Volition Python Coding Standards

**Original Author:** Adam Pletcher, Studio Architect - Technical Art, [adam.pletcher@dsvolition.com](mailto:adam.pletcher@dsvolition.com)  
**Last updated:** , by

If you’re a C/C++ programmer, please see the [Volition C/C++ Coding Standards](http://vsp.dsvolition.com/studio/programming/Shared%20Documents/Volition%20C%20Coding%20Standards.docx).

**CONTENTS**

[Overview 3](#_Toc500502167)

[Python Working Environment 3](#_Toc500502168)

[Wing IDE 3](#_Toc500502169)

[Indentation 4](#_Toc500502170)

[Vertical Alignment 4](#_Toc500502171)

[Headers, Templates 4](#_Toc500502172)

[Language Guidelines 4](#_Toc500502173)

[Imports 4](#_Toc500502174)

[Only import packages and modules 5](#_Toc500502175)

[Do not rename modules or packages on import 6](#_Toc500502176)

[Exceptions 6](#_Toc500502177)

[Do not use catch-all except statements 6](#_Toc500502178)

[Keep try blocks short 7](#_Toc500502179)

[Use finally clause to clean-up 7](#_Toc500502180)

[Raising Exceptions 8](#_Toc500502181)

[Create User-Defined Exceptions 8](#_Toc500502182)

[Global Variables 9](#_Toc500502183)

[Constants 10](#_Toc500502184)

[Classes 10](#_Toc500502185)

[Strings 11](#_Toc500502186)

[Nested/Local Functions and Classes 11](#_Toc500502187)

[List Comprehensions 12](#_Toc500502188)

[Default Iterators and Operators 12](#_Toc500502189)

[Keyword Arguments 12](#_Toc500502190)

[True/False Evaluation 13](#_Toc500502191)

[Deprecated Language Features 14](#_Toc500502192)

[Style Guidelines & Best Practices 14](#_Toc500502193)

[Indentation 14](#_Toc500502194)

[Naming 14](#_Toc500502195)

[Comments 15](#_Toc500502196)

[Doc Strings 15](#_Toc500502197)

[Block and In-Line Comments 16](#_Toc500502198)

[Todo Comments 16](#_Toc500502199)

[Main 17](#_Toc500502200)

[Parentheses 17](#_Toc500502201)

[Whitespace 18](#_Toc500502202)

[Vertical 18](#_Toc500502203)

[Surrounding 19](#_Toc500502204)

[Semicolons 19](#_Toc500502205)

[Working with XML 19](#_Toc500502206)

[Perforce API 20](#_Toc500502207)

[Parting Words 21](#_Toc500502208)

# Overview

This document is a counterpart to the [Volition C/C++ Coding Standards](http://vsp.dsvolition.com/studio/programming/Shared%20Documents/Volition%20C%20Coding%20Standards.docx), but specific to Python.

Python is actively used at Volition, by both Tech Artists/Designers and Programmers. This guide is a list of do’s and don’ts for our Python programs. It draws influences from the [Volition C/C++ Coding Standards](http://ctg.volition.net/volition_coding_standards), [Google Python Style Guide](http://google-styleguide.googlecode.com/svn/trunk/pyguide.html), [Python.org Style Guide for Python Code](http://www.python.org/dev/peps/pep-0008/), and most of all, observations on what’s worked best in existing Python code at the studio.

This doc is not intended to be gospel. The intention is only to illustrate ideals and establish a spirit of better, more legible code-writing across the studio.

One of the most-quoted cornerstones of the language is the [Zen of Python](http://www.python.org/dev/peps/pep-0020/) :

Beautiful is better than ugly.  
 Explicit is better than implicit.  
 Simple is better than complex.  
 Complex is better than complicated.  
 Flat is better than nested.  
 Sparse is better than dense.  
 Readability counts.  
 Special cases aren't special enough to break the rules.  
 Although practicality beats purity.  
 Errors should never pass silently.  
 Unless explicitly silenced.  
 In the face of ambiguity, refuse the temptation to guess.  
 There should be one-- and preferably only one --obvious way to do it.  
 Although that way may not be obvious at first unless you're Dutch.  
 Now is better than never.  
 Although never is often better than \*right\* now.  
 If the implementation is hard to explain, it's a bad idea.  
 If the implementation is easy to explain, it may be a good idea.  
 Namespaces are one honking great idea -- let's do more of those!

