Pydantic

Pydantic makes type hints validate your data at runtime, covered here from coercion and constraints to validators and serialization.

Declare a BaseModel

Subclass BaseModel; your type hints become the schema.

PURPOSE	COMMAND	RESULT
Define a model from type hints.	`class User(BaseModel):` `  id:int  name:str`	
Add a default value.	`name:str = "anon"`	
Make a field nullable.	`nickname:str | None = None`	
Construct an instance.	`u =User(id=1, name="ada")`	
List the declared fields.	`User.model_fields`	
Access attributes (typed).	`u.id # int  u.name # str`	

each annotated attribute becomes a validated field; v2 replaces the inner class Config with model_config = ConfigDict(...)

Validate Input

model_validate parses untrusted dicts and JSON into typed objects.

PURPOSE	COMMAND	RESULT
Validate a Python dict.	`User.model_validate({"id": 1, ...})`	
Validate a JSON string.	`User.model_validate_json(s)`	
Construct from keywords.	`User(id=1, name="ada")`	
Validate a list of items.	`TypeAdapter(list[User]).validate_python(rows)`	
Bad input never passes silently.	`User.model_validate({"id": "oops"})`	
Forbid coercion (strict).	`model_validate({"id": "1"},strict=True)`	

model_validate (dict) and model_validate_json (text) replace v1 parse_obj / parse_raw; TypeAdapter validates non-model types

Coerce & Constrain

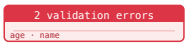
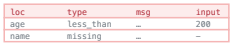
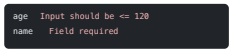
Field() adds bounds, defaults, and metadata to a type.

PURPOSE	COMMAND	RESULT
Coerce a string to the type.	`age:int... Account(age="30")`	
Bound a numeric field.	`age: int =Field(ge=0, le=120)`	
Constrain string length.	`name: str =Field(min_length=1, max_length=50)`	
Safe mutable default.	`tags: list[str] =Field( default_factory=list)`	
Regex via Annotated.	`Annotated[str,Field(pattern=r"^\d{5}$")]`	
Map external key names.	`Field(validation_alias= AliasChoices("user_id","userId"))`	

pydantic coerces sensible values by default; Field() layers ge/le, min_length/max_length, pattern, aliases, and safe defaults

Read a ValidationError

One exception lists every problem with loc, msg, and type.

PURPOSE	COMMAND	RESULT
Catch the failure.	`exceptValidationErroras e:`	
Count the problems.	`e.error_count()`	
Read structured error rows.	`e.errors()`	
Pinpoint the bad field.	`e.errors()[0]["loc"]`	
Human-readable summary.	`print(e)`	
JSON-shaped error report.	`e.json()`	

pydantic collects every problem into one ValidationError; each error dict carries loc, type, msg, and input

Field & Model Validators

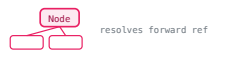
Hook custom rules per field or across the whole model.

PURPOSE	COMMAND	RESULT
Normalize one field.	`@field_validator("name") → strip().lower()`	
Validate after coercion.	`@field_validator("age", mode="after")`	
Cross-field rule.	`@model_validator(mode="after") → self`	
Pre-process raw input.	`@model_validator(mode="before")`	
Derive a read-only field.	`@computed_field`over a @property	
Raise to reject a value.	`raise ValueError("bad")`	

`@field_validator` (wrap a classmethod) runs per field; `@model_validator` runs across the instance; raise ValueError to reject

Nest & Reuse Models

Models compose: nest them, list them, union them.

PURPOSE	COMMAND	RESULT
Nest one model in another.	`addresses: list[Address]`	
Validate nested dicts deeply.	`Person.model_validate({"addresses":[…]})`	
Accept one of several types.	`value:int | str`	
Tagged (discriminated) union.	`Field(discriminator="kind") Cat | Dog`	
Reuse fields via inheritance.	`class Admin(User): level: int`	
Recursive / self-reference.	`children: list["Node"] →model_rebuild()`	

nested dicts validate all the way down into real sub-objects; a discriminator routes tagged unions without guessing

Serialize Out

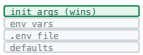
model_dump for a dict, model_dump_json for a JSON string.

PURPOSE	COMMAND	RESULT
Dump to a Python dict.	`u.model_dump()`	
Dump to a JSON string.	`u.model_dump_json()`	
JSON-safe values in a dict.	`u.model_dump(mode="json")`	
Drop empty / unset fields.	`u.model_dump(exclude_none=True)`	
Hide a field from output.	`secret: str =Field(exclude=True)`	
Emit the JSON Schema.	`User.model_json_schema()`	

model_dump keeps rich Python types; model_dump_json renders them as strings; mode="json" gives JSON-safe values in a dict

Config with pydantic-settings

BaseSettings loads typed config from env vars and .env files.

PURPOSE	COMMAND	RESULT
Install the separate package.	`uv add pydantic-settings`	
Declare a typed settings class.	`class Settings(BaseSettings):` `  port:int = 8000`	
Pull values from the environment.	`Settings() ← APP_PORT=9000`	
Namespace and load a .env file.	`SettingsConfigDict(env_prefix="APP_",` `  env_file=".env")`	
Source precedence.	`init args > env vars > .env > defaults`	
Validate config like any model.	`APP_PORT=oops →ValidationError`	

BaseSettings (a separate package in v2) reads typed config from env and .env, coercing and validating it like any BaseModel