5 Basic things you should know of PYDANTIC

Jam

learn the building blocks of AI

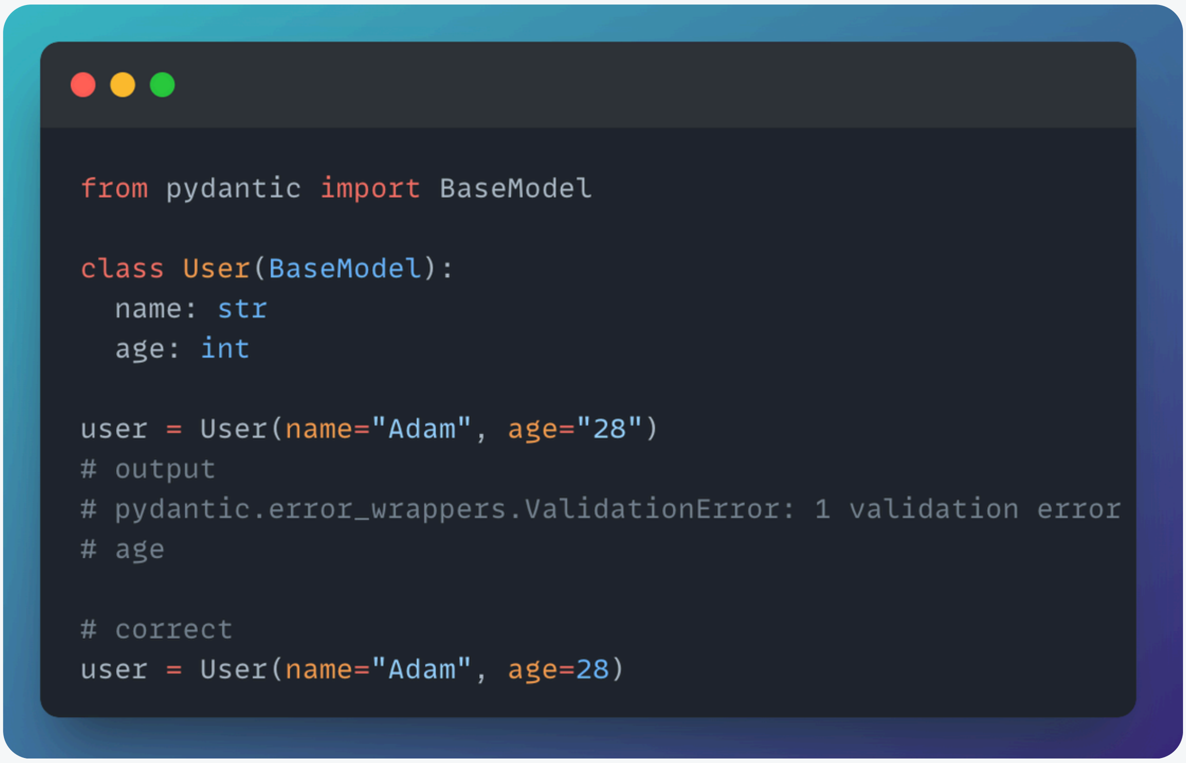
WITH AI

Introduction to Pydantic

01

OVERVIEW

Pydantic is a Python library that uses type annotations for data validation and settings management



```python
from pydantic import BaseModel

class User(BaseModel):
  name: str
  age: int

user = User(name="Adam", age="28")
# output
# pydantic.error_wrappers.ValidationError: 1 validation error
# age

# correct
user = User(name="Adam", age=28)
```

