Start coding typed Python

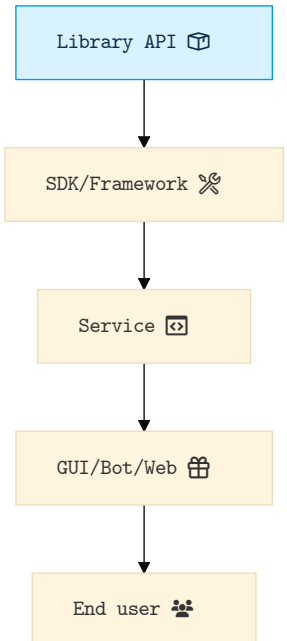
Anton Grishin (@alchemmist)

EOSP #L2, winter 2026. CU x CPM

Table of contents

Interface levels lib

From low-level primitives to user-facing products.



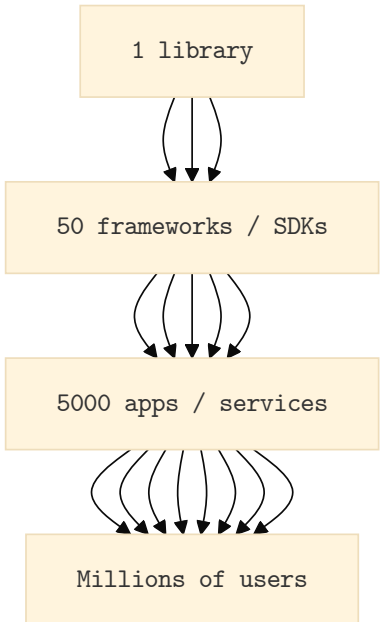
- Library it's a lowest interface.
- Library defines stable primitives and contracts.
- Upper layers can change without rewriting the core.
- The lower the layer, the wider the potential reuse.
- Backward compatibility pressure is highest at this layer.

Most popular examples `lib`

Library	What it is
`libc`	Standard C runtime library. Present in almost every C/Unix program via core primitives (`printf`, `malloc`, `memcpy`). Stable contracts underpin entire OS ecosystems.
`OpenSSL`	Cryptography and TLS implementation library. Used by browsers, curl, git, Python requests. Security fixes or bugs immediately affect huge parts of the internet.
`zlib`	General-purpose compression library. Powers gzip/deflate in HTTP, ZIP archives, PNG files. Small API, massive invisible reach.
`SQLite`	Embedded relational database engine. Shipped inside browsers, mobile apps, IDEs. Adoption spreads through embedding, not direct usage.

Why library gives widest reach lib

The adoption funnel expands through integrations.



- Low-level API is reused by many teams and products
- Consumers may not know your name, but still use your code daily
- This is strategic leverage: one core, many channels
- Library quality decisions compound across every dependent product

What makes a library truly reusable `lib`

Reach appears only if the interface is engineered well.

- Stable API contracts (minimal breaking changes)
- Clear boundaries and composable primitives
- Predictable error model and explicit return values
- Strong typing and tests at public boundaries
- Good docs and examples for quick integration
- Versioning discipline and changelog quality
- Backwards compatibility as default mindset
- Performance and security as part of API design

Typed Python `typing`

Why and how to write typed Python?

Dynamic vs static typing

typing

1. Where we will get error?
2. What's better?

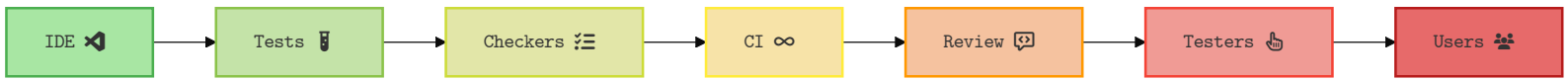
C — error before run:

```c
int discount(int total_cents, int percent) {
    return total_cents - total_cents * percent / 100;
}

discount("1000", 15);
// ⤴ compile-time error
```

Python (*without checks*) — error during execution path:

```python
def discount(total_cents, percent):
    return total_cents - total_cents * percent / 100
    # ⤴ runtime error

price = discount("1000", 15)
```

Error detection levels



What's better C or Python?

- C.
- Because we see error at early step: compilation.
- But Python may be get closer to that with typing and analyzers.

