

GDG Quito · ESPE

Typed Python

Types, good practices and where Python stands *today*
against other languages.



Hi, I'm

Sebastian Alvarez

Fullstack Software Engineer
- **GDG Quito Organizer**

I build Python backends (FastAPI, Django), cloud infrastructure and web products. Typing Python is the first thing I set up in every project.

• GDG Quito

• Quito, Ecuador

@listerineh

Agenda

- **01** Type systems
- **02** Type hints in Python
- **03** Good practices and tools
- **04** Python today against other languages
- **05** Practical examples

01

Type systems

Two different questions: *when* types are checked and *how strict* they are.

Kinds of typing



Static

Types are checked before running, by the compiler or a type checker.



Dynamic

A variable's type is decided while the program runs.



Strong

Restricts which operations each type allows.



Weak

Converts between types implicitly so the operation works.

Two independent axes

When is it checked?

- **Static:** before running
- **Dynamic:** while running
- **Gradual:** optional types on top of a dynamic language

How strict is it?

- **Strong:** no surprise conversions
- **Weak:** the language converts for you
- It's a spectrum, not a yes or no

Where each language falls

- **Rust, Java, Go:** static and strong
- **C and C++:** static, but with implicit conversions: weaker
- **TypeScript:** gradual, with types checked at compile time
- **Python and Ruby:** dynamic and strong
- **JavaScript and PHP:** dynamic and weak

Python: dynamic and strong

PYTHON

```
x = 1          # x is an int
x = "one"      # now it's a str: dynamic

"1" + 1
# TypeError: can only concatenate str (not "int") to str
```

In JavaScript, `"1" + 1` gives `"11"`: that's weak typing. Python would rather fail.

02

Type hints in Python

Optional annotations that tools read.

Type hints **don't change**
how **your code runs.**

Python is still dynamic: types are for tools and for people.

The basics

PYTHON

```
def greet(name: str, times: int = 1) -> str:  
    return f"Hi, {name}! " * times
```

```
ages: dict[str, int] = {"ana": 31}  
ids: list[int] = [1, 2, 3]
```

```
def find(id: int) -> User | None:  
    ...
```

Since 3.9, `list[int]` and `dict[str, int]`; since 3.10, `X | None` instead of `Optional[X]`.

A decade of types

- **3.5 (2015):** PEP 484, type hints and the `typing` module are born
- **3.9 and 3.10:** `list[int]` and `int | str`, no imports needed
- **3.11:** `Self`, `LiteralString`, `Required` and `NotRequired`
- **3.12:** dedicated generics syntax: `class Box[T]` and `type`
- **3.13:** `TypeIs`, `ReadOnly` and defaults for type parameters
- **3.14:** deferred evaluation of annotations (PEP 649)
- **3.15 (2026):** closed `TypedDict`s (PEP 728) and `TypeForm` (PEP 747)

Generics: then and now

PYTHON

```
# Then (3.5+)
T = TypeVar("T", bound=str)
class Greeting(Generic[T]):
    def say(self, text: T) -> T: ...

# Now (3.12+)
class Greeting[T: str]:
    def say(self, text: T) -> T: ...
```

PEP 695 syntax needs neither `TypeVar` nor `Generic`: the parameter lives on the class.

Protocol: duck typing with types

PYTHON

```
class Notifier(Protocol):
    def send(self, message: str) -> None: ...

class Email: # doesn't inherit from Notifier
    def send(self, message: str) -> None: ...

def notify(n: Notifier) -> None: ...
notify(Email()) # valid: it has the right shape
```

Structural typing: if it has the right shape, it fits. Good old *duck typing*, checked.

Types that model the domain

PYTHON

```
UserId = NewType("UserId", int)
Status = Literal["active", "suspended"]

class User(TypedDict):
    id: UserId
    name: str
    status: Status
```

Business rules live in the types: a typo in `"suspended"` is caught before production.