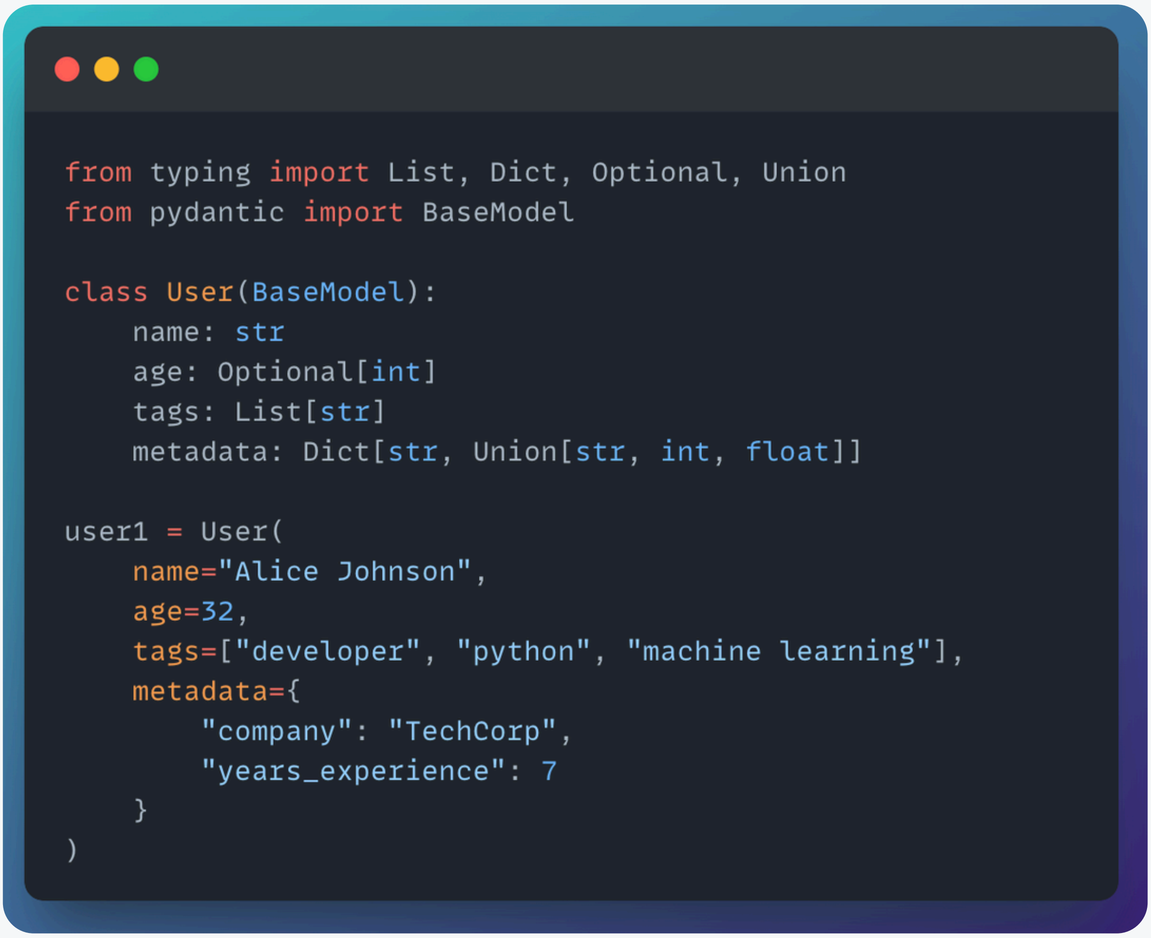
- Pydantic will validate the data and raise errors if any field doesn't meet the specified requirements.

Beyond “str”,“int” 02

OVERVIEW

Pydantic supports various field types:

- Primitive types: str, int, float, bool
- Collection types: list, tuple, set, dict
- Optional types: Optional from the typing module for fields that can be None



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

class User(BaseModel):
    name: str
    age: Optional[int]
    tags: List[str]
    metadata: Dict[str, Union[str, int, float]]

