Baseline Edition **TR 24772–4**

ISO/IEC JTC 1/SC 22/WG23 N1235

Date: 2022-11-16

ISO/IEC WD 24772–4

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Deleted: 1450

Deleted: 03-09

Programming languages — Avoiding vulnerabilities in programming languages – Part 4: Catalogue of vulnerabilities for the programming language Python

Document type: International standard
Document subtype: if applicable
Document stage: (10) development stage
Document language: E

Élément introductif — Élément principal — Partie n: Titre de la partie

WG 23/N1235

Deleted: 150

Participating in writeup 12 December 2022
Stephen Michell – convenor WG 23
Larry Wagoner - USA
Sean McDonagh – USA
Erhard Ploedereder
Tullio Vardanega – Italy

Deleted: ¶
¶
¶
¶

Regrets

Based on Document N1230 with edits by SJM and LDW and at meeting 14 Dec 2022.

All issues discussed are captured in the document, either as comments or resolved issues. The previous version of this document is N1207.

Deleted: 147.

Key for comments:
X xx – needs to be addressed
Y yy – addressed, need group to review
E ee – comment asks Erhard to address
L ll – comment asks Larry to address
N nn – comment asks Nick to address
S ss – comment asks Sean to address
T tt – comment asks Stephen to address

ISO copyright office
Case postale 56, CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Deleted: 150

CONTENTS

Deleted: 16

Deleted: 17

Deleted: 17

Deleted: 17

Deleted: 18

Deleted: 19

Deleted: 22

Deleted: 22

Deleted: 22

Deleted: 23

Deleted: 23

Deleted: 24

Deleted: 25

Deleted: 27

Deleted: 27

Deleted: 30

Deleted: 31

Deleted: 32

Deleted: 32

Deleted: 32

Deleted: 33

Deleted: 33

Deleted: 34

Deleted: 34

Deleted: 35

Deleted: 36

Deleted: 36

Deleted: 38

Deleted: 39

Deleted: 39
Deleted: 41
Deleted: 45
Deleted: 45
Deleted: 46
Deleted: 49
Deleted: 50
Deleted: 51
Deleted: 51
Deleted: 52
Deleted: 53
Deleted: 54
Deleted: 55
Deleted: 58
Deleted: 58
Deleted: 60
Deleted: 60
Deleted: 60
Deleted: 61
Deleted: 62
Deleted: 63
Deleted: 63
Deleted: 66
Deleted: 66
Deleted: 67
Deleted: 70
Deleted: 71
Deleted: 72
Deleted: 72
Deleted: 73
Deleted: 74
Deleted: 74
Deleted: 75
Deleted: 75

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772-4 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Deleted: 150

This document is part of a series of documents that describe how vulnerabilities arise in programming languages. ISO/IEC 24772-1 addresses vulnerabilities that can arise in any programming language and hence is language independent. The other parts of the series are dedicated to individual languages.

This document provides guidance for the programming language Python, so that application developers considering Python or using Python will be better able to avoid the programming constructs that can lead to vulnerabilities in software written in the Python language and their attendant consequences. This document can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This document can also be used in comparison with companion documents and with the language-independent report, ISO/IEC 24772-1, *Information Technology – Programming Languages— Guidance to avoiding vulnerabilities in programming languages,* to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

It should be noted that this document is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such document can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

Programming Languages — Avoiding vulnerabilities in programming languages — Part 4: Catalogue of vulnerabilities for programming language Python

1. Scope

This Technical Report specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.
Vulnerabilities are described in this Technical Report document the way that the vulnerability described in the language-independent TR 24772–1 are manifested in Python.
Python is not an internationally specified language, in the sense that it does not have a single International Standard specification. The language definition is maintained by the Python Software Foundation at https:python.org for the version of Python referenced in this document.
The analysis and guidance provided in this document is targeted to Python version 3.10.4, available from
https://www.python.org/doc/versions/?msclkid=72795f4dd15811ec9e440b65e4f93088
Implementations of earlier versions of Python exist and are in active usage, however, Python is not always backward compatible especially between v2.x and v3.x. Readers are cautioned to be aware of the differences as they apply to guidance provided herein. To determine possible vulnerabilities for future releases of Python, research the documentation on the Python web site given above.

2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.
ISO/IEC/IEEE 60559:2011 Information technology -- Microprocessor Systems -- Floating-Point arithmetic
ISO/IEC 10967-1:2012 Information technology -- Language independent arithmetic -- Part 1: Integer and floating point arithmetic
ISO/IEC 10967-2:2001 Information technology -- Language independent arithmetic -- Part 2: Elementary numerical functions
ISO/IEC 10967-3:2006 Information technology -- Language independent arithmetic -- Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions

"The Python Language Reference", https://docs.python.org/3/reference

"The Python Standard Library", https://docs.python.org/3/library/index.html

"Python/C API Reference Manual", http://docs.python.org/py3k/c-api

"Embedding Python in Another Application",
http://docs.python.org/3/extending/embedding.html

Commented [WLD1]: Yyy Need decision on whether we are putting in a version number or simply stating that this annex is targeted at the latest version.

Commented [SM2R1]: We probably should refer to the latest version published just before we publish.

Commented [WLD3R1]: Ok. Consider this a note to do that just before we publish.

Commented [MS4R1]: The latest version of Python, as of 12-08-2021 is **v3.10**, so v3.8 is obsolete if we do decide to include the version number.

Also, we should probably include disclaimers for *other* implementations of Python such as **Jython**, **IronPython**, and **PyPy**. Some implementations have significant differences from the standard CPython version. For example, Jython and IronPython don't have a GIL so this changes how concurrency is handled.

Commented [WLD5]: Yyy Copied these paragraphs from the Java annex. Only change was changing the word "Java" to "Python" and other minor modifications.

Commented [p6]: Stands at 3.9

3. Terms and definitions, symbols and conventions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382:2015, TR 24772–1:2019, and the following apply. Other terms are defined where they appear in *italic* type. ISO and IEC maintain terminology databases for use in standardization are available at:

- IEC Glossary, std.iec.ch/glossary
- ISO Online Browsing Platform, www.iso.ch/obp/ui

3.1 assignment statement
statement that assigns an object to a name (variable)
Note: The simple syntax is `a = b`, the augmented syntax applies an operator at assignment time (for example, `a += 1`) and therefore cannot create a new variable reference since it operates using the current value referenced by a variable. Other syntaxes support multiple targets, that is,

```
x = y = z = 1
```

binding (or rebinding) an instance attribute, that is,

```
x.a = 1
```

and binding (or rebinding) a container element, that is,

```
x[k] = 1
```

).

3.2 body
the portion of a compound statement that follows the header. It may contain other compound (nested) statements

3.3 boolean
truth value where `True` corresponds to any non-zero value and `False` corresponds to zero
Note: Commonly expressed numerically as 1 (true), or 0 (false) but referenced as `True` and `False`.

3.4 built-in
function provided by the Python language intrinsically without the need to import it (for example, `str, slice, type`)

3.5 class
program defined type which is used to instantiate objects and provide attributes that are common to all the objects that it instantiates

3.6 comment
information for readers that is ignored by the language processor
Note: Comments are preceded by a hash symbol "#".

3.7 complex number
number made up of two parts each expressed as floating-point numbers, a real and an imaginary part, in which the imaginary part is expressed with a trailing upper or lower case "J"

3.8 compound statement
program structure that contains and controls one or more statements

3.9 CPython
the standard implementation of Python coded in ANSI portable C

3.10 dictionary
built-in mapping consisting of zero or more key:value "pairs"
Note: Values are stored and retrieved using keys which can be of mixed types (with some caveats beyond the scope of this annex). The contents of a dictionary are ordered, changeable, and cannot contain duplicates.

3.11 docstring
one or more lines in a unit of code that serve to document the code
Note: Docstrings are retrievable at run-time and surround the documentation text by `'''` three single quotes`'''` or `"""`three double quotes`"""`

3.12 exception
object that encapsulates the attributes of an error or abnormal event
Note: Raising an exception is a process that creates the exception object and propagates it through a process that is optionally defined in a program. Lacking an exception 'handler", Python terminates the program with an error message.
3.13 floating-point number
real number expressed with a decimal point, an optional exponent expressed as an upper or lower case ”e” or “E” or both
Note: for example, `1.0, 27e0, .456`
3.14 function
a grouping of statements, either built-in or defined in a program using the `def` statement, which can be called as a unit
3.15 garbage collection
process by which the memory used by unreferenced objects and their namespaces is reclaimed
Note: Python provides a `gc` module to allow a program to direct when and how garbage collection is done.
3.16 global
variable that is scoped to a module and can be referenced from anywhere within the module including within functions and classes defined in that module
3.17 guerrilla patching
changing the attributes and/or methods of a module’s class at run-time from outside of the module
Note: Colloquially known as Monkey Patching.
3.18 immutable
unchangeable within a single execution of the program
Note: int, float, bool, str, and tuples are immutable objects in Python.
3.19 import
mechanism that is used to make the contents of a module accessible to the importing program
3.20 inheritance
definition of a class as a subclass of other classes such that inheriting class acquires methods and components from the superclass without explicitly defining them
Note: Inheritance uses a method resolution order (MRO) to resolve references to the correct inheritance level (that is, it resolves attributes (methods and variables)).
3.21 instance
single occurrence of a class that is created by calling the class as if it was a function (for example, `a = Animal()`
3.22 integer
a whole number of any length
Note: An integer can be of any length but is more efficiently processed if it can be internally represented by a 32 or 64 bit integer. Integer literals can be expressed in binary, decimal, octal, or hexadecimal formats.
3.23 keyword

identifier that is reserved for special meaning to the Python interpreter and that cannot be used as a name of an object or a function or a method (for example, `if`, `else`, `for`, `class`)

3.24 lambda expression
single return function statement within another statement instead of defining a separate function and referencing it
Note: Example of a lambda function:

```
x = lambda a : a + 10
print(x(15))
```
The print statement will print out 25.

3.25 list
ordered sequence of zero or more items which can be modified (that is, is mutable) and indexed

3.26 literal
string or number (for example, `'abc', 123, 5.4`)
Note: A string literal can use either double quote (“) or single apostrophe pairs (‘) to delimit a string.

3.27 membership
property of belonging by occurring in a sequence
Note: Python has built-ins to test for membership (for example, `if a in b`). Classes can provide methods to override built-in membership tests.

3.28 module
file containing source language or statements in Python or in another language and that has its own namespace and scope and may contain definitions for functions and classes
Note: A module is only executed when first imported and upon reloading.

3.29 mutability
characteristic of being changeable
Note: Lists and dictionaries are two examples of Python objects that are mutable.

3.30 name
Reference to a Python object such as a number, string, list, dictionary, tuple, set, built-in, module, function, or class

3.31 namespace
place where names reside with their references to the objects that they represent
Note: Examples of objects that have their own namespaces include: blocks, modules, classes, and functions. Namespaces provide a way to enforce scope and thus prevent name collisions since each unique name exists in only one namespace.

3.32 none
null object

3.33 number
integer, floating point, decimal, or complex number

3.34 operator
symbol that represents an action or operation on one or more operands
Note: For example `*` is an arithmetic operator that represents multiplication

3.35 overriding
attribute in a subclass to replace a superclass attribute

3.36 package:
collection of one or more other modules in the form of a directory

3.37 pickling
process of serializing objects using the `pickle` module

3.38 polymorphism
meaning of an operation (generally a function/method call) that depends on the objects being operated upon, not the *type* of object
Note: One of Python’s key principles is that object interfaces support operations regardless of the type of object being passed. For example, string methods support addition and multiplication just as methods on integers and other numeric objects do.

3.39 recursion
the ability of a function to call itself

Note: Python supports recursion to a level of 1,000 unless that limit is modified using the `setrecursionlimit` function.

3.40 scope

program region where a name is available for use within the overall program

Note: All names within Python exist within a specific namespace which is tied to a single block, function, class, or module in which the name was last assigned a value.

3.41 script

unit of code generally synonymous with a *program* but usually connotes code run at the highest level

Note: As in "*scripts run modules"*.

3.42 self

name given to a class' instance variable

3.43 sequence

ordered container of items that can be indexed or sliced using positive numbers

Note: Python provides three built-in sequences: strings, tuples, and lists. New sequences can also be defined in libraries, extension modules, or within classes.

3.44 set

unordered sequence of zero or more items which do not need to be of the same type

Note: Sets can be frozen (immutable) or unfrozen (mutable).

3.45 short-circuiting operator

behaviour of the operators `and` and `or` where the evaluation of the right-hand expression can be skipped if the left side evaluates to true in the case of the `or` or false in the case of `and`

Note: For example, in the expression

```
a or b,
```

there is no need to evaluate `b` if `a` is `True`, likewise in the expression

```
a and b,
```

there is no need to evaluate `b` if `a` is `False`.

3.46 statement

expression that generally occupies one line

Note: Multiple statements can occupy the same line if separated by a semicolon (`;`) but this is very unconventional in Python where each line typically contains one statement.

3.47 string

built-in sequence object consisting of one or more characters

Note: Unlike many other languages, Python strings cannot be modified (that is, they are "immutable") and do not have a termination character.

3.48 tuple

an immutable sequence of Python objects

Note: For example, `a, (a,),a,b,c`, `(1,2,3)` or `("A", "B", "C")`. Tuples may contain different object types (for example, `(1, "a", 5.678)`).

3.49 variable

a reference to the memory location of an object that contains a value

Note: Python variables (names) are not like variables in most other languages - they are dynamically referenced to objects. Python allows optional explicit type declarations to be added to variables, function parameters and return values. The Python language itself does not enforce these annotations but they can be used by third-party type checkers, as well as IDEs. Any Python variable may be reassigned to objects of different types at different times.

4. Using this document

ISO/IEC 24772-1:20xx clause 4.2 documents the process of creating software that is safe, secure and trusted within the context of the system in which it is fielded. As this document shows, vulnerabilities exist in the Python programming environment, and organizations are responsible for understanding and addressing the programming language issues that arise in the context of the real-world environment in which the program will be fielded.
Organizations following this document, meet the requirements of clause 4.2 of ISO/IEC 24772-1, repeated here for the convenience of the reader:

1. Identify and analyze weaknesses in the product or system, including systems, subsystems, modules, and individual components;
2. Identify and analyze sources of programming errors;
3. Determine acceptable programming paradigms and practices to avoid vulnerabilities using guidance drawn from clauses 5.3 and 6 in this document;
4. Determine avoidance and mitigation mechanisms using clause 6 of this document as well as other technical documentation;
5. Map the identified acceptable programming practices into coding standards;
6. Select and deploy tooling and processes to enforce coding rules or practices;
7. Implement controls (in keeping with the requirements of the safety, security and general requirements of the system) that enforce these practices and procedures to ensure that the vulnerabilities do not affect the safety and security of the system under development.

Tool vendors follow this document by providing tools that diagnose the vulnerabilities described in this document. Tool vendors also document to their users those vulnerabilities that cannot be diagnosed by the tool.
Programmers and software designers follow to this document by following the architectural and coding guidelines of their organization, and by choosing appropriate mitigation techniques when a vulnerability is not avoidable.

5 General language concepts and primary avoidance mechanisms

5.1 General Python language concepts

The key concepts discussed in this section are not entirely unique to Python, but they are implemented in Python in ways that are not always intuitive.

5.1.1 Dynamic Typing

A frequent source of confusion is Python's dynamic typing and its effect on variable assignments (*name* is synonymous with *variable* in this annex). In Python there are no static declarations of variables. Variables are created, rebound, and deleted dynamically. Further, variables are not the objects that they point to - they are just references to objects, which can be, and frequently are, bound to other objects at any time:

```
a = 1 # a is bound to an integer object whose value is 1
a = 'abc' # a is now bound to a string object
```

In Python, variables have no type – they reference objects which have types thus the statement `a = 1` creates a new variable called `a` that references a new object whose value is `1` and type is

integer. That variable can be deleted with a `del` statement or bound to another object any time as shown above. Refer to clause 6.2 Type system [IHN] for more on this subject. For the purpose of brevity, this annex often treats the term variable (or name) as being the object, which is technically incorrect but simpler. For example, in the statement `a = 1`, the numeric object `a` is assigned the value `1`. In reality the name `a` is assigned to a newly created *object* of type integer which is assigned the value `1`.
Even when explicit type declarations are present, they are not checked at runtime, and are instead checked using separate typechecking tools. The following code will execute without any problems, but the assignment of a string to a variable explicitly declared as holding an integer will cause static type analysis to fail:

```
a: int = 1 # Programmer declares a will always refer
           # to an int object
a = 'abc'  # Typechecker reports error when a is bound
            #to a string object
```

Similarly, there is no type checking for argument passing to user-defined functions and methods. Type errors are diagnosed during the execution of the function or method when an illegal operation is attempted or a call is made to a function or method that is not defined.

5.1.2 Mutable and Immutable Objects

Note that in the statement: `a = a + 1`, Python creates a *new* object whose value is calculated by adding `1` to the value of the current object referenced by `a`. If, prior to the execution of this statement `a`'s object had contained a value of `1`, then a new integer object with a value of `2` would be created. The integer object whose value was `1` is now marked for deletion using garbage collection (provided no other variables reference it). Note that the value of `a` is not updated in place, that is, the object referenced by `a` does not simply have `1` added to it as would be typical in other languages. The reason this does not happen in Python is because integer objects, as well as string, number and tuples, are immutable – they cannot be changed in place. Only lists, sets, and dictionaries can be changed in place – they are mutable. In practice this restriction of not being able to change a mutable object in place is mostly transparent but a notable exception is when immutable objects are passed as a parameter to a function or class. See clause 6.22 Initialization of Variables [LAV] for a description of this.
The underlying actions that are performed to enable the *apparent* in-place change do not update the immutable object – they create a new object and bind (or "point") the variable to the new object. This can be shown as below (the `id` function returns an object's address):

```
a = 'abc'
print(id(a)) #=> 30753768
a = 'abc' + 'def'
print(id(a)) #=> 52499320
print(a) #=> abcdef
```

The updating of objects referenced in the parameters passed to a function or class is governed by whether the object is mutable, in which case it is updated in place, or immutable in which case a local copy of the object is created and updated which has no effect on the passed object. This is described in more detail in clause 6.32 Passing Parameters and Return Values [CSJ].

5.1.3 Variables, objects and their values

Python provides the ability to dynamically create variables when they are first assigned to an object. In fact, assignment is the *only* way to bring a variable into existence. Function parameters

are implicitly assigned by the interpreter when the function is called. All values in a Python program are accessed through a variable reference which points to a memory location which is always an object (for example, number, string, list, and so on). A variable is said to be bound to an object when it is assigned to that object. A variable can be rebound to another object which can be of any type. For example:

```
a = 'alpha' # assignment to a string
a = 3.142 # rebinding "a" to a float
a = b = (1, 2, 3) # rebinding to a tuple
print(a) #=> (1, 2, 3)
del a
print(b) #=> (1, 2, 3)
print(a) #=> NameError: name 'a' is not defined
```

The first three statements show dynamic binding in action. The variable `a` is bound to a string, then to a float, then to another variable which in turn is assigned a tuple of value `(1, 2, 3)`. The `del` statement then unbinds the variable `a` from the tuple object which effectively deletes the `a` variable (if there were no other references to the tuple object it too would have been deleted because an object with zero references is *marked* for garbage collection (but is not necessarily deleted immediately)). In this case, we see that `b` is still referencing the tuple object so the tuple is not deleted. The final statement above shows that an exception is raised when an unbound variable is referenced.

The way in which Python dynamically binds and rebinds variables is a source of some confusion to new programmers and even experienced programmers who are used to static binding where a variable is permanently bound to a single memory location.

