ISO/IEC JTC 1/SC 22/WG 23 N 0347

Proposed Annex for Python Language

Date	2011-06-20
Contributed by	Kevin Coyne
Original file name	Python_Annex_v31.docx

Annex Python

Python. Vulnerability descriptions for the language Python Standards and terminology

Python.1 Identification of standards and associated documents

Bibliography

Enums for Python (Python recipe). (n.d.). Retrieved from ActiveState: http://code.activestate.com/recipes/67107/

Isaac, A. G. (2010, 06 23). *Python Introduction*. Retrieved 05 12, 2011, from https://subversion.american.edu/aisaac/notes/python4class.xhtml#introduction-to-the-interpreter

Lutz, M. (2009). *Learning Python.* Sebastopol, CA: O'Reilly Media, Inc.

Lutz, M. (2011). *Programming Python.* Sebastopol, CA: O'Reilly Media, Inc.

Martelli, A. (2006). *Python in a Nutshell.* Sebastopol, CA: O'Reilly Media, Inc.

Norwak, H. (n.d.). *10 Python Pitfalls*. Retrieved 05 13, 2011, from 10 Python Pitfalls: http://zephyrfalcon.org/labs/python_pitfalls.html

Pilgrim, M. (2004). *Dive Into Python.*

Python Gotchas. (n.d.). Retrieved from http://www.ferg.org/projects/python_gotchas.html

source, G. (n.d.). *Big List of Portabilty in Python*. Retrieved 6 12, 2011, from stackoverflow: http://stackoverflow.com/questions/1883118/big-list-of-portability-in-python

The Python Language Reference. (n.d.). Retrieved from python.org: http://docs.python.org/reference/index.html#reference-index

Python.2 General Terminology and Concepts

Python.2.1 General Terminology

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 (e.g., `a += 1`) and therefore cannot create a variable since it operates using the current value referenced by a variable. Other syntaxes support multiple targets (e.g., `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` equivalences to any non-zero value and `False` equivalences to zero. Commonly expressed numerically as 1 (true), or 0 (false) but referenced as `True` and `False`.
Bound method	A method that is called using a class instance.
Built-in	A function provided by the Python language intrinsically without the need to import it (e.g., `str, slice, type`).
Callables	Any Python object type that supports a function call. Callables include functions, built-ins, types, class objects and their methods, and generators.
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.
Container	An object that embeds, or contains, other objects.
CPython	The standard implementation of Python coded in ANSI portable C.
DBM	A file that supports keyed access.
Decorator	A way to specify that the function definition that immediately follows it belongs to the class specified.
Delegation	Overriding a superclass' method by calling the overridden superclass method to provide some of the logic. Most often done with the `__init__` method.
Delimiter	Combinations of one or more non-alphabetic characters used to bound strings, statements, and other constructs such as lists, tuples, and dictionaries.
Descriptor	An object that provides a method to override the default get method used to access and set attributes of an instance of the object's class. Descriptors are implemented as a separate class which controls the semantics of accessing and changing attributes of an instance.
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 document).
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 an upper or lower case "e or E" or both (e.g., `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.
Generator	A special form of a function that, instead of returning a value, instead returns an iterator object that can then be used by calling its next method which in turn advances the object's execution until the next `yield` statement is executed (typically at the end of the list).
Header	The start of a compound statement. A header is always terminated by a colon (:) and is followed by the body which is composed of one or more statements (simple or compound).
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 (i.e., superclass). Inheritance uses a method resolution order (MRO) to resolve references to the correct inheritance level (i.e., it resolves attributes (methods and variables)).
Instance	A single occurrence of a class that is created by calling the class as is it was a function (e.g., `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 (e.g., `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 (i.e., is mutable) and indexed.
List comprehension	An expression that provides a concise way to iterate through an interable object. In its simplest form it looks like: `for x in y` where `x` is an interaction counter and `x` is the object being iterated. There are more complex forms of list comprehensions beyond the scope of this document.
Literals	A string or number (e.g., `'abc', 123, 5.4`). Note that a string literal can use either double quote (“) or single apostrophe pairs (‘) to delimit a string.
Mapping	An unordered collection of objects that are indexed using a unique key. Only one mapping type is provided in Python as a built-in - the dictionary, but other types can be added using classes, libraries and extension modules.
Marshaling	See serialization.
Membership	If an item occurs within a sequence it is said to be a member. Python has built-ins to test for membership (e.g., `if a in b`). Classes can provide methods to override built-in membership tests.
Module	A file of source code written in Python or in another language in which case it's known as an extension.