user1 = User(
    name="Alice Johnson",
    age=32,
    tags=["developer", "python", "machine learning"],
    metadata={
        "company": "TechCorp",
        "years_experience": 7
    }
)
```

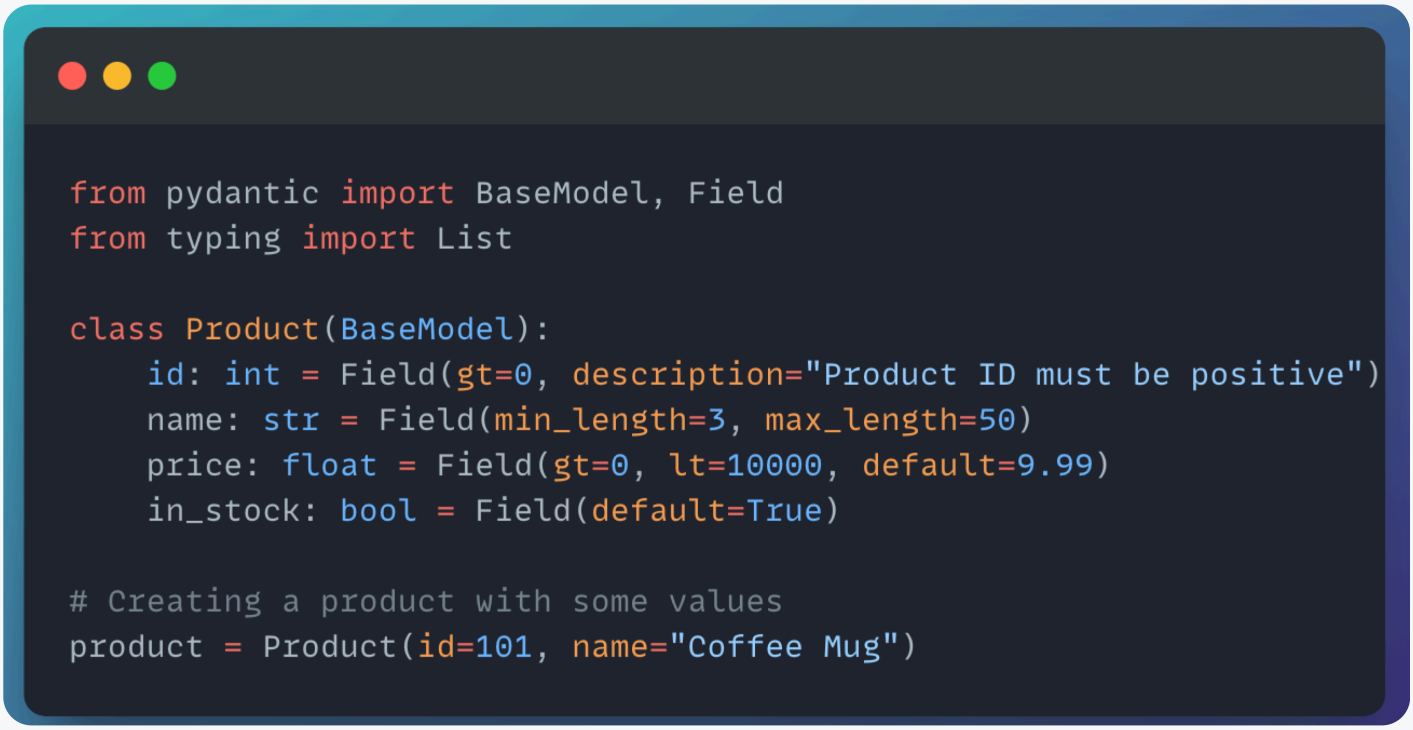
- Fields are required by default unless explicitly marked as Optional.
- Missing required fields will raise ValidationError.

Field Constraints and Default Values

03

OVERVIEW

Field constraints and default values allow you to define validation rules and fallback values directly in your Pydantic models, ensuring data quality while providing flexibility.



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

class Product(BaseModel):
    id: int = Field(gt=0, description="Product ID must be positive")
    name: str = Field(min_length=3, max_length=50)
    price: float = Field(gt=0, lt=10000, default=9.99)
    in_stock: bool = Field(default=True)

# Creating a product with some values
product = Product(id=101, name="Coffee Mug")
```