Variables in an expression are replaced with object references when that expression is evaluated, therefore a variable must be explicitly assigned before being referenced, otherwise a run-time exception is raised:

```
a = 1
if a == 1 : print(b) # error – b is not defined
```

When line 1 above is interpreted an object of type `integer` is created to hold the value `1` and the variable `a` is created and linked to that object. The second line illustrates how an error is raised if a variable (`b` in this case) is referenced before being assigned to an object.

```
a = 1
b = a
a = 'x'
print(a,b) #=> x 1
```

Variables can share references as above – `b` is assigned to the same object as `a`. This is known as a shared reference. If `a` is later reassigned to another object (as in line 3 above), `b` will still be assigned to the initial object that `a` was assigned to when `b` shared the reference, in this case `b` would equal to 1.

The subject of shared references requires particular care since its effect varies according to the rules for in-place object changes. In-places object changes are allowed only for mutable (that is, alterable) objects. Numeric objects and strings are immutable (unalterable). Lists and dictionaries are mutable which affects how shared references operate as below:

```
a = [1,2,3]
b = a
a[0] = 7
print(a) # [7, 2, 3]
```

```
print(b) # [7, 2, 3]
```

In the example above, a and b have a shared reference to the same list object so a change to that list object affects both references. If the shared reference effects are not well understood, the change to b can cause unexpected results.
For further discussion of aliasing, see 6.32 Passing parameters and return values [CSJ] and 6.38 Deep vs shallow copying [YAN]). For further discussion of concurrent access to values, see 6.61 Concurrency - data access [CGX].
The Python language, by design, allows for dynamic binding and rebinding. Because Python performs a syntactic analysis and not a semantic analysis (with one exception which is covered in clause 6.21 Namespace issues [BJL] Applicability to language) and because of the dynamic way in which variables are brought into a program at run-time, the Python language runtimes cannot warn that a variable is referenced but never assigned a value. The following code illustrates this:

```
if a > b:
    import x
else:
    import y
```

Depending on the current value of a and b, either module x or y is imported into the program. If x assigns a value to a variable z and module y references z then dependent on which import statement is executed first (an import always executes all code in the module when it is first imported), an unassigned variable reference exception will or will not be raised.
Programmers can use `ResourceWarning` to detect the implicit cleanup of resources and `tracemalloc` to report the location of the resource allocation.
Python only checks whether a variable already exists when it is encountered in a statement that attempts to access its value. It was intentionally part of the Python language design to resolve names at runtime when they are used. This allows for the scoping semantics where names may be resolved in either the current local scope, an outer lexically nested function scope, the module `global`, or the built-in namespace. Python therefore has no way to know if a variable is referenced before or after an assignment. For example:

```
if y > 0:
    print(x)
```

The above statement is legal even if x has not been previously defined (that is, assigned a value) in the current scope or an outer lexically nested function scope in a way that is visible to the compiler. However, at runtime, an exception `UnboundLocalError` is raised when a local variable is read before it is assigned. The exception is raised only if the statement is executed and `y > 0`, and x is not present in the current local scope, module globals or the built-in namespace. Thus, this scenario would not lend itself to static analysis because, as in the case above, it may be perfectly logical to not ever print x unless `y > 0`, or the program may use means that are opaque to the compiler to ensure that x is available in the module scope or the built-in namespace by the time it is needed (for example, it may be set from another module, or programmatically via the `globals()` built-in).
There is no ability to use a variable with an uninitialized value because *assigned* variables always reference objects which always have a value and *unassigned* variables do not exist. Therefore, Python raises an exception at runtime when an unassigned (that is, non-existent) variable is referenced.
Initialization of function arguments can cause unexpected results when an argument is set to a default object which is mutable:

```
def x(y=[]):
    y.append(1)
    print(y)
x([2]) #=> [2, 1], as expected (default was not needed)
x() # [1]
x() # [1, 1] continues to expand with each subsequent call
```

The behaviour above is not a bug - it is a defined behaviour for mutable objects but it is a very bad idea in almost all cases to assign mutable objects as default values.

5.1.4 Inheritance

Inheritance is a powerful part of Object Oriented Programming (OOP). Python supports single inheritance and multiple inheritance.
Python supports inheritance through a dynamic hierarchical search of class namespaces starting at the class of a given object and proceeding upward through its superclasses. Python supports method overriding; it does not support method overloading by default.
In binding the name of a method call, normally only the name of the called function is considered. As a special case, the decorator `@dispatch` can be used to enable method overloading, but only within the namespace of a single class. The decorator does not allow for overloading of methods along an inheritance hierarchy. Consider:

```
from multipledispatch import dispatch

@dispatch(int,int)
def product(first,second):
    result = first*second
    print(result);

@dispatch(float,float,float)
def product(first,second,third):
    result  = first * second * third
    print(result);

product(2,3) # => 6
product(2.2,3.4,2.3) # => 17.204
```

Without the "@dispatch" decorators, only the second method 'product' would be considered in subsequent name binding. With the decorators, the types of the parameters are taken into account as well in binding the method name of a call.
As the name resolution takes only the method name into account, a method definition either redefines (hides) an equally named inherited method of the class of the object or, if none is found, it represents a new method.

```
class A:
    def method1(self):
        print('method1 of class A')

class B(A):
    def method1(self):
        print('Modified method1 of class A by class B')
b = B()
b.method1() #=> Modified method1 of class A by class B
```

Multiple inheritance is also supported. Name resolution uses a strategy known as "Method Resolution Order (MRO)". The MRO is also commonly recognized as C3 Linearization. For

simpler cases that do not involve “diamond structures” i.e. superclasses that are shared by other superclasses, the MRO generally follows a depth-first, left-to-right ordering protocol resulting in a single path through the inheritance tree. For diamond structures, the rules become more complicated as shown by the examples below. In these cases, the MRO is difficult to establish manually and its outcome differs substantially from the usual rules in other OO-languages. In general, the MRO lookup sequence for binding names in classes is a mixture of left-most depth-first and selective breadth-first traversal, the latter ensuring that all search paths back to a given parent node are explored before this parent node is visited.

Consider the following example of multiple inheritance:

```
class A:
    def __init__(self):
        self.id = 'Class A'
    def getId(self):
        return "from A " + self.id

class B:
    def __init__(self):
        self.id = 'Class B'
    def getId(self):
        return "from B " + self.id

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)

c = C()
print(c.getId()) # => from A Class B
    # when class C(B,A) is used, the output is -> from B Class B
```

Even though both Class `A` and Class `B` carry a component `id`, the joint child `C` class has a single instance of `id`. Thus, the assignments executed by `A.__init__(self)` and `B.__init__(self)` operate on this single instance overwriting each other.

The built-in function `super()` introduces more flexibility. In Python, `super()` also relies on MRO. Updating the previous example using `super()` is shown below and the output is now “`Class A`”. Reversing the inheritance call to `class C(B, A)` would predictably result in “`Class B`.” The MRO for the scenario below is calculated using the `__mro__` attribute for `class C` resulting in (`C` -> `A` -> `B`). It is important to make sure that each class calls the `__init__` of its superclass so that it is properly initialized.

```
class A:
    def __init__(self):
        super().__init__()
        self.id = 'Class A'
    def getId(self):
        return self.id

class B:
    def __init__(self):
        super().__init__()
        self.id = 'Class B '
    def getId(self):
        return self.id
```

```
class C(A, B):
    def __init__(self):
        super().__init__()
    def getId(self):
        return self.id

c = C()
print(c.getId()) # => Class A
print(C.__mro__) # => (<class '__main__.C'>, <class
'__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

In general, the MRO lookup sequence for binding names in classes is a mixture of left-most depth-first and selective breadth-first traversal; the latter ensuring that all search paths back to a given parent node are explored before this parent node is visited. As noted earlier, in these cases the MRO is difficult to establish manually. Additionally, Python renders certain MRO's illegal which further complicates the understanding of the rules. For example, in a class hierarchy described by

```
class O: pass
class P: pass
class A(P): pass
class B(P): pass
class Z(O): pass
class Y(Z): pass
class W(O): pass

class C(Y, A, B, W): pass # This works fine

c = C()
c.meth()

class C(Z, Y, A, B, W): pass # => TypeError:
        #         Cannot create a consistent method resolution
        #         order (MRO) for bases Z, Y, A, B, W
```

the MRO for resolving the method name `c.meth()` is the linear sequence

`C – Y – Z – A – B – P – W – O – object.`

On the other hand, in the last line above, Python cannot establish a consistent MRO for `class C(Z, Y, A, B, W)`, because `Z` is a superclass of `Y` and Python throws the `TypeError` exception. Notice that `object` is always the last class in every MRO chain.
Note that Python will always diagnose a failure to declare a legal class, as shown above.

5.1.5 Concurrency

Python's `threading` module provides the ability to perform cooperative multithreading from within a single native thread. Due to the restrictions of Python's Global Interpreter Lock (GIL) in some implementations, only one thread at a time is permitted to run. Even though multithreading in those systems based use multiple Central Processing Unit (CPU) cores, it can be useful in situations where the CPU becomes idle such as in I/O-bound applications. It is important to handle

potential thread exceptions when starting new threads, and care needs to be taken so that each thread is only started once.
Python's `multiprocessing` module provides multiprocessing capability that allows independent processes to run on multiple cores. Unlike threading, these independent processes do not have shared memory and are not prone to the relevant data races. It is important to handle potential multiprocessing exceptions when starting new processes, and if a process terminates as the result of an exception, it cannot be restarted .
Python's `asyncio` module is the newest approach to handling asynchronous concurrency, introduced in Python 3.4. This new `asyncio` processing model is typically faster than implementations that use traditional threads and multiprocessing, and it is safer since `asyncio` operations all run in the same thread. Python event loops are automatically generated by `asyncio.run()`." Multiple event loops are possible but not recommended when using `asyncio` as the execution model relies on a single thread, and adding multiple event loops does not provide additional functionality or performance. Note that restrictions on the use of multiple cores mentioned above also apply to `asyncio` operations.
Interprocess communication is almost always necessary in multicore systems. If a program is implemented that uses both processes and event loops, it should be noted that event loops are not a suitable means of interprocess communication. It is plausible, however, that the masters of event loops may need to communicate with one another and this should happen outside of the event loop processing.
A thread with the flag daemon set to true is called a daemon thread and never terminates.
Futures are Python objects that represent the eventual result of asynchronous and concurrent operations. Futures are also available using the `concurrent.futures` module, which provides a common interface for asynchronous execution of threads using `ThreadPoolExecutor`, or processes using `ProcessPoolExecutor`. When executors are used, the overheads of repeatedly creating threads or processes are avoided. For CPU bound tasks, the `ProcessPoolExecutor` class can provide better performance. Futures in asyncio are awaitable objects and are not thread safe. Coroutines `await` on future objects until they provide a valid result, error message, or are cancelled.

Commented [SM7]: "concurrent" rather than "asynchronous?" If it applied to asynch_io only, then async would be ok

Commented [SM8]: SSS – Sean, add words about futures applying to asynchronous.

Commented [MS9R8]: `Ref:` https://docs.python.org/3/library/asyncio-future.html

Also, `concurrent.futures.Future` and `asyncio.Future` are similar but do have differences:
- result() and exception() do not take a timeout argument and raise an exception when the future isn't done yet.
- Callbacks registered with add_done_callback() are always called via the event loop's call_soon_threadsafe().
- This class is not compatible with the wait() and as_completed() functions in the concurrent.futures package.

In my opinion, these differences do not need to be added to the document.

5.2 Primary guidance for Python

5.2.1 Recommendations in interpreting guidance from ISO/IEC 24772-1:2019

Python has some fundamental differences with standard imperative languages, which are the majority of languages covered by these guidance documents, and the general guidance offered by those guidance documents does not always apply to Python.
In such cases, this guidance document will make the recommendation to "follow the applicable guidance of ISO/IEC TR 24772-1:2019 clause 6.x.5", even though that leaves it to the reader to determine what is applicable.

5.2.2 Top avoidance mechanisms

Each vulnerability listed in clause 6 provides a set of ways that the vulnerability can be avoided or mitigated. Many of the mitigations and avoidance mechanisms are common. This subclause provides the most effective and most common mitigations, together with references to which vulnerabilities they apply. The references are hyperlinked to provide the reader with easy access

to those vulnerabilities for rationale and further exploration. The mitigations provided here are in addition to the ones provided in ISO/IEC TR 24772-1:2019, clause 5.4.
The expectation is that users of this document will develop and use a coding standard based on this document that is tailored to their risk environment.

Number	Recommended avoidance mechanism	Reference(s)
1	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]
2	Use type annotations to help provide static type checking prior to running code.	6.5 [CCB], 6.2 [IHN], 6.11 [HFC]
3	Avoid the use of `auto()` for enums intended to be used for indexing into lists.	6.5 [CCB]
4	Assume that when examining code, that a variable can be bound (or rebound) to another object (of same or different type) at any time.	6.18 [WXQ]
5	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]
6	Use Python's built-in documentation (such as docstrings) to obtain information about a class' method before inheriting from it	6.41 [RIP]
7	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]
8	When using multiple threads, check for race conditions and deadlocks by using fuzzing techniques during development.	6.61 [CGX], 6.63 [CGM]
9	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]

Commented [SM10]: yyy Ensure that all of the recommendations are substantiated in 6.x for all items in this table.

Commented [WLD11R10]: Need to defer action on this until the table is close to finalized and we are removing the last of the comments.

Commented [WLD12R10]: Reviewed and corrected list.

Commented [p13]: Still open

6. Specific Guidance for Python

6.1 General

This clause contains specific advice for Python about the possible presence of vulnerabilities as described in ISO/IEC TR 24772-1:2019 and provides specific guidance on how to avoid them in Python code. This section mirrors ISO/IEC TR 24772-1:2019 clause 6 in that the vulnerability "Type system [IHN]" is found in 6.2 of ISO/IEC TR 24772-1:2019, and Python specific guidance is found in clause 6.2 and subclauses in this document.

Note that the guidance provided in this document applies to Python as specified in the Python 3.9.0 documentation. Python is extended by a number of commonly used libraries that can have behaviours different from those documented by the Python standard. This document does not address these additional libraries.

6.2 Type system [IHN]

6.2.1 Applicability to language

The vulnerabilities related to insufficient use of the type system as specified in ISO/IEC TR 24772-1:2019 clause 6.2 apply to Python.
Python abstracts all data as objects and every object has a type (in addition to an identity and a value). Extensions to Python, written in other languages, can define new types, and Python code can also define new types, either programmatically through the types module, or by using the dedicated class statement.
Python is also a strongly typed language – operations cannot be performed on an object that is not valid for that type. Checks performed to ensure an appropriate type are performed dynamically when the operation on the object is invoked. For operations that are not valid for the type an exception will be raised at runtime. Programmers can use `isinstance(), type(),` and other behavioural based type checkers to verify that the type is valid or convertible, and then convert to the desired type. In many cases, the conversion call is the type check (e.g. `itr = iter(arg)` is a common way of accepting any iterable as input, and throwing `TypeError` otherwise).

```
a = 'abc' # a refers to a string object
if isinstance(a, str): print('a type is string')
```

By default, a Python program is free to assign (bind), and reassign (rebind), any variable to any type of object at any time. This is considered safe in general since the type of the object is carried in the object and if a variable is rebound, then any future calls using that variable will check the type recorded in the object to decide the validity of the operation. Reference clause 6.36 Ignored error status and unhandled exceptions [OYB] for a discussion of the vulnerabilities associated with failed checks.
Variables are created when they are first assigned a value (see clause 6.17 Choice of clear names [NAI] for more on this subject). Variables are generic in that they do not have a type. They simply reference objects which hold the object's type information.
Automatic conversion occurs only for numeric types of objects. Python converts (coerces) from the simplest type up to the most complex type whenever different numeric types are mixed in an expression. For example:

```
a = 1
b = 2.0
c = a + b; print(c) #=> 3.0
```

In the example above, the + operation converts the value of `a` to its floating point equivalent, `1.0`, adds it to `b`, and stores the floating-point value, `3.0`, into `c` (which is thus a floating-point number). A programmer may erroneously expect that `c` is an integer and use it accordingly which can lead to unexpected results.
Some of these issues are visible to the programmer. For example, `x = 1/2` will create an object of type float with a numeric value of `0.5`, while `x = 1//2` will truncate to the integer `0`.

Gradual typing in Python allows optional annotations to be added to dynamic variables to assign them types so that they can be statically checked. This lets Python programs contain both dynamic variables, while adding the error-checking benefits of static variables. Python tools provide static type checkers that assist users in avoiding the misuse of declared types in Python. Python also has the issue that change of logical representation (e.g. meters to feet) are not enforced by the general type system Programmers can use dedicated libraries to manage such types or can create their own using classes.

6.2.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.2.5.
- 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 representations [STR]

6.3.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.3 applies to Python. Python provides hexadecimal, octal and binary built-in functions. `oct` converts to octal, `hex` to hexadecimal and `bin` to binary:

```
print(oct(256)) # 0o400
print(hex(256)) # 0x100
print(bin(256)) # 0b100000000
```

The notations shown as comments above are also valid ways to specify octal, hex and binary values respectively:

```
print(0o400) #=> 256
a = 0x100+1; print(a) #=> 257
```

The built-in `int` function can be used to convert strings to numbers and optionally specify any number base:

```
int('256') # the integer 256 in the default base 10
int('400', 8) #=> 256
int('100', 16) #=> 256
int('24', 5) #=> 14
```

Python stores integers that are beyond the implementation's largest integer size as an internal arbitrary length so that programmers are only limited by performance concerns when very large integers are used (and by memory when extremely large numbers are used). For example:

```
a=2**100 #=> 1267650600228229401496703205376
```

Python is not susceptible to the vulnerability associated with shifting the underlying number as described in ISO/IEC TR 24772-1:2019 clause 6.3 because Python treats positive integers as

being infinitely padded on the left with zeroes and negative numbers (in two's complement notation) with 1's on the left when used in bitwise operations:

```
a<<b # a shifted left b bits
a>>b # a shifted right b bits
```

There is no overflow check required for left shifts since bits are added as required. For right shifts of positive numbers, the result will decrease by powers of two with a limit of zero. Note that right shifts of negative numbers eventually result in -1 if the number of positions shifted is sufficiently large.
The vulnerability associated with endianness can be mitigated by identifying the endian protocol. Use `sys.byteorder` to determine the native byte order of the platform. The call returns `big` or `little`.

6.3.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.3.5
- 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 [PLF]

6.4.1 Applicability to language

The vulnerabilities described in ISO/IEC TR 24772-1:2019 clause 6.4 apply to Python.
Python supports floating-point arithmetic with a specified mantissa of 53 bits. Literals are expressed with a decimal point and or an optional `e` or `E`:

```
1., 1.0, .1, 1.e0
```

Python provides decimal fixed-point and floating-point libraries for use where appropriate.

6.4.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.4.5.
- Code algorithms to account for the fact that results can vary slightly by implementation.

6.5 Enumerator issues [CCB]

6.5.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.5 partially applies to Python.

Commented [14]: Yyy SSS ttt AI – Stephen - Feel free to file a docs enhancement bug about this, as we really don't make that info easy to find (it's mentioned in https://docs.python.org/3.7/reference/datamodel.html#the-standard-type-hierarchy, but the main float() docs never actually say "IEEE754 double-precision floating point values" anywhere, nor do they link to the relevant language reference section)

Commented [WLD15]: Doesn't seem to be an issue with this document – it is an issue with the Python.org docs. Suggest removing comment.

Commented [p16]: Comment to be deleted; only a reminder for Stephen to file bug report

An `enum` module was introduced in Python v3.4 which allows for better iteration and value comparison than most previous user-developed methods. An example of the new `enum` module is:

```
from enum import Enum
class ColorEnum(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
    YELLOW = 4
print(ColorEnum.BLUE) #=> ColorEnum.BLUE

from enum import Enum
class ColorEnum(Enum):
    RED = 1
    GREEN = 3
    BLUE = 2
    YELLOW = 4
print(ColorEnum.BLUE)
GREEN < BLUE #syntax error
Green.Value > BLUE.Value? #=> TRUE,
```

Values can be assigned to the names either manually or automatically using `auto()`. Using `auto()` ensures that each name is assigned a unique and sequential value and the initial assignment starting at 1 (not 0).

