Task 1 (Pydantic)

What is Pydantic and what problem does it solve?

Pydantic is a Python library for **data validation and parsing**, built on top of **Python type hints**. It ensures that the data your application receives — from APIs, config files, databases, user input, or ML pipelines — is **valid, correctly typed, and well-structured**, without requiring you to write manual validation code.

> **Problem it solves:**
> In real-world applications, data is often messy (wrong types, missing fields, extra fields, improper formats). Pydantic **validates data automatically**, converts it to the right types, and raises meaningful errors when something is wrong — saving you time and preventing runtime bugs.

What are Python Type Hints?

Python type hints, also known as type annotations, are a feature introduced in Python 3.5 that allows developers to indicate the expected data types of variables, function arguments, and return values.

How to use Python Type Hints?

Type hints are added using a colon (`:`) followed by the type, for variables, and in the function signature for parameters and return types. For more advanced features of **Typing Hints** we can use the **Typing** module which as several features for different scenarios.

```python
# Variable annotation
name: str = "Alice"
age: int = 30

# Function parameter and return type annotation
def greet(name: str) -> str:
    return f"Hello, {name}!"

def calculate_sum(a: int, b: int) -> int:
    return a + b
```

Example of why to use Type Hints

```python
# # without python type hints
def calculate_total_without_hints(items, discount):
    return sum(items) * (1 - discount)
calculate_total_without_hints([100, 200, 300], "10%")
# # with python type hints
def calculate_total(items: list[float], discount: float) -> float:
    return sum(items) * (1 - discount)
calculate_total([100, 200, 300], "10%")

#on running the above 2 codes using mypy we can easily see that the 1st one doesn't show any error during static analysis whereas the 2nd one does show errors during that phase.
```

Advanced Typing Hints

The `typing` module provides more advanced type hints for complex scenarios:

- **List, Dict, Tuple, Set:** To specify types within collections (e.g., `List[str]`, `Dict[str, int]`).
- **Union:** To indicate that a variable can be one of several types (e.g., `Union[int, float]`, or `int | float` in Python 3.10+).
- **Optional:** To indicate that a variable can be a specific type or `None` (e.g., `Optional[str]`).
- **Callable:** For type-hinting functions or other callable objects.
- **TypeVar:** For defining generic types.

Important Note: Type hints are primarily for static analysis and documentation; Python's runtime does not enforce them. Incorrect types assigned to type-hinted variables will not raise runtime errors by default.

Why is Pydantic so popular?

Reason	Description
Powered by type hints	Uses Python type annotations to define schemas. You don't need to learn a new syntax; validation and serialization are driven by `int`, `str`, `list`, `Dict`, `Optional`, etc. Works naturally with IDE autocompletion & mypy.
High speed (Rust-based validation)	Pydantic v2 uses a Rust core (via `pydantic-core`), making it **one of the fastest validation libraries in Python**.

Reason	Description
JSON Schema support	Pydantic models can output **JSON Schema**, enabling smooth integration with tools like OpenAPI, FastAPI, and frontend clients.
Strict vs Lax mode	You can choose: • **Strict mode:** no type coercion (e.g., `"42"` is NOT accepted as `int`) • **Lax mode:** smart type coercion (e.g., `"42"` → `42` automatically)
Supports dataclasses, TypedDicts and more	Works seamlessly with `@dataclass`, `TypedDict`, `Enum`, and many standard typing features.
Customization through validators/serializers	Developers can plug in custom logic (e.g., regex checks, range checks, auto formatting) using validators.
Large ecosystem and adoption	Used by major frameworks like **FastAPI, SQLModel, LangChain, Django Ninja, HuggingFace**, etc.
sada*Battle-tested**	Extremely mature and trusted in production at scale, including all major tech companies.

Advantages of using Pydantic?

