ISO/IEC JTC 1/SC 22/WG23 N0894

Date: 2018-09-26

ISO/IEC TR 24772–4

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 4: Vulnerability descriptions 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*

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

**Copyright notice**

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

*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*

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

Contents

[Foreword vi](#_Toc7089361)

[Introduction vii](#_Toc7089362)

[1. Scope 1](#_Toc7089363)

[2. Normative references 1](#_Toc7089364)

[3. Terms and definitions, symbols and conventions 2](#_Toc7089365)

[3.1 Terms and definitions 2](#_Toc7089366)

[4. Language concepts 5](#_Toc7089367)

[5. General uidance for Python 6](#_Toc7089368)

[5.1 Top avoidance mechanisms 6](#_Toc7089369)

[6. Specific Guidance for Python 8](#_Toc7089370)

[6.1 General 8](#_Toc7089371)

[6.2 Type System [IHN] 8](#_Toc7089372)

[6.3 Bit Representations [STR] 10](#_Toc7089373)

[6.4 Floating-point Arithmetic [PLF] 11](#_Toc7089374)

[6.5 Enumerator Issues [CCB] 11](#_Toc7089375)

[6.6 Conversion Errors [FLC] 12](#_Toc7089376)

[6.7 String Termination [CJM] 13](#_Toc7089377)

[6.8 Buffer Boundary Violation [HCB] 13](#_Toc7089378)

[6.9 Unchecked Array Indexing [XYZ] 13](#_Toc7089379)

[6.10 Unchecked Array Copying [XYW] 13](#_Toc7089380)

[6.11 Pointer Type Conversions [HFC] 13](#_Toc7089381)

[6.12 Pointer Arithmetic [RVG] 13](#_Toc7089382)

[6.13 Null Pointer Dereference [XYH] 13](#_Toc7089383)

[6.14 Dangling Reference to Heap [XYK] 13](#_Toc7089384)

[6.15 Arithmetic Wrap-around Error [FIF] 14](#_Toc7089385)

[6.16 Using Shift Operations for Multiplication and Division [PIK] 14](#_Toc7089386)

[6.17 Choice of Clear Names [NAI] 14](#_Toc7089387)

[6.18 Dead Store [WXQ] 16](#_Toc7089388)

[6.19 Unused Variable [YZS] 17](#_Toc7089389)

[6.20 Identifier Name Reuse [YOW] 17](#_Toc7089390)

[6.21 Namespace Issues [BJL] 19](#_Toc7089391)

[6.22 Initialization of Variables [LAV] 22](#_Toc7089392)

[6.23 Operator Precedence and Associativity [JCW] 23](#_Toc7089393)

[6.24 Side-effects and Order of Evaluation of Operands [SAM] 23](#_Toc7089394)

[6.25 Likely Incorrect Expression [KOA] 25](#_Toc7089395)

[6.26 Dead and Deactivated Code [XYQ] 26](#_Toc7089396)

[6.27 Switch Statements and Static Analysis [CLL] 26](#_Toc7089397)

[6.28 Demarcation of Control Flow [EOJ] 27](#_Toc7089398)

[6.29 Loop Control Variables [TEX] 27](#_Toc7089399)

[6.30 Off-by-one Error [XZH] 28](#_Toc7089400)

[6.31 Structured Programming [EWD] 29](#_Toc7089401)

[6.32 Passing Parameters and Return Values [CSJ] 30](#_Toc7089402)

[6.33 Dangling References to Stack Frames [DCM] 31](#_Toc7089403)

[6.34 Subprogram Signature Mismatch [OTR] 31](#_Toc7089404)

[6.35 Recursion [GDL] 32](#_Toc7089405)

[6.36 Ignored Error Status and Unhandled Exceptions [OYB] 32](#_Toc7089406)

[6.37 Type-breaking Reinterpretation of Data [AMV] 33](#_Toc7089407)

[6.38 Deep vs. Shallow Copying [YAN] 33](#_Toc7089408)

[6.39 Memory Leaks and Heap Fragmentation [XYL] 34](#_Toc7089409)

[6.40 Templates and Generics [SYM] 34](#_Toc7089410)

[6.41 Inheritance [RIP] 35](#_Toc7089411)

[6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP] 35](#_Toc7089412)

[6.43 Redispatching [PPH] 35](#_Toc7089413)

[6.44 Polymorphic variables [BKK] 35](#_Toc7089414)

[6.45 Extra Intrinsics [LRM] 36](#_Toc7089415)

[6.46 Argument Passing to Library Functions [TRJ] 36](#_Toc7089416)

[6.47 Inter-language Calling [DJS] 37](#_Toc7089417)

[6.48 Dynamically-linked Code and Self-modifying Code [NYY] 37](#_Toc7089418)

[6.49 Library Signature [NSQ] 38](#_Toc7089419)

[6.50 Unanticipated Exceptions from Library Routines [HJW] 38](#_Toc7089420)

[6.51 Pre-processor Directives [NMP] 39](#_Toc7089421)

[6.52 Suppression of Language-defined Run-time Checking [MXB] 39](#_Toc7089422)

[6.53 Provision of Inherently Unsafe Operations [SKL] 39](#_Toc7089423)

[6.54 Obscure Language Features [BRS] 39](#_Toc7089424)

[6.55 Unspecified Behaviour [BQF] 42](#_Toc7089425)

[6.56 Undefined Behaviour [EWF] 42](#_Toc7089426)

[6.57 Implementation–defined Behaviour [FAB] 43](#_Toc7089427)

[6.58 Deprecated Language Features [MEM] 44](#_Toc7089428)

[6.59 Concurrency – Activation [CGA] 45](#_Toc7089429)

[6.60 Concurrency – Directed termination [CGT] 45](#_Toc7089430)

[6.61 Concurrent Data Access [CGX] 46](#_Toc7089431)

[6.62 Concurrency – Premature Termination [CGS] 46](#_Toc7089432)

[6.63 Lock Protocol Errors [CGM 47](#_Toc7089433)

[6.64 Reliance on External Format String [SHL] 47](#_Toc7089434)

[7. Language specific vulnerabilities for Python 47](#_Toc7089435)

[8. Implications for standardization or future revision 48](#_Toc7089436)

[Bibliography 48](#_Toc7089437)

[Index 50](#_Toc7089438)

This page intentially left blank

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

# Introduction

This Technical Report provides guidance for the programming language Python v3.7, so that application developers considering Python or using Python will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the Python language and their attendant consequences. This guidance 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 report can also be used in comparison with companion Technical Reports and with the language-independent report, TR 24772–1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with TR 24772–1, which discusses programming language vulnerabilities in a language independent fashion.

