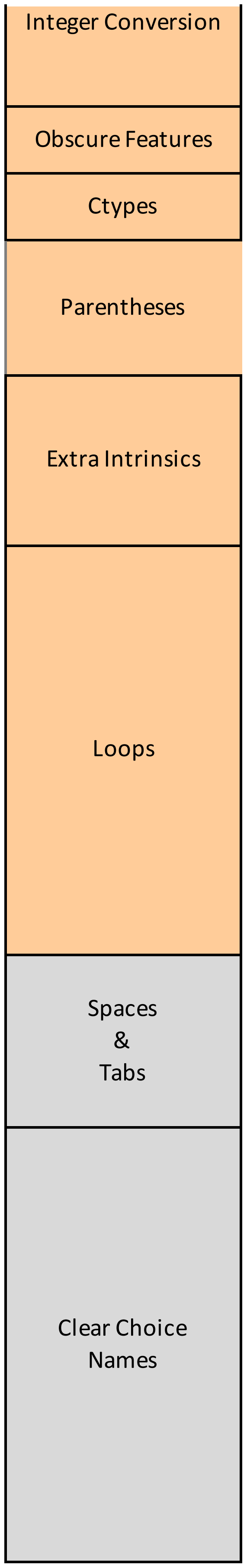
Category
Global References
Concurrency Mixing
Pickle
Exec & Eval
Enums

Mutable Objects
Right Shift
Monkey Patching
Wrap-around
Concurrency Premature Termination
Type Hints and Checkers
Exceptions

Asyncio
Inheritance and Redispatching
Dynamically-Linked and Self-Modifying Code
__mro__
Threads
Pipes
Undefined Behavior
Trusted Modules

Trusted Modules
Concurreny Premature Termination
join()
Concurrency Lock Protocols
Input Strings & Arguments
Implementation-defined Behavior
Integer Size
Unspecified Behavior
Critical Sections & Locks
Deep vs Shallow Copy


Integer Conversion
Obscure Features
Ctypes
Parentheses
Extra Intrinsics
Loops
Spaces & Tabs
Clear Choice Names

Recursion
Entry Point
Debug Mode
Shared References
Dead Code
Bit Representation
Floating-Point Arithmetic
Dead Store
Concurrency Directed Termination
Pointer Type Conversion
Modifying Constants
Inter-Language Calling
Missing Initialization of Variables
Memory Leaks
Library Signature
Concurrent Data Access
Imports
__init__
Polymorphic Variables

Obscure Language Features
Conversion Errors
Name Reuse
Side-effects
Unstructured Programming
Name Space
Parameters and Return Values
Dangling Reference
Namespace

Mitigations - Most Effective and Most Common
Avoid implicit references to global values from within functions to make code clearer. In order to update globals within a fu class, place the global statement at the beginning of the function definition and list the variables so it is clearer to the reader variables are local and which are global (for example, global a, b, c).
Avoid the use of the global and nonlocal specifications because they are generally a bad programming practice for reasons be scope of this annex and because their bypassing of standard scoping rules make the code harder to understand.
Avoid using global variables and consider using the queue.Queue(), threading.queue, asyncio.queue or multiprocessing.Que functions to exchange data between threads or processes respectively.
Be aware of when a variable is local versus global.
If global variables are used in multi-threaded code, use locks around their use. Access to the shared data can be protected by testing-and-setting a lock, then manipulating the data, and then releasing the lock when finished and before exiting. The use does not guarantee security since locks are only effective if all other threads check for the locks. A locked critical section in o can be modified by another thread if it does not first check for the lock.
When using global variables in multi-threaded code, use threading_local() which creates a local copy of the global variable w thread.
If shared variables must be used in multithreaded applications, use model checking or equivalent methodologies to prove th of race conditions.
Avoid mixing concurrency models within the same program, or if unavoidable, use with extreme caution.
Avoid the use of pickle for long term storage.
Avoid the use of the pickle module and logging.dictConfig and consider using JSON and MessagePack as alternatives.
Create a whitelist of Python built-in functions that are deemed to be expected and acceptable in uses of pickle and forbid an functions.
Avoid the use of protocol 0. Expand this mitigation so that it stands on it's own
Avoid the use of the exec and eval functions.
Avoid using exec or eval and never use these with untrusted code.
Avoid the use of auto() for enums intended to be used for indexing into lists.
Avoid using enums created by auto() to access lists
If using auto() for defining enums, be very careful in converting to list members.
If using auto() for defining enums, ensure that auto() is used everywhere.
Use the enumerate() built-in method when both container elements and their position within the iteration sequence are rec
Be aware that for immutable arguments, local copies are created when assignment occurs within the function while for mut arguments, assignments operate directly on the original argument.
Create copies of mutable objects before calling a function if changes are not wanted to mutable arguments.
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 creat

