Mindmap to Learn Pydantic

1. **Introduction to Pydantic**
   * **Overview**: Pydantic is a data validation and settings management library for Python, leveraging Python type hints to provide runtime data validation and parsing.
   * **Installation**: Install Pydantic using pip: pip install pydantic.
2. **Basic Concepts**
   * **Data Validation**: Pydantic validates data based on type hints.
     + Example: from pydantic import BaseModel; class User(BaseModel): name: str; age: int.
   * **Data Parsing**: Automatically converts data into desired types.
     + Example: user = User(name="John", age="30") converts age to int.
3. **Models**
   * **Creating Models**: Define models by subclassing BaseModel.
     + Example: class Item(BaseModel): name: str; price: float.
   * **Model Initialization**: Create instances with provided data.
     + Example: item = Item(name="Laptop", price=999.99).
4. **Field Types and Validation**
   * **Type Annotations**: Use Python type annotations for field types.
     + Example: class Person(BaseModel): name: str; age: int.
   * **Default Values**: Provide default values for fields.
     + Example: class User(BaseModel): name: str = "John Doe"; age: int = 30.
   * **Field Validation**: Use Pydantic's Field for advanced validation.
     + Example: from pydantic import Field; class User(BaseModel): age: int = Field(..., gt=0, lt=100).
5. **Model Methods**
   * **dict() Method**: Convert model instances to dictionaries.
     + Example: user\_dict = user.dict().
   * **json() Method**: Convert model instances to JSON strings.
     + Example: user\_json = user.json().
   * **copy() Method**: Create a copy of a model instance.
     + Example: user\_copy = user.copy().
6. **Model Configurations**
   * **Config Class**: Customize model behavior using the Config class.
     + Example: class User(BaseModel): class Config: allow\_mutation = False.
   * **Alias Generation**: Use field aliases.
     + Example: class User(BaseModel): name: str = Field(..., alias='username').
7. **Nested Models**
   * **Embedding Models**: Use other Pydantic models as fields.
     + Example: class Address(BaseModel): city: str; class User(BaseModel): name: str; address: Address.
   * **Recursive Models**: Models referencing themselves.
     + Example: class TreeNode(BaseModel): value: int; children: List['TreeNode'].
8. **Data Validation**
   * **Validators**: Custom validation methods using @validator.
     + Example: from pydantic import validator; class User(BaseModel): name: str; @validator('name'); def name\_must\_not\_be\_empty(cls, v): if not v: raise ValueError('must not be empty'); return v.
   * **Root Validators**: Validate the entire model.
     + Example: @root\_validator; def check\_all(cls, values): if values['password'] != values['confirm\_password']: raise ValueError('passwords do not match'); return values.
9. **Complex Field Types**
   * **Lists and Dictionaries**: Use complex types like lists and dictionaries.
     + Example: class User(BaseModel): friends: List[str]; attributes: Dict[str, Any].
   * **Optional Fields**: Use Optional for nullable fields.
     + Example: from typing import Optional; class User(BaseModel): middle\_name: Optional[str] = None.
10. **Advanced Usage**
    * **Custom Data Types**: Define custom data types.
      + Example: from pydantic import ConstrainedStr; class ShortStr(ConstrainedStr): max\_length = 10.
    * **Inheritance**: Inherit and extend Pydantic models.
      + Example: class Admin(User): admin\_level: int.
    * **Union Types**: Use Union for fields that can have multiple types.
      + Example: from typing import Union; class Response(BaseModel): data: Union[str, int, float].
11. **Serialization and Deserialization**
    * **Parsing Data**: Parse data from different formats.
      + Example: data = '{"name": "John", "age": 30}'; user = User.parse\_raw(data).
    * **Exporting Data**: Export data in different formats.
      + Example: user.json() or user.dict().
12. **Settings Management**
    * **Settings Model**: Use BaseSettings for managing settings.
      + Example: from pydantic import BaseSettings; class Settings(BaseSettings): app\_name: str = "My App"; env: str = "development".
    * **Environment Variables**: Load settings from environment variables.
      + Example: settings = Settings(), with environment variables APP\_NAME and ENV.
13. **Error Handling**
    * **ValidationError**: Handle validation errors.
      + Example: try: User(name="John", age="invalid") except ValidationError as e: print(e.json()).
    * **Custom Error Messages**: Customize error messages.
      + Example: @validator('age'); def check\_age(cls, v): if v < 0: raise ValueError('Age must be positive'); return v.
14. **Performance Optimization**
    * **Model Caching**: Enable model caching for faster performance.
      + Example: class Config: use\_cache = True.
    * **Strict Types**: Use strict types for better validation.
      + Example: class Config: strict = True.
15. **Integrations**
    * **FastAPI Integration**: Use Pydantic models with FastAPI for request and response validation.
      + Example: from fastapi import FastAPI; from pydantic import BaseModel; app = FastAPI(); class Item(BaseModel): name: str; @app.post("/items/") def create\_item(item: Item): return item.
    * **SQLModel Integration**: Use Pydantic models with SQLModel for database interactions.
      + Example: from sqlmodel import SQLModel, Field; class User(SQLModel): id: int = Field(default=None, primary\_key=True); name: str.
16. **Best Practices**
    * **Type Hints**: Use type hints extensively for clear and maintainable code.
      + Example: def get\_user(user\_id: int) -> User: pass.
    * **Model Validation**: Validate models at initialization.
      + Example: user = User(name="John", age=30).
    * **Documentation**: Document models and validation rules.
      + Example: Use docstrings and comments to explain validation logic.