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

**Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages — Vulnerability descriptions for the 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 analysis and guidance provided in this document is targeted to Python version 3.7. 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 guidance provided herein.

# 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*

# 3. Terms and definitions, symbols and conventions

## 3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382–1, TR 24772–1, 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

*assignment statement*:Used to create (or rebind) a variable to an object. The simple syntax is a=b, the augmented syntax applies an operator at assignment time (for example, a += 1) and therefore cannot create a variable since it operates using the current value referenced by a variable. Other syntaxes support multiple targets (that is, x = y = z = 1).

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

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

*built‐in*:A function provided by the Python language intrinsically without the need to import it (for example, str, slice, type).

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

*comment*:Comments are preceded by a hash symbol “#”.

*complex number*:A number made up of two parts each expressed as floating‐point numbers: a real and an imaginary part. The imaginary part is expressed with a trailing upper or lower case “J” or “j”.

*compound statement*:A structure that contains and controls one or more statements.

*CPython*: The standard implementation of Python coded in ANSI portable C.

*dictionary*:A built‐in mapping consisting of zero or more key/value "pairs". Values are stored and retrieved using keys which can be of mixed types (with some caveats beyond the scope of this annex).

*docstring*:One or more lines in a unit of code that serve to document the code. Docstrings are retrievable at run‐time.

*exception*:An object that encapsulates the attributes of an exception (an error or abnormal event). 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.

*floating‐point number*:A real number expressed with a decimal point, an exponent expressed as anupper or lower case ”e” or “E” or both (for example, 1.0, 27e0, .456).

*function*: A grouping of statements, either built‐in or defined in a program using the def statement, which can be called as a unit.

*garbage collection*:The process by which the memory used by unreferenced object and their namespaces is reclaimed. Python provides a gc module to allow a program to direct when and how garbage collection is done.

*global*: A 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.

*guerrilla patching*:Also known as Monkey Patching, the practice of changing the attributes and/or methods of a module’s class at run‐time from outside of the module.

*immutability*:The characteristic of being unchangeable. Strings, tuples, and numbers are immutable objects in Python.

*import*:A mechanism that is used to make the contents of a module accessible to the importing program.

*inheritance*:The ability to define a class that is a subclass of other classes (called the superclass). Inheritance uses a method resolution order (MRO) to resolve references to the correct inheritance level (that is, it resolves attributes (methods and variables)).

*instance*:A single occurrence of a class that is created by calling the class as if it was a function (for example, a = Animal()).

*integer*: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.

*keyword*:An identifier that is reserved for special meaning to the Python interpreter (for example, if, else, for, class).

*lambda expression*:A convenient way to express a single return function statement within another statement instead of defining a separate function and referencing it.

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

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

*membership*:If an item occurs within a sequence it is said to be a member. Python has built‐ins to test for membership (for example, if a in b). Classes can provide methods to override built‐in membership tests.

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

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

*name*: A variable that references a Python object such as a number, string, list, dictionary, tuple, set, built-in, module, function, or class.

*namespace*:A place where names reside with their references to the objects that they represent. 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.

*none*:A null object.

*number*:An integer, floating point, decimal, or complex number.

*operator*:Non‐alphabetic characters, characters, and character strings that have special meanings within expressions (for example, +, -, not, is).

*overriding*: Coding an attribute in a subclass to replace a superclass attribute.

*package*:A collection of one or more other modules in the form of a directory.

*pickling:* The process of serializing objects using the pickle module.

*polymorphism:* The meaning of an operation – generally a function/method call – depends on the objects being operated upon, not the *type* of object. 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.

*recursion:* The ability of a function to call itself. Python supports recursion to a level of 1,000 unless that limit is modified using the setrecursionlimit function.

*scope:* The visibility of a name is its scope. 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.

*script*:A unit of code generally synonymous with a *program* but usually connotes code run at the highest level as in “*scripts run modules”*.

*self*:By convention, the name given to a class’ instance variable.

*sequence*:An ordered container of items that can be indexed or sliced using positive numbers. Python provides three built‐in sequences: strings, tuples, and lists. New sequences can also be defined in libraries, extension modules, or within classes.

*set*: An unordered sequence of zero or more items which do not need to be of the same type. Sets can be frozen (immutable) or unfrozen (mutable).

*short‐circuiting operators*:Operators and and or can short‐circuit the evaluation of their operand if the leftside evaluates to true (in the case of the or) or false (in the case of and). Forexample, 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.

*statement*:An expression that generally occupies one line. 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.

*string*:A built‐in sequence object consisting of one or more characters. Unlike many other languages, Python strings cannot be modified (that is, they are "immutable") and they do not have a termination character.

*tuple*:A sequence of zero or more items (for example, (1,2,3) or ("A", "B", "C")). Tuples are immutable and may contain different object types (for example, (1, "a", 5.678)).

*variable*:Python variables (that is, names) are not like variables in most other languages ‐ they are never declared they are dynamically referenced to objects, and they may be bound to objects of different types at different times. Variables are bound explicitly (for example, a = 1 binds a to the integer 1) and unbound implicitly (for example, a=1; a=2). In the last example, a is bound to the object (value) 1 then implicitly unbound to that object when bound to 2 ‐ a process known as rebinding. Variables can also be unbound explicitly using the del statement (for example, del a, b, c).

# 4. 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 intuitive to new and experienced programmers alike.

**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 - they 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

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 subclause 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.

**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 references 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 subclause *6.22 Initialization of Variables [LAV]* for a description of this.

The underling actions that are performed to enable the *apparent* in-place change do not update the immutable object – they create a new object and “point” the variable to new object. This can be proven 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 subclause *6.32 Passing Parameters and Return Values [CSJ]*.

4.3 Creation of variables

Python provides the ability to dynamically create variables when they are first assigned a value. In fact, assignment is the *only* way to bring a variable into existence. All values in a Python program are accessed through a reference which refers 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 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 actually deleted immediately)). But 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.

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 subclause 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, Python 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 does not check to see if a statement references an uninitialized variable until runtime. This is by design in order to support dynamic typing which in turn means there is no ability to declare a variable. 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 at compile time even if x is not defined (that is, assigned a value). An exception is raised at runtime only if the statement is executed and y>0. This scenario does 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.

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 when an unassigned (that is, non-existent) variable is referenced.

Initialization of class 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’s a very bad idea in almost all cases to assign default values to mutable objects.