```
class ColorEnum(Enum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()
    YELLOW = auto()

for color in ColorEnum:
    print(color.value) #=> 1,2,3,4
```

If values are assigned manually they can occur out of sequence, but 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. For example:

```
class ColorEnum(Enum):
    RED = 1
    GREEN = 2
    BLUE = 2
    YELLOW = 3

for color in ColorEnum:
    print(color.name, color.value) #=> RED 1,GREEN 2,YELLOW
3
```

Notice that `BLUE` is completely ignored since it is a repeated value. Duplicate values can be detected and forced to raise a `ValueError` by using the `@unique` class decorator as shown below:

```
@unique
class ColorEnum(Enum):
    RED = 1
    GREEN = 2
    BLUE = 2
    YELLOW = 3

for color in ColorEnum:
    print(color.name, color.value)
                #=> ValueError: duplicate values found in
                #   <enum 'ColorEnum'>: BLUE -> GREEN
```

Mixing `auto()` with manual assignments can be prone to error for the same reason. For example:

```
from enum import Enum, auto
class Colors(Enum):
    RED = auto()
    BLUE = auto()
    GREEN = auto()
    PURPLE = 0
    YELLOW = 1

print(list(Colors)) #=> [<Colors.RED: 1>, <Colors.BLUE: 2>,
<Colors.GREEN: 3>, <Colors.PURPLE: 0>]
```

Notice that YELLOW is missing since its manually-assigned value of 1 had already been created automatically.

Another interesting scenario that involves lists and `auto()` is shown here:

```
from enum import IntEnum, auto
colors = ["RED", "GREEN"]
class Nums(IntEnum):
    ONE = auto()
    TWO = auto()
    THREE = auto()
print(colors[Nums.ONE]) #=> GREEN
```

On the other hand,

```
print(colors[Nums.ONE-1]) #=> RED
```

Notice that in this scenario the first item in the `colors` list (RED) cannot be accessed using `auto()`, unless 1 is subtracted from every enumeration constant created by `auto()`.

Given that enumeration is a useful programming device, many programmers choose to implement their own “enum” objects or types using a wide variety of methods including the creation of “enum” classes, lists, and even dictionaries. Use of enumeration requires careful attention to readability, performance, and safety.

In Python releases before 3.4, programmers used various other Python capabilities to implement the functionality of enumerations, each with its own set of vulnerabilities. New programs should use the provided functionality of `enum` as it is a more complete implementation. Programs created before Python 3.4 can consider updating their relevant code to use the `enum` module. For example, sets of strings can be used to simulate enumerations:

```
colors = {'red', 'green', 'blue'}
```

```
if ‘red’ in colors: print('valid color')
```

6.5.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.5.5.
- 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 [FLC]

6.6.1 Applicability to language

The vulnerabilities identified in ISO/IEC TR 62443-1:2019 clause 6.6 apply to Python, except those related to integer-based conversions since Python seamlessly handles integers as described below.
Python has updated how it handles coercion and instead of using the “lifting” technique that brings operands to a common type, it leaves the handling of different operand types to the operation. If a style slot is incapable of handling an argument type combination, the `Py_NotImplemented` singleton signals to the caller that the operation is not implemented for the type combination. This signals the caller to try other operation slots until it finds one that is compatible with the type combination being implemented. If there are no compatible combinations found, a `TypeError` exception is raised.
Native Python numerical types are converted using the following rules:

- If either argument is a complex number, the other is converted to the complex type otherwise, if either argument is a floating-point number, the other is converted to floating-point.
- Otherwise, both must be plain integers and no conversion is necessary.

Integers in the Python language are of a length bounded only by the amount of memory in the machine. Implementations may store integers in an internal format that has faster performance when the number is smaller than the largest integer supported by the implementation language and platform, but this detail is not exposed to the language user in Python.
Converting from a floating-point number to an integer, either implicitly (using the `int` function) or explicitly, will typically cause a loss of precision:

```
a = 3.0; print(int(a)) #=> 3 (no loss of precision)
a = 3.1415; print(int(a)) #=> 3 (precision lost)
```

Precision can also be lost when converting from very large integers with more than 53 bits of precision to a floating-point number. Losses in precision, whether from an integer to floating-point conversion or vice versa, do not generate errors but can lead to unexpected results especially when floating-point numbers are used for loop control.

Conversions of an excessively large integer or their string equivalent to a float will lead to the exception `OverflowError`. See clause 6.36 Ignored error status and unhandled exceptions [OYB].
Explicit conversion methods can also be used to explicitly convert between types though this is seldom required for numbers since Python will automatically convert as required. Examples include:

```
a = int(1.6666) # a converted to 1
b = float(1) # b converted to 1.0
c = int('10') # c integer 10 created from a string
d = str(10) # d string '10' created from an integer
e = ord('x') # e integer assigned integer value 120
f = chr(121) # f assigned the string 'y'
```

The vulnerability described in ISO/IEC TR 24772-1:2019 related to conversion between semantically incompatible types is applicable to Python, which does not express this notion, such as distinguishing feet from meters. The application developer can implement such mechanisms by wrapping important types in classes and checking class types before performing conversions to avoid resulting exceptions or miscalculations. An alternative method is to use one of the available open source libraries that provide the intended functionality that users can use in preference to creating their own.
Conversions between unrelated types are not possible in Python. For conversions up and down a class hierarchy, see 6.44 Polymorphic variables [BKK].

6.6.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.6.5.
- 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 [CJM]

6.7.1 Applicability to language

This vulnerability is not applicable to Python native programming, as Python does not use null terminated strings. Python strings are immutable objects whose length can be queried with built-in functions. Therefore, Python raises an exception for any access past the end or beginning of a string.

```
a = '12345'
b = a[5] #=> IndexError: string index out of range
```

Vulnerabilities associated with runtime exceptions are addressed in clause 6.36 Ignored error status and unhandled exceptions [OYB].

Python programs, however, may include extension modules written in C or C++, and any string types used for those modules will be C-based string types which have the vulnerability.

6.7.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.7.5.
- In particular, where C style strings or C++ style strings are used, follow the guidance of ISO/IEC TR 24772-1:2019.

6.8 Buffer boundary violation [HCB]

This vulnerability is not applicable to Python because Python's run-time checks the boundaries of arrays and raises an exception when an attempt is made to access beyond a boundary. Vulnerabilities associated with runtime exceptions are addressed in clause 6.36 Ignored error status and unhandled exceptions [OYB].

6.9 Unchecked array indexing [XYZ]

The vulnerability as described in ISO/IEC 24772-1:2019 clause 6.9 is not applicable to Python because Python's run-time checks the boundaries of arrays and raises an exception when an attempt is made to access beyond a boundary. Vulnerabilities associated with runtime exceptions are addressed in clause 6.36 Ignored error status and unhandled exceptions [OYB].

6.10 Unchecked array copying [XYW]

The vulnerability as described in ISO/IEC 24772-1:2019 clause 6.10 is not applicable to Python because assigning lists is done by reference. A deep copy of a list creates a new list object. There is a potential vulnerability associated with copying an object over part of itself when an object is complex, such as lists of lists. This is addressed in 6.38 Deep vs. shallow copying [YAN].

6.11 Pointer type conversions [HFC]

6.11.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2019 clause 6.11 is applicable to Python because Python permits code to instruct instances to "lie" about their type. Consuming code always has the option to decide whether to believe the real type or the claimed type, but naive code will believe any claims by default. As a simple example of code lying about its type, and thus changing the method implementation at runtime:

```
class Example:
```



```
    def method(self):
        print("From Example: ", type(self), self.__class__)
class Other:
    def method(self):
        print("From Other: ", type(self), self.__class__)
x = Example()
x.method()      #=> <class '__main__.Example'> <class
'__main__.Example'>
x.__class__ = Other # the type of the x instance (Example)
                # gets reassigned to 'Other'
x.method()      #=> <class '__main__.Other'> <class
'__main__.Other'>
```

6.11.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.11.5.
- 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 [RVG]

This vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.12 is not applicable to Python because Python does not have pointers and does not permit arithmetic on references.

6.13 Null pointer dereference [XYH]

This vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.13 does not apply to Python. The Python equivalent of a null pointer is the object "`None`". Accessing this object raises an exception. Vulnerabilities associated with runtime exceptions are addressed in clause 6.36 Ignored error status and unhandled exceptions [OYB].

6.14 Dangling reference to heap [XYK]

6.14.1 Applicability to language

This vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.14 only minimally applies to Python because Python uses garbage collection for memory reclamation, thus no dangling references can exist. Specifically, Python only uses namespaces to access objects,

therefore when an object is deallocated there are no names denoting the reclaimed object. Attempts to access those names anyway will raise runtime exceptions as usual. Vulnerabilities associated with runtime exceptions are addressed in clause 6.36 Ignored error status and unhandled exceptions [OYB].
Note: due to reference cycles and `__del__` methods, it is possible for objects that were scheduled for deallocation to gain new live references, and hence not be candidates for deallocation after all. Python runtimes are aware of this when it happens, and avoid deallocating the memory, ensuring that dangling references to heap memory are not created.
Python permits direct access to the internal data of objects by using the `memoryview()` function. The `memoryview()` function is useful on very large objects since it does not create a copy of the object data and, as a result, can perform certain tasks much faster. Managing this direct access to objects does require verification that the object data remains valid even if the object is no longer needed elsewhere in the program.

6.14.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.14.5.
- 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 [FIF]

6.15.1 Applicability to language

The vulnerability discussed in ISO/IEC TR 24772-1:2019 clause 6.15.3 does not apply to Python for integers.
Operations on integers in Python cannot cause wrap-around errors because integers have no maximum size other than what the memory resources of the system can accommodate.
Shift operations operate correctly, except that large shifts on negative numbers infill with '1's and will often result in a final answer of "-1".
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.
Attempts to convert large integers that cannot be represented as a double-precision IEEE 754 value to float will raise `OverflowError.`

```
bigint = 2 * 10 ** 308
float(bigint) #=> OverflowError: int too large to convert to
float
```

The vulnerabilities associated with unhandled exceptions is discussed in clause 6.36 "Ignored error status and unhandled exceptions [OYB]."

6.15.2 Avoidance mechanisms for language users

To mitigate the issues associated with floating-point types:

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.15.5.
- 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 operations for multiplication and division [PIK]

This vulnerability is not applicable to Python because there is no practical way to overflow an integer since integers have unlimited precision, left shifts are defined in terms of multiplication by powers of 2, and right shifts are defined in terms of floor division by powers of two.

```
print(-1 << 100) #=> -1267650600228229401496703205376
print(1 << 100)  #=> 1267650600228229401496703205376
print(-4 >> 3)   #=> -1 where 0 might be expected
```

6.17 Choice of clear names [NAI]

6.17.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.17 exists in Python. Python provides very liberal naming rules:

- Names may be of any length and consist of letters, numerals, and underscores only. All characters in a name are significant. Note that unlike some other languages where only the first *n* number of characters in a name are significant, ***all*** characters in a Python name are significant. This eliminates a common source of name ambiguity when names are identical up to the significant length and vary afterwards which effectively makes all such names a reference to one common variable.
- All names must start with an underscore or a letter.
- Names are case sensitive, for example, `Alpha`, `ALPHA`, and `alpha` are each unique names. While this is a feature of the language that provides for more flexibility in naming, it is also can be a source of programmer errors when similar names are used which differ only in case, for example, `aLpha` versus `alpha`.
- Names allow all Unicode "script" code points to be used as letters, and each numerical code point is considered distinct when used as part of a name, even if their visual rendering is similar. Some Unicode characters can cause confusion for humans in that what they read may not be the text that is processed by the language processor. For example, using homoglyphs, `Confused` (Cyrillic ES) versus `Confused` (Latin C), or aIpha (Latin capital I) versus alpha (Latin lowercase l) will be different names.

The following naming conventions are not part of the standard but are in common use:

- Class names start with an upper-case letter, all other variables, functions, and modules are in all lower case.
- Names starting with a single underscore (_) are not imported by the "`from` *`module`* `import *`" statement – this not part of the standard but most implementations enforce

it.

- Names starting and ending with two underscores (__) are system-defined names.
- Names starting with, but not ending with, two underscores are local to their class definition.
- Python provides a variety of ways to package names into namespaces so that name clashes can be avoided:
    - Names are scoped to functions, classes, and modules meaning there is normally no collision with names utilized in outer scopes and vice versa.
    - Names in modules (a file containing one or more Python statements) are local to the module and are referenced using qualification (for example, a function `x` in module `y` is referenced as `y.x`). Though local to the module, a module's names can be, and routinely are, copied into another namespace with a `from` *`module`* `import` statement.

Python's naming rules are flexible by design but are also susceptible to a variety of unintentional coding errors:

- Names are not required to be declared but they must be assigned values before they are referenced. This means that some errors will never be exposed until runtime when the use of an unassigned variable will raise an exception (see clause 6.22 Initialization of variables [LAV]).
- Names can be unique but may look similar to other names, for example, `alpha` and `aLpha`, `__x` and `_x`, `_beta__` and `__beta_` which could lead to the use of the wrong variable. Python will not detect this problem at compile-time.

Python utilizes dynamic typing with types determined at runtime. There are no type or variable declarations for an object by default, which can lead to subtle and potentially catastrophic errors:

```
x = 1
# lots of code…
if some rare but important case:
    X = 10
```

In the code above, the programmer intended to set (lower case) `x` to 10 and instead created a new upper-case `X` to `10` so the lower-case `x` remains unchanged. Python will not detect a problem because there is no problem – it sees the upper-case `X` assignment as a legitimate way to bring a *new* object into existence. It could be argued that Python could statically detect that `X` is never referenced and therefore indicate the assignment is dubious but there are also cases where a dynamically defined function defined downstream could legitimately reference `X` as a `global`.

6.17.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.17.5.
- 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 [WXQ]

6.18.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.18 applies to Python, since it is possible to assign a value to a variable and never reference that variable which causes a "dead store". This in itself is not harmful, other than the memory that it wastes, but if there is a substantial amount of dead stores then performance could suffer or, in an extreme case, the program could halt due to lack of memory

Similarly, if dead stores cause the retention of critical resources, such as file descriptors or system locks, then this retention may cause subsequent system failures.

Variables local to a function are deleted automatically when the encompassing function is exited but, though not a common practice, variables can be explicitly deleted when they are no longer needed using the `del` statement.

6.18.2 Avoidance mechanisms for users

- Follow the applicable guidance of ISO/IEC TR 24772-1:2019 clause 6.18.5.
- 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.

Commented [MS17]: 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 [YZS]

6.19.1 Applicability to language

The vulnerability as described in ISO IEC TR 24772-1:2019 clause 6.19 is applicable to Python.

6.19.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.19.5.

6.20 Identifier name reuse [YOW]

6.20.1 Applicability to language

Python has the concept of namespaces which are simply the places where names exist in memory. Namespaces are associated with functions, classes, and modules. When a name is created (that is, when it is first assigned a value), it is associated (that is, bound) to the namespace associated with the location where the assignment statement is made (for example, in a function definition). The association of a variable to a specific namespace is elemental to how scoping is defined in Python.
Scoping allows for the definition of more than one variable with the same name to reference different objects. For example:

```
avar = 1
def x():
    avar = 2
    print(avar) #=> 2
x()
print(avar) #=> 1
```

The variable `avar` within the function `x` above is local to the function only – it is created when `x` is called and disappears when control is returned to the calling program. If the function needed to update the outer variable named `avar` then it would need to specify that `avar` was a global before referencing it as in:

```
avar = 1
def x():
    global avar
    avar = 2
    print(avar) #=> 2
x()
print(avar) #=> 2
```

In the case above, the function is updating the variable `avar` that is defined in the calling module. There is a subtle but important distinction on the locality versus global nature of variables: *assignment* is always local unless `global` is specified for the variable as in the example above where `avar` is *assigned* a value of `2`. If the function had instead simply *referenced* `avar` without assigning it a value, then it would reference the topmost variable `avar` which, by definition, is always a global:

```
avar = 1
def x():
    print(avar)
x() #=> 1
```

The rule illustrated above is that attributes of modules (that is, variable, function, and class names) are global to the module meaning any function or class can reference them.
Scoping rules cover other cases where an identically named variable name references different objects:

- A nested function's variables are in the scope of the nested function only.
- Variables defined in a module are in *global* scope, which means they are scoped to the module only and are therefore not visible within functions defined in that module (or any

other function) unless explicitly identified as `global` at the start of the function.

Python has ways to bypass implicit scope rules:

- The `global` statement, which allows an inner reference to an outer scoped variable(s).
- The `nonlocal` statement, which allows a variable in an enclosing function definition to be referenced from a nested function.

The concept of scoping makes it safer to code functions because the programmer is free to select any name in a function without worrying about accidentally selecting a name assigned to an outer scope, which in turn could cause unwanted results. In Python, one must be explicit when intending to circumvent the intrinsic scoping of variable names. The downside is that identical variable names, which are totally unrelated, can appear in the same module, which could lead to confusion and misuse unless scoping rules are well understood.
Names can also be qualified to prevent confusion as to which variable is being referenced:

```
avar = 1
class xyz():
    avar = 2
    print(avar) #=> 2
print(xyz.avar, avar) #=> 2 1
```

The final `print` function call above references the `avar` variable within the `xyz` class and the global `avar`.

6.20.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.20.5.
- 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 [BJL]

6.21.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 21 is applicable to Python when modules are imported.
Python has a hierarchy of namespaces, which provides isolation to protect from name collisions, ways to explicitly reference down into a nested namespace, and a way to reference up to an encompassing namespace. Generally speaking, namespaces are isolated. For example, a program's variables are maintained in a separate namespace from any of the functions or classes it defines or uses. The variables of modules, classes, or functions are also maintained in their own protected namespaces. Namespaces may be nested.
For certain scenarios, the local namespace is dictated by the order of importation. For example, the scenarios below import two files (`a.py` and `b.py)` and each file contains a function named "`meth()`". Importing the files using "`from x import *`" results in the last `import` to be used. In the second scenario, using only the "`import x`" method allows the use of either `meth()` by prefacing it with the desired library name regardless of order presented in the file.

