use Pydantic?

Feature	Traditional Python	With Pydantic
Data Validation	Manual validation required	Automatic validation
Type Safety	Runtime errors possible	Compile-time type checking
Documentation	Manual documentation	Self-documenting models
JSON Handling	Manual serialization	Built-in JSON support
Error Messages	Generic error messages	Detailed validation errors

Table 1: Comparison: Traditional Python vs Pydantic

2 Basic Models and Data Types

2.1 Creating Your First Pydantic Model

Basic Model Example

```python
from pydantic import BaseModel

class Product(BaseModel):
    id: int
    name: str
    price: float
    in_stock: bool = True  # Default value

# Creating instances
product_one = Product(
    id=1,
    name="Laptop",
    price=999.99,
    in_stock=True
)

product_two = Product(
    id=2,
    name="Mouse",
    price=25.50
)  # in_stock will default to True
```

2.2 Supported Data Types

Type	Python Type	Description
Integer	`int`	Whole numbers
Float	`float`	Decimal numbers
String	`str`	Text data
Boolean	`bool`	True/False values
List	`List[T]`	Ordered collection of items
Dictionary	`Dict[K, V]`	Key-value pairs
Optional	`Optional[T]`	Value can be None
Union	`Union[T1, T2]`	Value can be one of multiple types
DateTime	`datetime`	Date and time objects

Table 2: Common Pydantic Data Types

3 Nested Models and Complex Structures

3.1 Simple Nested Models

Nested Models Example

```python
from pydantic import BaseModel
from typing import Optional

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

class Company(BaseModel):
    name: str
    address: Optional[Address] = None

class Employee(BaseModel):
    name: str
    company: Optional[Company] = None

# Usage
address = Address(
    street="123 Main St",
    city="New York",
    postal_code="10001"
)

company = Company(
    name="Tech Corp",
    address=address
)

employee = Employee(
    name="John Doe",
    company=company
)
```

3.2 Deeply Nested Structures

```python
class Country(BaseModel):
    name: str
    code: str

class State(BaseModel):
    name: str
    country: Country

class City(BaseModel):
    name: str
    state: State

class Address(BaseModel):
    street: str
    city: City
    postal_code: str

class Organization(BaseModel):
    name: str
```

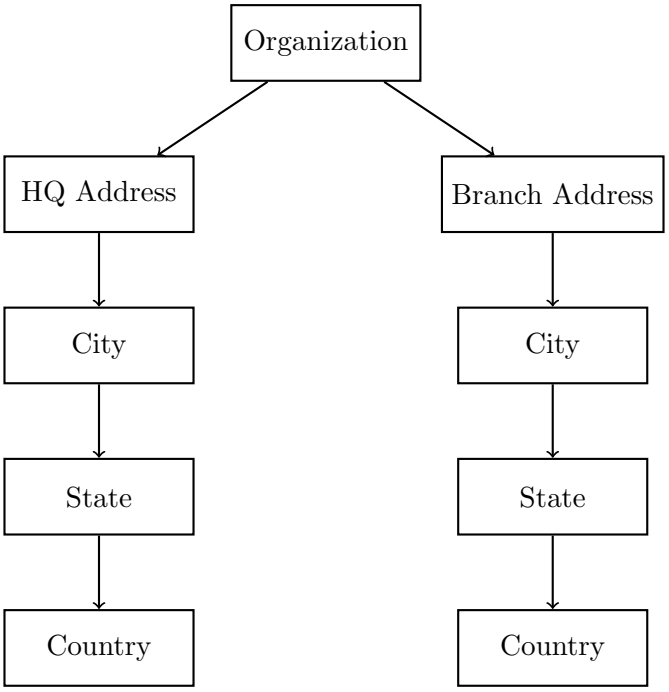

Figure 1: Deeply Nested Organization Structure

```
    head_quarter: Address
    branches: List[Address] = []
```

3.3 Mixed Data Types in Collections

Union Types for Mixed Content

```
from typing import Union, List

class TextContent(BaseModel):
    type: str = "text"
    content: str

class ImageContent(BaseModel):
    type: str = "image"
    url: str
    alt_text: str

class Article(BaseModel):
    title: str
    sections: List[Union[TextContent, ImageContent]]

# Usage
article = Article(
    title="My Blog Post",
    sections=[
        TextContent(content="This is a paragraph."),
        ImageContent(url="https://example.com/image.jpg", alt_text="Sample image"),
        TextContent(content="Another paragraph.")
    ]
)
```

4 Field Validation

4.1 Field Validators

Field validators allow you to add custom validation logic to individual fields.

