| No | Recommended avoidance mechanism | Reference(s) |
|---|---|---|
| 1 | Avoid the use of `pickle`, but if it must be used, only unpickle trusted data. | |
| 2 | When using `asyncio`, make all tasks non-blocking. | |
| 3 | Avoid using `exec` or `eval` and never use these with untrusted code | |
| 4 | Avoid mixing concurrency models within the same program or, if unavoidable, use with extreme caution. | |
| 5 | When using monkey patching, be aware that altering the behavior of objects at runtime can make code much more difficult to understand and easily introduce vulnerabilities. | |
| 6 | Do not use floating-point arithmetic when integers or Booleans would suffice especially for counters associated with program flow, such as loop control variables. | 6.4 [PLF], 6.15 [FIF], 6.6 [FLC] |
| 7 | Use type annotations to help provide static type checking prior to running code. | 6.5 [CCB], 6.2 [IHN], 6.11 [HFC] |
| 8 | ~~Avoid the use of auto() for enums intended to be used for indexing into lists.~~ Avoid mixing the use of `auto()` for `enums` with manual assignments, and when indexing into a list. | 6.5 [CCB] |
| 9 | ~~Assume that when examining code, that a variable can be bound (or rebound) to another object (of same or different type) at any time.~~ Do not use mutable objects as default values for arguments in a function definition unless absolutely needed and the effect is understood and be cognizant that assignments to objects, mutable and immutable, always create a new object. | 6.18 [WXQ] |
| 10 | Avoid implicit references to `global` values from within functions to make code clearer. In order to update global objects within a function or class, place the `global` statement at the beginning of the function definition and list the variables so it is clearer to the reader which variables are local and which are `global` (for example, global a, b, c). | 6.21 [BJL] |
| 11 | ~~Use Python's built-in documentation (such as docstrings) to obtain information about a class' method before inheriting from it~~ Inherit only from trusted classes and only use multiple inheritance that is linearizable with the `mro` rules. | 6.41 [RIP] |

**Deleted:** ,

| | | |
|---|---|---|
| 12 | Either avoid logic that depends on byte order or use the `sys.byteorder` variable and write the logic to account for byte order dependent on its value ('little' or 'big'). | 6.57 [FAB], 6.3 [STR] |
| 13 | ~~When using multiple threads, check for race conditions and deadlocks by using fuzzing techniques during development.~~ When using multiple threads, verify that all shared data is protected by locks or similar mechanisms, and use inter-communication mechanisms or `global` references to ensure safe terminations. | 6.61 [CGX], 6.63 [CGM] |
| 14 | If necessary, the preferred method for killing a thread is from within the thread itself using a watchdog message queue or global variable that signals the thread to terminate itself. This will enable the thread to perform proper cleanup and eliminate deadlocks. | 6.60 [CGT], 6.62 [CGS] |
| 15 | Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap-around errors. | |
| 16 | Always use named exceptions to avoid catching errors that are intended for other exception handlers and use context managers to enclose the code creating the exception. | |
| 17 | Follow the guidance of PEP 551 and PEP 578 to eliminate potentially dangerous default behaviour from calls into the Python runtime and in the use of audit hooks (see the General Recommendations contained in “PEP 551 -- Security transparency in the Python runtime” and “PEP 578 Python Runtime Audit Hooks”. | |