Mutability	The characteristic of being changeable. Lists and dictionaries are two examples of Python objects that are mutable.
Namespace	A place where names reside. Examples of objects that have their own namespaces include: modules, classes, and functions.
None	A null object.
Number	An integer, floating point, decimal, or complex number.
Object attribute	A method or value within an object accessed by the syntax: `objectname.attributename`.
Object item	An indexable or keyed object within a containing object. The syntax to reference an object is `a[b]` where `a` is the containing object and `b` is the key or index.
Operator	Non-alphabetic characters, characters, and character strings that have special meanings within expressions (e.g., `+, -, not, is`).
Overloading	Coding a method to replace Python's built-in operators.
Overriding	Coding an attribute in a subclass to replace a superclass attribute.
Package	A module that contains one or more other modules.
Persistence	The characteristic of being persistent from one session to another. Persistence is typically achieved using serialization techniques such as pickling, shelves, and DBM.
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.
Property	A mechanism that lets a class control the way in which specific attributes of a class are fetched and/or assigned.
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.
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, `self` is, 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.
Serialization	The process of converting an object to a byte stream representation.
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).
Shelve	The process of moving a pickle to an external DBMS.
Short-Circuiting Operators	Operators `and` and `or` can short-circuit the evaluation of their operand if the left side evaluates to true (in the case of the `or`) or false (in the case of `and`). For example, in the expression `a or b`, there is no need to evaluate `b` if `a` is `True`, likewise in the expression `a and b`, there is no need to evaluate `b` if `a` is `False`.
Simple Statement	A statement that does not contain other statements (e.g., `a = 1`).
Slicing	Indicating a subsequence of a sequence using the syntax `a[x:y]` where `a` is the object, `x` is the offset (based at 0) and `y` is the last character *not* to be included in

the slice.

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 (i.e., they are "immutable") and they do not have a termination character.

Tuple A sequence of zero or more items (e.g., `(1,2,3)` or `("A", "B", "C")`). Tuples are immutable and may contain different object types (e.g., `(1, "a", 5.678)`).

Unbound method A method that is called using the class i.e., not through an instance of a class. This is typically only used when needing to call a class method that is overridden at the subclass level.

Variable Python variables are not like variables in most other languages - they are never declared, and they are dynamically referenced to objects. They have no type and may be bound to objects of different types at different times. They are bound and explicitly (e.g., `a = 1` binds `a` to the integer `1`) and unbound implicitly (e.g., `a=1; a=2)`. In the last example, `a` is bound to the object (value) `1` then unbound to that object when bound to `2` - a process known as rebinding. Variables can also be unbound explicitly using the `del` statement (e.g., `del a, b, c`).

VHLL Very high level language. Python is a VHLL.

Python.2.2 Key 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. 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` which 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 Python.3 Type System [IHN] for more on this subject.

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, i.e., 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 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.

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
```

Python.3 Type System [IHN]

Python.3.1 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 Python.19 Dead Store [WXQ] 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 i.e., `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 (i.e., 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 (i.e., `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. On a 32-bit machine this would be the range -2^{30} to $2^{30}-1$. When an integer becomes too large to fit into that range it is converted to an extended 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.

Python.3.2 Guidance to language users

- 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 (e.g., when calling an extension that requires a specific type).

Python.4 Bit Representations [STR]

Python.4.1 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
```

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.

Python.4.2 Guidance to language users

- Keep in mind that using a very large integer will have a negative effect on performance;
- Do not assume that you will get an over or underflow halt when shifting using bitwise operators; use higher level functions such as multiplication and division when that is the intent; and
- Do not assume the bit orientation of the hardware stores bits left to right or right or right to left. If the progam logic depends on the direction bits are stored then be aware that results will differ based on the platform used (See Python.52 Implementation–defined Behaviour [FAB] for more on this subject).