# 5. General guidance for Python

## 5.1 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 TR 24772-1, 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** | **References** |
| 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. | 6.4.2 |
| 2 | Use of enumeration requires careful attention to readability, performance, and safety. There are many complex, but useful ways to simulate enums in Python [ (Enums for Python (Python recipe))]and many simple ways including the use of sets:  colors = {'red', 'green', 'blue'}  if red in colors: print('valid color')  Be aware that the technique shown above, as with almost all other ways to simulate enums, is not safe since the variable can be bound to another object at any time. If enum functions return error values, check the error return values before processing any other returned data. | 6.5.2 |
| 3 | Ensure that when examining code, that a variable can be bound (or rebound) to another object (of same or different type) at any time. | 6 |
| 4 | 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). | **6.20.2** |
| 5 | Use only spaces or tabs, not both, to indent to demark control flow. Avoid the form feed characters for indentation | 6.28.2 6.57.2 |
| 6 | Use Python’s built-in documentation (such as docstrings) to obtain information about a class’ method before inheriting from it | **6.41.2** |
| 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'). | 6.57.2 |
| 8 | When launching parallel tasks don’t raise a BaseException subclass in a callable in the Future class | 6.56.2 |
| 9 | Do not depend on the way Python may or may not optimize object references for small integer and string objects because it may vary for environments or even for releases in the same environment. | 6.55.2 |
| 10 | Be aware of short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression such as if the first expression evaluates to false in an and expression, then the remaining expressions, including functions calls, will not be evaluated. | 6.23.2 6.24.2 |

5.2 Guidance in interpreting guidance from TR 24772-1

Add words that explaine how you follow “applicable guidance”.

Python has some fundamental differences with standards imperative languages, which are the majority of languages covered by these documents. In some cases, general guidance does not apply to everything covered in a subsection, but some or most of the guidance.

In such cases we say “follow the applicable guidance of TR 24772-1 clause 6.x.5”, but that leaves it to the reader to determine what is applicable.

# 6. Specific Guidance for Python

## 6.1 General

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

## 6.2 Type System [IHN]

### 6.2.1 Applicability to language

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.

Python is also a strongly typed language – you cannot perform operations on an object that are not valid for that type. Python’s dynamic typing is a key feature designed to promote polymorphism to provide flexibility. Another aspect of dynamic typing is a variable does not maintain any type information – that information is held by the object that the variable references at a specific time. A Python program is free to assign (bind), and reassign (rebind), any variable to any type of object at any time.

Variables are created when they are first assigned a value (see subclause 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. Variables in an expression are replaced with the object they reference 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.

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 integer a is converted up to floating point (that is, 1.0) before the operation is performed. The object referred to by a is not affected – only the intermediate values used to resolve the expression are converted. If the programmer does not realize this conversion takes place he may expect that c is an integer and use it accordingly which could lead to unexpected results.

Automatic conversion also occurs when an integer becomes too large to fit within the constraints of the large integer specified in the language (typically C) used to create the Python interpreter. When an integer becomes too large to fit into that range it is converted to an unlimited precision integer of arbitrary length.

Explicit conversion methods can also be used to explicitly convert between types though this is seldom required 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'

Dynamic typing is a key feature of Python which promotes polymorphism for flexibility. Strict typing can, however, be imposed:

a = 'abc' # a refers to a string object

if isinstance(a, str): print('a type is string')

Using code to explicitly check the type of an object is strongly discouraged in Python since it defeats the benefit that dynamic typing provides - flexibility which allows functions to potentially operate correctly with objects of more than one type.

Gradual typing in Python allows optional annotations to be added to dynamic variables creating statically typed variables. This lets Python programs contain both dynamic variables, while adding the error-checking benefits of static variables. Python tools provide static type checkers such as PyCharm, mypy, and pytype that assist users in avoiding the misuse of declared types in Python.

### 6.2.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.3.5.
* Use static type checkers to detect typing errors

Note: The user community recommends *mypy* and *pytype.*

* Pay special attention to issues of magnitude and precision when using mixed type expressions;
* Be aware of the consequences of shared references;
* Be aware of the conversion from simple to complex; and
* Do not check for specific types of objects unless there is good justification, for example, when calling an extension that requires a specific type.

## 6.3 Bit Representations [STR]

### 6.3.1 Applicability to language

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

Potentially: Python experiences the vulnerability associated with shifting the underlying number as described in 62443-1 clause 6.3. 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 for shifting left or right so a program expecting an exception to halt it will instead unexpectedly continue leading to unexpected results.

### 6.3.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.3.5 (potentially) with regards to shift operations.
* Keep in mind that using a very large integer will have a negative effect on performance;

## 6.4 Floating-point Arithmetic [PLF]

### 6.4.1 Applicability to language

The vulnerabilities described in TR-24772-1 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

### 6.4.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.4.5
* Use floating-point arithmetic only when absolutely needed;
* Do not use floating-point arithmetic when integers or booleans suffice;
* Be aware that precision is lost for some real numbers (that is, floating-point is an approximation with limited precision for some numbers); and
* Be aware that results will frequently vary slightly by implementation.

## 6.5 Enumerator Issues [CCB]

### 6.5.1 Applicability to language

Python has an enumerate built-in type but it is not at all related to the implementation of enumeration as defined in other languages where constants are assigned to symbols. 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. One simple method is to simply assign a list of names to integers:

Red, Green, Blue = range (3)

print(Red, Green, Blue) # => 0 1 2

Code can then reference these “enum” values as they would in other languages which have native support for enumeration:

a = 1

if a == Green: print("a=Green")# => a=Green

There are disadvantages to the approach above though since any of the “enum” variables could be assigned new values at any time thereby undoing their intended role as “pseudo” constants. There are many forum discussions and articles that illustrate other, safer ways to simulate enumeration which are beyond the scope of this annex.

Use of enumeration requires careful attention to readability, performance, and safety. There are many complex, but useful ways to simulate enums in Python [ [1]]and many simple ways including the use of sets:

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

if "red" in colors: print('valid color')

A new 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)

The above example would print out: ColorEnum.BLUE

### 6.5.2 Guidance to language users

* Follow the guidance of TR 24772-1 clause 6.5.5
* Be aware that the technique shown above is not safe since the variable can be bound to another object at any time.
* Use the new enum module for better reliability and safety

## 6.6 Conversion Errors [FLC]

### 6.6.1 Applicability to language

The problem identified in TR 62443-1 clause 6.6 related to integer-based conversions does apply in Python since python seamlessly handles integers as described below:

1. Python converts numbers to a common type before performing any arithmetic operations. The common type is coerced using the following rules as defined in the standard (<http://docs.python.org/release/1.4/ref/ref5.html>):
   * 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, if either argument is a long integer, the other is converted to long integer;
   * 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. Integers are stored in an internal format that has faster performance when the number is smaller than the largest integer supported by the implementation language and platform.