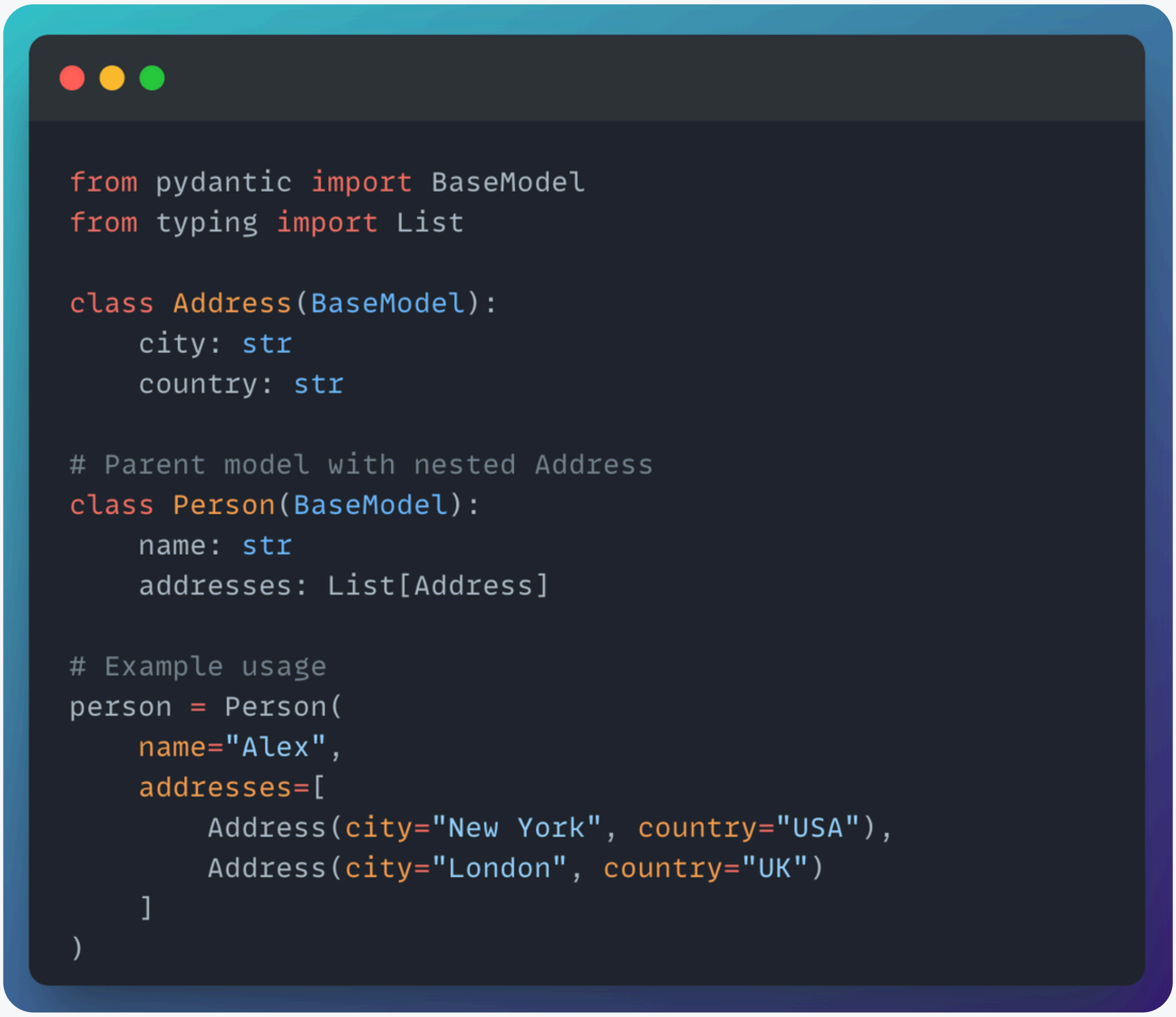
- Use Field( ) to add constraints like min/max values, string lengths, and regex patterns
- Specify defaults for optional fields with default= or use default_factory= for mutable defaults.

Nested Models

04

OVERVIEW

Pydantic allows models to be nested within each other, enabling complex data structures.



