Universal Python Architecture Guidelines

A distilled guide for writing well-structured Python code in any project.

Core Architectural Patterns

1. Layered Architecture

```
Presentation → Application → Domain → Infrastructure
```

Apply to:

- **Web apps: Routes → Services → Models → Database**
- **APIs: Endpoints → Business Logic → Entities → External APIs**
- **CLI tools: Commands → Workflows → Core Logic → File System**

Rule: Each layer only talks to the layer below it. Never skip layers.

---

2. Factory Pattern

python

```
class ClientFactory:
    @staticmethod
    def create(provider: str):
        if provider == "postgres":
            return PostgresClient()
        elif provider == "mongodb":

            return MongoClient()
```

Use when: You have multiple implementations of the same interface
Frameworks: Django (database backends), Flask (session interfaces)
Keep: All creation logic in one place, lazy imports for large dependencies

---

3. Mixin Pattern

python

```
class CacheMixin:
    def with_cache(self): ...

class LoggingMixin:
    def log(self): ...

class MyService(CacheMixin, LoggingMixin):
    # Gets caching and logging for free
```

Use for: Optional cross-cutting concerns (caching, logging, retry, validation)
Don't use for: Core business logic
Frameworks: Django class-based views use mixins heavily

---

4. Protocol-Based Interfaces (Duck Typing)

python

```
from typing import Protocol

class Configurable(Protocol):
    @property
    def timeout(self) -> int: ...
    @property
    def retries(self) -> int: ...
```

Use when: Defining "shapes" of data without forcing inheritance
Benefits: Works with any class that matches the signature
Python 3.8+: Use `Protocol` instead of abstract base classes when possible

---

5. Adapter Pattern

python

```
class LegacySystemAdapter:
    def __init__(self, old_system):
        self._system = old_system

    def new_method(self):
        return self._system.old_method_with_different_signature()
```

Use when: Translating between incompatible interfaces you can't change
Common in: Third-party API integrations, legacy system migrations
Frameworks: Django has adapters for different database engines

6. Repository Pattern

python

```python
class UserRepository:
    def get_by_id(self, user_id: int) -> User:
        # Fetch from database, return domain model

    def save(self, user: User) -> None:

        # Persist domain model
```

Use for: Abstracting data access
Returns: Domain models, never raw dictionaries or ORM objects
Frameworks: Works alongside Django ORM, SQLAlchemy

7. Mapper Pattern

python

```python
class APIMapper:
    @staticmethod
    def to_domain_model(api_response: dict) -> DomainModel:
        return DomainModel(
            id=api_response["external_id"],
            name=api_response["display_name"]

        )
```

Use for: Transforming external data to internal models
Keep: Separate from repositories (repositories fetch, mappers transform)
Benefits: Isolates external format changes

Configuration Management

Pattern

python

```python
@dataclass(frozen=True)
class Config:
    timeout: int = 30
    retries: int = 3
    @classmethod
    def from_env(cls) -> "Config": ...

    @classmethod
    def from_file(cls, path: str) -> "Config": ...

    def __post_init__(self):
        if self.timeout <= 0:
            raise ValueError("timeout must be positive")
```

Universal rules:

- ✅ Immutable (frozen dataclass)
- ✅ Multiple creation methods
- ✅ Validate in `__post_init__`
- ✅ Pass config explicitly (no globals)
- ❌ Never mutate after creation

Frameworks:

- Django: Override `settings.py` patterns with dataclasses
- FastAPI: Use Pydantic `BaseSettings`
- Flask: Replace `app.config` with typed config objects

Dependency Injection

Pattern

python

```
class Service:
    def __init__(
        self,
        repository: Repository,  # Injected
        logger: Logger | None = None  # Optional with default
    ):
        self.repository = repository
        self.logger = logger or get_default_logger()
```

Benefits:

- Easy to test (inject mocks)
- Explicit dependencies
- No hidden global state

Frameworks:

- Django: Use `django-injector` or pass dependencies to class methods
- FastAPI: Built-in with `Depends()`
- Flask: Use `flask-injector`

Error Handling Hierarchy

Pattern

python

```
class MyAppException(Exception):
    """Base exception for all app errors"""

class ValidationError(MyAppException):
    """Invalid input data"""

class NotFoundError(MyAppException):
    """Resource not found"""

class ExternalServiceError(MyAppException):

    """Third-party service failed"""
```

Rules:

- Create domain-specific base exception
- Specific exceptions inherit from it
- Include context (status codes, IDs)
- Never catch bare `Exception`

Usage:

python

```
try:
    result = service.get_user(user_id)
except NotFoundError:
    return 404
except ValidationError as e:
    return 400, {"error": str(e)}
except MyAppException:
    return 500  # Catch all other app errors
```

Validation at Boundaries

Principle

Validate data when it enters your system, not deep inside business logic.

