Python Code Evaluation Rubric and Design Guidelines

The following criteria define expectations for modern, idiomatic, and maintainable Python code in review and instructional contexts. This rubric reflects preferred design principles and trade-offs, not an exhaustive list of universal Python conventions. Evaluators should apply judgment and context, especially when assessing public-facing or legacy-compatible APIs. The rubric is organized from foundations to surface presentation: beginning with high-level development philosophy, then function behavior and control flow, followed by typing and data modeling, and concluding with documentation and style.

Development Philosophy (Zen of Python)

- Code should, where practical, align with the principles captured in the Zen of Python (import this). These principles — favoring clarity over cleverness, simplicity over unnecessary complexity, explicitness over implicit behavior, and readability over density — represent a widely shared cultural foundation of idiomatic Python.
- Evaluators should prefer designs that are straightforward to understand, easy to explain, and unsurprising to typical experienced Python developers, even when such designs are not the most abstract or theoretically "pure"; when trade-offs arise between elegance, practicality, and maintainability, solutions that are clear, predictable, and minimally surprising should generally be favored.
- Evaluators should be pedantic but practical: precise about semantic contracts and correctness, while avoiding unnecessary rigidity, over-abstraction, or formalism that does not materially improve clarity or maintainability.
- Clarity and explicitness apply most strongly to semantic contracts (what is guaranteed), rather than to overly formal prose. Prefer documentation that is precise where correctness depends on it, and otherwise written in clear, ordinary language.

Language Version and Features

- Code should target modern Python (≥ 3.12).
- Legacy or deprecated patterns should be avoided unless explicitly justified.
- Features introduced in Python 3.12 (or earlier) may be used when they meaningfully improve clarity, correctness, or expressiveness.
- Standard Library Usage:
    - Prefer modern, well-supported standard library APIs over legacy or superseded alternatives when they improve clarity, correctness, or maintainability.
    - When multiple standard library approaches are available, favor those aligned with current Python best practices and the targeted language version.

Function Design

- Functions should have a single primary responsibility.
- Functions should be Pythonic, readable, and free of common code smells.

- Functions should be efficient for their intended use, avoiding unnecessary work, while prioritizing readability and correctness over premature optimization.
- Functions must read clearly, behave predictably, and their documentation must match their actual behavior.
- Input normalization should generally be separated from core processing logic, unless separation would meaningfully harm clarity or performance.
- Parameter Ordering:
    - Function parameters should be ordered to reflect the conceptual structure of the API rather than incidental implementation details. Parameters should generally be declared in the following order:
        - Primary subject: the main object, entity, or resource the function conceptually acts upon.
        - Data being operated on: additional inputs that represent the information to be processed.
        - Configuration: flags, options, or settings that influence behavior.
        - Optional collaborators: auxiliary components such as loggers, callbacks, hooks, or observers.
    - This ordering should make the function's intent clearer at the call site and support more natural use of positional arguments while keeping optional or infrastructural concerns toward the end of the signature.
- Avoid reassigning function parameters unless the function is very short and the intent is immediately obvious; instead, use local variables for transformations or intermediate values.

Naming Conventions

- Function Naming:
    - Function names should clearly and accurately reflect their behavior.
    - Function names, behavior, and documentation must be tightly aligned.
    - Normalization Functions:
        - Functions whose purpose is to transform input into a canonical form so downstream logic does not need to care about superficial differences should include the word normalize or a clear synonym (e.g. canonicalize, coerce) in their name.
    - Boolean Predicate Functions:
        - When a function's primary purpose is enforcing or validating conditions, naming should reflect that intent, even if a boolean is returned.
        - Under this rubric, boolean-returning functions that perform validation, may raise errors, enforce conditions, or produce observable side effects should start with check_ or another explicit verb such as validate_ or ensure_.
        - Pure boolean predicates without side effects should use conventional prefixes such as is_, has_, or can_.

