	Python collated guidance	ISO/IEC/JTC 1/SC 22/WG 23 N1239
Clause	Advice	
6.2	Type system	
		Use static type checkers to detect typing errors. The Python community is one source of static type checkers.
		Pay special attention to issues of magnitude and precision when using mixed type expressions.
		Be aware of the consequences of shared references. See clause 6.24 Side-effects and order of evaluation of operands [SAM] and 6.38 Deep vs. shallow copying [YAN].
		Keep in mind that using a very large integer will have a negative effect on performance.
6.3	Bit representation	
		Be careful when shifting negative numbers to the right as the number will never reach zero.
		Localize and document the code associated with explicit manipulation of bits and bit fields.
		Use `sys.byteorder` to determine the native byte order of the platform.
6.4	Floating Point Arithmetic	
		Code algorithms to account for the fact that results can vary slightly by implementation.
6.5	Enumerator issues	
		Use type annotations to help provide static type checking prior to running the code.
		Avoid the use of auto() for enums intended to be used for indexing into lists.
		If using `auto()` for defining enums, ensure that `auto()` is used everywhere.

		If using `auto()` for defining enums, be very careful in converting to list members.
		Avoid using enums created by `auto()` to access lists
6.6	Conversion errors	
		Though there is generally no need to be concerned with an integer getting too large (rollover) or small, be aware that iterating or performing arithmetic with very large positive or small (negative) integers will hurt performance.
		Be aware of the potential consequences of precision loss when converting from floating-point to integer.
		Design coding strategies that allow the distinction of semantically incompatible types.
		Design classes that have operation handling methods carefully and ensure that `Py_NotImplemented` and `TypeError` exceptions are handled.
		Use or develop 'units' libraries to handle conversions between differing unit-based systems.
6.7	String termination (none)	
6.8	Buffer boundary violation (overflow) (none)	
6.9	Unchecked array indexing (none)	
6.10	Unchecked array copying - no specific guidance (none)	
6.11	Pointer type conversions	
		Do not alter the `__class__` attribute for instances of a class unless there are compelling reasons to do so. If alterations are required, document the reasons in docstring and local comments.
		Use type annotations and type hints in the code.
		Run a third-party static type checker.
6.12	Pointer arithmetic - (none)	
6.13	Null pointer dereference (none)	
6.14	Dangling reference to heap	
		When accessing data objects directly by using `memoryview()`, make sure that the data pointed to remains valid until it is no longer needed.

6.15	Arithmetic wrap-around error	
		Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap-around errors.
		Avoid using floating-point or decimal variables for loop control but if one of these types must be used, then bound the loop structures so as to not exceed the maximum or minimum possible values for the loop control variables.
		Test the implementation that is being used to see if exceptions are raised for floating-point operations and if they are then use exception handling to catch and handle wrap-around errors.
6.16	Using shift operators for multiplication and division (none)	
6.17	Choice of clear names	
		For more guidance on Python's naming conventions, refer to Python Style Guides contained in "PEP 8 – Style Guide for Python Code".
		Avoid names that differ only by case unless necessary to the logic of the usage, and in such cases document the usage.
		Adhere to Python's naming conventions.
		Do not use overly long names.
		Use names that are not similar (especially in the use of upper and lower case) to other names.
		Use meaningful names.
		Use names that are clear and visually unambiguous because the compiler cannot assist in detecting names that appear similar but are different.
		Ensure that 'show-all-hidden-characters' is enabled in the editor.
		Understand or eliminate all confusing Unicode characters, in particular, homoglyphs.
		Use caution when copying and pasting Unicode text.
6.18	Dead Store - no specific guidance	
		Assume that when examining code, that a variable can be bound (or rebound) to another object (of same or different type) at any time .