The most relevant one here is “readability counts”. [Guido](http://en.wikipedia.org/wiki/Guido_van_Rossum) commented once that code is read much more often than it is written. Readability is *crucial* when writing code, and should be your top priority.

Before we get into the rules, here’s when you would be justified in **breaking** them [taken from [PEP 8](http://www.python.org/dev/peps/pep-0008/)]:

1. When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.
2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess.

While there’s a lot of detail spelled out below, these are just standards and guidelines. You’ll certainly find cases where applying a guideline would make things worse. Use your best judgement.

# Python Working Environment

## Wing IDE

[Wing IDE Professional](http://www.wingware.com/wingide) is the standard Python IDE at Volition. It contains all the features we’ve required to date, including code completion, source assistant, remote debugging, etc. They also have great, amazingly fast customer support. Please install the latest release here:

[\\voliit1\it\Programs\WingIDE\wingide\_latest.lnk](file://voliit1/it/Programs/WingIDE/wingide_latest.lnk)

Then open Dev Config and hit Save. This will install a few custom things for Wing, including snippets/header templates, custom commands, and other standard settings we use across the studio.

Wing can be run for at least a couple weeks in full-featured trial mode. If you need a permanent license, please notify Adam Pletcher or Helpdesk.

## Indentation

Like all code at Volition, indentation settings must be **tabs 3.** Please do not mix spaces and tabs within a file.

The Python community most often uses **spaces 4** for indentation. When working inside third-party files, it’s fine to stick to their native standard. The tabs 3 Volition standard is only for files created by us.

If you find any V-made files that don’t use tabs 3, please convert them. Wing has a great Indentation tool for doing that.

### Vertical Alignment

Use best judgement when vertically aligning consecutive lines. This applies to :, #, =, and similar characters. Readability is generally the most important factor. However, be on the lookout for blocks of code that can become a maintenance headache to maintain alignment.

# Easy, but not as readable

**foo = 1000**

**long\_name = 2**

**some\_dict = {**

**'foo': 1,**

**'long\_name': 2,**

**}**

# Readable, but might be a hassle to add longer entries to, requiring realignment of older lines, etc.  
# Use good judgement

**Foo = 1000**

**long\_name = 2**

**some\_dict = {**

**'foo' : 1,**

**'long\_name' : 2,**

**}**

In a nutshell, if a long block of these is particularly hard to read, feel free to align them.

## Headers, Templates

Please use the Module, Class and Function headers found in the Wing Templates tool. They are designed to be very readable, but also work with the Sphinx auto-doc builder used to create docs [like these](file:///P:/CTG/documentation/index.html). It is particularly important to use the standard headers when writing library-level code in modules or packages.

# Language Guidelines

## Imports

Placement, Formatting

Always put all imports at the top of the file or module, after the file’s doc string, before constants or other code. Do not import inside code (unless the preceeding code is required to do the import successfully).

Imports should be grouped sensibly, roughly in order of most- to least-generic, with only one import per line.

# GOOD

**import** **os**

**import** **sys**

**import** **string**

**import** **wx**

**import** **wx.lib.agw**

**import** **vlib.os**

**import** **ctg.we**

**import** **ctg.ui.widgets**

# BAD (one per line only please)

**import** **ctg.we**, **ctg.ui.widgets**

**import** **os**, **sys**

# BAD (no in-line imports please, put them all at the top)

**def** get\_time( ):

**import** **time**

**return** time.clock( )

### Only import packages and modules

Do not directly import classes, functions, or other objects within a module. Wherever possible, please reference them through the module instead.

