Detailed Summary – mypy

1. What Mypy Actually Does

Mypy is a **static type checker**.
It **does not run** your code. It only **reads** and **analyzes** it.

Think of it as checking your logic before Python executes anything.

---

2. Internal Workflow of Mypy

a) Parsing

Converts your `.py` file into an AST (Abstract Syntax Tree - tree of your code).

```
x = 10
```

Mypy sees:

- a variable assignment
- variable name `x`, value type: `int`

---

b) Semantic Analysis

Resolves what each name refers to—local/global/function/class.

```
def f():
    x = 10
    return x

y = f()
```

Mypy connects:

- `x` in `f()` is local
- `y` receives an `int` (because f returns int)

---

c) Import Following

Looks inside every imported module so types match.

If you write:

```
from utils import get_user

user = get_user()
```

Mypy will open `utils.py` or its `.pyi` interface/stub to check the return type of `get_user()`.

Example:

If you have `my_module.py` with code, `my_module.pyi` might look like:

```
from typing import List, Dict

def process_data(data: List[int]) -> Dict[str, int]:
    ... # No implementation here, just the type signature
```

Tools then use this `pyi` file to understand `process_data` when analyzing your code.

---

d) Type Checking (Core)

Matches your hints vs how you actually use data.

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

add(10, "20")    # ✗ Mypy error
```

---

3. Type Inference

Mypy guesses types when you don't explicitly declare them.

```
def get_age() -> int:
    return 25

age = get_age()  # inferred: int

age.upper()      # ✗ Error: "upper" not on int
```

---

4. Third-party Libraries (Stub Files)

Many libraries don't have type hints.
Mypy uses **`.pyi` stub files** that contain only signatures.

Typeshed provides stubs for most standard libraries:

```
# requests.pyi (example)
def get(url: str) -> Response: ...
```

Mypy reads this instead of the real implementation.

---

5. Static vs Runtime Checking

Feature	Mypy (Static)	Runtime Checkers (Pydantic, DRF, etc.)
When	Before execution	During execution
Goal	Catch coding mistakes	Validate real data
Speed	Zero runtime cost	Small runtime cost
If wrong	Code still runs	Raises exception

---

6. Running Mypy

Install

```
pip install mypy
```

Example Code

```
def insert_patient_data(name: str, age: int) -> None:
    print(name, age)

insert_patient_data("susamay", "twenty")  # ❌ Error
```

Run

```
mypy patient_check.py
```

Output

```
error: Argument 2 has incompatible type "str"; expected "int"
```

---

7. Checking Folders

```
mypy src/
```

Using Strict Mode

```
mypy --strict .
```

---

8. Editor Integration

- **VS Code:** Install "Mypy" extension → highlights errors live
- **PyCharm:** built-in "static analysis" (mypy-like)

---

9. Configuring Mypy (Recommended)

Option A: `pyproject.toml` (Modern)

```
[tool.mypy]
python_version = "3.10"
warn_return_any = true
disallow_untyped_defs = true
check_untyped_defs = true
warn_unused_configs = true
```

Now simply run:

```
mypy .
```

Option B: `mypy.ini` (Traditional)

```
[mypy]
python_version = 3.10
show_error_codes = True
disallow_untyped_defs = True
strict_optional = True
warn_unused_ignores = True
ignore_missing_imports = True
```

---

10. Silencing Mypy (Optional)

```
insert_patient_data("susamay", "twenty")  # type: ignore[arg-type]
```

Use sparingly.

---

11. A "Mypy-Proof" Real Example

```
from typing import Optional
```

```
def insert_patient_data(name: str, age: int, email: Optional[str] = None) ->
bool:
    if age < 0:
        return False

    print(name, age)
    if email:
        print(email)
    return True

# Errors:
insert_patient_data("Susamay", "twenty")  # age: wrong type
insert_patient_data("Susamay", 20, 12345) # email: wrong type
```

---

12. Why Use Mypy?

Imagine you change a function's return type from `int` → `float` across **50 files**.

Mypy instantly shows every place that breaks, preventing hidden runtime bugs.

- **Dictionary = actual data you work with**
- **JSON = text version of data for sending/storing**

Feature	Dictionary	JSON
Type	Python object	String
Usage	Code runtime	Communication/storage
Comments allowed	Yes	❌ No
Keys	Can be any hashable type	Must be strings
Booleans	`True/False`	`true/false` (lowercase)
Null value	`None`	`null`
Trailing comma	Allowed	❌ Not allowed
Functions	Allowed	❌ Not allowed