		Avoid rebinding except where it adds identifiable benefit.
		Consider using `ResourceWarning` to detect implicit reclamation of resources.
		This was modified to match the "Recommended avoidance mechanism" table in Section 5. We should revisit the intent of this guidance.
6.19	Unused variable - no specific guidance	
6.20	Identifier name reuse	
		Do not use identical names unless necessary to reference the correct object.
		Avoid the use of the `global` and `nonlocal` specifications because they are generally a bad programming practice for reasons beyond the scope of this annex and because their bypassing of standard scoping rules make the code harder to understand.
		Use qualification when necessary to ensure that the correct variable is referenced.
6.21	Namespace issues	
		Use the full path name for imports, in preference to relative paths.
		When using the `import` statement, rather than use the `from X import *` form (which imports all of module X's attributes into the importing program's namespace), instead explicitly name the attributes that need to be imported (for example, `from X import a, b, c`) so that variables, functions and classes are not inadvertently overlaid.
		Avoid implicit references to global values from within functions to make code clearer. In order to update globals 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`).
		When interfacing with external systems or other objects where the declaration order of class members is relevant, use `__prepare__` to obtain the desired order for class member creation.
6.22	Missing initialization of variables	

		Ensure that it is not logically possible to reach a reference to a variable before it is assigned to avoid the occurrence of a runtime error.
6.23	Operator precedence and associativity	
		(none specific)
6.24	Side effects and order of evaluation	
		Avoid assignment to a variable equally named as the loop index counters within the loop.
		Be aware of Python's short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression.
		Do not change the size of a data structures while iterating over it. Instead, create a new list.
		Use the `assert` statement during the debugging phase of code development to help eliminate undesired conditions from occurring.
6.25	Likely incorrect expression	
		Add parentheses after a function call in order to invoke the function.
		Keep in mind that any function that changes a mutable object in place returns a `None` object – not the changed object since there is no need to return an object because the object has been changed by the function.
		Be sure to use an `await` statement for async coroutines and ensure that all routines are nonblocking.
6.26	Dead and deactivated code	
		Import just the attributes that are required by using the `from` statement to avoid adding dead code.
		Be aware that subsequent imports have no effect; use the `reload` statement instead of `import` if a fresh copy of the module is desired.
6.27	Switch statements and static analysis (none)	
6.28	Demarcation of control flow	
		• Use either spaces or tabs, not both, to demark control flow.

		Note: Python 3.0+ will refuse to compile code that uses a mixture of tabs and spaces for indentation.
6.29	Loop control variable abuse	
		Be careful to only modify variables involved in loop control in ways that are easily understood and in ways that cannot lead to a premature exit or an endless loop.
		When using the `for` statement to iterate through a mutable object, do not add or delete members because it could have unexpected results.
		Avoid using assignment expressions in the loop control statement (that is, `while` or `for`).
6.30	Off-by-one error	
		Be aware of Python's indexing by default from zero and code accordingly.
		Be careful that a loop will always end when the loop index counter value is one less than the ending number of the range.
		Use the `for` statement to execute over whole constructs in preference to loops that index individual elements.
		Use the `enumerate()` built-in method when both container elements and their position within the iteration sequence are required.
6.31	Unstructures programming	
		Use the `break` statement judiciously to exit from control structures and show statically that the code behaves correctly in all contexts.
		Restructure code so that the nested loops that are to be collectively exited form the body of a function, and use early function returns to exit the loops. This technique does not work if there is more complex logic that requires different levels of exit.
		Use context managers (such as `with`) to enclose code creating exceptions.
6.32	Passing parameters and return values	
		Create copies of mutable objects before calling a function if changes are not wanted to mutable arguments.

		Uses `types.MappingProxy` or `collections.ChainMap` to provide read-only views of mappings without the cost of making a copy.
		Be aware that for immutable arguments, local copies are created when assignment occurs within the function while for mutable arguments, assignments operate directly on the original argument.
		Be careful when passing mutable arguments into a function since the assignment sequence (order) within the function may produce unexpected results.
6.33	Dangling references to stack frames	
		Avoid using `ctypes` when calling C code from within Python and use `cffi` (C Foreign Function Interface) instead.
6.34	Subprogram signature mismatch	
		Adjust the maximum recursion depth to an appropriate value as needed.
6.35	Recursion	
		Prefer iteration to recursion, unless it can be proved that the depth of recursion can never be large.
6.36	Ignored error status and unhandled exception	
		Use Python's exception handling with care in order to not catch errors that are intended for other exception handlers. That is, always catch named exceptions.
		Use exception handling, but directed to specific tolerable exceptions, to ensure that crucial processes can continue to run even after certain exceptions are raised.
6.37	Type-breaking reinterpretation of data (N/A)	
6.38	Deep vs. shallow copying	
		Be aware the "slice" operator "`[:]`" and the container `copy()` methods only perform shallow copies.
		To obtain deep copies at all levels of a variable, use the `copy.deepcopy` standard library function.
6.39	Memory leaks and heap fragmentation	