Be careful when passing mutable arguments into a function since the assignment sequence (order) within the function may p unexpected results.
Be cognizant that assignments to objects, mutable and immutable, always create a new object.
Keep in mind that any function that changes a mutable object in place returns a None object – not the changed object since need to return an object because the object has been changed by the function.
When using the for statement to iterate through a mutable object, do not add or delete members because it could have unex results.
Be careful when shifting negative numbers to the right as the number will never reach zero.
Be careful when using Guerrilla patching to ensure that all uses of the patched classes and/or modules continue to function a expected; conversely, be aware of any code being used that patches classes and/or modules to avoid unexpected results.
Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap errors.
Consider using the or try or finally clauses in each thread method to notify a higher-level construct of the termination so tha corrective action if needed can be taken.
Employ static type checking by providing type hints for static analysis tools in areas involving inheritance.
Run a third-party static type checker.
Use software static analysis tools to detect such violations.
Use static analysis tools supported by type-checking hints.
Use static type checkers to detect typing errors. The Python community is one source of static type checkers.
Use systematic code reviews, organization-wide coding standards, and static analysis tools to prevent problems related to th redefinition of methods in object-oriented programming.
Use type annotations and type hints in the code.
Use type annotations to help provide static type checking prior to running the code.
Employ type hints to elicit compile-time analysis.
For any processes and threads that have already been started, ensure that additional starts on that same object are not attem avoid exceptions.
Handle all exceptions related to thread creation.
Handle exceptions; free locks; and clean up any processes that are the responsibility of this process.
Handle exceptions; free locks; and clean up nested threads and shared data before termination.
Test the implementation that is being used to see if exceptions are raised for floating-point operations and if they are then us exception handling to catch and handle wrap-around errors.
Use context managers (such as with) to enclose code creating exceptions.
Use exception handling, but directed to specific tolerable exceptions, to ensure that crucial processes can continue to run e certain exceptions are raised.
Use Python's exception handling with care in order to not catch errors that are intended for other exception handlers. That catch named exceptions.
Design classes that have operation handling methods carefully and ensure that Py_NotImplemented and TypeError exceptio handled.

For async functions, ensure that each async call executes one or more operations that relinquish control of the processor wh appropriate.
To reduce the chance of excessive delays, perform concurrent asyncio operations only on non-blocking code.
When multiple asyncio tasks access complex data shared among tasks which may require multiple iterations to fully update, partial data local to the task and perform the update only when all data is present.
When multiple asyncio tasks access data shared among tasks, always complete such access in each task prior to awaiting any
When pickling is applied to make objects persistent, use exception handling to cleanup partially written files.
When using asyncio, make all tasks non-blocking and use asyncio calls from an event loop.
Be sure to use an await statement for async coroutines and ensure that all routines are nonblocking.
Use Python’s built-in documentation (such as docstrings) to obtain information about a class’ methods before inheriting fro class provided that the documentation accurately reflects that implemented code.
Prefix method calls with the desired class wherever feasible.
For users who are new to the use of multiple inheritance in Python, carefully review Python’s rules, especially those of super names that prefix calls.
Inherit only from trusted classes, such as standard classes.
Only use multiple inheritance that is linearizable by the MRO rules.
Within a single class, avoid the definition of a second method with the same signature as an existing method.
Avoid dispatching whenever possible by prefixing the method call with the target class name, or with super().
Consider logging as many events as possible and ensure that such logs are moved off local machines frequently.
If the application is performing event logging as part of normal operations, consider logging all predetermined events in call external libraries.
Verify that the release version of the product does not use default entry points (python.exe on Windows, and pythonX.Y on o 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 uninte code from being executed.
Ensure that the file path and files being imported are from trusted sources.
Ensure that the file path used to locate a persisted file or DBMS is correct and never ingest objects from an untrusted source.
Follow the guidance of PEP 551 and PEP 578 to eliminate potentially dangerous default behaviour from calls into the Pytho and in the use of audit hooks (see the General Recommendations contained in “PEP 551 -- Security transparency in the Pytho runtime” and “PEP 578 Python Runtime Audit Hooks”.
Use __mro__ as an aid during development and during maintenance to help obtain the desired class hierarchies and verify lin
Use the __mro__ attribute to obtain information about the MRO sequence of classes followed by method calls.
Consider using __mro__ to check at runtime that the actual method binding matches the expected method binding and to r exception if they do not match.
When using multiple threads, consider using semaphores to manage access to critical sections of data.
When using multiple threads, consider using the ThreadPoolExecutor within the concurrent.futures module to help mainta control the number of threads being created.
When using multiple threads, verify that all shared data is protected by locks or similar mechanisms.
Use inter-thread or inter-process communication mechanisms to instruct another thread or process to terminate itself.
When using Pipe() in conjunction with threads, restrict the writing of a single pipe to a single thread, and similarly for readin
When launching parallel tasks do not raise a BaseException subclass in a callable in the Future class.
Do not inspect or change the content of a list when sorting a list using the sort() method.
Do not try to use the catch warnings function to suppress warning messages when using more than one thread.
Use only trusted modules as extensions.