    - **Evaluator Heuristic:** Evaluator Heuristic: If the function's name, behavior, and documentation do not make its intent obvious, then clarity has likely been sacrificed for brevity or convenience. Simple, explicit names that reduce the need to read surrounding code are preferred to clever, compact, or context-dependent ones. When in doubt, evaluators should be pedantic in service of readability.
- Variable Naming:
    - Variable names (including loop variables and comprehension bindings) should be chosen to communicate role and meaning, not merely satisfy syntax.
    - Prefer descriptive, domain-relevant names over abbreviated or opaque names unless the abbreviation is a widely understood term in context.
    - Avoid single-letter names except where convention strongly applies and readability is not harmed. Acceptable conventional cases include:
        - File handles in a with open(...) block (e.g., as f)
        - The throwaway name _ for intentionally unused values
    - Comprehension variables must follow the same naming standards as ordinary variables. In particular, the bound variable should reflect what each element represents, and names should not shadow built-ins or common standard-library identifiers unless explicitly justified.
    - Constants should generally use UPPER_SNAKE_CASE when they are semantically constant.
    - **Evaluator Heuristic:** If a reader must examine surrounding lines of code to understand what a variable represents or why it exists, then clarity has likely been sacrificed for brevity or convenience. Variable names that make their role obvious at the point of use are preferred to shorter, context-dependent, or overly generic names; when in doubt, evaluators should be pedantic in service of readability.

Determinism and Side Effects

- Functions should be deterministic unless non-determinism is inherent to their purpose (e.g., randomness, I/O, or time) and clearly documented.
- Side effects should be explicit, minimal, and well-documented.

Control Flow and Expressiveness

- Use modern language features (e.g. match statements) where they improve clarity over complex conditional logic.
- Code should favor explicit, readable control flow over cleverness.
- Ellipsis (...) vs. pass:
    - The ellipsis expression (...) and the pass statement have distinct semantic meanings and should not be used interchangeably.
    - Use ... to indicate that an implementation is intentionally absent. Appropriate contexts include:
        - Protocol method bodies

        - Abstract base class methods
        - Stub files (.pyi)
        - Type-only definitions or bodies that exist solely for typing purposes
    - In these cases, ... communicates that behavior is not defined here and is expected to be provided elsewhere.
    - Use pass to indicate an intentional no-op. Appropriate contexts include:
        - Optional hooks or extension points
        - Default implementations that deliberately perform no action
        - Methods that are explicitly permitted to do nothing
        - Empty blocks that are syntactically required by Python (e.g., an empty if, try, or class body)
    - Under this rubric, pass signals that "nothing happens by design," whereas ... signals that "this is unimplemented or specification-only." Mixing these meanings is considered a style and expressiveness issue.

Error Handling and Validation

- Exceptions should be specific and intentional.
- Broad except Exception clauses should be avoided unless used at well-defined boundaries.
- Exception chaining (raise X(...) from err) should be used when re-raising errors to preserve context.
- Errors should be reported through a consistent mechanism, and fallbacks must be documented.

Typing and Type Design

- Code should use current best-practice typing syntax, including:
    - Built-in generics (e.g. list[str], not List[str])
    - Prefer PEP 695 when defining reusable, generic APIs, not for trivial local functions
    - typing.Self, Protocol, and TypeAlias when they improve API clarity
- Type hints should prioritize clarity and correctness over completeness or exhaustiveness.
- Enums should improve the semantic clarity of type signatures. Introducing an Enum that does not meaningfully restrict or communicate valid choices is considered a design smell under this rubric.
- **Evaluator Heuristic:** If removing an Enum from a type signature would not change how a caller reasons about valid inputs or outputs, the Enum is likely misused.

Collection Type Hints (PEP-Aligned)