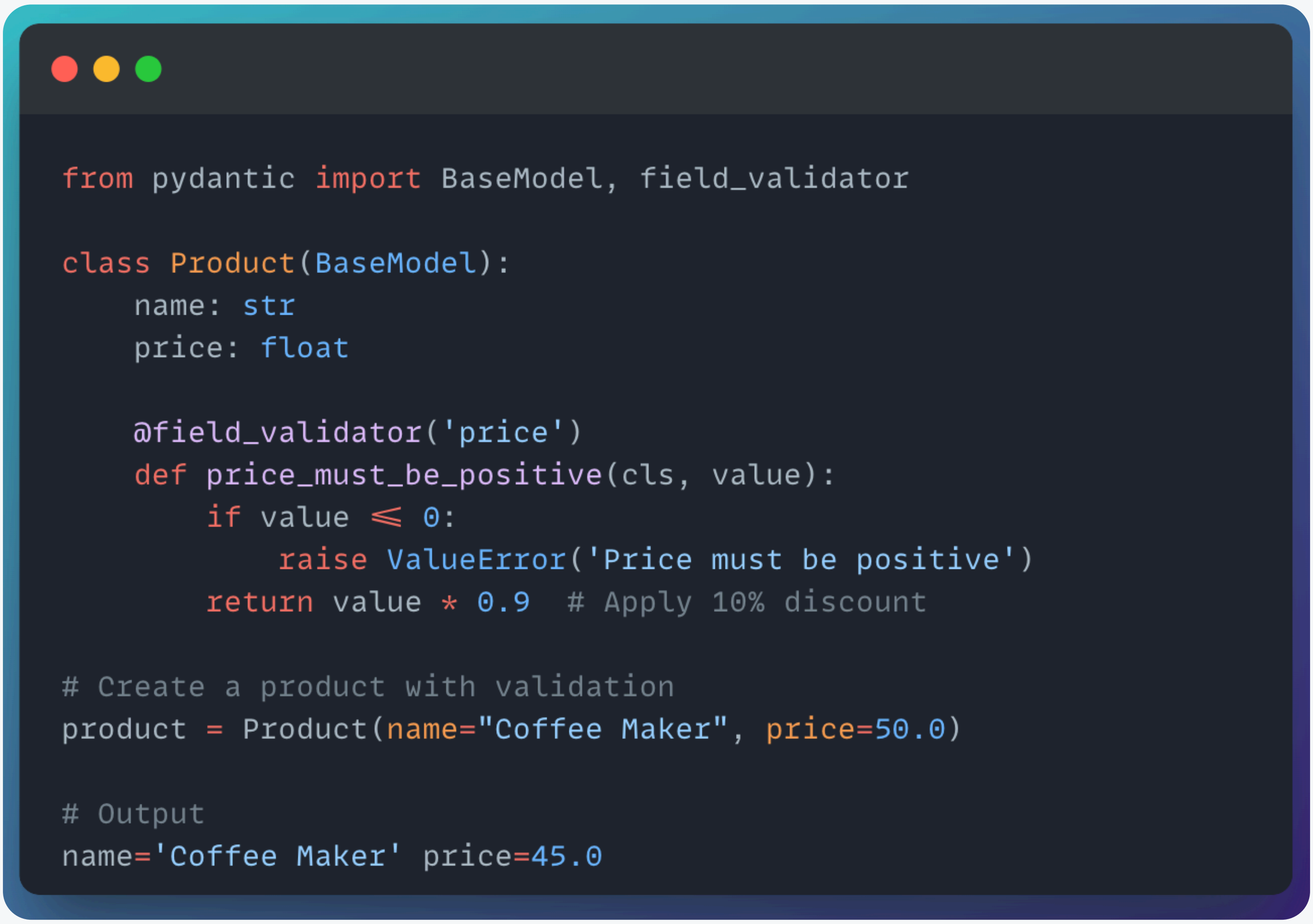
```python
from pydantic import BaseModel
from typing import List

class Address(BaseModel):
    city: str
    country: str

# Parent model with nested Address
class Person(BaseModel):
    name: str
    addresses: List[Address]

# Example usage
person = Person(
    name="Alex",
    addresses=[
        Address(city="New York", country="USA"),
        Address(city="London", country="UK")
    ]
)
```

- When defining nested models, Pydantic handles validation of the entire object tree, ensuring that data at all levels meets your specified requirements.

Custom Validators 05

OVERVIEW

Custom validators enable complex validation logic beyond simple type checking, allowing for data transformation, cross-field validation, and business rule enforcement.



```
from pydantic import BaseModel, field_validator

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

    @field_validator('price')
    def price_must_be_positive(cls, value):
        if value <= 0:
            raise ValueError('Price must be positive')
        return value * 0.9  # Apply 10% discount

# Create a product with validation
product = Product(name="Coffee Maker", price=50.0)

# Output
name='Coffee Maker' price=45.0
```

- Validators can both validate and transform input data
- Validation errors provide specific feedback about what went wrong
- Validators are executed in a predictable order during model creation