Field Validator Decorator

The `@field_validator` decorator is used to define custom validation functions for specific fields. These functions are called automatically during model instantiation.

Field Validation Examples

```python
from pydantic import BaseModel, field_validator

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

    @field_validator('first_name', 'last_name')
    def names_must_be_capitalize(cls, v):
        if not v.istitle():
            raise ValueError("Names must be capitalized")
        return v

class User(BaseModel):
    email: str

    @field_validator('email')
    def normalize_email(cls, v):
        return v.lower().strip()

class Product(BaseModel):
    price: str  # Input as string like "$4.44"

    @field_validator('price', mode='before')
    def parse_price(cls, v):
        if isinstance(v, str):
            return float(v.replace('$', '').replace(',', ''))
        return v
```

4.2 Validation Modes

Mode	Description
`after`	Runs after type conversion (default)
`before`	Runs before type conversion
`wrap`	Wraps the entire validation process

Table 3: Field Validator Modes

5 Model Validation

5.1 Model-Level Validators

Model validators operate on the entire model instance, allowing validation that depends on multiple fields.

Model Validator Example

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

class DateRange(BaseModel):
    start_date: datetime
    end_date: datetime

    @model_validator(mode="after")
    def validate_date_range(self):
        if self.end_date <= self.start_date:
            raise ValueError('End date must be after start date')
        return self

class SignupData(BaseModel):
    password: str
    confirm_password: str

    @model_validator(mode='after')
    def passwords_match(self):
        if self.password != self.confirm_password:
            raise ValueError("Passwords do not match")
        return self
```

5.2 Validation Flow Diagram

6 Computed Fields

6.1 Understanding Computed Fields

Computed fields are properties that are calculated based on other fields in the model. They are automatically included in serialization but are not part of the initialization.

Computed Field Benefits

- Automatic calculation based on other fields
- Included in JSON serialization
- Read-only properties
- Type-safe computation

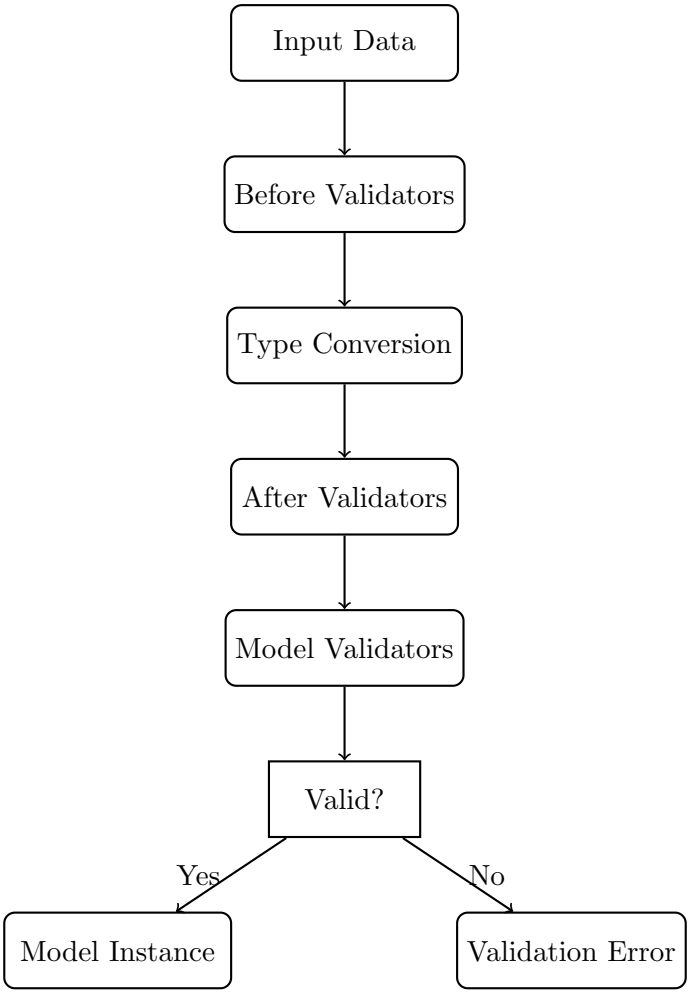

Figure 2: Pydantic Validation Flow

Computed Fields Examples