```
< -  file = a.py - >
def meth():
    print("From A")

< -  file = b.py - >
def meth():
    print("From B")
-----------------------
 from a import *
 from b import *
 from a import *
 meth() #=> From A
 --------------------------
 import a
 import b
 a.meth() #=> From A
```

See clause 6.41 Inheritance [RIP] for a discussion of multiple inherited methods with the same name.

Accessing a namespace's attribute (that is, a variable, function, or class name), is generally done in an explicit manner to make it clear to the reader (and Python) which attribute is being accessed:

```
n = Animal.num # fetches a class' variable called num
x = mymodule.y # fetches a module's variable called y
```

The examples above exhibit qualification – there is no doubt from where a variable is being fetched. Qualification can also occur from an encompassed namespace up to the encompassing namespace using the global statement:

```
def x():
    global y
    y = 1
```

The example above uses an explicit `global` statement which makes it clear that the variable `y` is not local to the function `x;` it assigns the value of `1` to the variable `y` in the encompassing module[1].

Python also has some subtle namespace issues that can cause unexpected results especially when using imports of modules. For example, assuming module `a.py` contains:

```
a = 1
```

And module `b.py` contains:

```
b = 1
```

[1] *V*alues are assigned to objects which in turn are referenced by variables but it's simpler to say the value is assigned to the variable. Also, the encompassing code could be at a prompt level instead of a module. For brevity this annex uses this simpler, though not as exact, wording.

Executing the following code is not a problem since there is no variable name collision in the two modules (the `from modulename import *` statement brings all of the attributes of the named module into the local namespace):

```
from a import *
print(a) #=> 1
from b import *
print(b) #=> 1
```

Later on, the author of the `b` module adds a variable named `a` and assigns it a value of **2**. `b.py` now contains:

```
b = 1
a = 2 # new assignment
```

The programmer of module `b.py` may have no knowledge of the `a` module and may not consider that a program would import both `a` and `b`. The importing program, with no changes, is run again:

```
from a import *
print(a) #=> 1
from b import *
print(a) #=> 2
```

The results are now different because the importing program is susceptible to unintended consequences due to changes in variable assignments made in two unrelated modules as well as the sequence in which they were imported. Also note that the "`from modulename import *`" statement brings all of the modules attributes into the importing code which can silently overlay like-named variables, functions, and classes.

A common misunderstanding of the Python language is that Python detects local names (a local name is a name that lives within a class or function's namespace) *statically* by looking for one or more assignments to a name within the class/function. If one or more assignments are found then the name is noted as being local to that class/function. This can be confusing because if only *references* to a name are found then the name is referencing a global object so the only way to know if a reference is local or global, barring an explicit global statement, is to examine the entire function definition looking for an assignment. This runs counter to Python's goal of Explicit is better than implicit (EIBTI):

```
a = 1
def f():
     print(a)
     a = 2
f() #=> UnboundLocalError: local variable 'a' referenced
before
        assignment
# now with the assignment commented out
a = 1
def f():
     print(a) #=> 1
     #a = 2
# Assuming a new session:
a = 1
def f():
```

```
    global a
    a = 2 * a
f()
print(a) #=> 2
```

Note that the rules for determining the locality of a name applies to the assignment operator = as above, but also to all other kinds of assignments which includes module names in an `import` statement, function and class names, and the arguments declared for them. See clause 6.19 Unused variable [YZS] for more detail on this.
Python can perform either absolute or relative imports. An absolute import specifies the resource to be imported using its full path from the project's root folder. A relative import specifies the resource is to be imported relative to the current location. Although the full path of an import can be long, the use of an absolute import defines explicitly what resource is being imported.
Name resolution follows a simple Local, Enclosing, Global, Built-ins (LEGB) sequence:

- First the local namespace is searched;
- Then the enclosing namespace (that is, a `def` or `lambda` (A `lambda` is a single expression function definition));
- Then the global namespace.
- Lastly the built-in's namespace.

Python v3.3 introduced `types.prepare_class()` which gives more control over how classes and metaclasses are created. The `__prepare__` function can be called prior to the creation of a metaclass instance giving complete control over how the class declarations are ordered. It also allows symbols to be inserted into the class namespace, which can be used elsewhere in the class, but these are only visible during class construction.

6.21.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.21.5.
- 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 Initialization of variables [LAV]

6.22.1 Applicability of language

This vulnerability applies only minimally to Python because all attempts to access an uninitialized variable result in an exception. There is no ability to use a variable with an uninitialized value because *assigned* variables always reference objects which always have a value and *unassigned* variables do not exist. Therefore, Python raises an exception at runtime when a name that is not bound to an object is referenced.
Static type analysis tools can be used to identify many accesses to names that are not bound to objects prior to execution.
Vulnerabilities associated with runtime exceptions are addressed in clause 6.36, Ignored error status and unhandled exceptions .

6.22.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.22.5.
- 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 [JCW]

6.23.1 Applicability to language

The vulnerability described in ISO/IEC TR 24772-1:2019 clause 6.23 applies to Python.
Python provides many operators and levels of precedence, so it is not unexpected that operator precedence and order of operation are not well understood and hence misused. For example:

```
     1 + 2 * 3 #=> 7, evaluates as 1 + (2 * 3)
     (1 + 2) * 3 #=> 9, parenthesis are allowed to coerce
precedence
```

.

6.23.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.23.5.

6.24 Side-effects and order of evaluation of operands [SAM]

6.24.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.24 exists in part in Python. Operands are evaluated left-to-right in Python and hence the evaluation order is deterministic,

but the vulnerabilities associated with short-circuit operators exist in Python. Additional vulnerabilities arise from Python semantics of loops that alter data structures.
Some of Python's data structures such as lists, dictionaries and sets, are mutable. Attempting to delete items from one of these data structures, from within a loop, will result in undesirable side-effects. The example below shows that using the loop index to delete items in the `numbers` list results in an indexing error since the loop index "`i`" is based on the full length of the original list.

```
def odd(x): return bool(x % 2)
numbers = [n for n in range(10)]

for i in range(len(numbers)):
   if odd(numbers[i]):
      # Deleting list items while looping results in error
      del numbers[i] #=> IndexError: list index out of
range
```

Numeric data types in Python are immutable and remain unchanged when used as an argument within a calling function. However, if the immutable argument within a calling function is made to be a global variable, then that argument is changed even though it is usually an immutable type. This potentially unexpected side-effect is illustrated in the following example. `double` passes the immutable integer "`y`" as an argument to the `double` function, but because it is declared as a `global` variable within the function, the immutable object is modified in the calling function.

```
def double(n):
   global y
   y = 2 * n

y = 5
double(y)
print(y) #=> 10
```

Potentially unexpected side-effects can also be experienced by changing an external list from a loop. For example, the following code shows that adding the color `black` to the `colors` list updates the list since lists are mutable objects. The `for` loop recognizes this new list member and continues with another pass through the loop with the index counter `i` now set to `black` resulting in the color `white` being added to the `colors` list.

```
colors = ["red"]
for i in colors:
    if i == "red":
        colors += ["black"]
    if i == "black":
        colors += ["white"]
print(colors) #=> ['red', 'black', 'white']
```

To avoid the unexpected side effects, is it recommended to use a copy of the list within the loop. In this scenario, `black` is added to the local `colors` list but since the loop index `i` never takes on a value other than `red`, the color `white` is never added to the `colors` list.

```
colors = ["red"]
for i in colors[:]: # Avoid side effects by using a local
list
```

```
        if i == "red":
            colors += ["black"]
        if i == "black":
            colors += ["white"]
    print(colors) #=> ['red', 'black']
```

Python allows reassignment of loop indexes, which can lead to unexpected results depending on the order of reassignment. For example, the following code illustrates two scenarios where the loop index "i" is reassigned within a loop. The first scenario uses the loop index *prior to* reassignment and prints out the expected sequence. The second scenario uses the loop index *after* reassignment and, since it creates a new object with a value of ten, this new value is printed out. Internally, the loop index counter remains intact and the loop exits after four iterations as expected.

```
    for i in range(1, 5):
        print(i) #=> 1,2,3,4
        i = 10

    for i in range(1, 5):
        i = 10 # new i is created, doesn't affect the loop
count
        print(i) #=> 10,10,10,10
```

Python supports sequence unpacking (parallel assignment) in which each element of the right-hand side (expressed as a tuple) is evaluated and then assigned to each element of the left-hand side (LHS) in left-to-right sequence. For example, the following is a safe way to exchange values in Python:

```
    a = 1
    b = 2
    a, b = b, a # swap values between a and b
    print (a,b) #=> 2, 1
```

Assignment of the targets (LHS) proceeds left-to-right so overlaps on the left side are not safe:

```
    a = [0,0]
    i = 0
    i, a[i] = 1, 2 #=> Index is set to 1; list is updated at
[1]
    print(a) #=> 0,2
```

Python Boolean operators are often used to assign values as in:

```
    a = b or c or d or None
```

`a` is assigned the first value of the first object that has a non-zero (that is, `True`) value or, in the example above, the value `None` if `b`, `c`, and `d` are all `False`. This is a common and well understood practice. However, trouble can be introduced when functions or other constructs with side effects are used on the right side of a Boolean operator:

```
    if a() or b()
```

If function `a` returns a `True` result then function `b` will not be called which may cause unexpected results. If necessary perform each expression first and then evaluate the results:

```
    x = a()
    y = b()
```

```
if x or y …
```

The `assert` statement in Python is used primarily for debugging and throws an exception, with optional comment, if predefined conditions are not met.

Be aware that, even though overlaps between the left hand side and the right hand side are safe, it is possible to have unintended results when the variables on the left side overlap with one another so always ensure that the assignments and left-to-right sequence of assignments to the variables on the left hand side never overlap. If necessary, and/or if it makes the code easier to understand, consider breaking the statement into two or more statements:

```
# overlapping
a = [0,0]
i = 0
i, a[i] = 1, 2 #=> Index is set to 1; list is updated at
[1]
print(a) #=> 0,2
# Non-overlapping
a = [0,0]
i, a[0] = 1, 2
print(a) #=> 2,0
```

As with many languages, Python performs short circuiting in Boolean expressions. In the case of "`x or y`", Python only evaluates `y` if `x` evaluates to false. Likewise, for "`x and y`", Python only evaluates `y` if `x` is true. If there are side effects in `y`, they only occur if y is evaluated.

6.24.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.24.5.
- 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 [KOA]

6.25.1 Avoidance mechanisms for language

The vulnerability as described in TR 24772-1 clause 6.25 applies to Python, but Python goes to some lengths to help prevent some of the likely incorrect expressions:

- Testing for equivalence cannot be confused with assignment:

```
a = b = 1
if (a=b): print(a,b) #=> syntax error
if (a==b): print(a,b) #=> 1 1
```

- Boolean operators use English words `not`, `and`, `or`; bitwise operators use symbols `~`, `&`,

and `|`, respectively. Python, however, does have some subtleties that can cause unexpected results:

- Skipping the parentheses after a function does not invoke a call to the function and will fail silently because it's a legitimate reference to the function object:

```
class a:
     def demo():
          print("in demo")
a.demo() #=> in demo
a.demo #=> <function demo at 0x000000000342A9C8>
x = a.demo
x() #=> in demo
```

  The two lines that reference the function without trailing parentheses above demonstrate how that syntax is a reference to the function *object* and not a call to the function.

- Built-in functions that perform in-place operations on mutable objects (that is, lists, dictionaries, and some class instances) do not return the changed object – they return `None`:

```
a = []
a.append("x")
print(a) #=> ['x']
a = a.append("y")
print(a) #=> None
```

- In async code, forgetting to use an `await` statement results in a warning about the unawaited coroutine.

Short-circuit operations can be a source of likely incorrect expressions as described in clause 6.24.

6.25.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.25.5.
- 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 [XYQ]

6.26.1 Applicability to language

There are many ways to have dead or deactivated code occur in a program and Python is no different in that regard. Except in very limited cases, Python does not provide static analysis to detect such code nor does the very dynamic design of Python's language lend itself to such analysis. The limited cases are those where a known-false constant value (for example 0, False) is used directly in a conditional flow control check (the branch will never be taken, so code does

not need to be emitted for it), and when a function unconditionally executes a top-level return statement (no code needs to be emitted for the section after the function returns).
The module and related `import` statement provide convenient ways to group attributes (for example, functions, names, and classes) into a file which can then be copied, in whole, or in part (using the `from` statement), into another Python module. All of the attributes of a module are copied when either of the following forms of the `import` statement is used. This is roughly equivalent to simply copying in all of code directly into the importing program, which can result in code that is never invoked (for example, functions which are never called and hence "dead"):

```
import modulename
from modulename import *
```

The `import` statement in Python loads a module into memory, compiles it into byte code, and then executes it. Subsequent executions of an import for that same module are ignored by Python and have no effect on the program whatsoever. The `reload` statement is required to force a module, and its attributes, to be loaded, compiled, and executed.

6.26.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.26.5.
- 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 [CLL]

The vulnerability does not apply to Python, which does not have a switch statement nor the concept of labels or branching to a demarcated "place".

6.28 Demarcation of control flow [EOJ]

6.28.1 Applicability to language

The vulnerabilities as described in ISO/IEC TR 24772-1:2019 clause 6.28 only minimally apply to Python. Python makes demarcation of control flow very clear because it uses indentation (using spaces or tabs – but not both within a given code block) and dedentation as the *only* demarcation construct:

```
a, b = 1, 1
if a:
    print("a is True")
else:
    print("False")
    if b:
        print("b is true")
print("back to main level")
```

The code above prints "`a is True`" followed by "`back to main level`". Note how control is passed from the first `if` statement's `True` path to the main level based entirely on indentation while in other languages that do not rely on indention, the second `if` would always execute and would print "`b is true`" since the second `if` would evaluate to `True`.

6.28.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.28.5.
- 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 variables [TEX]

6.29.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.28 applies only minimally to Python. Python `for` loops iterate over structures such as lists or ranges. Assignments to identically named variables in the loop go to local instances and do not affect the loop counter.
Python, however, shows other surprising behaviours. It is possible to alter the loop behaviour by creating or deleting the objects that are iterated over. When using the `for` statement to iterate though an iterable object such as a list, there is no way to influence the loop "count" because it's not exposed. The variable `a` in the example below takes on the value of the first, then the second, then the third member of the list:

```
x = ['a', 'b', 'c']
for a in x:
    print(a)
#=>a
#=>b
#=>c
```

It is possible, though not recommended, to change a mutable object as it is being traversed which in turn changes the number of iterations performed. In the case below the loop is performed only two times instead of the three times had the list been left intact:

```
x = ['a', 'b', 'c']
for a in x:
    print(a)
    del x[0]
print(x)
#=> a
#=> c
#=> ['c']
```

6.29.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.29.5.
- 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 [XZH]

6.30.1 Applicability to language

The Python language itself is vulnerable to off-by-one errors as is any language when used carelessly or by a person not familiar with Python's index starting at zero versus at one. Python does not prevent off-by-one errors but its runtime bounds checking for strings and lists does lessen the chances that doing so will cause harm. It is also not possible to index past the end or beginning of a string or list by being off-by-one because Python does not use a sentinel character and it always checks indexes before attempting to index into strings and lists and raises an exception when their bounds are exceeded.
The `range` function can be used to create a sequence over a range of numbers such as:

```
for x in range(10):
     print (x)
```

which will print the numbers 0 through 9. As many languages start indexing from 0, this is not likely a source of great confusion. It is more likely that confusion will arise when using a range starting with a value other than the default 0, such as:

```
for x in range(5, 10):
     print (x)
```

which will print the values 5 through 9.

6.30.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.30.5.
- 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 Structured programming [EWD]

6.31.1 Applicability to language

The vulnerabilities described in TR 24772-1:2019 clause 6.31 are substantially mitigated in Python. The language does not provide a statement for local or non-local transfers of control, however there is a library that provides `goto` capabilities.

A `break` statement for the premature exit from loops is provided. Multiple `break` and multiple `return` statements are permitted. Breaking out of multiple nested loops from the innermost loop can be problematic as the break only terminates the nearest enclosing loop.
Python is designed to make it simpler to write structured program by requiring indentation to show scope of control in blocks of code:

```
a = 1
b = 1
if a == b:
    print("a == b") #=> a == b
    if a > b:
        print("a > b")
else:
    print("a != b")
```

In many languages the last `print` statement would be executed because the `else` is associated with the immediately prior `if`, while Python uses indentation to link the `else` with its associated `if` statement. In the example above, the 'else' statement is associated with the first 'if' statement since it has the same level of indentation.

Note that context managers (such as those introduced by the `with` clause) can be used to consolidate where exceptions are evaluated and propagated, which lets developers write straight forward code without sprinkling "try … except … finally" structures throughout the code. For example, the following code ensures that the opened file is closed promptly, even if an exception occurs, or code in the body returns from a containing function, or breaks out of a containing loop:

```
with open("example.txt") as f:
    for line in f:
        print(line)
# File will be closed here, as well as on an exception,
break, continue, or return
```

6.31.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.31.5.
- 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 [CSJ]

6.32.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1 clause 6.32 minimally applies to Python.

Python functions return a value of `None` when no `return` statement is executed or when a `return` with no arguments is executed. Python detects attempts to return uninitialized arguments and raises the `NameError` exception.
Python passes arguments by assignment, which is similar to passing by reference. Python assigns the passed arguments to the function's local variables, but having the address of the caller's argument does not automatically allow the called function to change any of the objects referenced by those arguments – only *mutable* objects referenced by passed arguments can be changed. Aliasing can occur on the mutable actual objects designated by the parameters as follows:

```
class C():
    def __init__(self, number):
        self.comp = number

A=C(7)  # A.comp = 7
B=C(14) # B.comp = 14

def fun(X,Y):
   X.comp = 8
   Y.comp = 42
   print(X.comp) #=> may be 8, but also 42, depending on
call
   print(Y.comp) #=> always 42

fun(A, B) # call prints 8, 42
fun(A, A) # call prints 42, 42
fun(B, B) # call prints 42, 42
print(A.comp, B.comp) #=> 42 42
```

In the example above, class instances `A` and `B` are passed as arguments and their components are updated. While the local variables are discarded when the function goes out of scope, changes to the components of their designated objects remain in effect. The example shows that when identical objects are passed as function arguments, e.g. `fun(A, A)` or `fun(B, B)`, the `X` and `Y` aliases in the function definition are reassigned with identical values and since `Y.comp` always appears after `X.comp`, its value always gets returned to the calling function.
The example below uses two class instances `A` and `B`, each passed individually into a function that uses the `B` class instance. When the class `B` instance is passed to the function, it is aliased to both internal variables `X` and `B`, but when class `A` is passed to the function, it is only aliased to `X`.

```
class C():
    def __init__(self, number):
        self.comp = number

def fun(X):
   X.comp = 9
   B.comp = 43
   print(X.comp) # may be 9, but also 43, depending on call
   print(B.comp) # always 43

A = C(7) # A.comp = 7
B = C(14) # B.comp = 14
fun(A) # call prints 9 43
```

```
fun(B) # call prints 43 43
```

In the example below, the argument is mutable, and is therefore updated in place:

```
a = [1]

def f(x):
    x[0] = 2
    if a[0] == 2:
        print("surprise!")

f(a) #=> surprise
print(a) #=> [2]
```

Note that the list object `a` is not changed – it is the same object but its content at index `0` has changed, which causes the aliasing effect demonstrated by the "`if`" statement.
Aliasing of arguments with immutable types cannot happen in Python. The following example demonstrates that one can emulate a call by reference by assigning the returned object to the passed argument:

```
def doubler(x):
    return x * 2
x = 1
x = doubler(x)
print(x) #=> 2
```

This is not a true call by reference and Python does not replace the value of the object `x`, rather it creates a new object `x` and assigns it the value returned from the `doubler` function as proven by the code below which displays the address of the initial and the new object `x`:

```
def doubler(x):
    return x * 2
x = 1
print(id(x)) #=> 506081728
x = doubler(x)
print(id(x)) #=> 506081760
```

The object replacement process demonstrated above follows Python's normal processing of *any* statement which changes the value of an immutable object and is not a special exception for function returns.

6.32.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.32.5 to avoid aliasing effects.
- 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 [DCM]

6.33.1 Applicability to language

With the exception of interfacing with other languages, Python does not have the vulnerability as described in ISO/IEC TR 24772-1 clause 6.33. For example, Python has a foreign function library called `ctypes`, which allows C functions to be called in DLLs or shared libraries. It can provide the opportunity to read, and potentially change, memory locations:

```
import ctypes
memid = (ctypes.c_char).from_address(0X0B98F706)
```

Once `memid` is known, the potential exists to modify the memory location.

See clause 6.53 Provision of inherently unsafe operations [SKL] for the avoidance of such inherently unsafe operations. For safer interactions with C code, Python provides the `cffi` module.

6.33.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.33.5.
- Avoid using `ctypes` when calling C code from within Python and use `cffi` (C Foreign Function Interface) instead.

6.34 Subprogram signature mismatch [OTR]

6.34.1 Applicability to language

The vulnerability of a mismatch in type expectations as described in ISO/IEC TR 24772-1:2019 clause 6.34 exists in Python. An argument passed to a Python function may be of a type that does not match the needs of operations performed by the function on the formal parameter, resulting in a run-time exception. The other vulnerability of a mismatch in parameter numbers does not exist in Python, as Python checks the number of arguments passed. Variable numbers of positional and keyword arguments are supported by Python, but the method of accessing the arguments ensures that all access arguments exist.
Python supports the following argument structures:

1. positional,
2. `key=value` *(called a keyword argument),* or
3. both kinds of arguments, in which case positional arguments must precede the first keyword argument.

Python also supports a variable numbers of arguments and, other than the case of variable arguments, will check at runtime for the correct number of arguments making it impossible to corrupt the call stack in Python when using standard modules.
Python provides the mechanism `def foo(*a)` to permit `foo` to receive a variable number of positional arguments. In this case, the formal argument becomes a tuple and the actual parameters are extracted using tuple processing syntax. Furthermore, Python provides the

mechanism `def foo(**a)` to permit `foo` to receive a variable number of keyword arguments called a dictionary.
Python always calls the most recently defined function of a specified name. That is, there is no overloading of arguments. There is no type-checking of arguments as part of parameter passing and no concept of function overloading. Type errors are detected when the body executes operations not available for the type of the argument. Python provides a type membership test `isinstance(var_name, Class_or_primitive_type)`, that returns a Boolean that lets the user take alternative action based on the actual type of variable.
Python has many extension APIs and embedding APIs that include functions and classes providing additional functionality. These perform subprogram signature checking at runtime for modules coded in non-Python languages. Discussion of these APIs is beyond the scope of this annex but the reader should be aware that improper coding of any non-Python modules or their interfaces can cause call stack problems. Programmers should also be aware that the `cffi` module will believe the signature information it is given, which may or may not be accurate. For vulnerabilities associated with calling libraries written in other languages, see 6.47 Inter-language calling.

6.34.2 Avoidance mechanisms for language users

- Apply the guidance described in ISO/IEC TR 24772-1:2019 clause 6.47.5, Inter-language calling, when interfacing with C code or when calling library functions that interface with C code.
- Avoid using `ctypes` when calling C code from within Python and use `cffi` (C Foreign Function Interface) instead since it is more streamlined and safer.
- Document the expected types of the formal parameters (type hints) and apply static analysis tools that check the program for correct usage of types.
- Use type membership tests to prevent runtime exceptions due to unexpected parameter types.

6.35 Recursion [GDL]

6.35.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.35 is mitigated in Python since the depth of the recursion is limited. 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.

6.35.2 Guidance to language users

- Follow the guidance of ISO/IEC TR 24772-1:2019 clause 6.35.5.
- Adjust the maximum recursion depth to an appropriate value as needed.

6.36 Ignored error status and unhandled exceptions [OYB]

6.36.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.36 applies to Python. Unhandled Python exceptions in the main thread will cause the program to terminate, as discussed in ISO/IEC TR 24772-1:2019 clause 6.36.3. Unhandled exceptions in a concurrent part of a program will have effects that are dependent on the model of concurrency being used and the explicit way that the components are executed and communicate. See 6.62 Concurrency – Premature termination [CGS].

Something about legality of code that uses exceptions?

Commented [SM18]: SSS – Investigate if the language processor or any support tools that could determine that an exception is likely and provide a warning.

Commented [MS19R18]: I tested multiple tools and none of them offer support for this type of determination. I ran a Python tool named Prospector. It, in turn, runs six other tools: dodgy, mccabe, profile-validator, pycodestyle, pyflakes, and, Pylint.

6.36.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.36.5.
- 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 [AMV]

This vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.37 is not applicable to Python because assignments are made to objects and the object always holds the type – not the variable. Therefore, if multiple labels reference the same object, they all see the same type and there is no way to have more than one type for any given object.

6.38 Deep vs. shallow copying [YAN]

6.38.1 Applicability to language

Python exhibits the vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.38. The slice operator, e.g. "`x = y[:]`" and the copy methods, e.g. "`x = y.copy()`", copy the first level of a list, but leave deeper levels, such as sublists, shared. For producing deep copies, Python provides the `deepcopy` method.

The following example illustrates the issues in Python:

```
colours1 = ["orange", "green"]
colours2 = colours1
print(colours1)               --  ['orange', 'green']
print(colours2)               --  ['orange', 'green']
colours2 = ["violet", "black"]
print(colours1)               --  ['orange', 'green']
print(colours2)               --  ['violet', 'black']
```

If, however, one writes:

```
colours1 = ["orange", "green"]
colours2 = colours1
colours2[1] = "yellow"
print(colours1)               --  ['orange', 'yellow']
```

When `colours1` is created, Python creates it as a list type, and then has the list point to its elements. When `colours2` is created as a copy of `colours1`, they both point to the same list container. If one sets a new value to an element of the list, then any variable that points to that list sees the update, as shown in the second example. Example 1, on the other hand, shows that when a completely new list is created for `colours2` (replacing the equivalence of `colours1` and `colours2`), any further changes to `colours2` or `colours1` do not affect the other. Copying with the slice "`[:]`" operator provides a deeper level of copying under certain situations. It does create a new memory address for the top-level list, but when embedded sublists are involved, the slice operator still references the objects in the original list. The following example shows how changing a sublist within list L2 also unintentionally changes the same sublist in list L1.

```
L1 = [[1,2,3], [4,5,6], [7,8,9]]
L2 = L1[:]
L2[0][2] = [123456789]
print(L1) #=> [[1, 2, [123456789]], [4, 5, 6], [7, 8, 9]]
print(L2) #=> [[1, 2, [123456789]], [4, 5, 6], [7, 8, 9]]
```

Python also has a function called `deepcopy` that can be imported from the `copy` module and copies all levels of a structured object to a completely new object so that a list within a list can be independently accessed as shown in the example below:

```
import copy
L1 = [[1,2,3], [4,5,6], [7,8,9]]
L2 = copy.deepcopy(L1)
L2[0][2] = [123456789]
print(L1) #=> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(L2) #=> [[1, 2, [123456789]], [4, 5, 6], [7, 8, 9]]
```

6.38.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.38.5.
- 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 [XYL]

6.39.1 Applicability to language

The heap fragmentation vulnerability as described in ISO/IEC TR 24772-1:2019 exists in Python. The memory leak vulnerability of that clause is mitigated by Python automatic garbage collection as described below.
Python supports automatic garbage collection so in theory it should not have memory leaks. However, there are at least three general cases in which memory can be retained after it is no longer needed. The first is when implementation-dependent memory allocation/de-allocation algorithms cause a leak, which would be an implementation error and not a language error. The second general case is when objects remain referenced after they are no longer needed. This is a logic error which requires the programmer to modify the code to delete references to objects when they are no longer required.
There is a third subtle memory leak case wherein objects mutually reference one another without any outside references remaining – a kind of deadly embrace where one object references a second object (or group of objects) so the second object(or group of objects) can't be collected but the second object(s) also reference the first one(s) so it/they too can't be collected. This group is known as cyclic garbage. Python provides a garbage collection module called `gc` which has functions which enable the programmer to enable and disable cyclic garbage collection as well as inspect the state of objects tracked by the cyclic garbage collector so that these, often very subtle leaks, can be traced and eliminated.

6.39.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.39.5.
- 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 [SYM]

6.40.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.40 applies to Python, although Python does not have the applicable language characteristics as outlined in ISO/IEC TR 24772-1:2019 clause 6.40.4. Since Python is dynamically typed, essentially all functions in Python exhibit generic properties. Therefore, the mechanisms of failure outlined in ISO/IEC TR 24772-1:2019 clause 6.40.3 apply to Python.

6.40.2 Avoidance mechanisms for language users

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 [RIP]

6.41.1 Applicability to language

The vulnerabilities as described in ISO/IEC TR 24772-1:2019 clause 6.41 apply to Python. Python supports inheritance as described in clause 5.1.4.

It is important to make sure that each class calls the `__init__` of its superclass so that it is properly initialized. The built-in function `super()` provides access to the next class in the MRO sequence. See clause 5.1.4, which also includes an example.

The difficulties associated with establishing the MRO are also illustrated in clause 5.1.4.

There can be unexpected outcomes from the MRO as shown in the following code. The outcome might be expected to be `a=0`, but in reality the result is `a=2` since, as previously mentioned, methods in derived calls are always called before the method of the base class (`class T`).

```
class T():
    a = 0
class A(T):
    pass
class B(T):
    a = 2
class C(A,B):
    pass
c = C()
print(c.a) # => 2
```

There is no protection in Python against accidental redefinition, method capture, or accidental non-redefinition along the MRO sequence, so that these vulnerabilities apply.
Moreover, as the search for a binding is at run-time in dynamically established class hierarchies, a static analysis cannot predetermine the danger of these vulnerabilities to incur. Neither can a reviewer of the code without detailed analysis of the entire class hierarchy determine which method is called. The`__mro__` attribute can be queried in the code to determine the MRO sequence.
Hailed as a flexibility in Python literature, it is possible to add an additional sibling class into a given hierarchy, thereby redefining parent method definitions (or adding new ones), so that the elder sibling appears to have these capabilities from the viewpoint of all classes below. Thus, incorrect or malicious code can be inserted into already validated code.
As explained in clause 5.1.4, there are situations in which Python cannot establish a consistent MRO, in which case the `TypeError` exception is raised. For a discussion of vulnerabilities related to unhandled exceptions, see clause 6.36.
There are no language mechanisms to enforce class invariants when methods are redefined, so that class invariants can be easily violated by redefinitions.
To enforce the use of getter and setter methods to access class members, Python provides a mechanism to make members effectively private: the use of leading double underscores (without matching trailing underscores) for their name implies only local visibility in Python.
Any inherited methods are subject to the same vulnerabilities that occur whenever using code that is not well understood.

Static type analysis tools can detect issues associated with complex class hierarchies. Python's type hints provide valuable information to static analysis tools. Similarly, in multiple inheritance situations, displaying the MRO sequence assists developers in understanding the method binding.
See also 6.44 Polymorphic variables [BKK].

6.41.2 Avoidance mechanisms for language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.41.5.
- 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 or the contract model [BLP]

6.42.1 Applicability to language

Python is subject to violations of the Liskov substitution rule as documented in ISO/IEC TR 24772-1:2019 clause 6.42. The Python community provides static analysis tools for Python, which detect most instances of such violations.

6.42.2 Guidance to language users

Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.42.5. In particular, use software static analysis tools to detect such violations.

6.43 Redispatching [PPH]

6.43.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 exists in Python. By default, all calls in Python resolve to the method of the controlling object, a semantics that ISO/IEC TR 24772-1:2019 refers to as redispatching, and thus can result in infinite recursion between redefined and inherited methods, as described in ISO/IEC TR 24772-1:2019.
Redispatching can be prevented by:

- Prefixing the method call by the name of the desired class; or

- Prefixing the method call by `super()` to call on the method found along the MRO of the current class.

See clause 6.44 Polymorphic variables [BKK] for associated vulnerabilities.
The following example shows the infinitely recursive dispatching caused in `h()` and prevented in `f()`:

```
class A:
  def f(self):
    print("In A.f()”)
  def g(self):
    A.f(self) # call to f() in subclass B, will not dispatch
  def h(self):
    self.i()
  def i(self):
    self.h() # call to h() in subclass B, will dispatch
             # showing the vulnerability


class B(A):
  def f(self):
    self.g()
  def h(self):
    self.i() # call to i() in superclass A (infinite recursion)

a = A()
b = B()
b.f() #=> In A.f()
b.h() # RecursionError: maximum recursion depth exceeded
```

6.43.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.43.5.
- 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 [BKK]

6.44.1 Applicability to language

The vulnerabilities as described in ISO/IEC TR 24772-1:2019 clause 6.44 exist in Python in principle, although the mechanisms differ from the ones described in ISO/IEC TR 24772-1:2019.

Python is inherently polymorphic, in the sense that any called operation will attempt to apply itself to the given object and raise an exception if it cannot apply the operation. See clause 5.1.4 for more details. For the vulnerability of unhandled exceptions in the case no operation or method of the respective name is found in the object or class instance, see clause 6.36 Ignored error status and unhandled exceptions [OYB].
While Python has no casting operators as described in ISO/IEC TR 24772-1:2019, prefixing method calls with class names can achieve similar effects for these calls and cause respective vulnerabilities:

- Prefixing a call with the name of a specific class forces the binding of the method name to be taken from this class. There is, however, no check performed whether the named class is an ancestor class of the class of the `self` object, and thus safe to use ("upcast"). Any class is accepted, turning the feature into an unsafe cast in the terminology of ISO/IEC 24772-1. Subsequent failures occur in Python only when the class of `self` does not have members named by the implementation of the chosen method, or, if it does, malfunctions arise when the user semantics of these members are different in the two classes, e.g., a member `count` in two unrelated classes may stand for the count of very different entities, a method `engage` may engage an engine or engage a loving couple, depending on the class involved. Since parameters play no role in method resolution, they do not help in avoiding unintended matches.
- "`super`()" as a prefix to a call ignores local definitions and, instead, picks the binding from the next class in the applicable MRO (often a parent class as in most OO-languages, but occasionally a sibling of the parent class, as shown in the example in clause 5.1.4). As such, it is reasonably safe, since the classes are ancestors of the class of the object, albeit possibly not yielding the expected binding. The vulnerabilities of upcasts, as described in ISO/IEC TR 24772-1:2019, apply in any case. The `super()` function returns a temporary proxy object of the superclass so that its name does not need to be used in the child class. The example below shows how to explicitly call the `__init__` method in the `Foo` superclass by using both the superclass name and the `super()` function. Notice that the `self`-object reference parameter is required when using the `Foo` superclass name. Notice also that, by using `super()`, any changes to the parent class name will not matter as they do for the first call.

```
class Foo(object):
    def __init__(self, msg):
        print(msg)

class DerivedFoo(Foo):
    def __init__(self):
        Foo.__init__(self, '__init__ using Foo')
                                    # => __init__ using
Foo
        super().__init__('__init__ using super()')
                                    # => __init__ using
super()
```



```
DerivedFoo()
```

6.44.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.44.5.
- 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 intrinsics [LRM]

6.45.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.45 applies to Python. 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 as in this example:

```
x = 'abc'
print(len(x)) #=> 3
def len(x):
    return 10
print(len(x)) #=> 10
```

In the example above the built-in `len` function is overridden with logic that always returns 10. Note that the `def` statement is executed dynamically so the new overriding `len` function has not yet been defined when the first call to `len` is made therefore the built-in version of `len` is called in line 2 and it returns the expected result (3 in this case). After the new `len` function is defined it overrides all references to the builtin-in `len` function in the script. This can later be "undone" by explicitly importing the built-in `len` function with the following code:

```
from builtins import len
print(len(x)) #=> 3
```

It is very important to be aware of name resolution rules when overriding built-ins (or anything else for that matter). In the example below, the overriding `len` function is defined within another function and therefore is not found using the LEGB rule for name resolution (see clause 6.21 Namespace issues [BJL]):

```
x = 'abc'
print(len(x)) #=> 3
def f(x):
    def len(x):
        return 10
```

```
print(len(x)) #=> 3
```

6.45.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.45.5.
- 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 [TRJ]

6.46.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.46 applies to Python.

6.46.2 Guidance to language users

Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.46.5.

6.47 Inter-language calling [DJS]

6.47.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.47 is mitigated in Python, which has documented API’s for interfacing with other languages. In particular, Python has an API that extends Python using libraries coded in C or C++. The library or libraries are then imported into a Python module and used in the same manner as a module written in Python. The full API exposed to the “C” language by the CPython reference interpreter is documented in the “Python/C API Reference Manual”. The section in the Python/C API Reference Manual entitled “Extending Python with C or C++” provides a low level example of writing an extension module from scratch using that API.
Conversely, code written in C or C++ can embed Python. The standard for embedding Python is documented in “Embedding Python in Another Application”.

6.47.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 47.5, especially when interfacing to a language without a predefined API.
- 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.
  Note: Python maintainers recommend that developers use existing libraries and tools that automatically generate the Python interface code from simpler descriptions of intent, such as those covered in https://packaging.python.org/guides/packaging-binary-extensions/ such as Cython, cffi, and SWIG.
- 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 [NYY]

6.48.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.48 applies to Python. Python supports dynamic linking by design. The `import` statement fetches a file (known as a module in Python), compiles it and executes the resultant byte code at run time. This is the normal way in which external logic is made accessible to a Python program. Therefore, Python is inherently exposed to any vulnerabilities that cause a different file to be imported:

- Alteration of a file directory path variable to cause the file search locate a different file first.
- Overlaying of a file with an alternate file.

Python also provides an `eval` and an `exec` statement. The `exec` statement compiles and executes statements (example: `x=1,` a line that requires execution). The `eval` statement evaluates expressions (example, `1+1,` composed of operators and expressions). Both statements can be used to create self-modifying code:

```
x = "print('Hello ' + 'World')"
eval(x) #=> Hello World

program = \

     "a = 5"\

     "b = 10"\

     print("Sum =", a+b)"

exec(program) # Output: Sum = 15
```

Guerrilla patching, also known as monkey patching, is a way to dynamically modify a module or class at run-time to extend or subvert their processing logic and/or attributes. It can be a dangerous practice because once "patched" any other modules or classes that use the modified class or module may unwittingly be using code that does not do what is expected, which could cause unexpected results.[2].
Python Enhancement Proposals (PEP) 551 and 578 address issues involved with calling the default entry point and recommends language enhancements to provide better protection. They also provide guidance to eliminate the default behaviour.

6.48.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.48 clause 6.48.5.
- 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.

[2] Python, by default, has the potential to execute dangerous code without detection or verification. Python's default entry point (python.exe on Windows, and python 3.9 on other platforms) allows execution from the command line and does not have hooks enabled. Production software that uses modified entry points and logs as many events as possible can reduce most of these risks.

- 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 [NSQ]

6.49.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.49 is mitigated in Python, which provides an extensive API for extending or embedding Python using modules written in C, Java, and Fortran. Extensions themselves have the potential for vulnerabilities exposed by the language used to code the extension, which is beyond the scope of this document.
Python does not have a library signature-checking mechanism, but its API provides functions and classes to help ensure that the signature of the extension matches the expected call arguments and types. See 6.34 Subprogram signature mismatch [OTR].
However, Python v3.8 does provide an API that gives access to various runtime, import and compiler events. The information gathered from these events can be used to detect, identify and avoid malicious activity. For example, `sys.audithook` can be used to add a callback function for a predefined set of events. The callback function receives the name of the event as well as arguments that can be used for monitoring and filtering. These monitored events could be used to evaluate third party components for suspicious activity during runtime, reducing the inherent risks associated with external modules. These new hooks are especially useful in situations where third-party source code is either unavailable or too large to evaluate for malicious activity.

6.49.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.49.5.
- 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 [HJW]

6.50.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.50 applies to Python.

Python is often extended by importing modules coded in Python and other languages. For modules coded in Python, the risks include the interception of an exception that was intended for a module's imported exception handling code and vice versa.
For modules coded in other languages, the risks include:

- Unexpected termination of the program.
- Unexpected side effects on the operating environment.

6.50.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.50.5.

6.51 Pre-processor directives [NMP]

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.51 does not apply to Python since Python does not have a preprocessor.

6.52 Suppression of language-defined run-time checking [MXB]

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.52 is not applicable to Python because Python does not have a mechanism for suppressing run-time error checking. The only suppression available is the suppression of run-time warnings using the command line "`-W`" option that suppresses the printing of warnings but does not affect the execution of the program.

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.53 applies to Python. Even though there is no way to suppress error checking or bounds checking in Python, there are a few features that are inherently unsafe:

- Interfaces to modules coded in other languages since they could easily violate the security of the calling of embedded Python code (see 6.47 Inter-language calling).
- Use of the `exec` and `eval` dynamic execution functions (see 6.48 Dynamically-linked code and self-modifying code).
- Similarly, `logging.dictConfig` can end up running arbitrary code.
- Python permits user-defined modifications of the contents of module `builtins`. Doing so, however, can be unsafe unless the redefinition matches all of the semantics of the original built-in function, including future enhancements. Overriding Python's default behaviour, by either overriding Python's built-in functions or hiding it or a built-in variable by a user-defined variable of the same name, can have undesired side effects and can be difficult to debug.
- The `pickle` module is inherently unsafe since it allows arbitrary, and potentially malicious, code execution.
  - `Pickle` can spawn anything that Python can invoke including the web browser. To mitigate this risk, whitelists of Python `built-in` functions that are deemed to be

expected and acceptable can be created, and all other functions disallowed.

- Older Python 2 `pickle` protocols can be `ASCII` and slow (protocol=0) making them especially prone to DOS attacks. Python 3 defaults to higher protocols (2-4, binary). The anticipated protocol to be used is determined when pickled, not unpickled, but an attacker can choose various protocols. This risk can be reduced by not using protocol 0.
  - Pickle bombs (self-referencing payloads) can make a small payload expand to an extremely large object in memory resulting in DOS or other attacks. There are legitimate use cases for self-referencing payloads, but in order to minimize the chance of it being misused and potentially leading to a DOS attack, self-referencing payloads can be disallowed.
  - Usage of pickle for long-term storage increases the risk of attack, due in part to many more pickle payloads that are accepted than generated, and to evolving protocol and Python version changes.

6.53.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.53.5.
- 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 [BRS]

6.54.1 Applicability of language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.54 applies to Python. Some examples of obscure language features in Python are:

- Functions are defined when executed:

```
a = 1
while a < 3:
if a == 1:
   def f():
       print("a must equal 1")
else:
   def f():
```

```
        print("a must not equal 1")
f()
a += 1
```

The function `f` is defined and redefined to result in the output below:

```
a must equal 1
a must not equal 1
```

- A function's variables are determined to be local or global using static analysis: if a function only references a variable and never assigns a value to it then it is assumed to be global otherwise it is assumed to be local and is added to the function's namespace. This is covered in some detail in 6.22 Initialization of variables [LAV].
- A function's default arguments are assigned when a function is *defined*, not when it is *executed*:

```
def f(a=1, b=[]):
     print(a, b)
     a += 1
     b.append("x")
f()
f()
f()
```

The output from above is typically expected to be:

```
1 []
1 []
1 []
```

But instead it prints:

```
1 []
1 ['x']
1 ['x', 'x']
```

This is because neither `a` nor `b` are reassigned when `f` is *called* with *no* arguments because they were assigned values when the function was *defined*. The local variable `a` references an immutable object (an integer) so a new object is created when the `a += 1` statement is created and the default value for the `a` argument remains unchanged. The mutable list object `b` is updated in place and thus "grows" with each new call.

- The += operator does not work as might be expected for mutable objects:

```
x = 1
x += 1
print(x) #=> 2 (Works as expected)
```

But when we perform this with a mutable object:

```
x = [1, 2, 3]
y = x
print(id(x), id(y)) #=> 38879880 38879880
x += [4]
print(id(x), id(y)) #=> 38879880 38879880
```

```
x = x + [5]
print(id(x), id(y)) #=> 48683400 38879880
print(x, y) #=> [1, 2, 3, 4, 5] [1, 2, 3, 4]
```

The += operator changes `x` in place while the `x = x + [5]` creates a new list object which, as the example above shows, is not the same list object that `y` still references. This is Python's normal handling for all assignments (immutable or mutable) – create a new object and assign to it the value created by evaluating the expression on the right hand side (RHS):

```
x = 1
print(id(x)) #=> 506081728
x = x + 1
print(id(x)) #=> 506081760
```

- Equality (or equivalence) refers to two or more objects having the same value. It is tested using the == operator which can thought of as the 'is equal to test'. On the other hand, two or more *names* in Python are considered identical only if they reference the same object (in which case they would, of course, be equivalent too). For example:

```
a = [0,1]
b = a
c = [0,1]
a is b, b is c, a == c #=> (True, False, True)
```

`a` and `b` are both names that reference the same objects while `c` references a different object which has the same *value* as both `a` and `b`.

- Python's `pickle` module provides built-in classes for persisting objects to external storage for retrieval later. The complete object, *including its methods*, is serialized to a file (or DBMS) and re-instantiated at a later time by any program which has access to that file/DBMS. This has the potential for introducing rogue logic in the form of object methods within a substituted file or DBMS.
- Python supports passing parameters by keyword as in:

```
a = myfunc(x = 1, y = "abc")
```

This can make the code more readable and allows one to skip parameters. It can also reduce errors caused by confusing the order of parameters.
See also 6.59 Concurrency – activation.

6.54.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.54.5.
- 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 [BQF]

6.55.1 Applicability of language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.55 applies to Python to a small extent, as follows:

- The sequence of keys in a set is unspecified because the hashing function used to index the keys is likely to yield different sequences depending on the implementation.
- Python sets are unordered and unindexed, thus cannot be sorted. Any attempt to sort them has unspecified behaviour. In addition, other functions that depend on order, such as `min(), max(),` and `sorted()` have unspecified behaviour over sets.
- When creating persisting objects, if an exception is raised then an unspecified number of bytes may have already been written to the file.
- Pickling can result in unspecified behaviour as documented in clause 6.53.1 Provision of inherently unsafe operations [SKL].
- Python uses string Interning which is a process of storing only one copy of each distinct string value (up to 4096 characters in length) in memory. For efficiency reasons, whether a string will be *interned* and the interning mechanism that Python uses for strings and integers varies depending on object characteristics. For example, when a copy of a string that meets certain characteristics is created in Python, the copy points to the same object as the original:

```
a =
'SimpleStringWithOnlyASCIILetters_Digits123_And_Underscores
'
b =
'SimpleStringWithOnlyASCIILetters_Digits123_And_Underscores
'
print(a == b, a is b) #=> True True
```

All other strings, such as those longer than 4096 characters and contain any character that is not an ASCII letter, digit, or underscore, will not be interned:

```
a = 'Non-Simple String!' # ' ' and '!' prevent this
                         # string from being interned
b = 'Non-Simple String!'
print(a == b, a is b) #=> True False
```

If memory optimization is required for non-simple strings, optimization can be enforced by using the `intern()` function:

```
from sys import intern
a = intern('Non-Simple String!')
b = intern('Non-Simple String!')
print(a == b, a is b) #=> True True
```

For integers within the range [-5:256], Python optimizes duplicate assignments but, for all other values, each replicated variable points to its own unique object:

```
a = 257
b = 257
print(a is b) #=> False
```

Note: Running the preceding examples of interning in an IDE, such as PyCharm, will give different results since these interning rules may be overridden depending on the IDE. To achieve the results that are shown in these examples, the command line was used.

- Form feed characters used for indentation have an unspecified effect on the character count used to determine the scope of a block.

6.55.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 55.5.
- 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 [EWF]

6.56.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.56 applies to Python. Python has undefined behaviour in the following instances, among others:

- The behaviour of the `Future` class encapsulating the asynchronous execution of a callable is undefined if the `add_done_callback(fn)` method (which attaches the callable `fn` to the future) raises a `BaseException` exception.
- Modifying the dictionary returned by the `vars()` and `locals()` built-ins have undefined effects when used to retrieve the dictionary (that is, the namespace) for an object. The `vars()` built-in can accept an optional object as a parameter `vars(obj)` and, in this case, the returned value is not undefined but depends on the type of the parameter object.

- The `catch_warnings` function in the context manager can be used to temporarily suppress warning messages but it can only be guaranteed in a single-threaded application otherwise, when two or more threads are active, the behaviour is undefined.
- When sorting a list using the `sort()` method, attempting to inspect or mutate the content of the list will result in undefined behaviour.
- Undefined behaviour will occur if a thread exits before the main procedure, from which it was called.

6.56.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.56.5.
- 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 [FAB]

6.57.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.57 applies to Python. For example, Python has implementation-defined behaviour in the following instances:

- Byte order (little endian or big endian) varies by platform.
- Exit return codes are handled differently by different operating systems.
- The characteristics, such as the maximum number of decimal digits that can be represented, vary by platform.
- The filename encoding used to translate Unicode names into the platform's filenames varies by platform.
- Python supports integers whose size is limited only by the memory available. Extensive arithmetic using integers larger than the largest integer supported in the language used to implement Python will degrade performance.
- The type of garbage collection algorithm used, such as reference counting, mark and sweep, is implementation-defined. Depending upon the algorithm used, additional programmer action may be required to prevent memory leakage.
- The maximum value that a variable of type `Py_ssize_t` can take is implementation defined and documented by `sys.maxsize`.

6.57.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.57.5.
- 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 [MEM]

6.58.1 Applicability to language

The vulnerability as described in ISO/IEC TR 24772-1:2019 clause 6.58 applies to Python. For example, the following features were deprecated in Python:

- The `string.maketrans()` function is deprecated and is replaced by new static methods, `bytes.maketrans()` and `bytearray.maketrans()`. This change solves the confusion around which types were supported by the string module. Now, `str`, `bytes`, and `bytearray` each have their own `maketrans()` and `translate` methods with intermediate translation tables of the appropriate type.
- The syntax of the `with` statement now allows multiple context managers in a single statement:

```
with open('mylog.txt') as infile, open('a.out', 'w') as
outfile:
    for line in infile:
         if '<critical>' in line:
             outfile.write(line)
```

  With the new syntax, the `contextlib.nested()` function is no longer needed and is now deprecated.
- Deprecated `PyNumber_Int()`. Use `PyNumber_Long()` instead.
- Added a new `PyOS_string_to_double()` function to replace the deprecated functions `PyOS_ascii_strtod()` and `PyOS_ascii_atof()`.
- Added `PyCapsule` as a replacement for the `PyCObject` API. The principal difference

is that the new type has a well-defined interface for passing typing safety information and a less complicated signature for calling a destructor. The old type had a problematic API and is now deprecated.

- Warnings resulting from `DeprecationWarning` are shown by default but only when triggered by code running in the __main__ module.

6.58.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.58.

6.59 Concurrency – activation [CGA]

6.59.1 Applicability to language

The vulnerability as described in TR 24772-1 clause 6.59 applies to Python.
Python provides multiple concurrency models, see clause 5.1.5.
The vulnerabilities associated with the threading model are:

When a thread is created, if the new thread fails to be created for any reason, then an exception is thrown in the execution path of the creator, which can take corrective action. Hence this vulnerability does not exist for Python threads.

On the other hand, if a child thread has already been started, then attempting to start it again will result in an exception, and the behaviour of the program is implementation-defined. This applies even if the started thread has completed.

This scenario can lead to deadlock and race conditions when activating a thread, and is not always observable even during extensive testing, so it is important to prevent it during development so that it does not surface later.

The ThreadPoolExecutor enables a predetermined number of threads to be created in advance and available for work. Otherwise, creating and then destroying threads in Python has significant overhead associated with it so keeping a pool of threads available eliminates the creation/destruction process. The `join()` operation is also performed automatically so that is another benefit.

The vulnerabilities associated with the multiprocessing models are:

Since the processing model used is that of the underlying operating system and all process interactions are those of the OS, the vulnerabilities are those of the underlying OS.

Calling `set_start_method()` more than once on the same child process causes an exception. Calling it conditionally, for example with `'if __name__ == '__main__'` clause ensures that a process can be started only by a module called `'__main__'`.

The vulnerabilities associated with the `'asyncio'` model are:

Traditional threading or processes are not used in the creation of new 'async' entities, so the vulnerabilities associated with failing to initiate new concurrent entities do not apply. Vulnerabilities associated with communication between the 'async' entity and the initiating entity are addressed in 6.61 *Concurrency - data access [CGX]* and 6.63 *Concurrency – Lock protocol errors [CGM]*.

The `asyncio.run()` function manages the asyncio event loop. It cannot be called when another asyncio event loop is running in the same thread. Its design requires that it be used as the main entry point for asyncio programs and only be called once.

Deleted: or run (and completed)

Deleted: in the parent thread

Deleted: See 6.36 *Ignored runtime errors and unhandled exceptions* for vulnerabilities associated with exception handling.…

If any task in an event loop blocks, it runs the risk of never being restarted if the event loop ends before the block condition completes. Many functions in the Python standard library incur blocking, and therefore are subject to this issue. Therefore, many libraries also exist in non-blocking versions.
Managing multiple asyncio events can be error prone. Python provides a *debug mode* to help identify and catch common issues, as documented in [Ref]

COMMON VULNERABILITIES DISCUSSION

In each of the three forms of concurrency discussed above, there is a risk that some concurrent part of the program will incur an exception, which may or may not result in notification of the main body of the program. See 6.62 Concurrency -- Premature termination [CGS] for issues associated with such vulnerabilities.
The threat of deadlocks by mutual dependence among futures is analogous to deadlocks of threads and processes. For example:

```
from concurrent.futures import ThreadPoolExecutor
import time

def foo_a():
    time.sleep(1)
    print(b.result())
    return 1

def foo_b():
    print(a.result())
    return 2

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(foo_a) # waits indefinitely on b
b = executor.submit(foo_b) # waits indefinitely on a
```

Additional vulnerabilities can arise if a single Python program attempts to use multiple concurrency models, since the different models use different mechanisms for creation, scheduling, communication and termination.

6.59.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.59.5 for activation of processes or threads or asyncio tasks.
- 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.

Commented [SM21]: MMM - Stephen to try to write wording

Commented [SM22]: https://docs.python.org/3/library/asyncio-dev.html#asyncio-logger

Commented [SM23]: Candidate for removal. It is discussed in 6.62.

Deleted: vulnerabilitites

Deleted:

Commented [SM24]: Reference https://docs.python.org/3/library/asyncio-dev.html#asyncio-logger

- 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 [CGT]

6.60.1 Applicability to language

The vulnerability as described in TR 24772-1 clause 6.60 applies to Python.
As in 6.59.1, we separate the discussion into the three Python concurrency model.
Threading model

In Python, a thread may terminate by coming to the end of its executable code or by raising an exception. Python does not have a public API to terminate a thread. This is by design since killing a thread is not recommended due to the unpredictable behaviour that results. There are, however, dangerous work-arounds that can terminate Python threads by using calls to the operating system or the `ctypes` foreign function library. These workaround techniques can lead to deadlock, data corruption, and other unpredictable behaviour as described in ISO/IEC 24772-1 clause 6.60.
The preferred way to terminate an executing thread is to send it a message, signal or event to terminate itself, and then wait for the termination to occur (using `join()`, `is_alive()`).
The parent of a thread can determine if the child has completed either by repeated calls to `is_alive()` or by executing the `join()` statement. The `join()` operation has an optional timeout parameter to reduce the risk of infinite waiting and to provide the possibility for corrective action. The `join()` operation does not return a final result (except `None`), hence joining another thread or process multiple times does not affect the calling entity after the first call which awaits completion of the joined entity.
There are a number of possible errors associated with the joining of threads:

- Failure to join a completed thread can result in logic errors;
- Joining multiple children in an order different than the expected completion of those children can cause extended or indefinite delays;
- Attempting to join the current thread will result in an exception; and
- Any attempts to communicate with another thread after joining that entity can result in significant errors, such as a logic error, an exception or indefinite delays.

A particular challenge is the scenario of daemon threads. Inside a program, if a thread is created with the flag `daemon = true`, the termination of that thread is disconnected from the termination of the thread that created it. In addition, a `join()` on a daemon thread without a specified timeout will not return.

Multiprocessing model

Since processes are entities of the underlying operating system, terminating other processes is OS-specific. Processes terminate when they complete their program code, but do not notify the creating process; the programmer is responsible to communicate final results or a termination notice before each process terminates.

Commented [25]: yyy AI – Sean - Missing discussion on time consumption by termination/finalization code. Contradictory sentences in 6.60.1. Is it or is it not possible to kill another thread in Python???

Commented [MS26R25]: The only safe way to end a thread is from within the thread itself so that all locks can be released prior to termination. This can be done by having the thread periodically check a message queue or a global variable and then, if necessary, the thread itself can release its own locks and exit gracefully. Thus, the "unexpectedly delayed termination or the consumption of resources by the termination itself" depends on the implementation and application. Suggest deleting comment. Ref comment in 6.60.2

Commented [p27]: Needs work. Sean and Stephen to discuss.

Commented [SM28R27]: Close?

Commented [SM29]: yyy - What about subprocesses and tasks?

Commented [MS30R29]: Text modified for processes. Regarding tasks, the exception-inducing command for terminating Tasks can be found at: Coroutines and Tasks — Python 3.9.2 documentation

To cancel a running Task use the `cancel()` method. Calling it will cause the Task to throw a `CancelledError` exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.
`cancelled()` can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the `CancelledError` exception and was actually cancelled. ... [1]

Commented [p31]: This is VERY misleading, given terminate on processes and cancel calls on tasks/futures. Needs f ... [2]

Commented [p32]: In Python, there is no native method available to terminate a thread. Terminating an external thread is possible via OS calls or by using ctypes, but th ... [3]

Deleted: a

Deleted: conditions

Deleted: s

Deleted: The `join()` operation does not return a final result (except `None`), hence joining another thread or process multiple times does not affect the calling ent ... [4]

Deleted:

Deleted: Calling `join()` with a non-empty timeout together with `is_alive()` permits the calling thread to test the progress of a child. Calling `join()` with an ... [5]

Commented [MS33]: The ordering of the three concurrency models should probably remain consistent in Sections 5.59 thru 6.63. Section 5.59 has them listed in the following ... [6]

Commented [SM34R33]: We can delete this comment

The preferred way to terminate an executing a process is to send it a command to terminate itself, and then wait for the termination to occur using 'join'.
Processes that have been created typically need to return a result. This is accomplished via the `join()` method. See 6.61 Concurrency – data access [CGX]. There are a number of possible errors associated with the joining of threads or processes:

- Joining multiple child processes in an order different than the expected completion of those children can cause extended or indefinite delays.
- Attempting to `join()` the current process will result in deadlock.
- Using `join()` on a daemon process will result in a deadlock condition

Terminating a process in Python is possible but there are scenarios that may leave the system in a vulnerable state. Terminating a process that has acquired a lock or semaphore can result in a deadlock condition. For example, executing `terminate()` on a process that is using a pipe or queue may result in data corruption (See 6.6x TBD). Similarly, threads and processes that are externally terminated will not execute the 'finally' clause for that thread or process, which can result in logic errors, and if the terminated process has descendants, then the descendants will be orphaned.

A process can determine if another process has completed either by repeated calls to `multiprocessing.Process.is_alive()` or by calling `multiprocessing.Process.join()`. Calling `join()` with a non-empty timeout together with `is_alive()` permits the calling process to test the progress of the other processes. Calling join with an empty timeout value causes the process to await the completion of the other process.

Asyncio Model

- When the primary task terminates one or more dependent tasks; or

(The above came from 6.62 Premature termination and needs integration?)

For the second scenario, the directed termination of one or more dependent units may leave the program in an unexpected state. See 6.60 Concurrency – Directed termination [C??]

Termination of the event loop
When asyncio actions are scheduled and the parent is terminated, then the event loop is terminated with a runtime error possibly before some futures are delivered and program termination completes. If a controlled termination is required (external to the event loop), Python recommends to terminate the event loop owner with an exception, catch the exception, and send each asyncio event a `stop()` or a run_until_complete() directive to finish processing already-scheduled events and then cease processing. Once the event loop has completed it can be "`close()`"'d (after collecting results).
The following example shows another way to terminate an event loop that is interrupted by an exception. In general, such an exception would cause the concurrent iterations to be in an abnormal state. The associated "`finally`" clause cleans them up and terminates them.

```
Try:
    loop.run_forever()
```

Deleted: The parent of a thread can determine if the child has completed either by repeated calls to `is_alive()` or by executing the `join()` statement. Calling `join()` with a non-empty timeout together with `is_alive()` permits the calling thread to test the progress of a child. Calling join with an empty timeout value causes the threat to await the completion of the child thread.¶

Commented [SM35]: SSS – need a paragraph to document futures and ThreadPoolExecutor.

Commented [MS36R35]: This paragraph is at the end of this section

Commented [MS37R35]: This paragraph has been moved to '5.1.5 Concurrency'

Commented [SM38]: SSS – Sean, find a better place for this. While it is true, is is not specific to process creation.

Commented [MS39R38]: 6.61 seems like a good home for it as suggested in the text. Agree?

Commented [SM40]: Externally **what?** terminated?

Commented [MS41R40]: Even though killing threads in Python is not recommended, it is possible externally terminate threads using **signals**. As stated in: How to terminate running Python threads using signals | G-Loaded Journal

"The first most important thing to remember is that all signal handler functions must be set in the *main* thread, as this is the one that receives the signals. Registering signal handlers within the thread objects is wrong and *doesn't work*." Also, "Some care must be taken if both signals and threads are used in the same program. The fundamental thing to remember in using signals and threads simultaneously is: always perform signal() operations in the main thread of execution. Any thread can perform an alarm(), getsignal(), pause(), setitimer() or getitimer(); only the main thread can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the Python signal module, even if the underlying thread implementation supports sending signals to individual threads). This means that signals can't be used as a means of inter-thread communication. Use locks instead."

Commented [SM42]: From the docs: https://docs.python.org/3/library/asyncio-task.html

cancel(*msg=None*)
"Request the Task to be cancelled.
This arranges for a CancelledError exception to be thrown into the wrapped coroutine on the next cycle of the event loop.
The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a try … … except CancelledError … finally block. Therefore, unlike Future.cancel(), Task.cancel() does not guarantee that t ... [7]

Commented [SM43]: This could be accomplished with a global flag or by using one of the following wait conditions:
• FIRST_COMPLETED — Returns when the first task completes. ... [8]

Deleted: without explicit terminations

Deleted: A superior way is

```
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

A event loop can also await the completion of a selected set of tasks.