Feature / Advantage	Why it matters
Automatic type conversion	Converts input to correct type (e.g., string → int)
Structured error reporting	Easy debugging and cleaner API error responses
Nested model validation	Perfect for complex JSON / hierarchical data
Field constraints	e.g., `min_length`, `max_length`, `gt`, `lt`
Custom validators	Total control over data validation
Serialization	Export to `dict` or JSON easily (`model_dump`, `model_dump_json`)
Environment/Config validation	With `BaseSettings`, great for ML pipelines and .env configs

Getting started with Pydantic

```
python -m venv pydantic_env source pydantic_env/bin/activate
pip install pydantic
pip install "pydantic[email]"
```

So essentially we need Pydantic to perform Data validation that is, let's say data comes in to our application from multiple sources (user input, API, Database) and sometimes that data can be in a format which is not valid for our use-case (eg. users might submit a number as a string instead of an integer or an API might return null instead of numbers)

So traditionally we will have to write manual validation logic for this like this:

```
data={"name":"yash", "age":20,"email":"abc@gmail.com","is_active":false}

#.get() is used with dictionaries to retrieve data from a specified key.
Better than indexing because indexing causes KeyError if key is not found
def create_user(data):
    # Manual validation nightmare
    if not isinstance(data.get('age'), int):
        raise ValueError("Age must be an integer")
    if data['age'] < 0 or data['age'] > 150:
        raise ValueError("Age must be between 0 and 150")
    if not isinstance(data.get('email'), str) or '@' not in data['email']:
        raise ValueError("Invalid email format")
    if not isinstance(data.get('is_active'), bool):
        raise ValueError("is_active must be a boolean")

    # Finally create the user...
    return User(data['age'], data['email'], data['is_active'])
```

Pydantic is used in the industry because of its 3 main features:

1. Type Hints
2. Runtime Validation
3. Automatic serialization - basically we can convert our pydantic model to JSON encoded string or to a dictionary which is helpful when we want to export data to a database or to an API.

```
        #convert to dictionary
        .model_dump()
        #convert to json
        .model_dump_json()
```

Using Pydantic, we define your data structure once using Python's type annotation syntax, and Pydantic handles all the validation automatically:

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

class User(BaseModel):
    age: int
    email: EmailStr
    is_active: bool = True
    nickname: Optional[str] = None

# Pydantic automatically validates and converts data
user_data = {
    "age": "25",  # String gets converted to int
    "email": "john@example.com",
    "is_active": "true"  # String gets converted to bool
}

user = User(**user_data)
print(user.age)  # 25 (as integer)
print(user.model_dump())  # Clean dictionary output
```

Pydantic gives us the advantage of being very fast (since its core validation logic is written in Rust which makes this validation faster than Python validation in most cases). Therefore our applications can handle thousands of requests without validation being a bottleneck.

Also Pydantic allows for easy integration with modern frameworks such as FastAPI. Also when we define a Pydantic model, we can easily do its seriatlization to OpenAPI schema for free:

```python
from fastapi import FastAPI
from pydantic import BaseModel, EmailStr

app = FastAPI()

class UserCreate(BaseModel):
    name: str
    email: EmailStr
    age: int

@app.post("/users/")
async def create_user(user: UserCreate):
    # FastAPI automatically validates the request body
    # and generates API docs from your Pydantic model
    return {"message": f"Created user {user.name}"}
```

Solving a real problem: validating user registration data from a web form

```
from pydantic import BaseModel, EmailStr
from typing import Optional
from datetime import datetime

class User(BaseModel):
    name: str
    email: EmailStr
    age: int
    is_active: bool = True
    created_at: datetime = None

# Test with clean data
clean_data = {
    "name": "Alice Johnson",
    "email": "alice@example.com",
    "age": 28
}

user = User(**clean_data)
print(f"User created: {user.name}, Age: {user.age}")
print(f"Model output: {user.model_dump()}")
```

Let's understand this thoda sa:

- **Creating a Pydantic model**: Your `User` class inherits from `BaseModel`, which gives it all of Pydantic's validation and serialization capabilities. This inheritance turns a regular Python class into a data validation tool.
- **Field definitions**: Each line in the class defines a field with its expected type. The `name: str` syntax tells Pydantic that `name` should be a string, `age: int` means age should be an integer, and so on.
- **EmailStr explained**: `EmailStr` is a special Pydantic type that automatically validates email addresses. It comes from the `pydantic[email]` package you installed earlier and uses regular expressions to ensure the email format is valid. If someone passes "not-an-email", Pydantic will raise a validation error.
- **Default values**: Fields like `is_active: bool = True` have default values. If you don't provide these fields when creating a user, Pydantic uses the defaults. The `= None` for `created_at` makes this field optional.
- **Model instantiation**: When you call `User(**clean_data)`, the `**` unpacks your dictionary and passes each key-value pair as keyword arguments to the model constructor.

Now let's see pydantic's automatic type conversion:

```
#automatic type conversion in action
class User(BaseModel):
    name: str
    email: EmailStr
    age: int
    is_active: bool = True
    created_at: Optional[datetime] = None

messy_data = {
    "name":"Daniel Patel",
    "email":"daniel@gmail.com",
    "age":"34",
    "is_active":"true"
}
user= User(**messy_data)
print(f"Age type: {type(user.age)}")
print(f"Is active type: {type(user.is_active)}")
print(f"User is active? {user.is_active} and his Age is: {user.age}")
```

When Pydantic's validation fails, it generates clear error messages:

```
#validation error in pydantic
invalid_data={
    "name":"",
    "email":"abcdsa",
    "age":20
}
try:
    invalid_user = User(**invalid_data)
except ValidationError as e:
    print(e)
```

Python's Dataclass and how is it different from Pydantic

So `dataclass` (can be imported using built in `dataclasses module`) is a light way to create classes that mainly store data

So instead of writing code like this:

```
class User:
    def __init__(self, name, age):
        self.name = name
```

```
        self.age = age
```

We can write something like this:

```
from dataclasses import dataclass

@dataclass
class User:
    name: str
    age: int
```

This automatically generates:

- `__init__()`
- `__repr__()`
- `__eq__()`
- and other basic methods.
  It's **purely syntactic sugar** — no validation or type conversion happens at runtime.

`Example between the 2:`

```
from dataclasses import dataclass

#@dataclass is a decorator
@dataclass
class ProductDataclass:
   name: str
   price: float
   in_stock: bool

# Fast, simple, but no validation
product = ProductDataclass("Laptop", 999.99, True)

# This also works, even though types are wrong:
broken_product = ProductDataclass(123, "expensive", "maybe")
```

On the other hand Pydantic models add validation, serialization and enable framework integration

```
from pydantic import BaseModel, Field
```

```
#Field function is used to customize and add metadata to fields within a
Pydantic model.
class ProductPydantic(BaseModel):
    name: str = Field(min_length=1)
    price: float = Field(gt=0)  # Must be greater than 0
    in_stock: bool

# Automatic validation prevents bad data
try:
    product = ProductPydantic(name="", price=-10, in_stock="maybe")
except ValidationError as e:
    print("Validation caught the errors!")

# Valid data works perfectly
good_product = ProductPydantic(
    name="Laptop",
    price="999.99",  # String converted to float
    in_stock=True
)
```

When to choose each approach:

- Use dataclasses for internal data structures, configuration objects, or when performance is critical and you trust your data sources
- Use Pydantic for API endpoints, user input, external data parsing, or when you need JSON serialization

We can also wrap Python's dataclasses with Pydantic to give them validation powers:

```
from pydantic.dataclasses import dataclass

@dataclass
class User:
    name: str
    age: int

u = User(name="Alice", age="25")
print(u)
print(type(u.age))
```

Building Data models with Pydantic

Field validation and constraints

Task - Create a product catalogue API where price data comes from multiple vendors with different formatting standards (some might send prices as strings, others as floats, etc.)

```
from pydantic import BaseModel, Field
from decimal import Decimal
from typing import Optional

class Product(BaseModel):
    name: str = Field(min_length=1, max_length=100)
    price: Decimal = Field(gt=0, le=10000)  # Greater than 0, less than or
equal to 10,000
    description: Optional[str] = Field(None, max_length=500)
    category: str = Field(..., pattern=r'^[A-Za-z\s]+$')  # Only letters and
spaces
    stock_quantity: int = Field(ge=0)  # Greater than or equal to 0
    is_available: bool = True

# This works - all constraints satisfied
valid_product = Product(
    name="Wireless Headphones",
    price="199.99",  # String converted to Decimal
    description="High-quality wireless headphones",
    category="Electronics",
    stock_quantity=50
)

# This fails with clear error messages
try:
    invalid_product = Product(
        name="",  # Too short
        price=-50,  # Negative price
        category="Electronics123",  # Contains numbers
        stock_quantity=-5  # Negative stock
    )
except ValidationError as e:
    print(f"Validation errors: {len(e.errors())} issues found")
```

Each `Field()` parameter serves a specific purpose: `min_length` and `max_length` prevent database schema violations, `gt` and `le` create business logic boundaries, and `pattern` validates formatted data using regular expressions. The `Field(...)` syntax with ellipsis marks the required fields, while `Field(None, ...)` creates optional fields with validation rules.

Type Coercion vs Strict Validation (Strict vs Lax mode)

By default, Pydantic converts compatible types rather than rejecting them outright. This flexibility works well for user input, but some scenarios demand exact type matching. Here comes the Strict mode in Pydantic.