```python
from pydantic import BaseModel, computed_field, Field

class Product(BaseModel):
    price: float
    quantity: int

    @computed_field
    @property
    def total_price(self) -> float:
        return self.price * self.quantity

class Booking(BaseModel):
    user_id: int
    room_id: int
    nights: int = Field(..., ge=1)  # At least 1 night
    rate_per_night: float

    @computed_field
    @property
    def total_amount(self) -> float:
        return self.nights * self.rate_per_night

    @computed_field
    @property
    def booking_summary(self) -> str:
        return f"Room {self.room_id} for {self.nights} nights"

# Usage
booking = Booking(
    user_id=123,
```

7 Working with Collections

7.1 Lists and Dictionaries

Collection Types

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

class Cart(BaseModel):
    user_id: int
    items: List[str]
    quantities: Dict[str, int]
    metadata: Optional[Dict[str, str]] = None

class BlogPost(BaseModel):
    title: str
    content: str
    tags: List[str] = []
    comments: List['Comment'] = []
    image_urls: Optional[List[str]] = None

# Usage with dictionary unpacking
cart_data = {
    "user_id": 123,
    "items": ["Laptop", "Mouse", "Keyboard"],
    "quantities": {"Laptop": 1, "Mouse": 2, "Keyboard": 1}
}

cart = Cart(**cart_data)
```

7.2 Self-Referencing Models

For recursive structures like comments with replies:

Self-Referencing Models

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

class Comment(BaseModel):
    id: int
    content: str
    author: str
    replies: Optional[List['Comment']] = Field(default_factory=list)

# Required for forward references in Pydantic v1
Comment.update_forward_refs()

# Creating nested comments
comment = Comment(
    id=1,
    content="Great article!",
    author="Alice",
    replies=[
        Comment(
            id=2,
            content="I agree!",
            author="Bob",
            replies=[
                Comment(id=3, content="Thanks!", author="Alice")
            ]
        )
    ]
)
```

8 Serialization and Deserialization

8.1 Converting Models to Different Formats

Serialization Methods

Pydantic provides several methods to convert models to different formats:

- `.dict()`: Convert to Python dictionary
- `.json()`: Convert to JSON string
- `.model_dump()`: Modern method (Pydantic v2)
- `.model_dump_json()`: Modern JSON method (Pydantic v2)

Comprehensive Serialization Example

```python
from pydantic import BaseModel, Field
from typing import List
from datetime import datetime

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

class User(BaseModel):
    id: int
    name: str
    email: str
    is_active: bool = True
    created_at: datetime
    address: Address
    tags: List[str] = Field(default_factory=list)

    class Config:
        json_encoders = {
            datetime: lambda v: v.strftime('%d-%m-%Y %H:%M:%S')
        }

# Create a user instance
user = User(
    id=1,
    name="John Doe",
    email="john@example.com",
    created_at=datetime(2024, 3, 15, 13, 30),
    address=Address(
        street="123 Main St",
        city="New York",
        zip_code="10001"
    ),
    is_active=True,
    tags=["premium", "subscriber"]
)

# Serialization
python_dict = user.dict()
json_string = user.json()

print("Python Dictionary:")
print(python_dict)
print("\nJSON String:")
print(json_string)
```

8.2 Deserialization from Various Sources

Source	Method	Example
Dictionary	Constructor	`User(**data)`
JSON String	`parse_raw()`	`User.parse_raw(json_str)`
JSON File	`parse_file()`	`User.parse_file('user.json')`
Environment	Custom parser	Manual implementation

Table 4: Deserialization Methods

9 Advanced Field Configuration

9.1 Field Constraints and Validation

Field Constraints