Termination of asyncio tasks

To direct the termination of an asyncio task, one can set a shared variable that will direct asyncio task to terminate itself. As documented in ISO/IEC 24772-1 clause 6.60.3, the asyncio task can:

- Not detect the termination request;
- Detect and obey the termination request; or
- Detect and ignore the termination request.

In all cases, the vulnerabilities documented in ISO/IEC 24772-1 clause 6.60.3 apply to asyncio tasks.

Another mechanism is to asynchronously raise the `CancelledError` exception in an asyncio task via the `cancel` method in the `asyncio.Task` class. If this exception is ignored, the recipient task is transferred to its `finally` portion. Vulnerabilities associated with unhandled exceptions are addressed in clause 6.36 Ignored error status and unhandled exceptions[???]. If the exception is caught, the recipient task may:

- Complete;
- Report the error condition and complete; or
- Take alternative action and continue processing.

In any of the above cases, the vulnerabilities documented in ISO/IEC 24772-1 clause 6.60 apply to Python asyncio tasks.

Common Vulnerabilities

The termination of any concurrent activity can consume significant time and resources, e.g. because of finalization. Thus there is a risk of timing errors for the remaining concurrent entities.

6.60.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.60.5.
- 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.

Commented [SM44]: SSS - check if there is a mechanism to close individual asyncio tasks, or if one always closes the event loop. If the second, then this part is essentially blank.

Commented [SM45R44]: Previous response to this comment is now incorporated into the text

Deleted: n

Deleted: d

Deleted: d

Deleted: threads

Deleted: , and consider u

Deleted: ing

Deleted: ¶

Deleted: <#>Use care when externally terminating processes since `finally` clauses will not be executed, and descendant processes will not be terminated. ¶

Deleted: or

Deleted: ¶
Consider using `Process.terminate()` only on processes that never use shared resources and

Deleted: ensure that the termination is fail-safe and ends the process gracefully.

Deleted: ¶
Ensure that no thread is waiting on daemon threads to complete since these threads are always running.

6.61 Concurrency - data access [CGX]

6.61.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.61 applies to Python. The traditional accesses to shared data, and the locking and unlocking of locks that protect shared data are as described in ISO/IEC TR 24772-1:2019 clause 6.61.

Threading model

Threads and events can share memory, and care is required to coordinate the update and consumption of such memory. This is not restricted to "global" data since nesting of threads will effectively make all variables of the outermost thread 'shared'.

Some Python interpreters use a Global Interpreter Lock (GIL) which ensures that only a single bytecode is executed at a time. This may guarantee that access to primitive data objects are serialized, but does not guarantee serialization of data access between threads or asyncio tasks in general.

When using multiple threads, if certain events need to occur sequentially, putting these events into the same thread guarantees sequential access, reduces the need for locks and minimizes the chance for data corruption and race conditions.

When global variables are needed to communicate between functions within a single thread in a multithreaded application, visibility of the data to other threads (and the possibility of data corruption and race conditions) can be avoided by using the `threading.local()` function. This creates a local copy of the global variable in each thread that executes that call. Threads that do not create a local copy see (and can update) the global variable. Confusion can result if some threads maintain a local copy and others do not.

All other shared access to variables require that the data be locked before access and unlocked after. See 6.63 Protocol lock errors.

Multiprocessing model

Python processes do not share memory and therefore are not subject to data access errors between the processes, however, access errors can occur for objects such as those provided by multiprocessing.sharedctypes or maintained by the operating system and shared by processes, such as files. For such objects, the vulnerability exists.

Interprocess communication mechanisms such as pipes can exhibit concurrency control errors, see 6.63 Lock protocol errors [CGM]. Note that the use of pipes or queues to move significantly large amounts of data can reduce complexity related to global locks at the expense of performance, which can cause the application to run too slowly and/or miss deadlines.

Pipes and queues are designed such that one process writes to a pipe or queue and a second process reads from it. If one of the processes contains threads, and multiple threads attempt to access the same pipe or queue, then there is a risk of data corruption since the order of access cannot be guaranteed. Indeed, the use of more than one concurrency model in the same application makes the application susceptible to uncoordinated data accesses.

Asyncio model

A fundamental principle in writing asyncio tasks is that each iteration of a task (from the point where its data is ready for processing and where it suspends for the next iteration) is atomic with respect to the other tasks. It is a fundamental error to split calculations or

Deleted: <#>¶

Moved (insertion) [1]

Deleted: ¶
Pipes and queues are designed such that one process writes to a pipe or queue and a second process reads from it. If one of the processes contains threads, and multiple threads attempt to access the same pipe or queue, then there is a risk of data corruption since the order of access cannot be guaranteed. Indeed, the use of more than one concurrency model in the same application makes the application susceptible to uncoordinated data accesses.¶
Note that the use of pipes or queues to move significantly large amounts of data can reduce complexity related to global locks at the expense of performance. Either can cause the application to run too slowly and/or miss deadlines.

Commented [MS50]: Ref. Python Core Developer Raymond Hettinger:
Threading Example — PyBay 2017 Keynote documentation
RR1001

Commented [MS51]: RR 1004 – "Sometimes you need a global variable to communicate between functions. Global variables work great for this purpose in a single threaded program. In multi-threaded code, it mutable global state is a disaster. The better solution is to use a threading.local() that is global WITHIN a thread but not without."

Commented [SM52]: SSS check on various ways to declare and use threading.local data.

Deleted: ¶

Commented [MS53R52]: Below is a very basic example that illustrates how to use threading.local(). We can discuss the possibility of including it in the document (tutorial?):

```
import threading

userName = threading.local()

def Func(name_id):
   userName.val = name_id
   print(userName.val)

Thread1 = threading.Thread(target=Func("Name1"))
Thread2 = threading.Thread(target=Func("Name2"))

# start the threads
Thread1.start()
Thread2.start()

# wait for threads to complete
Thread1.join()
Thread2.join()
------- OUTPUT ------------
Name1
Name2
```

Commented [MS54R52]: The updated text addresses the general vulnerability concern (confusion) and an example here probably does not add much value. Accept this comment?

shared data access between iterations of the same task, since other tasks can access change the data between iterations.

When using asyncio, correct operation requires that all tasks relinquish control co-operatively with execution controlled by the Async IO manager. Since task switching is less arbitrary than thread context switching when cooperative transfers of control between coroutines are used., i.e. `await()` and `yield()` to provide predictable control over the task switching process.
It should be noted that when a task has performed an `await()` or a `yield()`, no assumptions can be made once restarted about any shared data that it was relying upon, and such data must be reread before further processing.

6.61.2 Avoidance mechanisms for language users

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.61.5.
- 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 [CGS]

6.62.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.62 applies to Python. Premature termination of any concurrent part of the program exposes all other portions of the program to the risk of logic errors, regardless of which concurrency model is used in the program. Python provides syntax to detect and diagnose many common premature termination scenarios that will let the program recover and continue, as discussed below.
Threading model
The termination of the main thread awaits the termination of all non-daemon children; it then terminates the daemon children and stops.
Exceptions in a thread at any level can be caught by a `try` clause at the outermost level of that thread; and `finally` clauses will be executed in the presence or absence of exception handling. Exceptions unhandled by a thread cause the invocation of the `thread.exceptHook()` method which can be programmed by the user. The default implementation of `thread.exceptHook()` causes silent termination of the thread.
All these mechanisms provide the opportunity to implement the necessary communication between threads about their termination state.

Deleted: Async IO tasks are prevented from making blocking calls, and switch cooperatively

Deleted: via

Commented [MS55]: Copied into 6.63 Protocol lock errors. Need to decide where to put it.

Moved up [1]: Pipes and queues are designed such that to a pipe or queue and a second process reads from it. If one of the processes contains threads, and multiple threads attempt to access the same pipe or queue, then there is a risk of data corruption since the order of access cannot be guaranteed. Indeed, the use of more than one concurrency model in the same application makes the application susceptible to uncoordinated data accesses.¶
Note that the use of pipes or queues to move significantly large amounts of data can reduce complexity related to global locks at the expense of performance. Either can cause the application to run too slowly and/or miss deadlines. ¶
Use `join()` on all threads that need to be completed before moving forward in the program otherwise there can be unexpected behaviour and possible data corruption. Do not use `join()` on any thread before starting it and only use it once per thread or an exception will be thrown, Do not use `join()` on a daemon thread since will result in a deadlock condition. ¶

Moved down [2]: If a child thread has put items in a queue and it has not used JoinableQueue.cancel_join_thread, then

Deleted: Async IO requires all calls to be non-blocking.

Deleted: ¶ ... [9]

Commented [p60]: Here, too, any guidance ought to ... [11]

Deleted: Whenever using a queue with multiprocessing, ensure that all items which have been put on the queue ... [10]

Deleted: Guidance to

Deleted: Follow the guidance contained in

Deleted: TR

Deleted: :2019

Moved (insertion) [4]

Deleted: no

Deleted: unprotected

Deleted: is used directly by more than one thread

Deleted: .

Deleted: are ordered correctly and

Deleted: in different threads

Moved up [4]: When using multiple threads, verify that no unprotected data is used directly by more than one thread.

Deleted: .¶

Deleted: If shared variables must be used in multithreaded applications, use model checking or equivalent ... [12]

Commented [MS61]: This scenario can be handled u ... [13]

Commented [MS62]: This scenario can be handled u ... [14]

Deleted: Unexpected exceptions that occur in the main thread can notify all child threads by using a global ... [15]

Any `join()` with the terminated thread is still possible but will not distinguish between normal and exceptional termination. Furthermore, predefined routines such as `threading.is_alive()`, `threading.active_count()`, and `threading.enumerate()` permit querying the state of other threads.

If termination occurs when a thread is accessing a pipe, then the pipe may become corrupted and further accesses can result in an exception or in undefined behaviour. If termination occurs when a thread is accessing a queue, then the queue may remain locked indefinitely and subsequent accesses can result in deadlock. See 6.63 Protocol lock errors.

When using `threading.pool` objects, it is important to properly manage the resources with a context manager or by calling `close()` and `terminate()` explicitly to prevent deadlock during finalization. Relying on Python's garbage collector to destroy the pool will not guarantee that the finalizer of the pool will be called.

Multiprocessing model

If the execution of a process incurs an exception and terminates prematurely, then any communicating processes can fail to receive expected results and can suffer from protocol errors, or themselves can wait indefinitely. OS calls to query the state of other processes are available, hence periodic checking if the other processes are still executable can be used.

`try – except` blocks exist for processes and are similar to `asyncio try – except` blocks.

Any process that terminates prematurely cannot be restarted. *(Check this is also in 6.59)*

Does a separate process terminating because of an exception notify the other processes, especially the main process? Does the termination of the main process cause all child processes to terminate? (Yes for daemonic children) What happens to pipes or queues that are connecting processes?

Something about handling exceptions – handle in method that creates the process or thread.

Unexpected exceptions must be handled when using processes. Exceptions can occur during process initialization, task execution, or task completion. The ProcessPoolExecutor is commonly used to create and manage a pool of worker processes and will be ...

If termination occurs when a process is accessing a pipe, then the pipe may become corrupted and further accesses can result in an exception or in undefined behaviour. If termination occurs when a process is accessing a queue, then the queue may remain locked indefinitely and subsequent accesses can result in deadlock. See 6.63 Protocol lock errors.

- When using `multiprocessing.pool` objects, it is important to properly manage the resources with a context manager or by calling `close()` and `terminate()` manually to prevent deadlock during finalization. Relying on Python's garbage collector to destroy the pool will not guarantee that the finalizer of the pool will be called.

Deleted: The `threading.excepthook()` function can be

Commented [MS63]: Ref: multiprocessing — Process-based parallelism — Python 3.9.6 documentation

Deleted: used to handle uncaught exceptions raised by `Thread.run()`. The following example shows one way that this technique could be implemented:¶

```
from time import sleep
import threading

def foo():
    print('In foo child thread ...')
    sleep(1)
    raise
Exception(threading.current_thread().name)

def custom_hook(args):
    print(f'Thread failed:
{args.exc_value}')

threading.excepthook = custom_hook
thread = threading.Thread(target=foo,
name='My foo thread')
thread.start()
thread.join()
print('Finishing main thread ...')
```

The output from the above example shows that the main thread was able to resume normal execution after handling the exception in the child thread:¶

```
In foo child thread ...
Thread failed: My foo thread
Finishing main thread ...
```

Commented [MS65]: Ref: multiprocessing — Process-based parallelism — Python 3.9.6 documentation

Asyncio model

Premature termination occurs as follows:

- When the primary task terminates due to an exception or unprogrammed event;
- When a dependent task raises an exception or terminates abnormally.

For the first scenario, all dependent tasks will be terminated when the main task terminates, see 6.36 Ignored error status or unhandled exception [???].

For the second scenario, the premature termination of dependent coroutines will almost always affect the execution of other coroutines and the main thread that contains the event loop, unless they also terminate. Otherwise tasks may remain in the event loop indefinitely or until the program terminates. If all programmed tasks are not cooperating effectively, then it is unlikely that the program will execute correctly.

The following methods can be helpful in handling asyncio exceptions:

- get_name() – useful for debugging especially when handling many coroutines
- exception() – returns None if there are no exceptions raised, otherwise returns the exception object. Usually used in the event loop to identify exceptions raised in coroutines.
- result() – returns the result of the coroutine and re-throws any exception that the coroutine raised. This allows propagation back to the caller.

The following example demonstrates a possible use of these methods and ensures that all coroutines are terminated properly:

```
import asyncio

async def foo():
    raise ValueError("foo value error")
    return("foo finished")

async def bar():
    await asyncio.sleep(1)
    return("bar finished")

async def main():
    foo_task = asyncio.create_task(foo(), name="Exception task")
    bar_task = asyncio.create_task(bar(), name="Waiting task")
    try:
        done, pending = await asyncio.wait(
            [foo_task, bar_task],
            return_when=asyncio.ALL_COMPLETED
        )
        for task in done:
            name = task.get_name()
            print(f"DONE: {name}")
            exception = task.exception()
            if isinstance(exception, Exception):
```

Commented [SM66]: Exceptions that occur in coroutines are not necessarily propagated to main() since it is a peer within the event loop. The exception does propagate up to the scheduler and then back to main() at which time the asyncio.wait parameters determine if main() is called on the next pass through the event loop, or whether all tasks have to complete before main() is called. See additional info in comment below.

Deleted: especially when an exception occurs in one or more coroutines. It is important to take

Commented [MS67]: Here is a very good reference that explains asyncio exception handling: https://python.plainenglish.io/how-to-manage-exceptions-when-waiting-on-multiple-asyncio-tasks-a5530ac10f02

Here are some recommendations taken directly from the above reference:

- " *Plan how you will handle pending tasks and exceptions. It is difficult to debug asyncio code when there is no code to explicitly handle these conditions.*
- *There is a high probability that tasks that throw exceptions will return earlier than those that return a value result. This means they will appear earlier, if not first, in the inerrable list. Therefore, the first task for which* `result()` *method is called may throw an exception, in which case you may want to process all other results and cancel pending tasks before re-throwing this error.*
- *The ordering of activities following* `asyncio.wait` *may be important in your implementation. Work out the most appropriate order for cancelling pending tasks, getting results, testing for and handling exceptions.*
- *If more than one task throws an exception you may need to work out how to aggregate them into a single error message that can be re-thrown once all other tasks have been processed."*

Deleted: precautionary steps to ensure that all exceptions are handled properly. Failure to handle exceptions for each coroutine can result in tasks remaining in the event loop indefinitely or until the program terminates.

Deleted: the

Deleted: if the

Deleted: d an exception

```
                print(f"{name} threw {exception}")
            try:
                result = task.result()
                print(f"{name} returned {result}")
            except ValueError as e:
                print(f"ValueError: {e}")
        for task in pending:
            task.cancel()
    except Exception as e:
        print("Outer Exception")

asyncio.run(main())
```

The above example runs successfully and produces the following output:

```
DONE: Waiting_task
Waiting_task returned bar finished
DONE: Exception_task
Exception_task threw foo value error
ValueError: foo value error
```

6.62.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.62.5.
- 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:
  - 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.
  - Handle exceptions, free locks, and clean up nested threads, and shared data before termination.
- For multiprocessing:
  - A
  - B
  - C
- For Asyncio:
  - Ensure consistent termination behaviour of all coroutines
  - B

Commented [MS68]: Using the **`task.result()`** method inside of a 2nd **`try:`** statement causes the exception to be re-thrown and ensures that all tasks are removed from the event loop prior to termination of the calling routine, **`main()`** in this example.

Commented [MS74]: Derived from the Python documentation. Ref: Ref: multiprocessing — Process-based parallelism — Python 3.9.6 documentation

Deleted: IA Python thread will terminate when its run() method terminates or if an unhandled exception occurs. Python does not permit other threads to abort or prematurely terminate other threads when using the threading library, but does provide terminate(), kill(), and close() methods in the multiprocessing library

Deleted: ¶
If termination occurs when a thread or process is accessing a pipe, then the pipe may become corrupted and further accesses can result in an exception or in undefined behaviour. If termination occurs when a thread or process is accessing a queue, then the queue may remain locked indefinitely and subsequent accesses can result in deadlock. See 6.63 Protocol lock errors.¶
If Process.terminate() or os.kill() is used to kill a process, and the associated process is using a pipe or queue, then the pipe or queue will likely be corrupted and may become unusable by other process. If the process has acquired a lock or semaphore, then terminating it will likely cause other processes to deadlock.¶
If a child process has put items in a queue and it has not used JoinableQueue.cancel_join_thread, then that process will not terminate until all buffered items have been flushed from the pipe, and future attempts to join that process may result in deadlock unless all items in the queue have been consumed. If the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children. Note that a queue created using a manager does not have this issue

Deleted: .¶
When using `multiprocessing.pool` objects, it is important to properly manage the resources with a context manager or by calling `close()` and `terminate()` manually to prevent deadlock during finalization. Relying on Python's garbage collector to destroy the pool will not guarantee that the finalizer of the pool will be called. ¶

Moved (insertion) [3]

Deleted: Use

Deleted: `f`

Deleted: keyword for

Deleted: that

Deleted: notifies

Deleted: Use

Moved up [3]: Protect data that would be vulnerable to premature termination, such as by using locks or

Deleted: ¶

Deleted: and

Deleted: and potentially

Deleted: Enable event logging and record all events prior to termination so that full traceability is preserved.

6.63 Concurrency - lock protocol errors [CGM]

6.63.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.63 applies to Python. Python provides locks and semaphores that are intended to protect critical sections of data. All calls to `lock.acquire()` with default parameters guarantee that the calling concurrent unit (thread, process or coroutine) will not continue until the lock is available. Python also provides event objects that permit programmed-specific notification between two concurrent units, as well as barriers and condition objects that permit the release of groups of concurrent units upon a single condition becoming true. However, there are vulnerabilities associated with Python's synchronization mechanisms:

- If a concurrent unit is killed in between `lock.acquire()` and `lock.release()`, every other concurrent unit unconditionally waiting on that lock will be deadlocked.
- Locations where locks are needed can be missed, unless shared resources are accessed exclusively by dedicated functions that act like a traditional monitor.
- The use of locks does not guarantee consistency of shared resources unless all relevant concurrent units check for the locks.
- Every critical section that starts with a `lock.acquire()` must be matched with a `lock.release()`, or the program, or some concurrent units, will deadlock.
- For calls of `lock.acquire(..)` that are parameterized with a time-limit or with the requirement for immediate locking, the omission of checking the result of `lock.acquire(..)` will allow the caller to proceed without acquiring a lock.