Implicit or explicit conversion floating point to integer, implicitly (or explicitly using the int function), 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 integer to floating point. Losses in precision, whether from integer to floating point or vice versa, do not generate errors but can lead to unexpected results especially when floating point numbers are used for loop control.

The vulnerability described in TR 24772-1 related to conversion between semantically incompatible types is applicable to Python, which does not express this notion, e.g. distinguishing feet from meters. The application developer can implement such mechanisms by wrapping important types in classes and explicitly checking class types before performing conversions, as shown by a simple example below.

def feet\_to\_meters(source);

return source/3.3

def Class feet;

ft = 0.0

def class meters

m = 0.0

def feet\_to\_meters(source, dest);

dest.m = source.ft/3.3

else

throw conversion\_error

f = new feet(5.0)

m = new meters

feet\_to\_meters(f,m)

print m.val

feet\_to\_meters(6.0, m)

dest.val = source.val /3.3

AI – Sean – give actual sample code that explores the ideas above.

### 6.6.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 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; and
* 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.

## 6.7 String Termination [CJM]

This vulnerability is not applicable, 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.

## 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.

## 6.9 Unchecked Array Indexing [XYZ]

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.

## 6.10 Unchecked Array Copying [XYW]

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.

## 6.11 Pointer Type Conversions [HFC]

This vulnerability is not applicable to Python because Python does not have conversions on references (pointers).

## 6.12 Pointer Arithmetic [RVG]

This vulnerability is not applicable to Python because Python does not provide arithmetic on references (pointers).

## 6.13 Null Pointer Dereference [XYH]

The Python equivalent of a null pointer is the object “None”. Accessing this object raises an exception. Hence this vulnerability is not applicable to Python. Vulnerabilities associated with runtime exceptions are addressed in clause 6.36.

## 6.14 Dangling Reference to Heap [XYK]

This vulnerability is not applicable 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.

## 6.15 Arithmetic Wrap-around Error [FIF]

### 6.15.1 Applicability to language

The vulnerability discussed in TR 24772-1 clause 6.15.3 does not apply to Python.

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.

### 6.15.2 Guidance to language users

To mitigate the issues associated with floating point types:

* 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 you must use these types 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 you are using 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.

>>> print(-1<<100)#=> -1267650600228229401496703205376

>>> print(1<<100) #=> 1267650600228229401496703205376

>>> print(-4>>3) #=> -1 where you might expect 0

## 6.17 Choice of Clear Names [NAI]

### 6.17.1 Applicability to language

This vulnerability 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; and
* 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.

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; and
* 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; and
* 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 never 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 subclause *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 ,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 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.17.5
* For more guidance on Python’s naming conventions, refer to Python Style Guides contained in PEP 8 at <http://www.python.org/dev/peps/pep-0008/> .
* 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; and
* Use names that are clear and visually unambiguous because the compiler cannot assist in detecting names that appear similar but are different.

## 6.18 Dead Store [WXQ]

### 6.18.1 Applicability to language

The vulnerability as described in TR 24772-1 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.

variables can bed using the del statement

### 6.18.2 Guidance to language users

* Follow the applicable guidance of TR 24772-1 clause 6.18.5.
* Avoid rebinding except where it adds identifiable benefit;
* Ensure that when examining code that you consider that a variable can be bound (or rebound) to another object (of same or different type) at any time; and
* Consider using ResourceWarning to detect implicit reclamation of resources.

## 

## 6.19 Unused Variable [YZS]

The applicability to language and guidance to language users sections of clause [6.19 Unused Variable [YZS]](#_6.19_Unused_Variable) write-up are applicable to unused variables in Python.

## 

## 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

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

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; and
* 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); and
* 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 Guidance to language users

* 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; and
* 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 TR 24772-1 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 very well 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.

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 where a variable is being fetched from. 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 module14F[[1]](#footnote-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

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

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 subclause 6.19 Unused Variable [YZS] for more detail on this.

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; and
* 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 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 Guidance to language users

* Follow the guidance from TR 24772-1 clause 6.21.5.
* Use absolute imports , where the full path is specified, in preference to relative imports.
* 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 you want to import (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 does not exist in Python because all attempts to access an uninitialized variable result in an exception. See clause 4.3 “Creation of variables” for further explanation. Vulnerabilities associated with runtime exceptions are addressed in clause 6.36.

### 6.22.2 Guidance to language users

* 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 TR 24772-1 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

Expressions that use and or or are evaluated left-to-right which can cause a short circuit:

a or b or c

In the expression above c is never evaluated if either a or b evaluate to True because the entire expression evaluates to True immediately when any sub expression evaluates to True. See 6.24 for further discussions of chort-circuit evaluation.

### 6.23.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.23.5;

## 6.24 Side-effects and Order of Evaluation of Operands [SAM]

### 6.24.1 Applicability to language

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. I

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

### 6.24.2 Guidance to language users

* Follow the guidance contained in 24772-1 clause 6.24.5.

Be aware of Python’s short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression.

* In a simultaneous assignment, ensure that the left-hand sides are independent of each other.
* 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 Applicability to language

Python goes to some lengths to help prevent 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 ~, &, | 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.

### 6.25.2 Guidance to language users

* 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. 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 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 Guidance to language users

* 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 tp 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 TR 24772-1 clause 6.28 do not apply to Python. Python makes demarcation of control flow very clear because it uses indentation (using spaces or tabs – but not both) 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 most other languages the final line would execute only when the second if evaluated to True.

### 6.28.2 Guidance to language users

* Use only spaces or tabs, not both, to indent to demark control flow.

## 6.29 Loop Control Variables [TEX]

### 6.29.1 Applicability to language

The vulnerability as documented in TR 24772-1 clause 6.28 exists in Python. In some cases the vulnerability is mitigated by the Python for construct.

Python provides two loop control statements: while and for. They each support very flexible control constructs beyond a simple loop control variable. Assignments in the loop control statement (that is, while or for) which can be a frequent source of problems, are not allowed in Python – Python’s loop control statements use expressions which *cannot* contain assignment statements.