# BAD **from** **foo** **import** \*

**from** **foo** **import** bar  
**from** **ctg.ui.widgets** **import** Spinner

The above are bad and can lead to serious maintenance issues. They make it hard to find module dependencies, and can confuse both Wing’s source analysis and users reading the code.

The first one is particularly nasty, as it blindly imports everything from a module… importing things and running code you do not need or want, including things added to that module later. Doing this with third-party code is also a security risk.

Instead, use **import x** for importing packages and modules. Refer to the contents of a module by using referencing the package/module name as a prefix

# GOOD **import** **ctg.ui.widgets**

**...**

**spin1 =** ctg.ui.widgets.Spinner(self**, -**1)

The above makes the code easier to follow, and any use of code from external modules is clear. It also more reliably works with go-to-definition features in the IDE.

If your package/module prefixes are getting long and unwieldy, this is also acceptable:

# ACCEPTABLE **from** **ctg.ui import widgets**

**...**

**spin1 =** widgets.Spinner(self**, -**1)

Additionally, be mindful of doing imports relative to the current module. Even if the module is in the same package as the current file, do not directly import the module without the full package name. This might cause the package to be imported twice (with unintended side effects) when the "main" module that is used to start an application lives inside a package and uses modules from that same package.

### Do not rename modules or packages on import

Python merrily allows this:

# BAD **import** **vlib.com as vcom**

Please resist the temptation. This makes the code hard to follow, particularly when tracing back to definitions in the modules you’ve imported. Similar pros/cons to the previous guideline above. Instead, do regular imports:

# GOOD **import** **vlib.com**

TODO: Add guidelines on writing modules and packages (ie. keep module names short, to facilitate the above guideline, etc.)

## Exceptions

Python errors are exception based. Exceptions are a means of breaking out of the normal flow of control to handle errors or other unexpected conditions. Guidelines for exceptions:

### Do not use catch-all except statements

Doing this can swallow errors you did not expect, causing problems for the user and making it difficult to debug. Python is very tolerant in this regard and an unqualified except will really catch *everything* including Python syntax errors.

# BAD **try**:

new\_icon\_filename = os.path.split( icon\_name )[ 0 ][ :-25 ] + '**\\**blah.png'

bmp = ctg.ui.wxBitmap\_from\_png( new\_icon\_filename )

**return** bmp

**except**:

**return** None

The author intended to handle the case where the icon filename specified did not exist on disk. This would cause the wxBitmap\_from\_png call to fail with an IOError, go to the except and return None from this function.

The problem is, the code in that try block can fail in ways he didn’t expect. The [ :-25 ] in the first line, for instance, makes assumptions that the folder it’s in has a name of a specific length. If that path is shorter, that index will throw an IndexError, but get swallowed by the generic except block and return None silently.

Instead, only handle exception types you expect:

# BETTER **try**:

new\_icon\_filename = os.path.split( icon\_name )[ 0 ][ :-25 ] + '**\\**blah.png'

bmp = ctg.ui.wxBitmap\_from\_png( new\_icon\_filename )

**return** bmp

**except IOError**:

**return** None

This would trap the IOError resulting from a non-existent file, but the potential IndexError would still throw an exception like it should.

The only time you should use a catch-all except is when you’re either re-raising the error, or otherwise bringing it to someone’s attention in an obvious manner. Here’s one example, showing how to give a friendly error message to the user before re-raising it:

# BETTER **try**:

new\_icon\_filename = os.path.split( icon\_name )[ 0 ][ :-25 ] + '**\\**blah.png'

bmp = ctg.ui.wxBitmap\_from\_png( new\_icon\_filename )

**return** bmp  
  
**except IOError**:  
 **return** None  
  
**except:**  
 wx.MessageBox( 'Unexpected error, please notify TechArt.', style = wx.ICON\_ERROR )  
 **raise**

Another example of appropriate use of a catch-all except, where unexpected errors result in returning None, but (this is the important part) it’s logged first so it can be seen by someone:

# BETTER  
log = vlib.log.LogHelper( 'magic tool', emailTo = 'sr3toolslager@thq.com' )

**try**:

new\_icon\_filename = os.path.split( icon\_name )[ 0 ][ :-25 ] + '**\\**blah.png'

bmp = ctg.ui.wxBitmap\_from\_png( new\_icon\_filename )

**return** bmp  
  
**except IOError**:  
 **return** None  
  
**except:**  
 log.error( 'Unexpected error getting bitmap' )  
 **return** None

You would only do the above if returning None from unexpected errors is acceptable for the client code, and when that’s preferable to it raising and stopping execution.

Note, the above examples say “better” and not “good”. See the next guideline for a more ideal version of it.

### Keep try blocks short

The more code you put inside a try block, the more likely it is to toss an error you did not expect. Take this example from above, for instance:

# BAD **try**:

new\_icon\_filename = os.path.join( os.path.split( icon\_name )[ 0 ], 'blah.png' )

bmp = ctg.ui.wxBitmap\_from\_png( new\_icon\_filename )

**return** bmp

**except IOError**:

**return** None

The except is designed to handle the case when the filename cannot be found on disk. There’s no reason for the first filename-building line to be part of the try block. This is better:

# GOODnew\_icon\_filename = os.path.join( os.path.split( icon\_name )[ 0 ], 'blah.png' )

**try**:

bmp = ctg.ui.wxBitmap\_from\_png( new\_icon\_filename )

**return** bmp

**except IOError**:

**return** None

### Use finally clause to clean-up

Where appropriate use the **finally** clause to clean-up, whether or not the exception was raised in the try block. This is good in cases where you need to close a file handle or a socket.

# GOOD **try**:

self.send\_data( self.socket, data )

**except** socket.error:

**print** 'Error sending data'

**raise**

**finally**:

self.socket.close( )

### Raising Exceptions

When your code encounters error conditions, raise an exception using the **raise** built-in. If there is a suitable [built-in exception type](http://docs.python.org/library/exceptions.html) that matches the condition, raise it.

# GOOD **raise** IOError( 'Config file not found on disk' )

The Python **assert** statement can also useful for raising exceptions when things aren’t as-expected:

# ACCEPTABLE **assert** isinstance( input\_obj, float )

If input\_obj was not a float, an **AssertionError** is raised:

Traceback (most recent call last):

File "<pyshell#4>", line 1, **in** <module>

assert isinstance( a, float )

AssertionError

The following is better and more informative, offering some additional info on what went wrong:

# GOOD **assert** isinstance( input\_obj, float ), '"input\_obj" not a float, found {0} instead.'.format( type( input\_obj ) )

Resulting in:

Traceback (most recent call last):

File "<pyshell#5>", line 1, **in** <module>

assert isinstance( input\_obj, float ), '"input\_obj" not a float, found {0} instead.'.format( type( input\_obj ) )

AssertionError: "input\_obj" not a float, found <type 'int'> instead.

### Create User-Defined Exceptions

An excellent way to offer more targeted error handling is to raise a [user-defined exception](http://docs.python.org/tutorial/errors.html#user-defined-exceptions) instead of one of the more general built-in exceptions. A custom exception should be specific to the module, package or other domain in which it occurred.

**class** **Category\_Error**( Exception ):

"""

Error to be raised from Category-related failures in resourcelib

"""

**def** \_\_init\_\_( self, value ):

self.value = value

**def** \_\_str\_\_( self ):

**return** repr( self.value )

...

**try**:

category = \_resourcelib.get\_category( cat\_name )

**except** RuntimeError **as** exc:

**raise** Category\_Error( exc.message )

This turns the normally generic RuntimeError into a more specific Category\_Error for the calling code to more effectively trap. You could also put additional logic inside the except block above to let certain RuntimeErrors re-raise, and not be lumped in with the rest.