Entry points:

- API endpoints
- CLI arguments
- File uploads
- Function parameters accepting external data

python

```
# ✅ Good: Validate at entry
def create_user(email: str, age: int) -> User:
    if not email or "@" not in email:
        raise ValidationError("Invalid email")
    if age < 0:
        raise ValidationError("Invalid age")
    # Now safe to use email and age

# ❌ Bad: Validate deep in logic
def save_to_database(user_data):
    # Database layer shouldn't validate business rules
```

Frameworks:

- Django: Use forms/serializers at view layer
- FastAPI: Use Pydantic models for request validation
- Flask: Validate in route handlers, not in business logic

Immutable Domain Models

Pattern

python

```python
@dataclass(frozen=True)
class User:
    id: int
    email: str
    created_at: datetime

    # Cannot be modified after creation
```

Benefits:

- Thread-safe
- Cacheable (can use as dictionary keys)
- Predictable (no hidden mutations)

When to use regular classes:

- Services (UserService, EmailService)
- Repositories
- Clients

When to use frozen dataclasses:

- Domain models (User, Order, Product)
- Configuration
- API responses

Resource Limiting

Pattern

python

```python
MAX_PAGE_SIZE = 100
MAX_UPLOAD_SIZE = 10 * 1024 * 1024  # 10MB
MAX_QUERY_RESULTS = 1000

def get_users(page_size: int = 20) -> list[User]:
    page_size = min(page_size, MAX_PAGE_SIZE)  # Cap at limit
    # Fetch and return
```

Apply to:

- Pagination (max results per page)
- File uploads (max file size)
- Database queries (max rows returned)
- API requests (timeout, max retries)

Why: Prevents resource exhaustion attacks

Logging Best Practices

Levels

python

```python
logger.debug("Cache key generated: {key}")   # Developer info
logger.info("User {user_id} logged in")      # Business events
logger.warning("Rate limit reached, retrying")  # Recoverable issues
logger.error("Payment failed: {error}")      # Failures

logger.critical("Database unreachable")      # System-level failures
```

Rules:

- ✅ Use structured logging: `logger.info("Action", extra={"user_id": 123})`
- ✅ Sanitize secrets before logging
- ✅ Log at boundaries (function entry/exit)
- ❌ Never log inside tight loops
- ❌ Never log raw exceptions without sanitizing

Sanitization:

python

```python
def sanitize_for_logging(text: str) -> str:
    text = re.sub(r"password=\S+", "password=[REDACTED]", text)
    text = re.sub(r"api_key=\S+", "api_key=[REDACTED]", text)

    return text
```

Retry Pattern with Backoff

Pattern

python

```python
from tenacity import retry, stop_after_attempt, wait_exponential

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=1, max=10),
    retry=retry_if_exception_type((ConnectionError, TimeoutError))
)
def call_external_api():
    # Retries on ConnectionError/TimeoutError only
    # Waits: 1s, 2s, 4s between retries
```

Rules:

- **Only retry transient errors (network, timeouts, 503)**
- **Don't retry business logic errors (validation, 404, 400)**
- **Use exponential backoff**
- **Cap max retries (prevent infinite loops)**

Frameworks:

- **Use `tenacity` library (works with any framework)**
- **Django: Integrate with Celery tasks**
- **FastAPI: Use as decorator on endpoints**

Caching Strategy

Pattern

python

```python
from functools import lru_cache
from cachetools import TTLCache

# Simple in-memory cache
@lru_cache(maxsize=128)
def expensive_computation(n: int) -> int:
    return n ** 2

# Time-based cache
cache = TTLCache(maxsize=100, ttl=300)  # 5 minutes

def get_user(user_id: int) -> User:
    if user_id in cache:
        return cache[user_id]
    user = fetch_from_database(user_id)
    cache[user_id] = user

    return user
```