The while statement leaves the loop control entirely up to the programmer as in the example below:

a = 1

while a:

print('in loop')

a = False # force loop to end after one iteration

else:

print('exiting loop')

The for statement does not provide a loop control variable and hence it cannot be modified by the programmer. It is possible, however, to alter the loop behavior 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 Guidance to language users

* 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.

## 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 from zero versus from 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.

### 6.30.2 Guidance to language users

* **Follow the guidance of TR 24772-1 clause 6.30.5.**
* Be aware of Python’s indexing from zero and code accordingly.
* Use the for statement to execute over whole constructs in preference to loops that index individual elements.

## 6.31 Structured Programming [EWD]

### 6.31.1 Applicability to language

Python is designed to make it simpler to write structured program by requiring indentation and dedentation 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 they associate the else with the immediately prior if while Python uses indentation to link the else with its associated if statement (that is, the one *above* it).

Python also encourages structured programming by *not* introducing any language constructs which could lead to unstructured code (for example, GO TO statements).

Python does have two statements that could be viewed as unstructured. The first is the break statement. It’s used in a loop to exit the loop and continue with the first statement that follows the last statement within the loop block. This is a type of branch but it is such a useful construct that few would consider it “unstructured” or a bad coding practice.

The second is the try/except block which is used to trap and process exceptions. When an exception is thrown a branch is made to the except block:

def divider(a,b):

return a/b

try:

print(divider(1,0))

except ZeroDivisionError:

print('division by zero attempted')

Note that “with” statements and context managers 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.

Python offers few constructs that could lead to unstructured code. However, judicious use of break statements is encouraged to avoid confusion.

### 6.31.2 Guidance to language users

* Use “with” statements and context managers to enclose regions, and use them to invoke code which may create exceptions.
* Use the break statement judiciously to exit from control structures and show statically that it behaves correctly in all contexts.

## 6.32 Passing Parameters and Return Values [CSJ]

### 6.32.1 Applicability to language

Python’s only subprogram type is the function. Even though the import statement does execute the imported module’s top level code (the first time it is imported), the import statement cannot effectively be used as a way to repeatedly execute a series of statements.

Python passes arguments by assignment which is similar to passing by pointer or reference. Python assigns the passed arguments to the function’s local variables but unlike some other languages, simply 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. Python has no concept of aliasing where a function’s variables are mapped to the caller’s variables such that any changes made to the function’s variables are mapped over to the memory location of the caller’s arguments.

a = 1

def f(x):

x += 1

print(x)#=> 2

f(a)

print(a)#=> 1

In the example above, an immutable integer is passed as an argument and the function’s local variable is updated and then discarded when the function goes out of scope therefore the object the caller’s argument references is not affected. In the example below, the argument is mutable and is therefore updated in place:

a = [1]

def f(x):

x[0] = 2

f(a)

print(a)#=> [2]

Note that the list object a is not changed – it’s the same object but its content at index 0 has changed.

The return statement can be used to return a value for a function:

def doubler(x):

return x \* 2

x = 1

x = doubler(x)

print(x)#=> 2

The example above also demonstrates a way to emulate a call by reference by assigning the returned object to the passed argument. 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.

Note that Python functions return a value of none when no return statement is executed or when a return with no arguments is executed.

### 6.32.2 Guidance to language users

* Create copies of mutable objects before calling a function if changes are not wanted to mutable arguments; and
* If a function wants to ensure that it does not change mutable arguments it can make copies of those arguments and operate on them instead.

## 6.33 Dangling References to Stack Frames [DCM]

### 6.33.1 Applicability to language

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.

## 6.33.2 Guidance to language users

* 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.

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

Python supports positional, *“keyword=value”*, or both kinds of arguments. It also supports 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 has extensive extension and embedding APIs that includes functions and classes to use when extending or embedding Python. These provide for subprogram signature checking at runtime for modules coded in non-Python languages. Discussion of this API is beyond the scope of this annex but the reader should be aware that improper coding of any non-Python modules or their interface could cause a call stack problem.

### 6.34.2 Guidance to language users

Apply the guidance described in TR 24772-1 clause 6.34.5.

## 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

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 TR 24772-1 clause 6.35.5

## 6.36 Ignored Error Status and Unhandled Exceptions [OYB]

### 6.36.1 Applicability to language

Python provides statements to handle exceptions which considerably simplify the detection and handling of exceptions. Rather than being a vulnerability, Python’s exception handling statements provide a way to foil denial of service attacks:

def mainpgm(x, y):

return x/y

for x in range(3):

try:

y = mainpgm(1,x)

except:

print('Problem in mainpgm')

# clean up code…

else:

print (y)

The example code above prints:

Problem in mainpgm

1.0

0.5

The idea above is to ensure that the main program, which could be a web server, can continue to run after an exception by virtue of the try/except statement pair.

Note that the “except” statement can handle an individual exception (except someNamedError): or all exceptions (except:). In the first case, outer level exceptions would be needed for complete recovery protocols, while in the second case, more work is required to determine the nature of the exception.

Note also that unhandled Python exceptions will cause the program to terminate, as discussed in TR 24772-1 subclause 6.26.3.

### 6.36.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.36.5;
* Use Python’s exception handling with care in order to not catch errors that are intended for other exception handlers, i.e. always catch named exceptions; and
* 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 is not applicable to Python because assignments are made to objects and the object always holds the type – not the variable, therefore all referenced objects has 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 TR 24772-1 clause 6.38.

The following example illustrates the issue 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 colour1 is created, Python creates it as a list type, then has the list point to its elements. When colour2 is created as a copy of colour1, 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 a complete new list is created for colour2 (replacing the equivalence of colour1 and colour2), and any further changes to colour2 or colour1 do not affect the other.

Python has a method called deepcopy that copies all levels of a structured variable to another variable.

### 6.38.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.38.5.
* Use the “slice” operator [:] to force a copy up to one nested level

*Note:* x = y[:] *copies the complete next level, but leaves deeper levels, such as sublists shared.*

* To force deep copies at all levels of a variable, use the “deepcopy” method.

## 6.39 Memory Leaks and Heap Fragmentation [XYL]

### 6.39.1 Applicability to language

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 (or even bugs) 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 very 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(s) 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.

### 6.38.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.39.5.
* Release all objects when they are no longer required.

## 6.40 Templates and Generics [SYM]

This vulnerability is not applicable to Python because Python does not implement these mechanisms.

## 6.41 Inheritance [RIP]

### 6.41.1 Applicability to language

Python supports inheritance through a hierarchical search of namespaces starting at the subclass and proceeding upward through the superclasses. Multiple inheritance is also supported. Any inherited methods are subject to the same vulnerabilities that occur whenever using code that is not well understood.