## Global Variables

These are tempting at times, but **avoid global variables**. At best they make your code hard to decipher, at worst they have the potential to change module behavior during imports and cause errors.

# BAD  
g\_tool = Tool( )

**class** **Tool**( object ):

**def** \_\_init\_\_( self ):

self.thing = 42

**def** setup( ):

**global** g\_tool

g\_tool.thing = 99

**def** print\_thing( ):

**global** g\_tool  
 **print** 'thing = {0}'.format( g\_tool.thing )

def main( ):

setup( )  
 print\_thing( )

**if** \_\_name\_\_ == '\_\_main\_\_':  
 main( )

Above shows passing the Tool instance around as a global object. Instead, try to keep your global namespace uncluttered with data objects. A better approach is to pass the instance itself into the methods that need to access it:

# GOOD

**class** **Tool**( object ):

**def** \_\_init\_\_( self ):

self.thing = 42

**def** setup( tool ):

tool.thing = 99

**def** print\_thing( tool ):

**print** 'thing = {0}'.format( tool.thing )

def main( ):

tool = Tool( ) # Make tool instance

setup( tool ) # Pass it in to setup function

print\_thing( tool ) # And to print

**if** \_\_name\_\_ == '\_\_main\_\_':  
 main( )

Alternatively, you could put get/set methods the Tool class and use those to access the data. This is a more traditional object-oriented approach, but really just a stylistic decision for the author:

# GOOD

**class** **Tool**( object ):

**def** \_\_init\_\_( self ):

self.\_thing = 42

**def** set\_thing( self, value ):

self.\_thing = value

**def** get\_thing( self ):

**return** self.\_thing

**def** setup( tool ):

tool.set\_thing( 99 )

def main( ):

tool = Tool( )

setup( tool )

**print** 'thing = {0}'.format( tool.get\_thing( ) )

**if** \_\_name\_\_ == '\_\_main\_\_':  
 main( )

There are very few exceptions to this, where use of a global is acceptable.

One notable, common exception is Constants…

## Constants

Constants are typically found at the top-level of a module, and used both in that module and from others after importing that module. Constants are defined by all capital letters with underscores:

# ACCEPTABLE  
**import** **os**

**import** **sys**

# Constants

PI = 3.14159

OPTIONS\_DEFAULT = 2

OPTIONS\_LEFT = 4

OPTIONS\_CENTER = 8

# Classes

**class** **Spam**( object ):

**pass**

When possible the values should be assigned right there and never changed. Rarely this is not possible, and some code needs to execute further down in order for the constant's value to be set properly. This is acceptable, as long as it's set that one time and not changed later.

# Constants

# Put placeholder at the top of module...

MAGIC\_NUMBER = None

<somewhere below>

MAGIC\_NUMBER = \_compute\_magic\_number( )

Another type of module-level object you may see occasionally is one used as a cache. Since these are fluid collections of values that often change, they should not be in all caps like a constant. Just use lowercase-with-underscores like a normal variable, and comment its use clearly.

# BAD  
# Constants

OPTIONS\_DEFAULT = 2

OPTIONS\_LEFT = 4

OPTIONS\_CENTER = 8

ICON\_CACHE = { }

# GOOD

# Constants

OPTIONS\_DEFAULT = 2

OPTIONS\_LEFT = 4

OPTIONS\_CENTER = 8

# Cache of icon images loaded, keyed by their full filename

icon\_cache = { }

## Classes

If a class inherits from no other base classes, it must explicitly inherit from object.

# BAD

**class** **Foo**( ):

**pass**

# GOOD

**class** **Foo**( object ):

**pass**

**class** Spam( Foo ):  
 # Inherits from an existing class

**pass**

Inheriting from object is necessary to make it a [new-style class](http://docs.python.org/reference/datamodel.html#new-style-and-classic-classes), and for properties work properly. It will also protect your code from a potential incompatibility with Python 3. It also defines special methods that implement the default semantics of objects including \_\_new\_\_, \_\_init\_\_, \_\_delattr\_\_, \_\_getattribute\_\_, \_\_setattr\_\_, \_\_hash\_\_, \_\_repr\_\_, and \_\_str\_\_.