Typing in Python is for code quality `typing`

- Not for Python runtime speed by itself (or not 🤔).
- For early bug detection before production.
- For clearer interfaces and cleaner architecture.
- For better IDE/PDE hints and safer refactoring.
- For reducing trivial test cases about primitive mismatches.
- Recommendation: combine strict typing with focused business tests.

Primitive annotations `typing`

Start from boundary functions where wrong types enter the system.

main.py

```python
def compute_fee(amount: float, fee_rate: float) -> float:
    if fee_rate < 0:
        raise ValueError("fee_rate must be non-negative")
    return amount * float(str(fee_rate))
```

- Input types after `:`
- Return type after `->`
- Common primitives: `str`, `int`, `float`, `bool`, `bytes`
- Recommendation: for money, prefer `Decimal` over `float`
- Recommendation: annotate return type even when it seems obvious

Union types `typing`

Use unions when multiple input types are truly valid.

```python
import json

def parse_payload(raw: str | bytes) -> dict[str, object]:
    if isinstance(raw, bytes):
        raw = raw.decode("utf-8")
    return json.loads(raw)
```

```python
def parse_positive_int(value: str) -> int | None:
    if not value.isdigit() or int(value) <= 0:
        return None
    return int(value)
```

- Good: `int | None` for optional result
- Risky: broad return unions like `int | str | dict`
- If union grows too much, redesign API
- Recommendation: keep union small and semantically clear

Typing collections `typing`

Specify container element types.

```python
def top_repo(repos: list[tuple[str, int]]) -> tuple[str, int] |
    if not repos:
        return None
    return max(repos, key=lambda r: r[1])

def avg_review_time(hours: list[float]) -> float:
    return sum(hours) / len(hours) if hours else 0.0

def total_prs(by_user: dict[str, int]) -> int:
    return sum(by_user.values())
```

- Avoid generic `list` / `dict` without params
- Type details improve both safety and readability
- Recommendation: express semantic meaning in nested collection types
- Practice note: most bugs in metrics code happen in data shape assumptions

`Mapping` vs `MutableMapping`

typing

Signal read-only vs mutable behavior.

```python
from collections.abc import Mapping, MutableMapping

def read_config(cfg: Mapping[str, str]) -> str:
    return cfg["DATABASE_URL"]

def patch_config(cfg: MutableMapping[str, str]) -> None:
    cfg["DEBUG"] = "true"
    cfg.setdefault("LOG_LEVEL", "info")
```

- If function only reads: prefer `Mapping`
- If function mutates: require `MutableMapping`
- Recommendation: narrower input contract makes side effects explicit
- Practice note: this small choice improves code review quality a lot

`NamedTuple` typing

Immutable tuple with named fields.

```python
from typing import NamedTuple

class User(NamedTuple):
    id: int
    username: str
    merged_prs: int
```

```python
def print_user(user: User) -> str:
    return f"{user.username} ({user.id}) -> {user.merged_prs} me
```

- Good for compact immutable records
- If mutability is needed, prefer `dataclass`
- Recommendation: use for small query result rows and snapshots

`TypedDict` typing

Typed shape for dictionary-like JSON data.

```python
from typing import NotRequired, TypedDict

class RepoPayload(TypedDict):
    id: int
    name: str
    private: bool
    language: NotRequired[str | None]
```

```python
def from_payload(data: RepoPayload) -> str:
    lang = data.get("language") or "unknown"
    return f"{data['name']} ({lang})"
```

- Great for external payload validation at static level
- Can declare optional keys explicitly
- Recommendation: use `TypedDict` at transport layer, convert to domain model next
- Practice note: keeps API chaos away from business logic

`dataclass` typing

Readable domain objects with defaults and generated methods.

```python
from dataclasses import dataclass

@dataclass
class User:
    id: int
    username: str
    email: str | None = None
    score: int = 0

    def activate_bonus(self, bonus: int) -> None:
        if bonus < 0:
            raise ValueError("bonus must be >= 0")
        self.score += bonus
```

- Mutable by default, so protect invariants in methods
- Use `@dataclass(frozen=True)` for immutable entities
- Recommendation: prefer dataclass for DTO/domain state containers
- Practice note: once behavior grows heavy, switch to regular class

Enum `typing`

Closed set of allowed domain values.

```python
from enum import Enum

class Status(Enum):
    NEW = "new"
    IN_REVIEW = "in_review"
    DONE = "done"
    FAILED = "failed"
```

```python
def can_merge(status: Status) -> bool:
    return status in {Status.IN_REVIEW, Status.DONE}
```