```
Data_dict = {
  "name": "Sam",
  "age": 25,
  "active": True
}
```

json_data = '{"name": "Sam", "age": 25, "active": true}'

JSON → dict

```
import json
d = json.loads('{"a": 1}')
```

dict → JSON

```
import json
s = json.dumps({"a": 1})       # dump to json
```

Task	Pydantic v2 Syntax
Create model	`class M(BaseModel): ...`
Field constraints	`Field(..., ge=0)`
Validate raw dict	`model_validate(data)`
Validate JSON	`model_validate_json(json_str)`
Custom validator	`@field_validator("field")`
Config	`model_config = ConfigDict(...)`
Dump	`model_dump()`

Feature	**Field(...)**	**@field_validator**
Simplicity	High (one line)	Medium (requires a method)
Speed	Very Fast (Core Rust logic)	Slower (Python function call)
Logic Capability	Simple ranges/regex only	Anything you can write in Python
Metadata	Can add descriptions/titles	Logic only

Goal	Parameter	Example
Fixed Default	`default`	`Field(default="Unknown")`
Dynamic Default	`default_factory`	`Field(default_factory=list)`
No Default	`default=...`	`Field(default=...)`

name: Annotated[str, Field(default="Anonymous", # <--- Default value set here max_length=50, etc)]

OR

name: Annotated[str, Field(max_length=50)] = "Anonymous"

Truth Table

Default controls presence · Optional controls `None`

Annotation	Default	Required?	Can be `None`?	Example Valid Values
`str`	✗	☑	✗	`"abc"`
`str = "x"`	☑	✗	✗	`"abc"`
`str = None` (as None is not string, so use optional)	☑	✗	✗ (! type mismatch)	✗
`Optional[str]`	✗	☑	☑	`"abc"`, `None`
`Optional[str] = None`	☑	✗	☑	`"abc"`, `None`
`Optional[str] = "x"`	☑	✗	☑	`"abc"`, `"x"`, `None`
`bool`	✗	☑	✗	`True`, `False`
`bool = False`	☑	✗	✗	`True`, `False`
`Optional[bool]`	✗	☑	☑	`True`, `False`, `None`
`Optional[bool] = None`	☑	✗	☑	`True`, `False`, `None`
`List[str]`	✗	☑	✗	`["a", "b"]`
`List[str] = Field(default_factory=list)`	☑	✗	✗	`[]`, `["a"]`
`Optional[List[str]]`	✗	☑	☑	`["a"]`, `None`
`Optional[List[str]] = None`	☑	✗	☑	`["a"]`, `None`

...

allergies: Optional[List[str]] = None

allergies: Optional[List[str]] = [False, 36.0, None, "Dust"]

allergies: Optional[List[str]] = Field(default=[False, 36.0, None, "Dust"], validate_default=True)

Code	Validated?
Input data	☑ Always
Default value	❌ Not by default
Default + `validate_default=True`	☑ Yes
Pydantic v2 defaults	☑ Always

🧠 Core rules to remember

1. **`Optional[List[str]]`**
   - Means: the **field value** can be `None` **or** a `List[str]`
   - It does **not** validate anything by itself
2. **Pydantic v1 validates ONLY input data**
   - Defaults are **trusted**
   - Defaults are **not validated** unless explicitly enabled

Practically (runtime):

`bool = None` **is the same as** `Optional[bool]`

For static typing:

`Optional[bool]` is the correct and cleaner way.

So use –

married: Optional[bool] = None OR newer

married: bool | None = None

```
allergies: Optional[List[str]] = Field(default_factory=list)  # use this not
Field(default=[]), to avoid mutable default value passed, so can infect other
instances
```

Even cleaner for Python 3.9+

you don't even need to import `List` from `typing` anymore, use lower list

allergies: Annotated[list[str], Field(default_factory=list, max_length=5)]

Old (typing import)	New (Built-in)	Example
`List[str]`	**`list[str]`**	`allergies: list[str]`
`Dict[str, int]`	**`dict[str, int]`**	`scores: dict[str, int]`
`Optional[str]`	**`str | None`**	`name: str | None`

1. The Old Way (Pre-3.12)

You simply assigned a type to a variable. This works, but tools like IDEs sometimes struggle to distinguish between a "type" and a regular variable.

Python

