Ready Reference: Specs, Standards, and Submission Checklist

1 At-a-Glance Requirements

- **Language:** Python 3.10+
- **Style:** PEP 8 + Google or NumPy docstrings; PEP 257 for docstrings
- **Typing:** Type hints required for all public functions/classes
- **Tests:** pytest required; minimum 5 tests covering normal, edge, and error paths; at least one parametrized test
- **Logging:** Use the logging module; INFO for key events, DEBUG for diagnostics; never log secrets
- **Errors:** Use specific exceptions; validate inputs; no silent failures
- **AI Use:** PCA allowed; verify outputs; include a brief AI Decision Log
- **Deliverables:** Code, tests/, README (≤ 1 page), examples.py or README snippet, AI Decision Log
- **Submission windows:** Initial post by Thu 11:55 p.m. (Central); 2 peer replies by Sat; wrap-up by Sun

2 CheckiO Task Specs (if applicable)

- **Function signatures:** Do not change names or argument order specified by the task
- **Determinism:** No random or time-dependent behavior unless specified
- **I/O:** Avoid print/input in solution functions (use return values)
- **Performance:** Must pass platform time/space limits for given test sizes
- **Edge cases:** Handle empty input, None, extremes, duplicates, and invalid types (document constraints)
- **Examples:** Provide at least one minimal example per task in README or examples.py

3 Project Specs (module-level)

Scope: 3–6 public functions/classes or ~300–600 LOC module

Recommended structure:

```
project_root/
├── src/your_module.py (or package/)
├── tests/test_your_module.py
├── examples.py (optional if in README)
├── README.md
└── pyproject.toml (optional extension)
```

Required capabilities:

- Clear API surface (public vs private)
- Input validation and meaningful exceptions
- Logging configured via basicConfig or dictConfig (keep it simple)
- Unit tests covering normal, edge, and error flows
- Usage example that runs as-is

4 Coding Standards

PEP 8 highlights

- Line length ≤ 88–100 chars (choose and stick to one)
- snake_case for functions/variables; PascalCase for classes
- Prefer f-strings; avoid implicit string concatenation across lines
- Docstrings (choose one style and be consistent)

Google style example

```
def normalize(values: list[float], *, eps: float = 1e-9) -> list[float]:
    """
    Scale values to the [0, 1] range.
    Args:
        values: Input sequence of floats.
        eps: Small constant to avoid division by zero.
    Returns:
        Normalized values in the same order as input.
    Raises:
        ValueError: If values is empty or contains non-floats.
    """
    ...
```

Type hints

- Use builtins generics (`list[int]`, `dict[str, float]`)
- Mark optional as `Optional[T]` or `T | None`
- Add return types for all functions; `__init__` returns None

Comments

- Explain "why," not "what." Keep them current or remove.

5 Error Handling and Validation

- Validate at boundaries (public APIs)
- Prefer built-in exceptions (ValueError, TypeError, KeyError) with specific messages
- Avoid bare except; catch specific exceptions; re-raise with context if needed
- Don't swallow exceptions silently; log or raise meaningfully

6 Logging (minimal, appropriate)

Configuration (in module guard or CLI, not at import time):

```
import logging
logging.basicConfig(
    level=logging.INFO,
    format="%(levelname)s:%(name)s:%(message)s"
)
logger = logging.getLogger(__name__)
```

Usage:

```
logger.info("Loaded %d records", len(records))
logger.debug("Params: %s", params)  # ensure no secrets
logger.warning("Empty input received; using defaults")
```

7 Testing Requirements (pytest)

Minimum: 5 tests total; include normal, edge, and error paths; ≥1 parametrized test

Example parametrized test:

```
import pytest
@pytest.mark.parametrize("values,expected", [
    ([0, 10], [0.0, 1.0]),
    ([5, 5], [0.0, 0.0]),
])
def test_normalize_basic(values, expected):
    assert normalize(values) == expected
```

Error testing:

```
import pytest
def test_normalize_raises_on_empty():
    with pytest.raises(ValueError):
        normalize([])
```

8 README (≤ 1 page)

- **Purpose:** One-sentence overview + what problem it solves
- **Quick start:** `python -m venv .venv; pip install -r requirements.txt` or use `pyproject.toml`
- **Usage:**

```
from your_module import normalize
print(normalize([2, 4, 6]))
```

- **Limitations/assumptions:** State known constraints and edge cases
- **AI citation:** "Assistance from [PCA name/version]; final code reviewed/verified by author."

9 AI Decision Log (required, brief)

3–5 entries; each includes:

- Prompt (1–2 lines)
- AI suggestion summary (1–2 lines)
- Decision: Accept / Modify / Reject
- Rationale (1–2 lines)

Example entry:

Prompt	"Propose exceptions for parse_config()."
AI Suggestion	Use ValueError and FileNotFoundError; add schema check.
Decision	Modify
Rationale	Adopt FileNotFoundError; replace ValueError with custom ConfigError for clarity.

10 Submission Checklist

- Code runs without modification; no prints in library functions
- Docstrings consistent style; all public members typed
- Input validation + specific exceptions; no silent failures
- Logging added and level-appropriate; secrets excluded
- tests/ present; all tests passing locally
- README ≤ 1 page with usage example
- AI Decision Log included with at least one "Reject" or "Modify" entry
- Initial post by Thursday; two peer replies by Saturday; wrap-up by Sunday

11 Quality Targets (A-level indicators)

- Docstrings and type hints are complete and accurate
- Tests are meaningful, not redundant; include boundary and failure cases
- Logs provide useful runtime breadcrumbs without noise
- Clear, concise README that orients a new developer in under 60 seconds
- Thoughtful AI log demonstrating critical evaluation

Success = Clear code + Strong tests + Helpful docs + Good communication. Build it. Test it. Document it. Share it.