- Avoids random string values across codebase
- Recommendation: map external strings to enum right after parsing
- Practice note: enum is cheap and prevents many typo-level bugs

Custom classes in annotations

typing

Your own classes are first-class types too.

```python
class User:
    id: int
    username: str
    email: str
    friends: list["User"]

class Team:
    members: list[User]
```

```python
def add_friend(team: Team, user: User, friend: User) -> None:
    if user not in team.members or friend not in team.members:
        raise ValueError("both users must be in team")
    user.friends.append(friend)
```

- Domain classes in signatures make intent obvious
- Recommendation: avoid `dict[str, Any]` in core logic signatures
- When signatures use real entities, architecture boundaries become clearer
- Practice note: this reduces accidental cross-layer coupling over time

Abstract collection classes `typing`

Use behavior-based input contracts.

```python
from collections.abc import Iterable

def total_commits(per_week: Iterable[int]) -> int:
    return sum(per_week)

def top3(values: Iterable[int]) -> list[int]:
    return sorted(values, reverse=True)[:3]
```

- `Iterable`, `Sequence`, `Mapping` and others come from `collections.abc`
- Prefer importing these ABCs from `collections.abc` directly
- Recommendation: annotate by behavior required by the algorithm
- Practice note: this reduces accidental over-coupling to `list`

`Sequence` vs `Iterable` typing

The difference is operation guarantees.

```python
from collections.abc import Iterable, Sequence

def avg_any(values: Iterable[int]) -> float:
    total, count = 0, 0
    for v in values:
        total += v
        count += 1
    return total / count if count else 0.0
```

```python
def median3(values: Sequence[int]) -> int:
    return sorted(values)[len(values) // 2]
```

- `Iterable`: can loop
- `Sequence`: can loop + index + len
- Recommendation: do not require `Sequence` if one pass is enough
- Practice note: this keeps APIs compatible with generators/streams

Interface | Abstract class | Protocol `typing`

Three related tools, different coupling.

- **Interface**: only abstract methods
- **Abstract class**: abstract + implemented methods
- **Protocol**: implicit interface

```python
from collections.abc import Iterable

class UserRepo():
    def save(self, login: str) -> None:
    def all(self) -> Iterable[str]: ...

class SqliteRepo(UserRepo):
    def all(self) -> list[str]:
        self._db.get_logins()
    def save(self, login: str) -> None:
        self._db.save(login)

def export(repo: UserRepo) -> list[str]:
    return [u.upper() for u in repo.all()]

export(SqliteRepo()) # no problems
```

```python
from typing import Protocol

class UserRepo(Protocol):
    def save(self, login: str) -> None: ...
    def all(self) -> Iterable[str]: ...

class SqliteRepo:
    def all(self) -> list[str]:
        self._db.get_logins()
    def save(self, login: str) -> None:
        self._db.save(login)

def export(repo: UserRepo) -> list[str]:
    return [u.upper() for u in repo.all()]

export(SqliteRepo()) # no problems
```

`Callable` typing

Type callbacks and higher-order functions.

```python
from collections.abc import Callable

def retry(
    action: Callable[[], str],
    on_error: Callable[[Exception], None],
    attempts: int = 3,
) -> str:
    for _ in range(attempts):
        try:
            return action()
        except Exception as e:
            on_error(e)
    raise RuntimeError("all retries failed")
```

- Prefer explicit argument and return signatures
- `Callable[..., T]` is possible but weaker
- Recommendation: type callback signatures early in async/event code
- Practice note: production bugs often hide in callback contracts

Generics with `TypeVar` typing

Preserve relation between input and output types.

```python
from collections.abc import Mapping
from typing import TypeVar

T = TypeVar("T")

def pick_or_fail(items: Mapping[str, T], key: str) -> T:
    if key not in items:
        raise KeyError(key)
    return items[key]
```

- If mapping value is `Repo`, return type is inferred as `Repo`
- Without generics this collapses to `Any` and safety is lost
- Recommendation: use generics for reusable repository/service helpers

Bounded generics `typing`

Restrict allowed types with `bound=`.

```python
from collections.abc import Iterable
from datetime import datetime
from typing import TypeVar

TimestampT = TypeVar("TimestampT", bound=datetime)

def latest(items: Iterable[TimestampT]) -> TimestampT:
    return max(items)
```