In Python 3, all classes are new-style, and inheriting from object will no longer be necessary.

## Strings

Since moving to Python 2.6, it’s no longer desirable to use the % operator when formatting strings. It will be deprecated in Python 3, and we need to start using the new style of string formatting:

# BAD

**print** 'The *%s* *%s* fox.' % ( 'quick', 'brown' )

# GOOD

**print** ( 'The {0} {1} fox.'.format( 'quick', 'brown' ) )

Please see the [Format String Syntax](http://docs.python.org/library/string.html#formatstrings) for more information.

## Nested/Local Functions and Classes

Python allows functions and classes to be defined inside other functions, classes. These are at times difficult to read, and should be avoided when possible.

One common exception is recursive functions, where a short, inner function is sometimes a good and obvious fit:

**# ACCEPTABLE  
def** search\_tree( tree ):

**def** search\_branch( branch ):

branch\_results = [ ]

children = branch.get\_children( )

**for** child **in** children:

branch\_results.extend( search\_branch( child ) )

**return** branch\_results

root = tree.get\_root( )

results = search\_ branch( root )

**return** results

On the other hand, it’s still more clear without nesting:

**# GOOD**

**def** search\_branch( branch ):

branch\_results = [ ]

children = branch.get\_children( )

**for** child **in** children:

leaf\_results.extend( search\_branch( child ) )

**return** branch\_results

**def** search\_tree( tree ):

root = tree.get\_root( )

results = search\_branch( root )

**return** results

Try to avoid nested functions or classes in other contexts. As always, use good judgement here.

## List Comprehensions

List comprehensions are super awesome. When used correctly they provide a concise and efficient way to create/transform lists and iterators without resorting to the use of map(), filter(), or lambda.

Simple list comprehensions can be clearer and simpler than other list creation techniques. Generator expressions can be very efficient, since they avoid the creation of a list entirely.

On the other hand, complicated list comprehensions or generator expressions can be very hard to read.

When writing list comprehensions, make sure it fits reasonably on one line... the mapping expression, for clause, and filter expression. Multiple for clauses or filter expressions are not permitted. Use loops instead when things get more complicated.

# BAD (double loops)

result = [ (x, y) **for** x **in** range( 10 ) **for** y **in** range( 5 ) **if** x \* y > 10 ]

# GOOD

results = [ ]

**for** x **in** range( 10 ):

**for** y **in** range( 5 ):

**if** x \* y > 10:

results.append( (x, y) )

# BAD (multiple filters)

result = [ x **for** x **in** items **if** x.startswith( 'prefix' ) **and** x.endswith( 'suffix' ) **and** 'error' **not** **in** x.lower( ) ]

# GOOD

results = [ ]

**for** x **in** items:

**if** x.startswith( 'prefix' ) **and** x.endswith( 'suffix' ) **and** 'error' **not** **in** x.lower( ):

results.append( x )

## Default Iterators and Operators

Container types, like dictionaries, and sequence types like tuples and lists all define default iterators and membership test operators ("in" and "not in").

These default iterators and operators are simple and efficient. For example, using “in” or “not in” to test key membership in a dictionary is faster than using has\_key( ), which requires an attribute search and uses a relatively expensive function call. “in” and “not in” is shorter, more readable, and the style is consistent with tests for memberships in sequence types like lists and tuples.

Please use default iterators and operators for types that support them, like lists, tuples, sets, dictionaries, and files. The built-in types define iterator methods, too. Favor these over methods that return lists.

# GOOD **if** obj **in** alist: ...  
**if** key **not** **in** adict: ...  
**for** key **in** adict: ...

**for** line **in** afile: ...

**for** k, v **in** dict.iteritems( ): ...

# BAD

**if** **not** adict.has\_key( key ): ...  
**for** key **in** adict.keys( ): ...

**for** line **in** afile.readlines( ): ...

## Keyword Arguments

When supplying keyword arguments, please name them in the function call, and do not pass them by position.