Use only trusted modules.
Enable event logging and record all events prior to termination so that full traceability is preserved.
Ensure consistent termination behaviour of all coroutines
Consider using one or more of the threading.is_alive(), threading.active_count(), and threading.enumerate() methods to det thread's execution state is as expected.
Do not call join() on a daemon thread.
Ensure that join() is not used on a process before it is started since this will throw an exception.
Ensure that join() is not used on a process before it is started since this will throw an exception.
For threads, use join() as the final interaction with other thread(s) to ensure that the calling thread is blocked until all joined have either terminated normally, thrown an exception, or timed out (if implemented).
If exclusive access to any resource shared among multiple processes is needed, ensure the exclusivity by synchronization me provided by the multiprocessing module.
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.
Call the sys.getfilesystemcoding() function to return the name of the encoding system used.
Interrogate the sys.float.info system variable to obtain platform specific attributes and code according to those constraints.
Use the os.fsencode() and os.fsdecode() methods as a portable way to encode or decode a filename to the filesystem encodin used.
Use zero (the default exit code for Python) for successful execution and consider adding logic to vary the exit code according platform as obtained from sys.platform (such as, 'win32', 'darwin', or other).
Either avoid logic that depends on byte order or use the sys.byteorder variable and write the logic to account for byte order on its value ('little' or 'big').
Keep in mind that using a very large integer will have a negative effect on performance.
Though there is generally no need to be concerned with an integer getting too large (rollover) or small, be aware that iteratir performing arithmetic with very large positive or small (negative) integers will hurt performance.
Use sys.maxsize to determine the maximum value a variable of type Py_ssize_t can take. Usually on a 32-bit platform, the valu - 1 on a 32-bit platform and 2**63 - 1 on a 64-bit platform.
Use the intern() function to enforce optimization when memory optimization is required for non-simple strings.
Prefer the use of equality (==) to identity (is) and clearly document any use of identity.
Consider using the id function to test for object equality.
Do not depend on the sequence of keys in a dictionary to be consistent across implementations, or even between multiple e with the same implementation, in versions prior to Python 3.7.
Do not modify the dictionary object returned by a vars() and locals() call.
Verify that all sections of code that have access to critical sections check for a lock prior to accessing the resource.
Verify that all sections of code that have critical sections check the related lock prior to entering the critical section, includir known to be unsynchronized.
Protect data that would be vulnerable to premature termination, such as by using locks or protected regions, or by retaining consistent version of the data (checkpoints).
Do not yield within critical sections.
To obtain deep copies at all levels of a variable, use the copy.deepcopy standard library function.
Be aware the "slice" operator "[:]" and the container copy() methods only perform shallow copies.
Be aware of the potential consequences of precision loss when converting from floating-point to integer.
Pay special attention to issues of magnitude and precision when using mixed type expressions.