		Set each object to null when it is no longer required.
		If a program is intended for continuous operation, examine all object usage carefully, following the guidance of ISO/IEC TR 24772-1:2019, to show that memory is effectively reclaimed and reused.
		Use context managers to explicitly release large memory buffers that are no longer needed.
6.40	Templates and generics	
		Though Python does not meet the applicable language characteristics, the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.40.5 is good advice for avoiding issues that arise in a dynamically typed language.
6.41	Inheritance	
		Inherit only from trusted classes, such as standard classes.
		Only use multiple inheritance that is linearizable by the MRO rules.
		Make sure that each class calls the `__init__` of its superclass.
		Use the `__mro__` attribute to obtain information about the MRO sequence of classes followed by method calls.
		Use static analysis tools supported by type-checking hints.
		Employ type hints to elicit compile-time analysis.
		Prefix method calls with the desired class wherever feasible.
		Use Python's built-in documentation (such as docstrings) to obtain information about a class' methods before inheriting from the class provided that the documentation accurately reflects that implemented code.
		For users who are new to the use of multiple inheritance in Python, carefully review Python's rules, especially those of `super()` and class names that prefix calls.
6.42	Violations of the Liskov substitution principle	
		use software static analysis tools to detect such violations.
6.43	Redispatching	

		Avoid dispatching whenever possible by prefixing the method call with the target class name, or with `super()`.
		Within a single class, avoid the definition of a second method with the same signature as an existing method.
		Use systematic code reviews, organization-wide coding standards, and static analysis tools to prevent problems related to the redefinition of methods in object-oriented programming.
6.44	Polymorphic variables	
		Make sure that each class implements the __init__ method that calls the __init__ of its superclass.
		Employ static type checking by providing type hints for static analysis tools in areas involving inheritance.
		Use `__mro__` as an aid during development and during maintenance to help obtain the desired class hierarchies and verify linearity.
		Consider using `__mro__` to check at runtime that the actual method binding matches the expected method binding and to raise an exception if they do not match.
		Pay attention to warnings that identify variables written but never read.
6.45	Extra intrinics	
		Do not override built-in “intrinsics”.
		If it is necessary to override an intrinsic, document the case and show that it behaves as documented and that it preserves all the properties of the built-in intrinsic.
6.46	Argument passing to library functions	
		(No python-specific guidance)
6.47	Inter-language calling	

		Do not write Python extension modules by hand, as doing so is error-prone, and highly likely to lead to reference counting errors, memory leaks, dangling pointers, out-of-bounds memory accesses, and similar problems.
		Where available, use existing interface libraries that bridge between Python and the extension module language, for example, `PyO3` for Rust, `pybind11` for C++.
6.48	Dynamically-linked code and self-modifying code	
		Avoid using `exec` or `eval` and *never* use these with untrusted code.
		Be careful when using Guerrilla patching to ensure that all uses of the patched classes and/or modules continue to function as expected; conversely, be aware of any code being used that patches classes and/or modules to avoid unexpected results.
		Ensure that the file path and files being imported are from trusted sources.
		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”.
		Verify that the release version of the product does not use default entry points (python.exe on Windows, and pythonX.Y on other platforms) since these are executable from the command line and do not have hooks enabled by default.
		Consider using a modified entry point that restricts the use of optional arguments since this will reduce the chance of unintentional code from being executed.
		Avoid any unprotected settings from the working environment in an entry point.
		If the application is performing event logging as part of normal operations, consider logging all predetermined events in calling external libraries.
		Consider logging as many events as possible and ensure that such logs are moved off local machines frequently.
6.49	Library signature	
		Use only trusted modules as extensions.