- Function parameters that accept collections must be typed using the weakest abstraction that correctly expresses the function's semantic requirements, not merely the concrete implementation being passed.
- This guidance aligns with PEP 484 (Type Hints), PEP 544 (Protocols and Structural Subtyping), and PEP 585 (Built-in Generic Types), and reflects the design intent of the collections.abc hierarchy.
- Preferred Abstract Collection Types (ordered roughly from weakest to strongest with respect to ordering and mutability):
    - Iterable[T]
        - Use when the function only iterates over the values and makes no additional assumptions.
        - Semantic guarantees:
            - Provides iter only
        - Does not guarantee:
            - Length (len)
            - Boolean evaluation via if collection
            - Membership testing
            - Guaranteed reusability; inputs might be single-pass iterables
            - Indexing or ordering
        - Notes:
            - Suitable for streaming, generator, or one-pass inputs
            - Represents the lowest semantic commitment
            - Consistent with PEP 484's guidance to avoid over-specifying parameter types
    - Collection[T]
        - Use when the function requires len(), membership testing, or other behaviors typically provided by Collection.
        - Semantic guarantees:
            - iter
            - len
            - contains
        - Implications:
            - Implies a known, computable length.
            - The collection is reusable.
            - Implies the collection is reusable for future operations.
        - Notes:
            - Stronger semantic signal than Iterable without requiring ordering
            - Appropriate when len(), membership checks, or other Collection-like behaviors are used
            - Aligns with PEP 544's emphasis on behavior over concrete types
    - Sequence[T]
        - Use when the function depends on ordering, positional meaning, or index-based access.
        - Semantic guarantees:

            - Stable ordering
            - Indexing and slicing (getitem)
            - Repeatable iteration with positional consistency
        - Implications:
            - Ordering is meaningful and relied upon
            - Index-based access is required
        - Notes:
            - Strongest and most restrictive common collection abstraction
            - Should not be used unless ordering or indexing is semantically required
    - MutableSequence[T]
        - Use when the function requires ordered, indexable access and in-place mutation of elements, but does not depend on a specific concrete container type.
        - Semantic guarantees:
            - iter
            - len
            - contains
            - Stable ordering
            - Indexing (getitem)
            - In-place element mutation (setitem, insert, append, del)
        - Implications:
            - The collection is ordered and reusable
            - Positional meaning is relied upon
            - Callers must expect the collection's contents to be modified in place
            - Length and element values may change over the lifetime of the call
        - Notes:
            - Represents the weakest abstraction that truthfully expresses ordered mutation
            - Does not guarantee support for list-specific APIs such as list.sort() or slice assignment
            - Should not be used when mutation is incidental rather than part of the function's semantic contract
            - Appropriate when behavior must work for any compliant mutable sequence, not just list
            - Stronger semantic signal than Sequence[T], weaker and less specific than list[T]
            - Aligns with PEP 544's emphasis on behavioral contracts over concrete types
        - Example:

            - MutableSequence[T] is appropriate if a function calls .append(), but list[T] is required if the function calls .sort() or uses list-specific behavior
    - Concrete Collection Types (e.g., list, tuple, set)
        - Concrete collection types must only be used when required by the function's semantics, such as:
            - In-place mutation (e.g., append, sort, pop)
            - API contracts that require a specific type
            - Performance or memory characteristics unique to the implementation
        - Using a concrete type when an abstract type would suffice is considered over-specification and a design flaw under this rubric
        - Structured tuples:
            - Concrete tuple types are appropriate when the structure or arity of the elements is semantically significant (e.g., tuple[float, float]). When only ordering or indexability matters (but not arity), prefer Sequence[T] over a concrete tuple[T, ...]
    - **Evaluation Heuristic:**
        - Type hints must describe how a function uses a collection, not what concrete type happens to be passed today.
        - The correct annotation is the weakest accurate type that captures the semantic guarantees the function relies on.
        - This principle applies most strictly to parameter types; return types may more often be concrete to communicate allocation, ownership, or performance characteristics.

Asynchronous Code (When Applicable)

- async / await should be used correctly and consistently.
- Blocking operations should not be performed inside asynchronous functions.
- Async APIs should clearly signal their behavior through naming and documentation.

Data Modeling

- Prefer structured types over ad-hoc dictionaries.
- Use @dataclass (preferably with slots=True and frozen=True where appropriate) for simple data containers.
- Immutability should be favored when practical.
- Enum Usage:
    - Enums should be used to model a closed set of meaningful choices within a domain.
    - Enum members must represent valid options that are intentionally selected, passed into functions, returned from APIs, or otherwise participate in control flow or decision-making.