When high performance is dependent on knowing the range of integer numbers that can be used without degrading perform the sys.int_info struct sequence to obtain the number of bits per digit (bits_per_digit) and the number of bytes used to repre digit (sizeof_digit).
Be aware that a function is defined dynamically so its composition and operation may vary due to variations in the flow of co within the defining program.
Avoid using ctypes when calling C code from within Python and use cffi (C Foreign Function Interface) instead.
Add parentheses after a function call in order to invoke the function.
Do not override the names of built-in variables or functions.
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 al properties of the built-in intrinsic.
Avoid overriding Python’s default behaviour provided by the builtins module.
Avoid assignment to a variable equally named as the loop index counters within the loop.
Avoid using assignment expressions in the loop control statement (that is, while or for).
Avoid using floating-point or decimal variables for loop control but if one of these types must be used, then bound the loop so as to not exceed the maximum or minimum possible values for the loop control variables.
Be careful to only modify variables involved in loop control in ways that are easily understood and in ways that cannot lead t premature exit or an endless loop.
Be careful that a loop will always end when the loop index counter value is one less than the ending number of the range.
Ensure that there is only one asyncio event loop per program, although multiple events can be activated within the single lo Python event loops are automatically generated by asyncio.run().
Restructure code so that the nested loops that are to be collectively exited form the body of a function, and use early functio to exit the loops. This technique does not work if there is more complex logic that requires different levels of exit.
Use the for statement to execute over whole constructs in preference to loops that index individual elements.
Use either spaces or tabs, not both, to demark control flow.
Always use either spaces or tabs (but not both) for indentations.
Do not use form feed characters for indentation.
Consider using a text editor to find and make consistent, the use of tabs and spaces for indentation.
Note: Python 3.0+ will refuse to compile code that uses a mixture of tabs and spaces for indentation.
Adhere to Python’s naming conventions.
Avoid names that differ only by case unless necessary to the logic of the usage, and in such cases document the usage.
Do not use identical names unless necessary to reference the correct object.
Do not use overly long names.
Use meaningful names.
Use names that are clear and visually unambiguous because the compiler cannot assist in detecting names that appear simila different.
Use names that are not similar (especially in the use of upper and lower case) to other names.
For more guidance on Python’s naming conventions, refer to Python Style Guides contained in “PEP 8 – Style Guide for Pyth
Understand or eliminate all confusing Unicode characters, in particular, homoglyphs.
Use caution when copying and pasting Unicode text.
Ensure that ‘show-all-hidden-characters’ is enabled in the editor.

Adjust the maximum recursion depth to an appropriate value as needed.
Prefer iteration to recursion, unless it can be proved that the depth of recursion can never be large.
Avoid any unprotected settings from the working environment in an entry point.
Use the debug mode of the Python interpreter to detect concurrency errors .
Be aware of the consequences of shared references. See clause 6.24 Side-effects and order of evaluation of operands [SAM] an Deep vs. shallow copying [YAN].
Be aware that subsequent imports have no effect; use the reload statement instead of import if a fresh copy of the module is
Avoid rebinding except where it adds identifiable benefit.
Assume that when examining code, that a variable can be bound (or rebound) to another object (of same or different type) a
Use sys.byteorder to determine the native byte order of the platform.
Localize and document the code associated with explicit manipulation of bits and bit fields.
Code algorithms to account for the fact that results can vary slightly by implementation.
Consider using ResourceWarning to detect implicit reclamation of resources.
Ensure that all shared resources locked by the thread or process are released upon termination, for example, in an exception and/or in a finally block.
Design the code to be fail-safe in the presence of terminating processes, threads or tasks.
Avoid external termination of concurrent entities except as an extreme measure, such as the termination of the program.
Do not alter the __class__ attribute for instances of a class unless there are compelling reasons to do so. If alterations are requ document the reasons in docstring and local comments.
Do not assign new values to NotImplemented, Ellipsis or __debug__.
Do not write Python extension modules by hand, as doing so is error-prone, and highly likely to lead to reference counting e memory leaks, dangling pointers, out-of-bounds memory accesses, and similar problems.
Ensure that it is not logically possible to reach a reference to a variable before it is assigned to avoid the occurrence of a runti
Use context managers to explicitly release large memory buffers that are no longer needed.
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 2 1:2019, to show that memory is effectively reclaimed and reused.
If coding an extension, utilize Python's extension API to ensure a correct signature match.
If data accesses need to be serialized, ensure that they reside in the same thread, or provide explicit synchronization among t or processes for the data accesses.
When using the import statement, rather than use the from X import * form (which imports all of module X's attributes into importing program's namespace), instead explicitly name the attributes that need to be imported (for example, from X impo so that variables, functions and classes are not inadvertently overlaid.
Import just the attributes that are required by using the from statement to avoid adding dead code.
Make sure that each class calls the __init__ of its superclass.
Make sure that each class implements the __init__ method that calls the __init__ of its superclass.
Pay attention to warnings that identify variables written but never read.