Python.5 Floating-point Arithmetic [PLF]

Python.5.1 Applicability to language

Python supports floating-point arithmetic. Literals are expressed with a decimal point and or an optional `e` or `E`:

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

There is no way to determine the precision of the implementation from within a Python program but, in the CPython implementation, it's implemented as a C double which is approximately 53 bits of precision.

Python.5.2 Guidance to language users

- Use floating-point arithmetic only when absolutely needed;
- Do not use floating-point arithmetic when integers or booleans would suffice;
- Be aware that precision is lost for some real numbers (i.e., floating-point is an approximation with limited precision for some numbers);
- Be aware that results will frequently vary slightly by implementation (See Python.52 Implementation–defined Behaviour [FAB] for more on this subject); and
- Testing floating-point numbers for equality (especially for loops) can lead to unexpected results. Instead, if floating-point numbers are needed for loop control use >= or <= comparisons.

Python.6 Enumerator Issues [CCB]

Python.6.1 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 and that there is no enumeration construct in Python, 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 document.

Python.6.2 Guidance to language users

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')
```

If you require integers associated with each value then the following works:

```
blue, green, red = range(1,4)
print(red) #=> 3
```

Python.7 Numeric Conversion Errors [FLC]

Python.7.1 Applicability to language

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 more 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))# => 3.0 (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.

Python.7.2 Guidance to language users

- 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.

Python.8 String Termination [CJM]

In Python strings are immutable objects whose length is managed with built-in functions therefore Python does not permit accesses past the end, or beginning, of a string.

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

Python.9 Buffer Boundary Violation [HCB]

This vulnerability is not applicable to Python.

Python.10 Unchecked Array Indexing [XYZ]

This vulnerability is not applicable to Python.

Python.11 Unchecked Array Copying [XYW]

This vulnerability is not applicable to Python.

Python.12 Pointer Casting and Pointer Type Changes [HFC]

This vulnerability is not applicable to Python.

Python.13 Pointer Arithmetic [RVG]

This vulnerability is not applicable to Python.

Python.14 Null Pointer Dereference [XYH]

This vulnerability is not applicable to Python.

Python.15 Dangling Reference to Heap [XYK]

This vulnerability is not applicable to Python.

Python.16 Wrap-around Error [XYY]

This vulnerability is not applicable to Python.

Python.17 Sign Extension Error [XZI]

This vulnerability is not applicable to Python.

Python.18 Choice of Clear Names [NAI]

Python.18.1 Applicability to language

Python provides very liberal naming rules:

- Names are of any length and consist of letters, numerals and underscores only. 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 (e.g., `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 (e.g., `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;
- Names starting and ending with two underscores (__) are system-defined names; and
- 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 (e.g., 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 also Python.23 Initialization of Variables [LAV]).
- Names can be unique but may look similar to other names (e.g., `alpha` and `aLpha`, `__x` and `_x`, `_beta__` and `__beta_`) which could lead to the use of the wrong variable especially because Python does not support declarations:

```
veeeeeeeeerylongname = 'abc'
# do stuff
veeeeeeeeeerylongname = 'xyz'#name has an extra 'e'
print(veeeeeeeeerylongname)#=> abc
```

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`.

Python.18.2 Guidance to language users

- Avoid names that differ only by case unless necessary to the logic of 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.

Python.19 Dead Store [WXQ]

Python.19.1 Applicability to language

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.

Python has as a beneficial feature - 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 (e.g., numbers, string, lists etc.). 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 Python.22.1 Applicability to language Python.22.1 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.

Python.19.2 Guidance to language users

- Avoid rebinding except where it adds value;
- Ensure that when examining code that you take into account that a variable can be bound (or rebound) to another object (of same or different type) at any time; and
- Variables local to a function are deleted automatically when the encompassing function is exited but, though not a common practice, you can also explicitly delete variables using the `del` statement when they are no longer needed.

Python.20 Unused Variable [YZS]

The applicability to language and guidance to language users sections of the Python.19 Dead Store [WXQ] write-up are applicable here.

Python.21 Identifier Name Reuse [YOW]

Python.21.1 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 (i.e., when it is first assigned a value), it is associated (i.e., bound) to the namespace associated with the location where the assignment statement is made (e.g., 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:

```
a = 1
```

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

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

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

In the case above, the function is updating the variable `a` 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 `a` is *assigned* a value of `2`. If the function had instead simply *referenced* `a` without assigning it a value, then it would reference the topmost variable `a` which, by definition, is always a global:

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

The rule illustrated above is that attributes of modules (i.e., 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 onlyand 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 an enclosing function definition to reference a nested function's variable(s).

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:

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

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

Python.21.2 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 document 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.

Python.22 Namespace Issues [BJL]

Python.22.1 Applicability to language

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 (i.e., 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 as to from where an attribute is being fetched. Qualification can also occur from an encompassed namespace up to the encompassing namespace using the `global` statement:

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

The example above uses an explicit `global` statement which makes it clear that the variable `y` is not

local to the function `x` – it assigns the value of `1` to the variable `y` in the encompassing module. (Note that values 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 document uses this simpler, though not as exact, wording.)

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 the function's namespace) in functions *statically* by looking for one or more assignments to a name within the function. If one or more assignments are found then the name is

noted as being local to that 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 Python.21 Identifier Name Reuse [YOW] 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 (i.e. a `def` or `lamda (`A `lambda` is a single expression function definition));
- Then the global namespace; and
- Lastly the built-in's namespace.

Python.22.2 Guidance to language users

- When practicable, consider using the `import` statement without the `from` clause. This forces the importing program to use qualification to access the imported module's attributes;
- When using the `import` statement, rather than use the `from *` form (which imports all of the module's attributes into the importing program's namespace), instead use the `from` clause to explicitly name the attributes that you want to import (e.g., `from alpha import a, b, c`) so that variables, functions and classes are not inadvertently overlaid; and
- 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 (e.g., `global a, b, c`).

Python.23 Initialization of Variables [LAV]

Python.23.1 Applicability of language

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 (i.e., 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 (i.e., 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.

Python.23.2 Guidance to language users

- Ensure that it is not logically possible to reach a reference to a variable before it is assigned. The example above illustrates just such a case where the programmer wants to print the value of `x` but has not assigned a value to `x` – this proves that there is missing, or bypassed, code needed to provide `x` with a meaningful value at runtime.

Python.24 Operator Precedence/Order of Evaluation [JCW]

Python.24.1 Applicability to language

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 subexpression evaluates to `True`. The short circuit effect is non-consequential above but in the case below the effect is subtle and potentially destructive:

```
def x(i):
    if i:
        return True
    else:
        1/0  # Hard stop
a = 1
b = 0
while True:
    if x(a) or x(b):
        print('a or b is True')
```

The code above will go into an endless loop because `x(b)` is never evaluated.

Python.24.2 Guidance to language users

- Use parenthesis liberally to force intended precedence;
- Be aware that short-circuited expressions can cause subtle errors because not all sub-expressions may be evaluated; and
- Break large/complex statements into smaller ones using temporary variables for interim results.

Python.25 Side-effects and Order of Evaluation [SAM]

Python.25.1 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 is:

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

`a` is assigned the first value of the first object that has a non-zero (i.e., `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 the function `a` returns a `True` result then the function `b` will not be called which may cause unexpected results.

Python.25.2 Guidance to language users

- Be aware of Python's short-circuiting behaviour when expressions with side effects are used on the right side of a Boolean expression; if necessary perform each expression first and then evaluate the results:

```
x = a()
y = b()
if x or y …
```

Python.26 Likely Incorrect Expression [KOA]

Python.26.1 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. However Python 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 (i.e., 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
```

Python.26.2 Guidance to language users

- Be sure to add parentheses after a function call in order to invoke the function; and
- 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.

Python.27 Dead and Deactivated Code [XYQ]

Python.27.1 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. Further, 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 provides convenient ways to group attributes (e.g., 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 (e.g., 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.

Python.27.2 Guidance to language users

- Import just the attributes that are required by using the `from` statement to avoid adding dead code; and
- Be aware that subsequent imports have no effect; use the `reload` statement instead if a fresh copy of the module is desired.