XXXXXX

Threading and process models

```
# def increase(lock):
#     global database_value
#
#     lock.acquire()
#     local_copy = database_value
#     local_copy+= 1
#     time.sleep(0.1)
#     database_value = local_copy
#     lock.release() # don't forget this else deadlock

def increase(lock):
    global database_value

    with lock: # better option is to use a context manager
               # since it acquires and releases the lock
               # automatically
        local_copy = database_value
        local_copy+= 1
        time.sleep(0.1)
        database_value = local_copy
```

Commented [SM76]: SSS - split into Threading, Processes, and Asyncio tasks.

Thank you!

Commented [SM77R76]:

Deleted: threads

Deleted: threads

Deleted: thread

Deleted: an a

Deleted: r

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Deleted: thread

Deleted: that

Deleted: s

Deleted: ¶

Deleted: These vulnerabilities can be mitigated by using locks around critical sections of code, but the excessive use of locks becomes difficult to manage and will also negatively impact performance.

Deleted: Identifying all l

Deleted: complicated

Deleted: and t

Deleted: security

Deleted: since locks are only effective if

Deleted: threads

Deleted: The data in a locked critical section in one thread can be modified by another thread if it does not first check for (acquire) the lock.

Deleted: progra

Deleted: m

Deleted:

Formatted: Font: 10.5 pt

Formatted: List Paragraph, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

Formatted: Font: 10.5 pt

Commented [p78]: This is not about pros and cons of locks. This is about unsynchronized data access from any concurrent program units, whatever their names, and its consequences and should be written that way. What can go wrong in Python? And what can the user do about it? (Applies to 61.1. as a whole.)

Moved down [5]: To help ensure that thread locks are released, a context manager should be used as follows:¶
`database_value = 0`

Deleted: To help ensure that thread locks are released ... [16]

Commented [SM79]: Talk about semaphores, or ren ... [17]

Formatted: Font: 10.5 pt

Deleted: r

```
if __name__ == "__main__":
    lock = Lock()
    print('start value', database_value)

    thread1 = Thread(target=increase, args= (lock,)) # tuple so need
                                                     # the comma
    thread2 = Thread(target=increase, args= (lock,))
    # thread1 = Thread(target=increase()) note: this will produce the
                                          # correct result but is
                                          # incorrectly passed to
                                          # execute
    # thread2 = Thread(target=increase())

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    print('end value', database_value)

    print('end main')
```

Also notice in the above example, that passing in the full function name `increase()`, including the parentheses, incorrectly causes the function to run yet gives the correct result. Only pass in the function name `increase`, without parentheses, as the target parameter.

It is important to prevent Python processes or threads from waiting on daemon processes or threads since the daemons never complete until the program exits. To prevent a deadlock condition from occurring, use `join()` on the message queue and wait for all of the requested threads to be marked as done before proceeding.

If a child thread has put items in a queue and it has not used JoinableQueue.cancel_join_thread, then that thread will not terminate until all buffered items have been flushed from the queue to the underlying pipe, and future attempts to join that thread may result in a deadlock unless all items in the queue have been consumed.

When using asyncio, correct operation requires that all tasks relinquish control co-operatively, with execution controlled by the Async IO manager. Since task switching is less arbitrary than thread context switching when cooperative transfers of control between coroutines are used., i.e. `await()` and `yield()` to provide predictable control over the task switching process.

In addition, Python provides API's for interprocess communications, such as pipes and queues.. Some mechanisms are designed to be usable by multiple processes by encapsulating the interface to each in multiprocess-safe calls. For pipes, this does not apply as there is no synchronization between multiple readers or writers of the pipe.

Formatted: Font: 10.5 pt

Formatted: Font: 10.5 pt

Deleted:

Formatted: Font: 10.5 pt

Commented [p80]: This is not about pros and cons of locks. This is about unsynchronized data access from any concurrent program units, whatever their names, and its consequences and should be written that way. What can go wrong in Python? And what can the user do about it? (Applies to 61.1. as a whole.)

Moved (insertion) [5]

Deleted: To help ensure that thread locks are released, a context manager should be used as follows:¶
`database_value = 0`¶

Commented [SM81]: Need to address protocols errors for processes, async_io and concurrent models. Async_io and concurrent likely have less ways of failing but processes have many.

Commented [SM82R81]: This example shows a vulnerability that also happens in sequential code, hence belongs elsewhere. Sean, please look for a place.

Commented [MS83]: RR 1003

Deleted: (?)

Deleted: threads

Moved (insertion) [2]

Field Code Changed

Commented [SM84]: I think this is suspect. Discuss next time.

Commented [MS85]: Possibly move this to language reference section? Also, further research on asyncio behaviours is needed.

Commented [SM86]: Copied from 6.60.1

Deleted: ¶

Deleted: Multiprocessing model¶

Asyncio model

Even though asyncio is single-threaded and can execute only one coroutine at a time, it may sometimes be necessary to use locks. Asyncio coroutines are thread-safe as long as they are using resources that are not shared by other threads or processes. However, if shared data is accessed in critical sections within multiple concurrent coroutines, the data can become inconsistent if one coroutine enters its critical section while another is still within its related critical section. Asyncio provides the `asyncio.Lock` class to protect these critical sections, but these sections are not thread-safe or process-safe, hence cannot be safely shared by any other thread or process. The same instance of the `asyncio.Lock` class must be used by all coroutines that access a shared resource so that race conditions can be avoided.

Since asyncio tasks are not truly concurrent, guaranteeing that no yields are present in critical sections avoids the vulnerability.

6.63.2 Avoidance mechanisms for language users

- Use the avoidance mechanisms of in ISO/IEC TR 24772-1:2019 clause 6.63.5.
- 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.

6.64 Reliance on external format string [SHL]

6.64.1 Applicability to language

The vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.64 applies to Python. Externally controllable strings can result in unexpected behaviour such as buffer overruns, exposure of private data, and other malicious exploits. Python strings share most of the potential security vulnerabilities described in ISO/IEC TR 24772-1:2019 clause 6.64.

Commented [SM87]: Explain in clause 5.

Deleted: However, if a critical section of code is accessed by multiple coroutines concurrently, the state of this critical section may become ambiguous by the other suspended coroutines that share it.

Deleted: instruction

Deleted: instruction

Deleted: a critical section

Deleted: ¶

Formatted: Justified, Space Before: Auto, After: Auto

Formatted: English (US)

Deleted: **Guidance**

Deleted: **to**

Deleted: Follow

Deleted: guidance

Deleted: contained

Commented [88]: yyy The solution in most programming languages is to place all access to such shared data in subprograms that first test-and-set a lock, then manipulate the data, and then release the lock when finished and exit the subprogram. Exception handlers for all exceptions are placed in the subprogram which releases the lock before either exiting or propagating the exception.

Commented [p89]: Seconding this comment. Advice to write monitor methods to encapsulate lock handling and data access, which is Part 1 advice. To go back to locks instead is bad advice. Good advice is to avoid
Your own locks.

Deleted: using the data

Commented [p90]: A Python concept? Different from locks?

Commented [MS91R90]: Ref:
Synchronization Primitives — Python 3.9.6 documentation
Also
https://docs.python.org/3/library/multiprocessing.html#sharing-state-between-processes

Deleted: <#>When using multiple threads, check for race conditions and deadlocks by using fuzzing techniques during development. ¶

Commented [SM93]: This needs coverage in the subsubclause 1 above.

Commented [SM94]: These likely belong in 6.63 Protocol lock errors.

Commented [SM95R94]: Explanations needed in 6.63.1.

6.64.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.64.3.
- 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 [UJO]

6.65.1 Applicability to language

This vulnerability as documented in ISO/IEC TR 24772-1:2019 clause 6.65 only minimally applies to Python because Python only has a small number of constants.
Python does not allow the declaration of constants. However, Python has six constants declared as part of the language. The list is:

- `False`
- `True`
- `None`
- `NotImplemented`
  Note that per the Python language documentation: "Changed in version 3.9: Evaluating `NotImplemented` in a boolean context is deprecated. While it currently evaluates as true, it will emit a `DeprecationWarning`. It will raise a `TypeError` in a future version of Python."
- `Ellipsis` (same as the ellipsis literal ". . .")
- `__debug__`

Early versions of Python would allow these constants to be given a new value. Since Python version 3.0, the first three, `False`, `True` and `None`, have been declared as keywords in addition to being a constant so their values may no longer be changed. The remaining three, `NotImplemented`, `Ellipsis` and `__debug__`, can be assigned new values without raising a SyntaxError making them modifiable constants.

6.65.2 Guidance to language users

- Follow the guidance contained in ISO/IEC TR 24772-1:2019 clause 6.65.3.
- Do not assign new values to `NotImplemented`, Ellipsis or `__debug__`.

7. Language specific vulnerabilities for Python

7.1 General

Deleted: P

Commented [p96]: Does not belong here as text

Commented [WLD97R96]: Attempted to fix – could also move this note to the end of the list.

Commented [98]: yyy Note from Nick Coghlan:
Speaking of clocks & timing, there are some use cases that should be updated to use time.monotonic() rather than time.time() or time.clock()
: https://www.python.org/dev/peps/pep-0418/#time-monotonic

Windows applications should also be aware of the fact that Python 3.6
always uses utf-8 for binary filesystem and console interfaces:
https://docs.python.org/dev/whatsnew/3.6.html#pep-529-change-windows-filesystem-encoding-to-utf-8

Non-Windows applications should be aware of the fact that Python 3.7+
will attempt to coerce the C locale to C.UTF-8 (or an equivalent
locale), and that implementing that behaviour is an approved option
for redistributor's Python 3.6 implementations (e.g. the system Python
in Fedora implements the option).
https://www.python.org/dev/peps/pep-0538/ has the details of that.

Commented [WLD99R98]: See Sean's reply in 6.60. Suggest deleting this comment or moving it to 6.60.

7.2 Lack of Explicit Declarations

7.2.1 Description of application vulnerability

As explained in clause 5.1.4, an assignment to a not yet existing variable is legal and creates the variable and its object on the spot. This capability also extends to the data members of a class, thereby extending that class. Moreover, reassigning an existing label to a different object binds the label to the new object regardless of the type of the previous object. Hence, any arbitrary assignment to a variable is legal.

7.2.2 Cross reference

Commented [SM100]: All: Look up potential cross references

7.2.3 Mechanism of failure

A mistyped label name as the target of an assignment simply introduces a new label. For example, upon execution of

```
CountTheNumberOfObjects = 0
   # and later on …
  CountTheNumberofObjects = CountTheNumberOfObjects + 1
                                                        # Two different
```

variables!!!
Most programmers will miss the differences in the names and be highly surprised by the fact that `CountTheNumberOfObjects` will retain its initialized value, usually 0.
Thus any unintentional mistyping of identifiers on the left hand side of an assignment is required by the language to go unnoticed. However, reading the value of a yet unknown variable will result in runtime error "`NameError`".

7.2.4 Avoiding the vulnerability or mitigating its effects

(look to static analysis tools???)

7.3 Code representation differs between compiler view and reader view

7.3.1 Description of application vulnerability

There is an issue that was raised , that the 10646 character set includes characters that set the direction (L >R or R->L) may be able to be embedded in code. A compiler won't be fooled by the change, but it can be used to mask a variable name or to hide a line of code to look like a comment. If Python supports such characters we should write up.

Example

`Blow_Up(); <CR> BeNice()` #The lack of a <LF> may display only the `BeNice();` call

7.2.2 Cross reference

Nicholas Boucher, Ross Anderson; Trojan Source: Invisible Vulnerabilities,

8. Implications for standardization or future revision

Bibliography

[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2004.

[2] ISO/IEC TR 10000-1, *Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework.*

[3] ISO 10241 (all parts), *International terminology standards.*

[4] Steve Christy, *Vulnerability Type Distributions in CVE*, V1.0, 2006/10/04.

[5] Carlo Ghezzi and Mehdi Jazayeri, *Programming Language Concepts*, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998.

[6] John David N. Dionisio. *Type Checking*. http://myweb.lmu.edu/dondi/share/pl/type-checking-v02.pdf

[7] The Common Weakness Enumeration (CWE) Initiative, MITRE Corporation, http://cwe.mitre.org

[8] Goldberg, David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, vol 23, issue 1 (March 1991), ISSN 0360-0300, pp 5-48.

[9] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. Institute of Electrical and Electronics Engineers, New York, 2008.

[10] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008.

[11] Bo Einarsson, ed. Accuracy and Reliability in Scientific Computing, SIAM, July 2005 http://www.nsc.liu.se/wg25/book

[12] "Enums for Python (Python recipe)," [Online]. Available: http://code.activestate.com/recipes/67107

[13] M. Pilgrim, Dive Into Python, 2004.

[14] M. Lutz, Learning Python, Sebastopol, CA: O'Reilly Media, Inc., 2009.

[15] "The Python Language Reference," [Online]. Available: http://docs.python.org/reference/index.html#reference-index.

[16] A. Martelli, Python in a Nutshell, Sebastopol, CA: O'Reilly Media, Inc., 2006.

[17] M. Lutz, Programming Python, Sebastopol, CA: O'Reilly Media, Inc., 2011.

[18] A. G. Isaac, "Python Introduction," 23 06 2010. [Online]. Available: https://subversion.american.edu/aisaac/notes/python4class.xhtml#introduction-to-the-interpreter.

[19] H. Norwak, "10 Python Pitfalls," [Online]. Available: http://zephyrfalcon.org/labs/python_pitfalls.html.

[20] "Python Gotchas," [Online]. Available: http://www.ferg.org/projects/python_gotchas.html.

[21] G. source, "Big List of Portabilty in Python," [Online]. Available: http://stackoverflow.com/questions/1883118/big-list-of-portability-in-python.

[22] "Python/C API Reference Manual", http://docs.python.org/py3k/c-api

[23] "Embedding Python in Another Application", http://docs.python.org/3/extending/embedding.html

[24] M. Pilgrim, Dive Into Python, 2004.

[25] M. Lutz, Learning Python, Sebastopol, CA: O'Reilly Media, Inc, 2009.
[26] "The Python Language Reference," [Online]. Available: http://docs.python.org/reference/index.html#reference-index.
[27] Martelli, Python in a Nutshell, Sebastopol, CA: O'Reilly Media, Inc., 2006.
[28] M. Lutz, Programming Python, Sebastopol, CA: O'Reilly Media, Inc., 2011.
[29] G. Isaac, "Python Introduction," 23 06 2010. [Online]. Available: https://subversion.american.edu/aisaac/notes/python4class.xhtml#introduction-to-the-interpreter.
[30] H. Norwak, "10 Python Pitfalls," [Online]. Available: http://zephyrfalcon.org/labs/python_pitfalls.html.
[31] "Python Gotchas," [Online]. Available: http://www.ferg.org/projects/python_gotchas.html.
[32] G. source, "Big List of Portability in Python," [Online]. Available: http://stackoverflow.com/questions/1883118/big-list-of-portability-in-python.
[33] "PEP 551 -- Security transparency in the Python runtime", [Online]. Available: https://www.python.org/dev/peps/pep-0551/
[34] "PEP 8 -- Style Guide for Python Code", [Online]. Available: http://www.python.org/dev/peps/pep-0008/

Index

Page 75: [1] Commented [MS30R29] McDonagh, Sean 3/24/21 9:45:00 PM

Text modified for processes. Regarding tasks, the exception-inducing command for terminating Tasks can be found at: [Coroutines and Tasks — Python 3.9.2 documentation]

To cancel a running Task use the `cancel()` method. Calling it will cause the Task to throw a `CancelledError` exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.

`cancelled()` can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the `CancelledError` exception and was actually cancelled.

Page 75: [2] Commented [p31] ploedere 2/7/22 3:07:00 AM

This is VERY misleading, given terminate on processes and cancel calls on tasks/futures. Needs fixing, since all these external killings cause the vulnerabilities of directed termination.

Page 75: [3] Commented [p32] McDonagh, Sean 2/7/22 3:07:00 AM

In Python, there is no native method available to terminate a thread. Terminating an external thread is possible via OS calls or by using ctypes, but this is very dangerous and not recommended which is why Python does not support tread termination natively. As a side note, it's my understanding that Java once had the native ability to kill a thread but then they deprecated it. The dangers associated with killing threads are not specific to Python but are worth mentioning in my opinion.

Page 75: [4] Deleted Stephen Michell 11/16/22 2:29:00 PM

Page 75: [5] Deleted Stephen Michell 11/16/22 2:41:00 PM

Page 75: [6] Commented [MS33] McDonagh, Sean 7/19/22 10:47:00 AM

The ordering of the three concurrency models should probably remain consistent in Sections 5.59 thru 6.63. Section 5.59 has them listed in the following order: Threading, Multiprocessing, asyncio, and Common, which may be a good trend to establish for the subsequent sections.

Page 76: [7] Commented [SM42] Stephen Michell 9/7/22 2:23:00 PM

From the docs:
https://docs.python.org/3/library/asyncio-task.html

cancel(*msg=None*)
"Request the Task to be cancelled.

This arranges for a CancelledError exception to be thrown into the wrapped coroutine on the next cycle of the event loop.
The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a try … … except CancelledError … finally block. Therefore, unlike Future.cancel(), Task.cancel() does not guarantee that the Task will be cancelled, although suppressing cancellation completely is not common and is actively discouraged."

Page 76: [8] Commented [SM43] Stephen Michell 9/7/22 2:25:00 PM

This could be accomplished with a global flag or by using one of the following wait conditions:
• FIRST_COMPLETED — Returns when the first task completes.
• ALL_COMPLETED — Returns when all tasks are complete. If an exception is raised in a task then it is noted, however, instead of stopping execution, all other tasks are allowed to complete.
• FIRST_EXCEPTION —Same as ALL_COMPLETED with the change that the function immediately returns if an exception is raised by a task, irrespective of whether other tasks have been completed or not.

Page 79: [9] Deleted Stephen Michell 7/20/22 3:04:00 PM

Page 79: [10] Deleted Stephen Michell 10/4/21 3:19:00 PM

Page 79: [11] Commented [p60] ploedere 6/21/21 10:09:00 PM

Here, too, any guidance ought to have an explanation of the vulnerability avoided in the text of 61.1.

Needs work.

Stephen and Sean to communicate.

Page 79: [12] Deleted Stephen Michell 11/16/22 4:46:00 PM

Page 79: [13] Commented [MS61] McDonagh, Sean 7/20/22 6:31:00 AM

This scenario can be handled using exception hooks. See example in text

Page 79: [14] Commented [MS62] McDonagh, Sean 7/20/22 6:32:00 AM

This scenario can be handled using a global flag to allow child threads to either continue, or stop, as desired. For example:

```
from time import sleep
import threading

def foo():
    print('In foo child thread ...')
    for i in range(10):
        if stop_threads:
            break
        print(i)
        sleep(.1)

def main():
    global stop_threads
    thread = threading.Thread(target=foo, name='My foo thread')
    thread.start()
    sleep(.5)
```

```
    try:
        raise Exception()
    except:
        print(f"Exception in thread: {threading.current_thread().name}")
        stop_threads = True
    thread.join()
    print('Finishing main thread ...')

if __name__ == "__main__":
    stop_threads = False
    main()
```

-----OUTPUT----

In foo child thread ...

0

1

2

3

4

Exception in thread: MainThread

Finishing main thread ...

Page 79: [15] Deleted Stephen Michell 10/19/22 4:41:00 PM

Page 83: [16] Deleted Stephen Michell 12/14/22 4:19:00 PM

Page 83: [17] Commented [SM79] Stephen Michell 12/14/22 4:21:00 PM

Talk about semaphores, or remove the avoidance mechanism that discusses semaphores.