    - Enums must not be used solely as namespaced containers for constants, configuration values, or unrelated symbolic values. Using an Enum as a constant bag obscures intent and misrepresents the semantics of the data being modeled.
    - If values are not conceptually chosen from, prefer:
        - Module-level constants
        - Typed constants (e.g. Final[str], Final[int])
        - Dedicated configuration objects or data structures
    - The presence of an Enum in an API signals a constrained and meaningful domain. Introducing an Enum that does not provide this semantic restriction is considered a data modeling flaw under this rubric.

Documentation and Docstrings

- Documentation should be written in a structured docstring format.
    - reStructuredText is preferred; NumPy and Google styles are acceptable alternatives.
- Module docstrings should be written in the indicative mood, clearly describing what the module provides, defines, or implements. They should explain the module's purpose at a high level and be written in clear, ordinary prose structured into complete sentences and paragraphs.
- Class docstrings should be written in the indicative mood, typically as a noun phrase or descriptive statement, clearly describing what the class represents or encapsulates. They should focus on the class's role, meaning, and intended use, and be written in clear, ordinary prose.
- Function docstrings should be written in the imperative mood, in clear, ordinary prose, and in complete sentences.
- Inline literals that could be typed directly into Python should be marked using double back-ticks, including:
    - identifiers
    - expressions
    - values
    - function calls
    - modules
    - keywords
- :param documentation may be omitted when it merely restates the parameter name or type without adding substantive information about constraints, semantics, or usage.
- Return Value Documentation:
    - For value-producing and boolean predicate functions, descriptions should begin with "Return ...".
    - For boolean-returning functions, :return: documentation should use the form:
        - Return True if ...
        - The phrase "otherwise False" should generally be omitted (as it is typically redundant for boolean-only return values).

    - For functions returning collections or other structured values, return documentation must describe both the shape and the semantic meaning of the result.
        - A good return description should answer:
        - What is returned? (Type and contents of the value.)
        - What does inclusion or exclusion mean? (Success criteria, filtering rules, guarantees, or conditions under which items appear.)
        - Why would a caller care? (Especially when the return value reflects non-obvious control flow, partial success, validation outcomes, or suppressed errors.)
        - Vague descriptions such as "List of files" or "Result data" are insufficient unless the semantic meaning is obvious from the function's name and context.
    - For functions returning collections:
        - Be specific about the return type and behavior (e.g. "List of compiled ...", "Iterator yielding ...")
        - Clearly indicate lazy behavior when returning iterators or generators
    - If a function substitutes a fallback when the configuration value is missing or empty, the summary may mention the fallback for concision. The precise failure behavior (e.g., returning None, raising ValueError, etc.) must be stated in :return: and/or :raises:. The summary should avoid implying that fallback guarantees a valid parsed result.
        - Template:
            - Summary: "Return a <type> parsed from <source>, using <fallback> if missing/empty."
            - :return: "Parsed <type>, or None if parsing/validation fails."
    - Prefer a one-line docstring when the behavior can be completely described in a single, non-redundant sentence (often "Return/Compute/Format ..."), and add a multi-line docstring only when documenting parameters, edge cases, side effects, invariants, or non-obvious behavior.
    - :return: documentation may be omitted for functions that return None.
- :raises documentation may be omitted when it merely restates the exception type or a condition already clear from the function's name, signature, or simple implementation, and does not add substantive information about when or why the exception is raised.
- Docstrings should be precisely pedantic about behavior and guarantees, while generally avoiding pedantry about ordinary English phrasing.
    - Be pedantic about what the function guarantees:
        - Specify contracts that affect correct usage (e.g., exact invariants, edge cases, ordering guarantees, idempotence, determinism, lifetime/validity of returned objects, and important side effects).
        - Clarify any behavior that would be surprising or easy to misuse.
    - Be simple about how you say it:
        - Prefer plain, direct sentences over legalistic or over-specified wording when it does not change how a caller would use the API.

        - Avoid redundant qualifiers that do not materially affect understanding (e.g., "in the output" when the function's purpose already implies it).
        - Optimize for readability and fast comprehension by experienced Python developers.
- **Evaluator Heuristic:** If additional wording would not change how a competent caller uses the function, it is likely unnecessary.

Style and Readability

- Avoid unnecessary f-strings: do not prefix a string with f unless it contains an interpolation ({...}); remove the f after refactors that eliminate formatted expressions.