### 6.41.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.41.5.
* Inherit only from trusted classes; and
* Use Python’s built-in documentation (such as docstrings) to obtain information about a class’ method before inheriting from it.

## 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 TR 24772-1 clause 6.42. The Python community provides static analysis tools for Python, which detect a large class of such violations.

### 6.42.2 Guidance to language users

Follow the guidance contained in TR 24772-1 clause 6.42.5. In particular, use static analysis tools, either commercial or provided by the Python community to detect such violations.

## 6.43 Redispatching [PPH]

### 6.43.1 Applicability to language

This vulnerability applies to Python and can result in infinite recursion between redefined and inherited methods.

### 6.43.2 Guidance to language users

Follow the guidance contained in TR 24772-1 clause 6.43.5.

## 6.44 Polymorphic variables [BKK]

### 6.44.1 Applicability to language

TBD

*Python is inherently polymorphic, in the narrow sense of OO polymorphism, and in the general sense that any operation will attempt to apply itself to any object, and raise an exception if it cannot apply the operation to a given object.*

### 6.44.2 Guidance to language users

TBD

## 6.45 Extra Intrinsics [LRM]

### 6.45.1 Applicability to language

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:

x = 'abc'

print(len(x))#=> 3

def len(x):

return 10

print(len(x))#=> 10

If 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’s 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 subclause 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

* 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

Refer to subclause 6.34 Subprogram Signature Mismatch [OTR].

### 6.46.2 Guidance to language users

Refer to 6.34 Subprogram Signature Mismatch [OTR].

## 6.47 Inter-language Calling [DJS]

### 6.47.1 Applicability to language

Python has a documented API for extending Python using libraries coded in C or C++. The library(s) are then imported into a Python module and used in the same manner as a module written in Python. Python’s standard for interfacing to the “C” language is documented in <http://docs.python.org/py3k/c-api/>.

Conversely, code written in C or C++ can embed Python. The standard for embedding Python is documented in: <http://docs.python.org/py3k/extending/embedding.html>.

The Jython system is a Java-based implementation that interfaces with Java, and IronPython provides interfaces to Microsoft .NET languages.

### 6.47.2 Guidance to language users

* Use the language interface APIs documented on the Python web site for interfacing to C/C++, the Jython web site for Java, the IronPython web site for .NET languages, and for all other languages consider creating intermediary C or C++ modules to call functions in the other languages since many languages have documented API’s to C and C++.

## 6.48 Dynamically-linked Code and Self-modifying Code [NYY]

### 6.48.1 Applicability to language

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; and
* Overlaying of a file with an alternate.

Python also provides an eval and an exec statement each of which can be used to create self-modifying code:

x = "print('Hello " + "World')"

eval(x)#=> Hello World

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 they expect which could cause unexpected results.

### 6.48.2 Guidance to language users

* Avoid using exec or eval and *never* use these with untrusted code;
* Be careful when using Guerrilla patching to ensure that all users of the patched classes and/or modules continue to function as expected; conversely, be aware of any code that patches classes and/or modules that your code is using to avoid unexpected results; and
* Ensure that the file path and files being imported are from trusted sources.

## 6.49 Library Signature [NSQ]

### 6.49.1 Applicability to language

Python has 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 annex.

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].

### 6.49.2 Guidance to language users

* Use only trusted modules as extensions; and
* 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

Python is often extended by importing modules coded in Python and other languages. For modules coded in Python the risks include:

* Interception of an exception that was intended for a module’s imported exception handling code (and vice versa); and
* Unintended results due to namespace collisions (covered in 6.21 Namespace Issues [BJL] and elsewhere in this annex).

For modules coded in other languages the risks include:

* Unexpected termination of the program; and
* Unexpected side effects on the operating environment.

### 6.50.2 Guidance to language users

* Wrap calls to library routines and use exception handling logic to intercept and handle exceptions when practical.
* Document when wrapping library routine calls is unfeasible and ???

## 6.51 Pre-processor Directives [NMP]

### 6.51.1 Applicability to language

Python v3.8 will provide a new 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.

### 6.51.2 Guidance to language users

* During development, avoid using the 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. The entry point should not use any unprotected settings from the working environment.
* Consider logging all predetermined events and backing them up to a non-local file so that an attacker cannot delete them. All events should be recorded prior to abort operations so that full traceability is preserved.
* Consider using DeviceGuard and the open\_for\_import hook to validate the signatures of all files in the Python application.
* For more guidance on using pre-processor directives and hooks, refer to the General Recommendations contained in PEP 551 at [https://www.python.org/dev/peps/pep-0551/](https://urldefense.proofpoint.com/v2/url?u=https-3A__www.python.org_dev_peps_pep-2D0551_&d=DwMFaQ&c=31nHN1tvZeuWBT6LwDN4Ngk1qezfsYHyolgGeY2ZhlU&r=_hSCXI5-mXrGcbRiWbBwgeug3UbaT2XrXWFb_Ccpjkg&m=y37OtV4PdnybrQB11vd0_HWC9IKBhiN444-WUMw4XPw&s=A8KC-czaoMfA-9vwcSpZ9Jrw06wN3WuZWKsX1ZAE3Xs&e=)

## 6.52 Suppression of Language-defined Run-time Checking [MXB]

This vulnerability 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 which 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

Python has very few operations that are inherently unsafe. For example, there is no way to suppress error checking or bounds checking. However, there are two operations provided in Python that are inherently unsafe in any language:

* Interfaces to modules coded in other languages since they could easily violate the security of the calling of embedded Python code; and
* Use of the exec and eval dynamic execution functions (see 6.48 Dynamically-linked Code and Self-modifying Code [NYY]).

### 6.53.2 Guidance to language users

* Use only trusted modules; and
* Avoid the use of the exec and eval functions.

## 6.54 Obscure Language Features [BRS]

### 6.54.1 Applicability of language

Python has some obscure language features as described below:

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 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.

### 6.54.2 Guidance to language users

* 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 you absolutely need to and you understand the effect;
* Be aware that when using the += operator on mutable objects the operation is done in place;
* Be cognizant that assignments to objects, mutable and immutable, always create a new object;
* Understand the difference between equivalence and equality and code accordingly; and
* 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

Understanding how Python manages identities becomes less clear when a script is run using integers (or short strings):

a=1

b=a

c=1

a is b, b is c, a == c #=> (True, **True**, True)

In the example above c references the same object as a and b even though c was never assigned to either a or b. This is a nuance of how Python is optimized to cache short strings and small integers. Other than in a test for identity as above, this nuance has no effect on the logic of the program (for example, changing the value of c to 2 will not affect a or b). Refer also to 4. Language concepts.