```python
from pydantic import BaseModel, Field
from typing import List
from datetime import datetime

class User(BaseModel):
    username: str = Field(..., min_length=4, max_length=20)
    email: str = Field(..., regex=r'^[\w\.-]+@[\w\.-]+\.\w+$')
    age: int = Field(..., ge=18, le=120)  # Between 18 and 120
    score: float = Field(0.0, ge=0.0, le=100.0)
    tags: List[str] = Field(default_factory=list, max_items=10)
    created_at: datetime = Field(default_factory=datetime.now)

class Product(BaseModel):
    name: str = Field(..., description="Product name")
    price: float = Field(..., gt=0, description="Price must be positive")
    category: str = Field(..., regex=r'^[A-Za-z\s]+$')
    stock: int = Field(0, ge=0, description="Stock quantity")
```

9.2 Field Constraints Reference

Constraint	Type	Description
`gt`	Numeric	Greater than
`ge`	Numeric	Greater than or equal to
`lt`	Numeric	Less than
`le`	Numeric	Less than or equal to
`min_length`	String/List	Minimum length
`max_length`	String/List	Maximum length
`regex`	String	Regular expression pattern
`min_items`	List	Minimum number of items
`max_items`	List	Maximum number of items
`unique_items`	List	All items must be unique

Table 5: Common Field Constraints

10 Error Handling and Debugging

10.1 Understanding Validation Errors

Validation Error Structure

Pydantic validation errors contain:

- **loc**: Location of the error (field path)
- **msg**: Human-readable error message
- **type**: Error type code
- **ctx**: Additional context

Error Handling Example

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

class User(BaseModel):
    username: str = Field(..., min_length=4)
    age: int = Field(..., ge=18)
    email: str

try:
    user = User(
        username="ab",       # Too short
        age=16,              # Too young
        email="invalid"      # Invalid email format
    )
except ValidationError as e:
    print("Validation errors:")
    for error in e.errors():
        print(f"Field: {error['loc']}")
        print(f"Error: {error['msg']}")
        print(f"Type: {error['type']}")
        print("---")

    # Pretty print all errors
    print("\nFormatted errors:")
    print(e)
```

11 Performance Considerations

11.1 Best Practices for Performance

Practice	Description	Impact
Use built-in types	Prefer `int`, `str` over custom validators	High
Minimize validators	Only add custom validation when necessary	Medium
Cache model creation	Reuse model instances when possible	Medium
Avoid deep nesting	Limit nesting levels where possible	Medium
Use appropriate types	Use specific types like `EmailStr`	Low

Table 6: Performance Best Practices

11.2 Memory Usage Optimization

Memory Optimization

```python
from pydantic import BaseModel
from typing import Optional
import sys

# Memory-efficient model design
class OptimizedUser(BaseModel):
    id: int
    name: str
    email: Optional[str] = None  # Use Optional for fields that might be None

    class Config:
        # Use slots for memory efficiency
        allow_population_by_field_name = True
        use_enum_values = True

# Measure memory usage
user = OptimizedUser(id=1, name="John")
print(f"Memory usage: {sys.getsizeof(user)} bytes")
```

12 Integration Patterns

12.1 API Development with FastAPI

FastAPI Integration

```python
from pydantic import BaseModel, Field
from fastapi import FastAPI, HTTPException
from typing import List, Optional
from datetime import datetime

app = FastAPI()

class UserCreate(BaseModel):
    username: str = Field(..., min_length=3, max_length=20)
    email: str = Field(..., regex=r'^[\w\.-]+@[\w\.-]+\.\w+$')
    password: str = Field(..., min_length=8)

class UserResponse(BaseModel):
    id: int
    username: str
    email: str
    created_at: datetime
    is_active: bool = True

@app.post("/users/", response_model=UserResponse)
async def create_user(user: UserCreate):
    # Validation is automatic
    # Create user logic here
    return UserResponse(
        id=1,
        username=user.username,
        email=user.email,
        created_at=datetime.now()
    )

@app.get("/users/{user_id}", response_model=UserResponse)
async def get_user(user_id: int):
    # Get user logic here
    if user_id == 1:
        return UserResponse(
            id=user_id,
            username="john_doe",
            email="john@example.com",
            created_at=datetime.now()
        )
    raise HTTPException(status_code=404, detail="User not found")
```