- Useful when algorithm needs specific capabilities
- Bound keeps generic reusable but not unconstrained
- Recommendation: start concrete, generalize with `bound` later
- Bounded generics are a good compromise between flexibility and safety

`Literal` typing

Constrain exact accepted values.

```python
from typing import Literal

def export_report(
    format: Literal["csv", "json"],
    mode: Literal["full", "compact"] = "compact",
) -> bytes:
    ...
```

- Typos like `"jsno"` are caught statically
- Excellent for modes, statuses, fixed strategy names
- Recommendation: use `Literal` when values are tiny and closed
- If variants start growing, move from `Literal` to `Enum`

Static analyzers `typing`

Tool	Main strength	Typical tradeoff
`mypy`	mature ecosystem and plugins	needs strictness tuning
`pyright`	fast, strong diagnostics	less plugin flexibility
`pyrefly`	very fast Rust implementation	younger tool, behavior still moving
`ty`	modern Rust architecture (beta)	pre-release maturity

- Recommendation: pick one tool as CI gate, keep others optional
- Practice note: mixed analyzer requirements usually confuse teams
- In team projects, consistency of diagnostics is more important than tool preference

Install and run analyzers with `uv` `typing`

```
uv tool install mypy
uv tool install pyright
uv tool install pyrefly
uv tool install ty
```

```
uvx mypy .
uvx pyright .
uvx pyrefly check
uvx ty check
```

- Recommendation: in CI prefer `uv run mypy src tests`
- Use `uvx` for local experiments and tool comparison
- Pin one analyzer version in CI to avoid random baseline drift
- Keep local command aliases in `Makefile`/scripts for team consistency

Analyzer demo: intentional typing bugs `typing`

```python
from dataclasses import dataclass
from typing import NewType

UserId = NewType("UserId", int)

@dataclass
class User:
    id: UserId
    email: str
    is_active: bool

def discount(total: int, percent: int) -> int:
    return total - total * (percent / 100)

def send_invoice(user: User, amount: int) -> str:
    if not user.is_active:
        return None
    return f"invoice for {user.email}: {amount}"

def main() -> None:
    user = User(id=42, email=123, is_active="yes")
    total = discount("1000", 10)
    send_invoice(user, "500")
```

`pyright` catches multiple issues before runtime:

- `float` returned where `int` expected
- `None` returned where `str` expected
- invalid `User` field types
- invalid function call argument types
- domain identifier misuse (`NewType` violation)

Recommendation:

- fix type contracts first, then clean call sites
- do not silence these errors with broad ignores

Stub files (`.pyi`) `typing`

Type code you cannot or do not want to edit.

`calc.py`

```python
def fetch_user(user_id):
    # legacy code we do not want to refactor now
    ...
```

`calc.pyi`

```python
def fetch_user(user_id: int) -> dict[str, str]: ...
```

- Recommendation: use stubs for generated/legacy/third-party code
- Keeps typing strict without touching risky implementation paths

TypeAlias

typing

Improve readability for complex or business-specific types.

```python
from typing import TypeAlias

UserId: TypeAlias = int
MetricName: TypeAlias = str
MetricsPayload: TypeAlias = dict[MetricName, int | float]
```

```python
type UserId = int
type MetricName = str
type MetricsPayload = dict[MetricName, int | float]
```

- Right block is Python 3.12+ syntax
- Recommendation: alias business terms to make signatures readable
- Practice note: this pays off immediately in reviews and docs

`TypeAlias` vs `NewType` `typing`

Same runtime family, different static guarantees.

```python
from typing import NewType, TypeAlias

UserId: TypeAlias = int
RepoId = NewType("RepoId", int)
```

```python
def get_user(user_id: UserId) -> None:
    ...

def get_repo(repo_id: RepoId) -> None:
    ...
```

- `TypeAlias` is only a synonym
- `NewType` protects domain IDs from accidental mix
- Recommendation: use `NewType` for IDs crossing module boundaries
- Practice note: prevents subtle bugs in service glue code

How to use typing correctly `typing`

- Types must fail on wrong code, otherwise they do not protect you
- Start small, increase strictness gradually
- Strict typing and strict tests work as two safety nets
- Recommendation: enforce strictness per module, not all at once
- Recommendation: avoid `Any` in new business logic without clear reason

Returning `None` explicitly typing

Always annotate return type, even if it is obvious from implementation.

```python
def log_metrics(metrics: dict[str, int]) -> None:
    for name, value in metrics.items():
        print(f"{name}: {value}")

tmp = log_metrics({"prs": 12}) + 1
```

```
error:
  Operator "+" not supported for types "None" and "Literal[1]"
```

- Without return annotation, intent is ambiguous for readers
- Recommendation: explicitly annotate all public function returns
- Practice note: this improves code navigation speed

Function input vs output `typing`

A function should be more explicit about concrete return type than about accepted input type.