Cache key design:

python

```
def make_cache_key(cls_name: str, method: str, *args) -> str:
    key_data = {
        "class": cls_name,
        "method": method,
        "args": [str(arg) for arg in args]
    }
    # Hash to fixed length

    return hashlib.sha256(json.dumps(key_data, sort_keys=True).encode()).hexdigest()
```

Frameworks:

- **Django: Use `django.core.cache`**
- **Flask: Use `flask-caching`**
- **FastAPI: Use `@lru_cache` or Redis**

Security Checklist

Input Validation

python

```
# ✅ Validate all external input
def create_user(email: str):
    if not re.match(r"^[\w\.-]+@[\w\.-]+\.\w+$", email):
        raise ValidationError("Invalid email format")
    if ".." in email or "/" in email:

        raise ValidationError("Suspicious email pattern")
```

SSRF Prevention

python

```
FORBIDDEN_HOSTS = {"localhost", "127.0.0.1", "0.0.0.0"}
PRIVATE_NETWORKS = ["10.", "192.168.", "172.16."]
```

```python
def validate_url(url: str):
    parsed = urlparse(url)
    if parsed.hostname in FORBIDDEN_HOSTS:
        raise SecurityError("Cannot access localhost")
    for network in PRIVATE_NETWORKS:
        if parsed.hostname.startswith(network):
            raise SecurityError("Cannot access private network")
```

Secret Sanitization

python

```python
def sanitize_secrets(text: str) -> str:
    patterns = [
        (r"password=\S+", "password=[REDACTED]"),
        (r"api_key=\S+", "api_key=[REDACTED]"),
        (r"token=\S+", "token=[REDACTED]"),
    ]
    for pattern, replacement in patterns:
        text = re.sub(pattern, replacement, text, flags=re.IGNORECASE)
    return text
```

Testing Patterns

Dependency Injection Enables Easy Testing

python

~~~python
# Production code
def process_payment(payment_gateway: PaymentGateway, amount: float):
    return payment_gateway.charge(amount)

# Test code
class FakePaymentGateway:
    def charge(self, amount: float):
        return {"status": "success"}

def test_payment():
    gateway = FakePaymentGateway()  # Inject fake
    result = process_payment(gateway, 100.0)
    assert result["status"] == "success"
```

### Test Organization
~~~

```
tests/
├── unit/          # Fast, isolated, no external dependencies
├── integration/    # Test multiple components together
└── e2e/           # Full system tests
```

## **Anti-Patterns to Avoid**

❌ **Don't**	✅ **Do**
Global config: `from config import CONFIG`	Inject config: `__init__(self, config: Config)`
God classes with 50+ methods	Small classes with single responsibility
Circular imports	Clean dependency tree
Mixing concerns (models calling APIs)	Layers with clear boundaries
Catching bare `Exception`	Catch specific exceptions
Mutable domain models	Frozen dataclasses
Validation deep in logic	Validate at entry points
Hardcoded dependencies	Dependency injection

## **Quick Decision Tree**
```
Need multiple implementations? → Factory
Need optional behavior? → Mixin
Need to define interface? → Protocol
Need to translate interfaces? → Adapter
Need to abstract data access? → Repository
Need to transform external data? → Mapper
Need to retry operations? → Retry with backoff
Need to cache results? → Caching with TTL
Need configuration? → Immutable dataclass

Need domain models? → Frozen dataclass
```

Framework-Specific Applications

Django

- Use factories for model managers
- Use mixins for view behaviors (LoginRequired, etc.)
- Replace settings with typed config dataclasses
- Use repositories to abstract ORM queries

FastAPI

- Built-in dependency injection with `Depends()`
- Use Pydantic for domain models (similar to frozen dataclasses)
- Apply repository pattern for database access
- Use protocols for service interfaces

Flask

- Use blueprints as application layer
- Apply factory pattern for app creation
- Use `flask-injector` for dependency injection
- Implement repositories for database access

Summary: The Golden Rules

1. Layers: Presentation → Application → Domain → Infrastructure
2. Validation: At boundaries, fail fast
3. Dependencies: Inject, don't create
4. Configuration: Immutable, typed, validated
5. Domain Models: Frozen dataclasses
6. Errors: Specific exceptions, clear hierarchy
7. Resources: Always set limits
8. Logging: Structured, sanitized, at boundaries
9. Retry: Transient errors only, with backoff
10. Security: Validate input, sanitize output, block dangerous operations

Apply these patterns when you have the problem they solve, not just because they exist. Start simple, add complexity only when needed.

Retry