Python.28 Switch Statements and Static Analysis [CLL]

Python.28.1 Applicability to language

By design Python does not have a switch statement nor does it have the concept of labels or branching to a demarcated “place”. Python enforces structure by not providing these constructs but it also provides several statements to select actions to perform based on the value of a variable or expression. The first of these are the `if/elseif/else` statements which operate as they do in other languages so this warrants no further coverage here.

Python provides a `break` statement which allows a loop to be broken with an immediate branch to the first statement after the loop body:

```
a = 1
while True:
    if a > 3:
         break
    else:
        print(a)
        a += 1
```

The loop above prints 1, 2 and 3, each on separate lines, then terminates upon execution of the `break` statement.

Python.28.2 Guidance to language users

- Use `if/elseif/else` statements to provide the equivalent of switch statements.

Python.29 Demarcation of Control Flow [EOJ]

Python.29.1 Applicability to language

Python makes demarcation of control flow very clear because it uses indentation (using spaces or tabs – but not both) 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`.

Python.29.2 Guidance to language users

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

Python.30 Loop Control Variables [TEX]

Python.30.1 Applicability to language

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 (i.e., `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 is unusual in that it does not provide a loop control variable therefore it is not possible to vary the sequence or number of loops that are performed other than by the use of the `break` statement (covered in Python.28 Switch Statements and Static Analysis [CLL]) which can be used to immediately branch to the statement after the loop block.

When using the `for` statement to iterate though an interable object such as a list, there is no way to influence the loop "count" because it's not exposed. The variable `x` 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 loops 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']
```

Python.30.2 Guidance to language users

- Be careful to only modify loop control variables 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.

Python.31 Off-by-one Error [XZH]

Python.31.1 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.

Python.31.2 Guidance to language users

- Be aware of Python's indexing from zero and code accordingly.

Python.32 Structured Programming [EWD]

Python.32.1 Applicability to language

Python is designed to make it simpler to write structured program by requiring indentation to show scope of control in blocks of code:

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

In many languages the last `print` statement would be executed because they associate the `else` with the immediately prior `if` while Python uses indentation to link the `else` with its associated `if` statement (i.e. the one *above* it).

Python also encourages structured programming by *not* introducing any of the following constructs which could easily lead to unstructured code:

- Labels and branching statements such as `GO TO`;
- `Case`, `GO TO DEPENDING, EVALUATE`, `switch` and other statements that branch dependent on a variable's value; and
- `ALTER` which changes `GO TO` *`label`* to branch to a different label.

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')
```

Python.32.2 Guidance to language users

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

Python.33 Passing Parameters and Return Values [CSJ]

Python.33.1 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. Another way to consider Python's argument handling is to think of the passing of immutable objects as passing by *value* and the passing of mutable objects as being passed by *pointer*:

```
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.

Python functions can also optionally return a value:

```
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.

Python.33.2 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

---

Python.34 Dangling References to Stack Frames [DCM]

This vulnerability is not applicable to Python.

Python.35 Subprogram Signature Mismatch [OTR]

Python.35.1 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 document but the reader should be aware that improper coding of any non-Python modules or their interface could cause a call stack problem

Python.35.2 Guidance to language users

- Ensure that you are using a trusted source when using non-standard imported modules.

Python.36 Recursion [GDL]

Python.36.1 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.

Python.36.2 Guidance to language users

- Avoid recursion when practical unless the bounds are well-defined and protected; and
- Consider utilizing the default limit of recursion to a level of 1,000 (default limit) or less by modifying the limit using the `setrecursionlimit` function.

Python.37 Returning Error Status [NZN]

Python.37.1 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, is allowed to continue to run after an exception by virtue of the `try/except` statement pair.

Python.37.2 Guidance to language users

- Use Python's exception handling with care in order to not catch errors that are intended for other exception handlers; 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.

Python.38 Termination Strategy [REU]

Python.38.1 Applicability to language

Python has a rich set of exception handling statements which can be utilized to implement a termination strategy that assures the best possible outcome ranging from a hard stop to a clean-up and fail soft strategy. Refer to Python.37 Returning Error Status [NZN] for an example of an implementation that cleans up and continues.