```
#automatic type conversion in action

class User(BaseModel):
    name: str
    email: EmailStr
    age: int
    is_active: bool = True
    created_at: Optional[datetime] = None

messy_data = {
    "name":"Daniel Patel",
    "email":"daniel@gmail.com",
    "age":"34",
    "is_active":"true"
}
user= User(**messy_data)
print(f"Age type: {type(user.age)}")
print(f"Is active type: {type(user.is_active)}")
print(f"User is active? {user.is_active} and his Age is: {user.age}")
```

```
#using Pydantic's Strict mode
#Strict mode vs Lax mode in Pydantic
from pydantic import StrictStr, StrictInt, StrictBool
class User(BaseModel):
    name: str
    email: EmailStr
    age: int
    is_active: bool = True
    created_at: Optional[datetime] = None

class StrictUser(BaseModel):
    name: StrictStr
    email: EmailStr = Field(..., strict=True)
    age: StrictInt
    is_active: StrictBool = True
    created_at: Optional[datetime] = Field(..., strict=True)

messy_data = {
    "name":"Daniel Patel",
    "email":"daniel@gmail.com",
    "age":"34",
```

```
    "is_active":"true"
}

user= User(**messy_data)
print(f"Age type: {type(user.age)}")
print(f"Is active type: {type(user.is_active)}")
print(f"User is active? {user.is_active} and his Age is: {user.age}")

try:
    strict_user= StrictUser(**messy_data)
except ValidationError as e:
    print(e)
```

Nested models and complex data

Real applications handle complex, interconnected data structures. An e-commerce order contains customer information, shipping addresses, and multiple product items, each requiring its own validation:

```
from typing import List
from datetime import datetime

class Address(BaseModel):
   street: str = Field(min_length=5)
   city: str = Field(min_length=2)
   postal_code: str = Field(pattern=r'^\d{5}(-\d{4})?$')
   country: str = "USA"

class Customer(BaseModel):
   name: str = Field(min_length=1)
   email: EmailStr
   shipping_address: Address
   billing_address: Optional[Address] = None

class OrderItem(BaseModel):
   product_id: int = Field(gt=0)
   quantity: int = Field(gt=0, le=100)
   unit_price: Decimal = Field(gt=0)

class Order(BaseModel):
   order_id: str = Field(pattern=r'^ORD-\d{6}$')
   customer: Customer
   items: List[OrderItem] = Field(min_items=1)
   order_date: datetime = Field(default_factory=datetime.now)
```

```
# Complex nested data validation
order_data = {
    "order_id": "ORD-123456",
    "customer": {
        "name": "John Doe",
        "email": "john@example.com",
        "shipping_address": {
            "street": "123 Main Street",
            "city": "Anytown",
            "postal_code": "12345"
        }
    },
    "items": [
        {"product_id": 1, "quantity": 2, "unit_price": "29.99"},
        {"product_id": 2, "quantity": 1, "unit_price": "149.99"}
    ]
}

order = Order(**order_data)
print(f"Order validated with {len(order.items)} items")
```

Optional Fields and None handling

Sometimes User creation demands complete information while updates should accept partial changes

```
from typing import Optional

class UserCreate(BaseModel):
    name: str = Field(min_length=1)
    email: EmailStr
    age: int = Field(ge=13, le=120)
    phone: Optional[str] = Field(None, pattern=r'^\+?1?\d{9,15}$')

class UserUpdate(BaseModel):
    name: Optional[str] = Field(None, min_length=1)
    email: Optional[EmailStr] = None
    age: Optional[int] = Field(None, ge=13, le=120)
    phone: Optional[str] = Field(None, pattern=r'^\+?1?\d{9,15}$')

# PATCH request with partial data
update_data = {"name": "Jane Smith", "age": 30}
user_update = UserUpdate(**update_data)