Understand the difference between equivalence and equality and code accordingly.
Ensure that a function is defined before attempting to call it.
Use or develop ‘units’ libraries to handle conversions between differing unit-based systems.
Design coding strategies that allow the distinction of semantically incompatible types.
Use qualification when necessary to ensure that the correct variable is referenced.
Use the assert statement during the debugging phase of code development to help eliminate undesired conditions from occu
Do not change the size of a data structures while iterating over it. Instead, create a new list.
Be aware of Python’s short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean ex
Use the break statement judiciously to exit from control structures and show statically that the code behaves correctly in all
Use the full path name for imports, in preference to relative paths.
Uses types.MappingProxy or collections.ChainMap to provide read-only views of mappings without the cost of making a cop
Where available, use existing interface libraries that bridge between Python and the extension module language, for exampl Rust, pybind11 for C++.
When accessing data objects directly by using memoryview(), make sure that the data pointed to remains valid until it is no l needed.
When interfacing with external systems or other objects where the declaration order of class members is relevant, use __pre obtain the desired order for class member creation.

Summaries (High-Risk Only)
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).
Avoid mixing concurrency models within the same program, or if unavoidable, use with extreme caution.
Avoid the use of pickle, but if it must be used, only unpickle trusted data.
Avoid using exec or eval and never use these with untrusted code.
Avoid mixing the use of auto() for enums with manual assignments, and when indexing into a list.
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.
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.
Be cognizant that most arithmetic and bit manipulation operations on non-integers have the potential for undetected wrap-around errors.
Use type annotations to help provide static type checking prior to running code.
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.

When using asyncio, make all tasks non-blocking
Inherit only from trusted classes and only use multiple inheritance that is linearizable with the MRO rules.
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”.
When using multiple threads, verify that all shared data is protected by locks or similar mechanisms, and use inter-thread or inter-process communication mechanisms to ensure safe terminations.

Notes	Risk	Occurrences	Severity	Mitigation	Uniqueness
Many of these mitation strategies are common for other languages and improper handling of global variables can lead to race conditions. Python does have unique guidance pertaining to threads such as threading_local().	High	7	x	x	x
Mixing concurrency models in Python is an advanced technique that can be useful but can also increase the risk	High	1	x	x	x
Red-box warning in the docs. The pickle module is not secure. Only unpickle data you trust. The best safe mitigation strategy for pickle is to avoid it all together if possible	High	3	x	x	x
All Python uses of exec are vulnerable	High	2	x	x	x
Mixing auto() with manual assignments can be prone to error and, if values are assigned manually, care must be taken to ensure that there are no repeat values since only the first unique value is recognized and all subsequent repeated vales are ignored.	High	4	x	-	x

	High	8	x	x	x
	High	1	x	-	x
Altering the behavior of objects at runtime can sometimes be useful but but it can make code much more difficult to understand and easily intorduce vulnerabilities	High	1	x	-	x
Normally the OverflowError exception is raised for floating-point wrap-around errors but, for implementations of Python written in C, exception handling for floating-point operations cannot be assumed to catch this type of error because they are not standardized in the underlying C language. Because of this, most floating-point operations cannot be depended on to raise this exception.	High	1	x	x	x
	High	1	x	-	x
Manually added hints in Pyton can enable 3rd-party static analysis tools.	High	9	x	x	x
These mitigations are common to many languages but implemented uniquely in Python. Unhandled exceptions can result in vulnerabilities.	High	9	x	x	x