Types at runtime: Pydantic

PYTHON

```
from pydantic import BaseModel, PositiveInt

class SignUp(BaseModel):
    name: str
    age: PositiveInt

SignUp(name="Ana", age="31") # age=31, converted
SignUp(name="Ana", age=-4)  # ValidationError
```

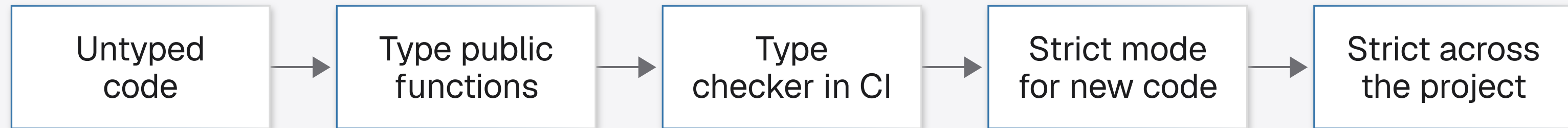
The same type hints, validated while running: that's how FastAPI and much of the modern ecosystem work.

03

Good practices

How to type without fighting the types.

Gradual adoption



You don't need to type everything on day one: every step already pays off.

Good practices

- Type the **boundaries** first: public functions, APIs and data models
- Use `list[int]` and `X | None`; `List` and `Optional` belong to old code
- Avoid `Any`: prefer `object`, a `Protocol` or a generic
- Validate external data at runtime and trust the types inside
- A type checker in CI, with strict mode on new code
- Every `# type: ignore` carries the error code and a reason

Tools



mypy

The reference type checker, since 2012.



Pyright

From Microsoft; the engine behind Pylance in VS Code.



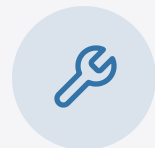
ty

From Astral (uv, Ruff), written in Rust. In beta.



Pyrefly

From Meta, also in Rust and built for monorepos.



Ruff

Lint and formatter; its ANN rules require annotations.



Pydantic

Runtime validation from type hints.

Configuration in pyproject.toml

TOML

```
[tool.mypy]
strict = true
[[tool.mypy.overrides]]
module = "legacy.*"
ignore_errors = true

[tool.ruff.lint]
extend-select = ["ANN", "UP"]
```

Strict for everything new; legacy code joins little by little. `UP` upgrades `Optional[X]` to `X | None`.

Pros and cons of typing Python

Pros

- Documents the code without extra comments
- Autocomplete and safe refactors in the IDE
- Errors found before production
- Clearer architecture and teams that understand each other
- Better context for AI coding assistants

Cons

- More code and a learning curve
- On their own they validate nothing at runtime
- Untyped libraries or incomplete stubs
- They can slow down quick scripts and prototypes

04

Python today

Where it stands against other languages.

Python in numbers

#1

in the TIOBE index

19.23%

TIOBE share

58%

of developers use it

On GitHub, TypeScript overtook Python and JavaScript in 2025.

Typed languages are gaining ground in the AI era.

Python against other languages

- **TypeScript:** the same gradual model, but the checker ships with the compiler; in Python it's a separate tool
- **Go:** simple static types, a single binary and native concurrency; great for services and infrastructure
- **Rust:** the strictest type system; Python uses it under the hood (Ruff, uv, ty, Pydantic's core)
- **Java and Kotlin:** mature static typing in enterprises; Python wins in data, AI and prototypes
- **Python:** the AI and data ecosystem, with a type system that now covers almost everything
- listerinh.dev/slides

What's new in Python

- **3.13:** a *free-threaded* build without the GIL and a JIT compiler, both experimental
- **3.14:** officially supported free-threading, deferred annotations and *t-strings*
- **3.15:** deferred imports with `lazy import`, `frozendict` and UTF-8 by default
- **uv:** installs Python and dependencies in seconds
- **ty** and **Pyrefly:** Rust type checkers for large projects

05

Practical examples

From untyped code to `mypy --strict`.

Example: before

PYTHON

```
def total(items, discount=None):  
    subtotal = sum(i["price"] * i["qty"] for i in items)  
    return subtotal * (1 - (discount or 0))
```

What is `items`? Is `discount` a percentage or an amount? The code doesn't say.

Example: after

PYTHON

```
@dataclass(frozen=True)
class Item:
    price: Decimal
    qty: int

def total(items: list[Item], discount: Decimal = Decimal(0)) -> Decimal:
    subtotal = sum((i.price * i.qty for i in items), Decimal(0))
    return subtotal * (1 - discount)
```

The checker now knows what an `Item` is, and money stops using `float`.

Questions? Let's connect

The slides stay at listerineh.dev/slides.

linkedin.com/in/listerineh

listerineh.dev