12.2 Database Integration

SQLAlchemy Integration

```python
from pydantic import BaseModel
from sqlalchemy import Column, Integer, String, Boolean, DateTime
from sqlalchemy.ext.declarative import declarative_base
from datetime import datetime

Base = declarative_base()

# SQLAlchemy model
class UserDB(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    username = Column(String(50), unique=True)
    email = Column(String(100), unique=True)
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime, default=datetime.utcnow)

# Pydantic models
class UserCreate(BaseModel):
    username: str
    email: str

class User(BaseModel):
    id: int
    username: str
    email: str
    is_active: bool
    created_at: datetime

    class Config:
        from_attributes = True  # Enable ORM mode

# Usage
def create_user(db_user: UserDB) -> User:
    return User.from_orm(db_user)
```

13 Testing with Pydantic

13.1 Unit Testing Models

Testing Pydantic Models

```python
import pytest
from pydantic import ValidationError
from datetime import datetime

def test_user_creation():
    """Test successful user creation"""
    user_data = {
        "id": 1,
        "name": "John Doe",
        "email": "john@example.com",
        "created_at": datetime.now()
    }

    user = User(**user_data)
    assert user.id == 1
    assert user.name == "John Doe"
    assert user.email == "john@example.com"

def test_user_validation_error():
    """Test validation error handling"""
    with pytest.raises(ValidationError) as exc_info:
        User(
            id="invalid",  # Should be int
            name="",       # Should not be empty
            email="invalid-email"  # Invalid format
        )

    errors = exc_info.value.errors()
    assert len(errors) >= 2  # At least 2 validation errors

def test_user_serialization():
    """Test model serialization"""
    user = User(
        id=1,
        name="John Doe",
        email="john@example.com",
        created_at=datetime(2024, 1, 1)
    )

    user_dict = user.dict()
    assert "id" in user_dict
    assert user_dict["name"] == "John Doe"

    json_str = user.json()
    assert "John Doe" in json_str
```

14 Common Patterns and Recipes

14.1 Configuration Management

Settings Management

```python
from pydantic import BaseSettings, Field
from typing import List, Optional

class DatabaseSettings(BaseModel):
    host: str = "localhost"
    port: int = 5432
    username: str
    password: str
    database: str

class AppSettings(BaseSettings):
    # Application settings
    app_name: str = "My Application"
    debug: bool = False
    secret_key: str = Field(..., min_length=32)

    # Database settings
    database: DatabaseSettings

    # API settings
    api_key: Optional[str] = None
    allowed_hosts: List[str] = ["localhost", "127.0.0.1"]

    # Environment-based configuration
    environment: str = Field("development", regex="^(development|staging|
    production)$")

    class Config:
        env_file = ".env"
        env_nested_delimiter = "__"

# Usage
settings = AppSettings(
    secret_key="your-super-secret-key-here-32-chars",
    database=DatabaseSettings(
        username="myuser",
        password="mypass",
        database="mydb"
    )
)
```