When persisting objects using pickling, if an exception is raised then an unspecified number of bytes may have already been written to the file.

### 6.55.2 Guidance to language users

* Do not rely on the content of error messages – use exception objects instead;
* When persisting object using pickling use exception handling to cleanup partially written files; and
* Do not depend on the way Python may or may not optimize object references for small integer and string objects because it may vary for environments or even for releases in the same environment.

## 6.56 Undefined Behaviour [EWF]

### 6.56.1 Applicability to language

Python has undefined behaviour in the following instances:

* Caching of immutable objects can result in (or not result in) a single object being referenced by two or more variables. Comparing the variables for equivalence (that is, if a == b) will always yield a True but checking for equality (using the is built-in) may, or may not, dependent on the implementation:

a = 1

b = 2-1

print(a == b, a is b) #=> (True, ?)

* The sequence of keys in a dictionary is undefined because the hashing function used to index the keys is unspecified therefore different implementations are likely to yield different sequences.
* The [Future](http://docs.python.org/release/3.2/library/concurrent.futures.html?highlight=undefined%20behavior#concurrent.futures.Future) class encapsulates the asynchronous execution of a callable. The behaviour is undefined if the add\_done\_callback(fn) method (which attaches the callable fn to the future) raises a [BaseException](http://docs.python.org/release/3.2/library/exceptions.html#BaseException) subclass.
* Modifying the dictionary returned by the vars built-in has undefined effects when used to retrieve the dictionary (that is, the namespace) for an object.
* Form feed characters used for indentation have an undefined effect on the character count used to determine the scope of a block.
* 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.
* The order of sort of a list of sets, using list.sort(), is undefined as is the use of the function used on a list of sets that depend on total ordering such as min(), max(), and sorted().
* Undefined behaviour will occur if a thread exits before the main procedure from which it was called itself exits.

### 6.56.2 Guidance to language users

* Understand the difference between testing for equivalence (for example, ==) and equality (for example, is) and never depend on object identity tests to pass or fail when the variables reference immutable objects;
* Do not depend on the sequence of keys in a dictionary to be consistent across implementations.
* When launching parallel tasks don’t raise a [BaseException](http://docs.python.org/release/3.2/library/exceptions.html#BaseException) subclass in a callable in the Future class;
* Never modify the dictionary object returned by a vars call;
* Never use form feed characters for indentation;
* Consider using the id function to test for object equality;
* Do not try to use the catch\_warnings function to suppress warning messages when using more than one thread; and
* Never 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

Python has implementation-defined behaviour in the following instances:

* Mixing tabs and spaces to indent is defined differently for UNIX and non-UNIX platforms;
* 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; and
* 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, so it may be useful to know the integer size of the implementation.

### 6.57.2 Guidance to language users

* Always use either spaces or tabs (but not both) for indentations;
* Consider using the -tt command line option to raise an IndentationError;
* Consider using a text editor to find and make consistent, the use of tabs and spaces for indentation;
* 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').
* 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.
* 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).

## 6.58 Deprecated Language Features [MEM]

### 6.58.1 Applicability to language

The following features were deprecated in the latest (as of this writing) version of E 3.1. These are documented at <http://docs.python.org/release/3.1.3/whatsnew/3.1.html>:

* The [string.maketrans()](http://docs.python.org/release/3.1.3/library/string.html#string.maketrans) function is deprecated and is replaced by new static methods, [bytes.maketrans()](http://docs.python.org/release/3.1.3/library/stdtypes.html#bytes.maketrans) and [bytearray.maketrans()](http://docs.python.org/release/3.1.3/library/stdtypes.html#bytearray.maketrans). This change solves the confusion around which types were supported by the [string](http://docs.python.org/release/3.1.3/library/string.html#module-string) module. Now, [str](http://docs.python.org/release/3.1.3/library/functions.html#str), [bytes](http://docs.python.org/release/3.1.3/library/functions.html#bytes), and [bytearray](http://docs.python.org/release/3.1.3/library/functions.html#bytearray) each have their own maketrans and translate methods with intermediate translation tables of the appropriate type.
* The syntax of the [with](http://docs.python.org/release/3.1.3/reference/compound_stmts.html#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()](http://docs.python.org/release/3.1.3/library/contextlib.html#contextlib.nested) function is no longer needed and is now deprecated.
* Deprecated [PyNumber\_Int()](http://docs.python.org/release/3.1.3/c-api/number.html#PyNumber_Int). Use [PyNumber\_Long()](http://docs.python.org/release/3.1.3/c-api/number.html#PyNumber_Long) instead.
* Added a new [PyOS\_string\_to\_double()](http://docs.python.org/release/3.1.3/c-api/conversion.html#PyOS_string_to_double) function to replace the deprecated functions [PyOS\_ascii\_strtod()](http://docs.python.org/release/3.1.3/c-api/conversion.html#PyOS_ascii_strtod) and [PyOS\_ascii\_atof()](http://docs.python.org/release/3.1.3/c-api/conversion.html#PyOS_ascii_atof).
* Added [PyCapsule](http://docs.python.org/release/3.1.3/c-api/capsule.html#PyCapsule) as a replacement for the [PyCObject](http://docs.python.org/release/3.1.3/c-api/cobject.html#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.

### 6.58.2 Guidance to language users

* When practicable, migrate Python programs to the current standard.

## 6.59 Concurrency – Activation [CGA]

### 6.59.1 Applicability to language

Python offers several approaches for handling concurrency, and each method has its own advantages and disadvantages. 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), only one thread at a time is permitted to run. Even though multithreading cannot 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. Python’s multiprocessing module provides multiprocessing capability and does allow independent processes to run on multiple cores. Unlike threading, these independent processes do not have shared memory and are not prone to the same data race conditions that threads can have. Python’s asyncio module is the newest approach to handling asynchronous concurrency and was introduced in Python 3.4. This new Async IO processing model is typically safer and faster than implementations that use traditional threads and multiprocessing.