# Serialize only provided fields
```

```
patch_data = user_update.model_dump(exclude_none=True)
print(f"Fields to update: {list(patch_data.keys())}")
```

Serialization converts Pydantic objects back into dictionaries or JSON strings for storage or transmission. The `model_dump()` method handles this conversion, with `exclude_none=True` removing unprovided fields. This pattern works perfectly for PATCH requests where clients send only the fields they want to change, preventing accidental data overwrites in your database.

Custom Validation and Real-World Integration

Sometimes, we can have a user registration form where password requirements vary based on subscription plans, or an API that receives address data from multiple countries with different postal code formats. These scenarios require custom validation logic that captures your specific business rules while integrating smoothly with web frameworks and configuration systems.

Field Validators and model validation

For custom business logic use = `@field_validators` decorator as it transforms your validation functions into part of the model itself.

In **Pydantic v2**, the `@field_validator` decorator is used to define **custom validation logic** for a specific field.

✅ It allows you to:

- Add **extra constraints** beyond simple `Field()` rules.
- Access **other fields** (if needed) through the `info` argument.
- Raise custom error messages.

Task - a user registration system where different subscription tiers have different password requirements:

```
from pydantic import BaseModel, field_validator, Field
import re

class UserRegistration(BaseModel):
    username: str = Field(min_length=3)
    email: EmailStr
    password: str
    subscription_tier: str = Field(pattern=r'^(free|pro|enterprise)$')

    @field_validator('password')
    @classmethod
    def validate_password_complexity(cls, password, info):
```

```
        tier = info.data.get('subscription_tier', 'free')

        if len(password) < 8:
            raise ValueError('Password must be at least 8 characters')

        if tier == 'enterprise' and not re.search(r'[A-Z]', password):
            raise ValueError('Enterprise accounts require uppercase letters')

        return password
```

`@field_validator` is typically used when we want to custom validate a field using rules that depend on multiple fields.

But if we want validation of multiple fields instead of just one we use the `@model_validator` decorator that runs after all the individual fields are validated.

```
from datetime import datetime
from pydantic import model_validator

class EventRegistration(BaseModel):
    start_date: datetime
    end_date: datetime
    max_attendees: int = Field(gt=0)
    current_attendees: int = Field(ge=0)

    @model_validator(mode='after')
    def validate_event_constraints(self):
        if self.end_date <= self.start_date:
            raise ValueError('Event end date must be after start date')

        if self.current_attendees > self.max_attendees:
            raise ValueError('Current attendees cannot exceed maximum')

        return self
```

A `@model_validator` method doesn't receive `password` or `info` — instead, it receives the **entire model instance** (`self` when `mode='after'`) or the raw input data (`cls, data` when `mode='before'`).

The validator must return `self` to indicate successful validation.

Configuration management with environment variables

How to access the .env files safely using Pydantic so that our production applications are secure and deployment friendly?

Answer - Using `Pydantic's BaseSettings`

```
# .env file
DATABASE_URL=postgresql://user:password@localhost:5432/myapp
SECRET_KEY=your-secret-key-here
DEBUG=False
ALLOWED_HOSTS=["localhost", "127.0.0.1", "yourdomain.com"]
```

Then define the settings model

```
from pydantic_settings import BaseSettings
from pydantic import Field
from typing import List

class AppSetting(BaseSettings):
    database_url: str= Field(description='Database connection url')
    secret_key: str= Field(description='Secret key for JWT tokens')
    debug: bool= Field(default=False, alias='APP_DEBUG')
    allowed_hosts: List[str]= Field(default=["localhost"])

    class Config:
        env_file=".env"
        case_sensitive=False
#load settings automatically from environment and .env file
settings = AppSetting()
```

Since we are loading data from .env file tehrefore we also use the `config subclass` .

The `BaseSettings` class automatically reads from environment variables, `.env` files, and command-line arguments. Environment variables take precedence over `.env` file values, making it easy to override settings in different deployment environments. The `case_sensitive = False` setting allows flexible environment variable naming.

For complex applications, you can organize settings into logical groups:

```
class DatabaseSettings(BaseSettings):
   url: str = Field(env="DATABASE_URL")
   max_connections: int = Field(default=5, env="DB_MAX_CONNECTIONS")

class AppSettings(BaseSettings):
   secret_key: str
   debug: bool = False
   database: DatabaseSettings = DatabaseSettings()
```

```
    class Config:
        env_file = ".env"

settings = AppSettings()
# Access nested configuration
db_url = settings.database.url
```

The `.env` file approach works with deployment platforms like Heroku, AWS, and Docker, where environment variables are the standard way to configure applications