14.2 Data Pipeline Patterns

ETL Pipeline with Pydantic

```python
from pydantic import BaseModel, validator, Field
from typing import List, Optional, Dict, Any
from datetime import datetime
import json

# Input data model
class RawData(BaseModel):
    timestamp: str
    user_id: int
    event_type: str
    payload: Dict[str, Any]

# Processed data model
class ProcessedEvent(BaseModel):
    event_id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    timestamp: datetime
    user_id: int
    event_type: str
    processed_payload: Dict[str, Any]
    processing_metadata: Optional[Dict[str, str]] = None

    @validator('timestamp', pre=True)
    def parse_timestamp(cls, v):
        if isinstance(v, str):
            return datetime.fromisoformat(v)
        return v

    @validator('event_type')
    def validate_event_type(cls, v):
        allowed_types = ['click', 'view', 'purchase', 'signup']
        if v not in allowed_types:
            raise ValueError(f'Event type must be one of {allowed_types}')
        return v

# Pipeline processor
class DataPipeline:
    def __init__(self):
        self.processed_count = 0
        self.error_count = 0

    def process_batch(self, raw_events: List[Dict[str, Any]]) -> List[
    ProcessedEvent]:
        processed = []

        for raw_event in raw_events:
            try:
                # Parse raw data
                raw_data = RawData(**raw_event)

                # Transform to processed event
                processed_event = ProcessedEvent(
                    timestamp=raw_data.timestamp,
                    user_id=raw_data.user_id,
                    event_type=raw_data.event_type,
                    processed_payload=self._process_payload(raw_data.
    payload),
                    processing_metadata={
                        "processed_at": datetime.now().isoformat(),
                        "processor_version": "1.0"
                    }
                )

                processed.append(processed_event)
                self.processed_count += 1

```

15 Migration and Version Management

15.1 Model Versioning Strategies

Versioning Approaches

- **Field Addition**: Add new optional fields
- **Field Deprecation**: Mark fields as deprecated
- **Model Inheritance**: Create versioned model classes
- **Alias Support**: Support multiple field names

Model Versioning Example

```python
from pydantic import BaseModel, Field
from typing import Optional, Union
from datetime import datetime

# Version 1 - Original model
class UserV1(BaseModel):
    id: int
    name: str
    email: str
    created_at: datetime

# Version 2 - Added optional fields
class UserV2(BaseModel):
    id: int
    name: str
    email: str
    created_at: datetime

    # New fields (optional for backward compatibility)
    phone: Optional[str] = None
    is_verified: bool = False
    last_login: Optional[datetime] = None

# Version 3 - Field name changes with aliases
class UserV3(BaseModel):
    id: int
    full_name: str = Field(alias="name")  # Changed field name
    email: str
    created_at: datetime = Field(alias="createdAt")  # Support camelCase

    phone: Optional[str] = None
    is_verified: bool = False
    last_login: Optional[datetime] = Field(None, alias="lastLogin")

    class Config:
        allow_population_by_field_name = True

# Migration helper
class UserMigrator:
    @staticmethod
    def migrate_v1_to_v2(user_v1: UserV1) -> UserV2:
        return UserV2(**user_v1.dict())

    @staticmethod
    def migrate_v2_to_v3(user_v2: UserV2) -> UserV3:
        data = user_v2.dict()
        data['full_name'] = data.pop('name')
        return UserV3(**data)
```

16 Real-World Use Cases

16.1 API Request/Response Models

Complete API Model Set

```python
from pydantic import BaseModel, Field, validator
from typing import List, Optional, Dict
from datetime import datetime
from enum import Enum

class OrderStatus(str, Enum):
    PENDING = "pending"
    CONFIRMED = "confirmed"
    SHIPPED = "shipped"
    DELIVERED = "delivered"
    CANCELLED = "cancelled"

class ProductBase(BaseModel):
    name: str = Field(..., min_length=1, max_length=100)
    price: float = Field(..., gt=0)
    description: Optional[str] = None
    category: str

class ProductCreate(ProductBase):
    stock_quantity: int = Field(..., ge=0)

class ProductResponse(ProductBase):
    id: int
    stock_quantity: int
    created_at: datetime
    updated_at: Optional[datetime] = None

class OrderItemCreate(BaseModel):
    product_id: int
    quantity: int = Field(..., gt=0)

class OrderItemResponse(BaseModel):
    id: int
    product: ProductResponse
    quantity: int
    unit_price: float
    total_price: float

class OrderCreate(BaseModel):
    customer_email: str = Field(..., regex=r'^[\w\.-]+@[\w\.-]+\.\w+)
    items: List[OrderItemCreate] = Field(..., min_items=1)
    shipping_address: str
    notes: Optional[str] = None

class OrderResponse(BaseModel):
    id: int
    customer_email: str
    items: List[OrderItemResponse]
    status: OrderStatus
    total_amount: float
    shipping_address: str
    notes: Optional[str] = None
    created_at: datetime
    updated_at: Optional[datetime] = None

    @validator('total_amount')
    def validate_total_amount(cls, v):
        if v < 0:
            raise ValueError('Total amount must be non-negative')
        return v

class OrderUpdate(BaseModel):
    status: Optional[OrderStatus] = None
```