		If coding an extension, utilize Python's extension API to ensure a correct signature match.
6.50	Unanticipated exceptions from library routines	
		(No specific Python guidance)
6.51	Pre-processor directives (N/A)	
6.52	Suppression of language-defined checks (N/A)	
6.53	Provision of inherently unsafe operations	
		Use only trusted modules.
		Avoid the use of the `exec` and `eval` functions.
		Avoid overriding Python's default behaviour provided by the `builtins` module.
		Create a whitelist of Python `built-in` functions that are deemed to be expected and acceptable in uses of `pickle` and forbid any other functions.
		Do not override the names of `built-in` variables or functions.
		Avoid the use of the `pickle` module and `logging.dictConfig` and consider using `JSON` and `MessagePack` as alternatives.
		Avoid the use of `pickle` for long term storage.
		Avoid the use of protocol 0.
		Disallow the use of self-referencing payloads.
6.54	Obscure language features	
		Ensure that a function is defined before attempting to call it.
		Be aware that a function is defined dynamically so its composition and operation may vary due to variations in the flow of control within the defining program.
		Be aware of when a variable is local versus global.
		Do not use mutable objects as default values for arguments in a function definition unless absolutely needed and the effect is understood.
		Be aware that when using the += operator on mutable objects the operation is done in place with a new object id being created.

		Be cognizant that assignments to objects, mutable and immutable, always create a new object.
		Understand the difference between equivalence and equality and code accordingly.
		Ensure that the file path used to locate a persisted file or DBMS is correct and *never* ingest objects from an untrusted source.
6.55	Unspecified behaviour	
		When pickling is applied to make objects persistent, use exception handling to cleanup partially written files.
		Prefer the use of *equality* (==) to *identity* (`is`) and clearly document any use of identity.
		Use the `intern()` function to enforce optimization when memory optimization is required for non-simple strings.
		Consider using the `id` function to test for object equality.
		Do not use form feed characters for indentation.
6.56	Undefined behaviour	
		Do not depend on the sequence of keys in a dictionary to be consistent across implementations, or even between multiple executions with the same implementation, in versions prior to Python 3.7.
		When launching parallel tasks do not raise a BaseException subclass in a callable in the Future class.
		Do not modify the dictionary object returned by a `vars()` and `locals()` call.
		Do not try to use the `catch warnings` function to suppress warning messages when using more than one thread.
		Do not inspect or change the content of a list when sorting a list using the `sort()` method.
6.57	Implementation-defined behaviour	

		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`').
		Always use either spaces or tabs (but not both) for indentations.
		Consider using a text editor to find and make consistent, the use of tabs and spaces for indentation.
		Use zero (the default exit code for Python) for successful execution and consider adding logic to vary the exit code according to the platform as obtained from `sys.platform` (such as, '`win32`', '`darwin`', or other).
		Interrogate the `sys.float.info` system variable to obtain platform specific attributes and code according to those constraints.
		Call the `sys.getfilesystemcoding()` function to return the name of the encoding system used.
		Use the `os.fsencode()` and `os.fsdecode()` methods as a portable way to encode or decode a filename to the filesystem encoding that is used.
		When high performance is dependent on knowing the range of integer numbers that can be used without degrading performance use the `sys.int_info struct` sequence to obtain the number of bits per digit (`bits_per_digit`) and the number of bytes used to represent a digit (`sizeof_digit`).
		Use `sys.maxsize` to determine the maximum value a variable of type `Py_ssize_t` can take. Usually on a 32-bit platform, the value is 2**31 - 1 on a 32-bit platform and 2**63 - 1 on a 64-bit platform.
6.58	Deprecated language features	
		(No specific Python guidance)
6.59	Concurreny -- Activation	
		For any processes and threads that have already been started, ensure that additional starts on that same object are not attempted to avoid exceptions.

		Avoid mixing concurrency models within the same program, or if unavoidable, use with extreme caution.
		Handle all exceptions related to thread creation.
		Ensure that there is only one `asyncio` event loop per program, although multiple events can be activated within the single loop. Python event loops are automatically generated by `asyncio.run()`.
		When using `asyncio`, make all tasks non-blocking and use `asyncio` calls from an event loop.
		Use the debug mode of the Python interpreter to detect concurrency errors .
		To reduce the chance of excessive delays, perform concurrent `asyncio` operations only on non-blocking code.
		When using multiple threads, consider using the `ThreadPoolExecutor` within the `concurrent.futures` module to help maintain and control the number of threads being created.
		For async functions, ensure that each async call executes one or more operations that relinquish control of the processor when appropriate.
6.60	Concurrency -- directed termination	
		Avoid external termination of concurrent entities except as an extreme measure, such as the termination of the program.
		Use inter-thread or inter-process communication mechanisms to instruct another thread or process to terminate itself.
		Ensure that all shared resources locked by the thread or process are released upon termination, for example, in an exception handler and/or in a finally block.
		Design the code to be fail-safe in the presence of terminating processes, threads or tasks.
		Do not call `join()` on a daemon thread.
6.61	Concurrent data access	