- Inputs can be abstract (`Iterable`, `Mapping`, protocols)
- Outputs should usually be concrete (`list`, `dict`, `User`)
- Recommendation: be liberal in accepted input, strict in returned shape
- Practice note: explicit outputs reduce integration misunderstandings

Repository structure review `code`

Now map typing rules to our real project scaffold: `lib-demo`.

Live demo plan `live-demo`

Add one metric end-to-end: API payload -> typed entity -> metric function -> test -> `mypy`.

Live demo: step-by-step `live-demo`

1. Add typed payload model with `TypedDict`
2. Parse GitHub response to domain entities
3. Add metric function (example: `total_open_issues`)
4. Add unit test for metric
5. Configure and run `mypy`

```
uv sync
uv add --dev mypy
uv run mypy src tests
uv run pytest
```

Live demo: metric example `live-demo`

```python
from collections.abc import Iterable
from gh_contrib_demo.core.entities.issue import Issue


def total_open_issues(items: Iterable[Issue]) -> int:
    return sum(1 for _ in items)
```

```python
from gh_contrib_demo.core.entities.issue import Issue
from gh_contrib_demo.core.metrics import total_open_issues


def test_total_open_issues() -> None:
    issues = [Issue(id=1, number=10), Issue(id=2, number=11)]
    assert total_open_issues(issues) == 2
```

Live demo: strict `mypy` baseline `live-demo`

`pyproject.toml`

```
[tool.mypy]
python_version = "3.12"
strict = true
warn_return_any = true
warn_unused_ignores = true
```

- Start strict early in the library layer
- Fix root causes, avoid blanket `# type: ignore`

Why `uv` is a savior `uv`

`uv` is not just another installer. It is a consolidation point for Python workflows.

- Official docs position it as package + project manager written in Rust
- Goal: one fast tool instead of fragmented chains of utilities
- Important in education: less setup overhead, more engineering time

The dependency pain in Python uv

Historical pipeline (simplified):

```
manual installs / setup.py
  -> easy_install (setuptools era)
  -> pip + virtualenv
  -> pip-tools / pipx / twine
  -> conda, pyenv, poetry, rye, ...
```

- `easy_install` was a big step, but is now deprecated
- `pip` improved packaging workflows and requirements files
- Then ecosystem split into many specialized tools
- Result: powerful stack, but high cognitive load for beginners
- Team problem: everyone has different local setup ritual

Why `uv` in practice [uv]

From official docs, key capabilities:

- 10-100x faster than `pip` in many workflows
- single tool replacing `pip`, `pip-tools`, `pipx`, `poetry`, `pyenv`, `twine`, `virtualenv`, and more
- universal lockfile and project dependency management
- script support with inline metadata
- install/manage Python versions (`uv python ...`)
- pip-compatible interface (`uv pip ...`)

Why this matters in our course:

- one command vocabulary for all students
- reproducible environments in every homework/repo
- fewer "works on my machine" failures

```
# Python versions
uv python install 3.12
uv python pin 3.12

# project lifecycle
uv sync
uv add requests
uv add --dev mypy pytest ruff
uv lock
uv run pytest

# scripts and one-off tools
uv run script.py
uvx ruff check .

# migration path for pip users
uv pip compile requirements.in -o requirements.txt
uv pip sync requirements.txt
```

Summary `uv`

`uv` practical cheat sheet:

```
# Day 1 in a new repo
uv python install 3.12
uv python pin 3.12
uv sync
uv run pytest

# Add deps during feature work
uv add httpx
uv add --dev mypy ruff
uv lock

# One-off tools and scripts
uvx pyright .
uv run scripts/check_metrics.py
```

- Recommendation: standardize these commands in docs/Makefile
- Outcome: predictable onboarding and reproducible CI behavior

Homework

- Read `longreads/typing-python.en.md` fully
- Install `uv` and run checks in `lib-demo`
- Add one typed metric with tests in your own branch
- Run `mypy` and fix all reported errors

layout: end