16.2 Configuration File Processing

Configuration Models

```python
from pydantic import BaseModel, Field, validator
from typing import List, Dict, Optional, Union
from pathlib import Path
import yaml
import json

class DatabaseConfig(BaseModel):
    host: str = "localhost"
    port: int = Field(5432, ge=1, le=65535)
    database: str
    username: str
    password: str
    pool_size: int = Field(10, ge=1, le=100)
    ssl_mode: str = Field("prefer", regex="^(disable|allow|prefer|require)$")

class RedisConfig(BaseModel):
    host: str = "localhost"
    port: int = Field(6379, ge=1, le=65535)
    db: int = Field(0, ge=0, le=15)
    password: Optional[str] = None
    ttl: int = Field(3600, ge=1)  # 1 hour default

class LoggingConfig(BaseModel):
    level: str = Field("INFO", regex="^(DEBUG|INFO|WARNING|ERROR|CRITICAL)$")
    file_path: Optional[Path] = None
    max_size: str = Field("10MB", regex=r'^\d+[KMG]B)
    backup_count: int = Field(5, ge=1, le=10)

class SecurityConfig(BaseModel):
    secret_key: str = Field(..., min_length=32)
    algorithm: str = "HS256"
    access_token_expire_minutes: int = Field(30, ge=1)
    refresh_token_expire_days: int = Field(30, ge=1)

class AppConfig(BaseModel):
    app_name: str = "My Application"
    version: str = "1.0.0"
    debug: bool = False
    host: str = "0.0.0.0"
    port: int = Field(8000, ge=1, le=65535)

    database: DatabaseConfig
    redis: RedisConfig
    logging: LoggingConfig
    security: SecurityConfig

    # Feature flags
    features: Dict[str, bool] = Field(default_factory=dict)

    @validator('version')
    def validate_version(cls, v):
        import re
        if not re.match(r'^\d+\.\d+\.\d+, v):
            raise ValueError('Version must follow semantic versioning (x.y.z)')
        return v

class ConfigLoader:
    @staticmethod
    def load_from_file(file_path: Union[str, Path]) -> AppConfig:
        file_path = Path(file_path)

        if not file_path.exists():
```

17 Troubleshooting Common Issues

17.1 Common Validation Errors and Solutions

Error Type	Common Cause	Solution
`type_error`	Wrong data type provided	Check input data types
`value_error`	Custom validation failed	Review validator logic
`missing`	Required field not provided	Add field or make optional
`extra`	Unknown field in input	Use `Config.extra = "ignore"`
`json_invalid`	Invalid JSON format	Validate JSON syntax

Table 7: Common Pydantic Errors

17.2 Debugging Techniques

Debugging Pydantic Models

```python
from pydantic import BaseModel, ValidationError, Field
import traceback
import json

class DebugModel(BaseModel):
    name: str
    age: int = Field(..., ge=0, le=150)
    email: str

    class Config:
        # Enable detailed error messages
        validate_assignment = True
        # Allow extra fields for debugging
        extra = "allow"

def debug_validation(model_class, data):
    """Helper function to debug validation issues"""
    try:
        instance = model_class(**data)
        print(f"    Validation successful: {instance}")
        return instance
    except ValidationError as e:
        print("    Validation failed!")
        print(f"Error count: {len(e.errors())}")
        print("\nDetailed errors:")

        for i, error in enumerate(e.errors(), 1):
            print(f"\n{i}. Field: {' -> '.join(str(x) for x in error['loc'])}")
            print(f"   Input: {error.get('input', 'N/A')}")
            print(f"   Error: {error['msg']}")
            print(f"   Type: {error['type']}")
            if 'ctx' in error:
                print(f"   Context: {error['ctx']}")

        # Pretty print the original data
        print(f"\nOriginal data:")
        print(json.dumps(data, indent=2, default=str))

        return None
    except Exception as e:
        print(f"    Unexpected error: {e}")
        traceback.print_exc()
        return None

# Example usage
test_data = [
    {"name": "John", "age": 25, "email": "john@example.com"},  # Valid
    {"name": "", "age": -5, "email": "invalid-email"},         # Invalid
    {"name": "Jane", "age": "not-a-number", "email": "jane@example.com"},  # Type error
]

for i, data in enumerate(test_data, 1):
    print(f"\n{'='*50}")
    print(f"Test case {i}:")
    print(f"{'='*50}")
    debug_validation(DebugModel, data)
```