To explain… Python has no function overloading, but relies on keyword arguments to make function call signatures more flexible.

**def** do\_something( value, double = False ):

**if** double:

**print** value \* 2

**else**:

**print** value

Typically you would call it like this:

**>>>** do\_something( 42, double = True ) # GOOD

84

The twist is, Python allows you to supply keyword arguments by position as well:

**>>>** do\_something( 42, True ) # BAD

84

This can be very confusing with functions that take several arguments or keyword arguments, and make debugging difficult. Please name all keyword arguments in your function calls.

Also avoid using mutable object types as default values in keyword arguments. Those values are created at function *definition* time, not when the function is called. This can lead to unexpected side effects. Instead, please use None as the default, and inside the function check for that, setting it to an empty list or other mutable type as desired. Some examples:

# GOOD **def** do\_something( value, things = None ):

**if** things is None:

things = [ ]  
 ...

# BAD **def** do\_something( value, things = [ ] ):

...

## True/False Evaluation

Use the “implicit” True/False whenever possible, using '**is**' and '**is not**'.

Do not use **==** or **!=** to compare singletons like None. An '**is**' or '**is not**' identity test with a singleton object will be more optimal than the **==** or **!=** operators, as **==** and **!=** will try to use the \_\_eq\_\_ and \_\_cmp\_\_ methods on each object on either side of the operator. Those are unnecessary steps when you just want to determine if something **is** or **is not** None.

*# GOOD*

**if** users:

**print** 'users'

**if** **not** users:

**print** 'no users'

**if** foo == 0:

self.handle\_zero( )

**if** i % 10 == 0:

self.handle\_multiple\_of\_ten( )

*# BAD*

**if** len( users ) == 0:

**print** 'no users'

**if** foo **is** **not** None **and** **not** foo:

self.handle\_zero( )

**if** **not** i % 10:

self.handle\_multiple\_of\_ten( )

Python evaluates certain values as False when in a boolean context. An easy rule-of-thumb is that all "empty" values are considered False… so 0, None, [ ], { }, "" all evaluate as False in a boolean context.

This notation may initially look strange to C/C++ developers, but conditions using Python booleans are easier to read and less error-prone. In most cases, they're also faster.

There are a few caveats that you should keep in mind though:

1. Never use == or != to compare singletons like None. Use is or is not.
2. Beware of writing if x: when you really mean if x is not None:—e.g., when testing whether a variable or argument that defaults to None was set to some other value. The other value might be a value that's false in a boolean context.
3. Never compare a boolean variable to False using ==. Use if not x: instead. If you need to distinguish False from None then chain the expressions, such as if not x and x is not None:.
4. For sequences (strings, lists, tuples), use the fact that empty sequences are false, so if not seq: or if seq: is preferable to if len( seq ): or if not len( seq ):.
5. When handling integers, implicit false may involve more risk than benefit (i.e., accidentally handling None as 0). You may compare a value which is known to be an integer (and is not the result of len()) against the integer 0.

## Deprecated Language Features

Use string.format instead of % for print formatting. Use list comprehensions and for loops instead of filter, map, and reduce. Use string methods instead of the string module where possible.

We only use Python releases that support these new styles, so there is no reason not to use them.

# BAD

**print** 'the *%s* brown fox' % ( 'quick' )

words = string.split( foo, ':' )

map( **lambda** x: x[ 1 ], filter( **lambda** x: x[ 2 ] == 5, my\_list ) )

# GOOD

'the {0} brown fox'.format( 'quick' )

words = foo.split( ':' )

[ x[ 1 ] **for** x **in** my\_list **if** x[ 2 ] == 5 ]