		When using multiple threads, verify that all shared data is protected by locks or similar mechanisms.
		If data accesses need to be serialized, ensure that they reside in the same thread, or provide explicit synchronization among the threads or processes for the data accesses.
		Avoid using global variables and consider using the `queue.Queue()`, `threading.queue, asyncio.queue` or `multiprocessing.Queue()` functions to exchange data between threads or processes respectively.
		When multiple asyncio tasks access data shared among tasks, always complete such access in each task prior to awaiting any event.
		When multiple asyncio tasks access complex data shared among tasks which may require multiple iterations to fully update, retain any partial data local to the task and perform the update only when all data is present.
		If shared variables must be used in multithreaded applications, use model checking or equivalent methodologies to prove the absence of race conditions.
6.62	Concurrency -- premature termination	
		Protect data that would be vulnerable to premature termination, such as by using locks or protected regions, or by retaining the last consistent version of the data (checkpoints).
		Enable event logging and record all events prior to termination so that full traceability is preserved.
		For threads:
		Handle exceptions; free locks; and clean up nested threads and shared data before termination.
		Consider using the `or try` or `finally` clauses in each thread method to notify a higher-level construct of the termination so that any corrective action if needed can be taken.

		Consider using one or more of the `threading.is_alive()`, `threading.active_count()`, and `threading.enumerate()` methods to determine if a thread's execution state is as expected.
		For multiprocessing:
		Handle exceptions; free locks; and clean up any processes that are the responsibility of this process.
		Consider using the `or try` or `finally` clauses in each thread method to notify a higher-level construct of the termination so that any corrective action if needed can be taken.
		Consider using one or more of the `threading.is_alive()`, `threading.active_count()`, and `threading.enumerate()` methods to determine if a thread's execution state is as expected.
		For Asyncio:
		Ensure consistent termination behaviour of all coroutines
6.63	Protocol lock errors	
		Verify that all sections of code that have critical sections check the related lock prior to entering the critical section, including API calls known to be unsynchronized.
		Ensure that `join()` is not used on a process before it is started since this will throw an exception.
		When using `Pipe()` in conjunction with processes or threads inside multiple processes, restrict the writing of a single pipe to a single process or thread, and similarly for reading.
		If exclusive access to any resource shared among multiple processes is needed, ensure the exclusivity by synchronization mechanisms provided by the `multiprocessing` module.

		If global variables are used in multi-threaded code, use locks around their use. Access to the shared data can be protected by first testing-and-setting a lock, then manipulating the data, and then releasing the lock when finished and before exiting. The use of locks does not guarantee security since locks are only effective if all other threads check for the locks. A locked critical section in one thread can be modified by another thread if it does not first check for the lock.
		Verify that all sections of code that have access to critical sections check for a lock prior to accessing the resource.
		When using global variables in multi-threaded code, use `threading_local()` which creates a local copy of the global variable within each thread.
		When using multiple threads, consider using semaphores to manage access to critical sections of data.
		When using Pipe() in conjunction with processes or threads, restrict the writing of a single pipe to a single process or thread, and similarly for reading.
		For threads, use `join()` as the final interaction with other thread(s) to ensure that the calling thread is blocked until all joined threads have either terminated normally, thrown an exception, or timed out (if implemented).
		Ensure that `join()` is not used on a thread or a process before it is started since this will throw an exception.
		When using `Pipe()` in conjunction with threads, restrict the writing of a single pipe to a single thread, and similarly for reading.
		Do not `yield` within critical sections.
6.64	Uncontrolled format string	
		Implement checks to limit the size of input strings.
		Limit the number of input arguments to the expected values.
		Review the Python format string specifiers and do not allow formats that should not be input by the user.
6.65	Modifying constants	

		Do not assign new values to `NotImplemented`, Ellipsis or `__debug__`.

DMJ	38	E

					*