```
from typing import Annotated
from pydantic import Field

# Simple assignment
Username = Annotated[str, Field(min_length=3, max_length=20)]

class User(BaseModel):
    name: Username  # Works fine
```

2. The New Way (Python 3.12+)

The `type` keyword explicitly tells Python: "This is a type definition, not a variable." It also supports **Generics** much more easily.

Python

```
from typing import Annotated
from pydantic import Field, BaseModel

# Explicit Type Alias
type Username = Annotated[str, Field(min_length=3, max_length=20)]

# Explicit Generic Type Alias (Very powerful!)
type MaxList[T] = Annotated[list[T], Field(max_length=5)]

class Patient(BaseModel):
    # This says: A list of strings, but capped at 5
    allergies: MaxList[str]
```

```
    # This says: A list of integers, but capped at 5
    recent_readings: MaxList[int]
```

- **`T`**: A placeholder for any type. (Conventional Generic, T = Type)
- **`MaxList[T]`**: A "Generic Alias."
- **`MaxList[str]`**: A "Concrete Type" (where the placeholder has been filled).

Comparison: With vs. Without `T`

Without Generics (The repetitive way): You would have to define a new alias for every single data type you want to limit.

Python

```
type MaxStringList = Annotated[list[str], Field(max_length=5)]
type MaxIntList = Annotated[list[int], Field(max_length=5)]
type MaxFloatList = Annotated[list[float], Field(max_length=5)]
```

With Generics (The `T` way): You define the logic **once**, and it adapts to whatever you give it.

Python

```
type MaxList[T] = Annotated[list[T], Field(max_length=5)]

class Survey(BaseModel):
    comments: MaxList[str]  # T becomes str
    scores: MaxList[int]    # T becomes int
```

A Quick Summary Checklist for you:

- **Imports:** Use `from typing import Annotated`.
- **Built-ins:** Use `list[]`, `dict[]`, and `| None`.
- **Aliases:** Use `type MyName = Annotated[...]`.
- **Fields:** Use `field_name: MyName`.
- **Lists:** Always use `default_factory=list` to stay safe from shared memory bugs.

Default auto-corersions –

Source -> Target	Works?
`"25"` → `int`	✔
`"12.5"` → `float`	✔
`123` → `str`	✔
`"true"/int/"yes"/"y"/"on"/0.1/` Non-empty Collections → `bool`	✔
Single value → `List[...]`	✔
JSON string → `Dict`	✔
str → datetime/date/time	✔
str → Enum	✔
dict → BaseModel	✔

🚫 **Things Pydantic does NOT auto-convert**

❌ **"1,2,3" → List[int]**

(Pydantic does not split CSV-type strings)

❌ **"abc" → Dict**

Unless it's valid JSON.

❌ **Completely invalid formats**

(e.g. "maybe" → bool)

Case	Model fields	Input	Result
`Inner({"x": "10"})`	1 field	dict	✔ Pydantic treats dict as full input
`Patient({...})`	multiple fields	dict	❌ treated as positional arg for first field

❌ **Never pass dict directly as positional unless model has exactly one field.**

---

name: str = Field(max_length=50) OR simple name: str # pydantic v1 style still can be used

OR below Annotated - recommended for pydantic v2 clearly separates the type from the constraints and works better with tools like: FastAPI, mypy

Annotated [data_type, Field(default, constraints, *metadatas like title, description, examples)] = default-value-prioritized

- `...` = “repeat this type forever”.
- `tuple[int, ...]` = any-length tuple of ints.

```
tuple[int, str, float ...] = first three elements: int, str, float, remaining elements: float
```

```
list[int | str]       # list of ints OR strings
```

```
def f(x: Any):
  x.nonexistent_method()  # mypy: OK
def g(x: object):
  x.nonexistent_method()  # mypy: ERROR
```

INSTEAD OF BELOW

`@field_validator('name', mode='after')`: Runs **after** Pydantic parses it (this is the default).

Another Way: The `Annotated` way

If you find yourself using `transform_name` in many different models, you can move the logic out of the class entirely:

Python

```
from pydantic import AfterValidator

# Define the logic once
def to_upper(v: str) -> str:
    return v.upper()

# Create a reusable type
UpperCaseStr = Annotated[str, AfterValidator(to_upper)]

class User(BaseModel):
    name: UpperCaseStr # Now 'name' is automatically uppercased!
```