asyncio offers many powerful capabilities but can be fairly complicated to implement correctly	High	6	x	x	x
	High	7	x	x	x
	High	7	x	x	x
MRO in Python is very different that most other languages and can be complicated to follow	High	3	x	x	x
Some of these mitigations are common among other languages, but there are also Python-specific mechanisms required to avoid potentially serious vulnerabilities such as race conditions	High	4	x	x	x
Using pipes in Python can be complicated and must be used carefully to avoid vulnerabilities.	High	1	x	x	x
	Low	3	-	-	x
	Med	2	x	-	-

	Med	2	x		
	Med	4	x	-	x
	Med	4	x	-	x
	Med	1	-	-	-
Python uniquely implemetns format strings however the general guidelines appropiate for most other languages apply.	Med	3	x	-	x
	Med	5	-	-	x
	Med	3	-	-	x
	Med	5	-	-	x
Identifying and protecting critical sections is crucial but commonly done for other languages	Med	4	-	-	-
Best Practice, this mitigation should be routine in Python	Med	2	x	-	x
Knowledge that Python uses integers that are virtually unlimited helps					

Knowledge that Python uses integers that are virtually unlimited helps to mitigate vulnerabilities.	Med	3	x	-	x
Knowledge that Python functions are defined dynamically helps to mitigate vulnerabilities.	Med	1	x	-	x
cffi (modules converted to Python) is preferred; ctypes are error-prone and often easily avoided	Med	1	x	-	x
Excluding parentheses "()" after a function name refers to that function by it's object name and is sometimes required to achieve the desired functionality; including parentheses in these situations may introduce vulnerabilities.	Med	1	x	-	x
Python provides a set of built-in intrinsics, which are implicitly imported into all Python scripts. Any of the built-in variables and functions can therefore easily be overridden and result in vulnerabilities.	Med	4	x	-	x
Many of these loop mitation strategies are common for other languages except for the `asyncio` event loop mitigation which is unique to Python. Improper handling of loops can often lead to unexpected vulnerabilities but can be readily avoided.	Med	8	x	-	x
Python 3 does not permit the mixing of tabs and spaces and will throw an error at runtime. Additionally, many Python IDEs will alert the developer of mixed spaces and tabs, or automatically perform the conversion to one or the other.	Low	5	-	-	x
Python’s naming rules are flexible by design but are also susceptible to a variety of unintentional coding errors. However, using care while creating names will mitigate the occurance of unexpected, and potentially vunlerable, behavior.	Low	11	-	-	x

Recursion is supported in Python and is, by default, limited to a depth of 1,000, which can be overridden using the `setrecursionlimit` function. If the limit is set high enough, a runaway recursion could exhaust all memory resources leading to a denial of service, but this is very easy to control.	Low	2	-	-	x
Best Practice, this mitigation should be routine in Python	Low	1	-	-	x
	Low	1	-	-	x
Basic knowledge of Python will mitigate most of the vulnerabilities associated with shared references.	Low	1	x	-	x
Using reload instead of import when appropiate is unique to Python but easily implemented. Understanding that variables in Python are simply labels with no intrisic value significantly increases the effectiveness of this mitigation.	Low	3	-	-	x
sys.byteorder and adequate documentation mitigates these vulnerabilities	Low	2	-	-	x
	Low	1	-	-	x
	Low	2	-	-	-
Python does not have native calls to terminate certain concurrent entities sucs as threads, so this is less likely to occur	Low	2	-	-	x
	Low	1	-	-	x
	Low	1	-	-	x
	Low	1	-	-	-
Straightforward to implement and can be identified with tools	Low	1	-	-	-
	Low	3	-	-	x
	Low	1	-	-	x
	Low	1	-	-	-
Reduces dead code	Low	2	-	-	x
	Low	2	-	-	x
	Low	1	-	-	-

	Low	2	-	-	-
	Low	2	-	-	-
	Low	1	-	-	-
There are numerous scenarios in Python that can have unexpected side effects.	Low	3	x	-	x
	Low	1	-	-	-
	Low	1	-	-	-
	Low	1	-	-	x
	Low	1	-	-	x
	Low	1	-	-	x