18 Best Practices Summary

18.1 Development Best Practices

1. **Use Type Hints Consistently**
   - Always specify return types for computed fields
   - Use Union types for fields that accept multiple types
   - Import types from `typing` module for Python ¡ 3.9

2. **Design for Validation**
   - Add field constraints early in development
   - Use custom validators for business logic
   - Provide meaningful error messages

3. **Structure Models Logically**
   - Group related fields in nested models
   - Use composition over deep inheritance
   - Keep models focused and cohesive

4. **Handle Optional Fields Properly**
   - Use `Optional[T]` for nullable fields
   - Provide sensible defaults where appropriate
   - Document when `None` is a meaningful value

5. **Performance Considerations**
   - Avoid excessive nesting when possible
   - Cache frequently used model instances
   - Use appropriate data structures for collections

18.2 Security Best Practices

Security Considerations

- **Input Sanitization**: Always validate and sanitize user input
- **Sensitive Data**: Be careful with sensitive fields in serialization
- **Size Limits**: Set appropriate limits on string and collection sizes
- **Regex Safety**: Use safe regex patterns to avoid ReDoS attacks
- **Error Messages**: Don't leak sensitive information in error messages

19 Conclusion

Pydantic provides a powerful, type-safe way to handle data validation and serialization in Python applications. Its integration with Python's type system makes code more maintainable and self-documenting, while its performance and flexibility make it suitable for a wide range of applications.

19.1 Key Takeaways

- Pydantic models provide automatic data validation based on type hints
- Field validators and model validators enable custom business logic
- Computed fields allow calculated properties with automatic serialization
- Nested models support complex data structures with full validation
- Integration with frameworks like FastAPI provides powerful API development capabilities
- Proper error handling and debugging techniques improve development experience
- Following best practices ensures maintainable and secure applications

19.2 Next Steps

To further your Pydantic knowledge:

1. Explore Pydantic v2 features and migration strategies
2. Practice with real-world API development using FastAPI
3. Implement data pipelines using Pydantic for validation
4. Study advanced configuration management patterns
5. Contribute to open-source projects using Pydantic

Remember: Good validation is not just about preventing errors—it's about creating robust, maintainable, and user-friendly applications.

20 Appendix: Quick Reference

20.1 Common Imports

```
from pydantic import (
    BaseModel,
    Field,
    validator,
    field_validator,
    model_validator,
    computed_field,
    ValidationError
)
from typing import List, Dict, Optional, Union
from datetime import datetime
from enum import Enum
```

20.2 Field Constraint Quick Reference

```
# Numeric constraints
age: int = Field(..., ge=0, le=120)
price: float = Field(..., gt=0)
score: float = Field(..., ge=0.0, le=100.0)

# String constraints
username: str = Field(..., min_length=3, max_length=20)
email: str = Field(..., regex=r'^[\w\.-]+@[\w\.-]+\.\w+)

# Collection constraints
tags: List[str] = Field(..., min_items=1, max_items=10)
items: List[int] = Field(..., unique_items=True)
```

20.3 Validation Patterns

```
# Field validator
@field_validator('field_name')
def validate_field(cls, v):
    if condition:
        raise ValueError('Error message')
    return v

# Model validator
@model_validator(mode='after')
def validate_model(self):
    if self.field1 > self.field2:
        raise ValueError('Field1 must be less than field2')
    return self

# Computed field
@computed_field
@property
def computed_value(self) -> str:
    return f"{self.field1}-{self.field2}"
```