Python.38.2 Guidance to language users

- Use Python's exception handling statements to implement an appropriate termination strategy.

Python.39 Type-breaking Reinterpretation of Data [AMV]

This vulnerability is not applicable to Python.

Python.40 Memory Leak [XYL]

Python.40.1 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 – this is beyond the scope of this document. 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 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.

Python.40.2 Guidance to language users

- Release all objects when they are no longer required.

Python.41 Templates and Generics [SYM]

This vulnerability is not applicable to Python.

Python.42 Inheritance [RIP]

Python.42.1 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.

Python.42.2 Guidance to language users

- 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.

Python.43 Extra Intrinsics [LRM]

Python.43.1 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 See Python.22 Namespace Issues [BJL]):

```
x = 'abc'
```

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

Python.43.2 Guidance to language users

- Do not override built-in “intrinsics” unless absolutely necessary

---

Python.44 Argument Passing to Library Functions [TRJ]

Python.44.1 Applicability to language

Refer to Python.35 Subprogram Signature Mismatch [OTR].

Python.44.2 Guidance to language users

Refer to Python.35 Subprogram Signature Mismatch [OTR].

Python.45 Dynamically-linked Code and Self-modifying Code [NYY]

Python.45.1 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
```

Python.45.2 Guidance to language users

- Avoid using `exec` or `eval` and *never* use these with untrusted code; and
- Ensure that the file path and files being imported are from trusted sources.

Python.46 Library Signature [NSQ]

Python.46.1 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 document.

Python does not have a library signature checking mechanism but its API provides functions and classes to help ensure that the signature of the extension matches the expected call arguments and types. See Python.35 Subprogram Signature Mismatch [OTR].

Python.46.2 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.

Python.47 Unanticipated Exceptions from Library Routines [HJW]

Python.47.1 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 Python.21 Identifier Name Reuse [YOW] and elsewhere in this document).

For modules coded in other languages the risks include:

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

Python.47.2 Guidance to language users

- Wrap calls to library routines and use exception handling logic to intercept and handle exceptions when practicable.

Python.48 Pre-processor Directives [NMP]

This vulnerability is not applicable to Python.

Python.49 Obscure Language Features [BRS]

Python.49.1 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 Python.22 Namespace Issues [BJL].

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.49.2 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.

Python.50 Unspecified Behaviour [BQF]

Python.50.1 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 (e.g., changing the value of `c` to 2 will not affect `a` or `b`). Refer also to Python.2.2 Key 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.

Python.50.2 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.

Python.51 Undefined Behaviour [EWF]

Python.51.1 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 (i.e., `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` 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` subclass.
- Modifying the dictionary returned by the `vars` built-in has undefined effects when used to retrieve the dictionary (i.e., the namespace) for an object.
- Formfeed 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, a when 2 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 any 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.

Python.51.2 Guidance to language users

- Understand the difference between testing for equivalence (e.g. `==`) and equality (e.g., `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` subclass in a callable in the `Future` class;
- Never modify the dictionary object returned by a `vars` call;
- Never use form characters used for indentation;
- 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.

Python.52 Implementation–defined Behaviour [FAB]

Python.52.1 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.

Python.52.2 Guidance to language users

- Always use either spaces or tabs (but not both) for indentations;
- 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` (e.g., '`win32`', '`darwin`' etc.).
- 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`).

Python.53 Deprecated Language Features [MEM]

Python.53.1 Applicability to language

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

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

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

- With the new syntax, the `contextlib.nested()` function is no longer needed and is now deprecated.
- Deprecated `PyNumber_Int()`. Use `PyNumber_Long()` instead.
- Added a new `PyOS_string_to_double()` function to replace the deprecated functions `PyOS_ascii_strtod()` and `PyOS_ascii_atof()`.
- Added `PyCapsule` as a replacement for the `PyCObject` API. The principal difference is that the new type has a well defined interface for passing typing safety information and a less complicated signature for calling a destructor. The old type had a problematic API and is now deprecated.

Python.53.2 Guidance to language users

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