### 6.59.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.59.5.
* For any thread that has already been started, ensure that additional starts on that same thread are not attempted. Multiple attempts to start any single thread object will raise a runtime error.
* If a thread is unable to be created and an exception is thrown, always handle the exception.
* For any process that has already been started, ensure that additional starts on that same process are not attempted. Multiple attempts to start any process object will raise a runtime error.
* Starting Async IO tasks using the asyncio module can only occur on a thread that is not running. During development, it is recommended to run the Async IO code in debug mode. This will help detect never-awaited coroutines, non-threadsafe Async IO APIs, excessive execution times for I/O and callback functions, and never-retrieved exceptions. To reduce the chance of excessive delays, all concurrent Async IO operations need to be performed on non-blocking code.

## 6.60 Concurrency – Directed termination [CGT]

### 6.60.1 Applicability to language

In Python, a thread may terminate by coming to the end of its executable code or by raising an exception. Python does not have an API to kill a thread. This is by design since killing a thread is not recommended due to the unpredictable behavior that results. If a thread is killed in between an acquire() and release(), every other thread that waits on that lock will be deadlocked. Terminating processes in Python is possible but there are scenarios that may leave the system in a vulnerable state.

### 6.60.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.60.5.
* Avoid killing threads since it is only safe if extreme measures are taken.
* If necessary, the preferred method for killing a thread is natively 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.
* Use care when terminating processes since finally clauses will not be executed, and descendant processes will not be terminated. Design the code to be fail-safe since terminating a process may corrupt data associated with pipes and queues.

## 6.61 Concurrency - Data Access [CGX]

### 6.61.1 Applicability to language

The preemptive task-switching nature of threads can create opportunities for certain vulnerabilities such as race conditions and deadlocks. 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. Identifying all locations where locks are needed can be complicated and 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. Since threads use shared memory, the overhead costs are typically less than they are for multiprocessing scenarios and often run faster.

Processes, unlike threads, do not need locks and are easier to terminate safely. However, because processes do not have a shared state, communicating between processes comes at a higher overhead cost.

Unlike threads, Async IO switches cooperatively from an Async IO manager and, since task switching is less arbitrary, there is less of a need for locks. Asynchronous code uses await and yield to provide predictable control over the task switching process. Async IO is safer and faster than other task switching techniques, but it does require all calls to be non-blocking.

### 6.61.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.61.5.
* Use join() 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 before it is started since this will throw an exception. Verify that the opportunity does not exist for any thread to perform multiple joins since this would result in a deadlock condition. Be sure that no thread is waiting on daemon threads to complete since these threads are always running. Performing a join() on a daemon thread will result in a deadlock condition and it is recommended to use a join() on the message queue instead.
* If two or more items need to occur sequentially, ensure that they are ordered correctly and reside in the same thread.
* When using multiple processes, avoid using global variables and consider using the multiprocessing.Queue() function to share data between processes.
* When using multiple threads, avoid using global variables and consider using the queue.Queue() function to share data between threads.
* When using multiple threads, verify that no unprotected data is used directly by more than one thread.
* When using multiple threads, consider using the ThreadPoolExecutor within the concurrent.futures module to help maintain and control the number of threads being implemented.
* When using multiple threads, check for race conditions and deadlocks by using fuzzing techniques during development.
* If shared variables must be used in multithreaded applications, use model checking or equivalent methodologies to prove the absence of race conditions.
* For all new applications that require concurrency, consider using Async IO instead of threads or processes whenever possible. The reliability, speed, and maintainability of Async IO code is superior even though there is a steep learning curve.
* When converting existing code to Async IO, yield and await statements must be added to the code.
* When using Async IO, all tasks must be non-blocking and use Async IO calls from an event loop. Locks and other synchronization techniques are usually not needed when implementing Async IO.

## 6.62 Concurrency – Premature Termination [CGS]

### 6.62.1 Applicability to language

A 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.

### 6.62.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.62.5.
* Use the finally keyword for each thread method that notifies a higher-level construct of the termination so that corrective action can be taken.
* Use 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.
* 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.
* Handle exceptions and clean up nested threads and potentially shared data before termination.

## 6.63 Concurrency - Lock Protocol Errors [CGM]

### 6.63.1 Applicability to language

Python provides locks and semaphores that are intended to protect critical sections of data. Python also provides event objects that permit programmed-specific notification between two threads, as well as barriers and condition objects that permit the release of groups of threads upon a single condition becoming true.

### 6.63.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.63.5.
* If global variables are used in multi-threaded code, be sure to use locks around them. Identify all locations where locks should be used but realize that 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 using the data.
* 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.

## 6.64 Reliance on External Format String [SHL]

### 6.64.1 Applicability to language

Externally controllable strings can result in unexpected behavior such as buffer overruns, exposure of private data, and other malicious exploits. Python strings share most of the potential security vulnerabilities described in TR 24772-1 clause 6.64.

### 6.64.2 Guidance to language users

* Follow the guidance contained in TR 24772-1 clause 6.64.3.
* 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.

# 7. Language specific vulnerabilities for Python

# 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>

|  |  |
| --- | --- |
| [1] | "Enums for Python (Python recipe)," [Online]. Available: http://code.activestate.com/recipes/67107/. |
| [2] | M. Pilgrim, Dive Into Python, 2004. |
| [3] | M. Lutz, Learning Python, Sebastopol, CA: O'Reilly Media, Inc, 2009. |
| [4] | "The Python Language Reference," [Online]. Available: http://docs.python.org/reference/index.html#reference-index. |
| [5] | A. Martelli, Python in a Nutshell, Sebastopol, CA: O'Reilly Media, Inc., 2006. |
| [6] | M. Lutz, Programming Python, Sebastopol, CA: O'Reilly Media, Inc., 2011. |
| [7] | A. G. Isaac, "Python Introduction," 23 06 2010. [Online]. Available: https://subversion.american.edu/aisaac/notes/python4class.xhtml#introduction-to-the-interpreter. [Accessed 12 05 2011]. |
| [8] | H. Norwak, "10 Python Pitfalls," [Online]. Available: http://zephyrfalcon.org/labs/python\_pitfalls.html. [Accessed 13 05 2011]. |
| [9] | "Python Gotchas," [Online]. Available: http://www.ferg.org/projects/python\_gotchas.html. |
| [10] | G. source, "Big List of Portabilty in Python," [Online]. Available: http://stackoverflow.com/questions/1883118/big-list-of-portability-in-python. [Accessed 12 6 2011]. |



# Index

CGM – Protocol Lock Errors, 47

CGS – Concurrency – Premature Termination, 46

Language Vulnerabilities

Concurrency – Premature Termination [CGS], 46

Protocol Lock Errors [CGM], 47

Uncontrolled Fromat String [SHL], 47

LHS (left-hand side), 23

SHL – Uncontrolled Format String, 47

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. [↑](#footnote-ref-1)