# Style Guidelines & Best Practices

## Indentation

Like all code at Volition, indentation settings must be **tabs 3.**

## Naming

Naming conventions are as follows:

|  |  |  |
| --- | --- | --- |
| **Type** | **Public** | **Internal, Semi-Private** (not for use by outside code) |
| Package filenames | lower\_with\_under |  |
| Module filenames | lower\_with\_under | \_lower\_with\_under |
| Classes | Cap\_Words | \_Cap\_Words |
| Functions | lower\_with\_under( ) | \_lower\_with\_under( ) |
| Exceptions | Cap\_Words |  |
| Global, module-level constants | CAPS\_WITH\_UNDER | \_CAPS\_WITH\_UNDER |
| Global, module-level variables | Cap\_first\_word\_then\_lower | \_Cap\_first\_word\_then\_lower |
| Global class constants | CAPS\_WITH\_UNDER | \_CAPS\_WITH\_UNDER |
| Global class variables | lower\_with\_under | \_lower\_with\_under |
| Instance Variables | lower\_with\_under | \_lower\_with\_under (semi-private) or \_\_lower\_with\_under (private) |
| Method Names | lower\_with\_under( ) | \_lower\_with\_under( ) (semi-private) or \_\_lower\_with\_under() (private) |
| Function/Method Parameters | lower\_with\_under |  |
| Local Variables | lower\_with\_under |  |

NOTE: For naming of “private” code, notice we’re using the single leading underscore (Python’s “semi-private”) rather than the double-leading underscore (Python’s “private”). We do this because the double underscore causes method names to be [obfuscated](http://stackoverflow.com/questions/1162234/what-is-the-benefit-of-private-name-mangling-in-python), and as a TA group we agreed this was irritating, especially during debugging sessions, and generally not necessary for us. Instead, we use single underscore “semi-private” notation to indicate private stuff that shouldn’t be used by outside code.

Names to avoid:

* single character names, except for counters or iterators
* dashes (-) in any package/module name
* \_\_double\_leading\_and\_trailing\_underscore\_\_ names. These are reserved by Python. Note, single leading double underscores with none trailing is fine, and is the supported method of making a class member private (which just means “not exportable” in Python)

## Comments

Be sure to use the right style for module, function, method and in-line comments.

### Doc Strings

Python has a unique commenting style using doc strings. A doc string is a string that is the first statement in a package, module, class or function. These strings can be extracted automatically through the \_\_doc\_\_ member of the object and are used by pydoc and Sphinx. Doc strings also appear inside Wing in the Source Assistant pane, and when typing help( object ) in a Python interpreter.

Our convention for doc strings is to use the three double-quotes at the start and end, on lines by themselves. They also use reStructured Text markup, for the Sphinx doc-builder we use. Here is an example of the class doc string template, right after inserting it using Wing:

**class** **Spam**( object ):

"""

Enter a description of the class here.

\*\*Arguments:\*\*

:``Argument``: Enter a description for the argument here.

\*\*Keyword Arguments:\*\*

:``Argument``: Enter a description for the keyword argument here.

\*\*Examples:\*\* ::

Enter code examples here. (optional field)

\*\*Todo:\*\*

\* Enter thing to do. (optional field)

\*\*Author:\*\*

Adam Pletcher, adam@volition-inc.com, 6/2/2010 3:56:22 PM

"""

**def** \_\_init\_\_( self ):

**pass**

“Description”, “Arguments”, “Keyword Arguments”, “Returns” (functions/methods only), and “Author” are all required. If you have a function that does not require Arguments, or Keyword Arguments etc. please put None, with no text decoration.

\*\*Arguments:\*\*

None

\*\*Returns:\*\*

None

The description should give a brief, but complete overview of the module/class/function, at the big-picture level. Basically what it does, not specifically how it does it.

The doc string for a function, for instance, should give enough information to write a call to the function without looking at a single line of the function's code. Args should be individually documented, an explanation following each. The doc string should specify the expected types where specific types are required.

Doc strings should also contain links to documents with more details on the code architecture, where applicable. Otherwise, specifics are better left to the in-line and block comments in the code itself.

Doc Strings use reStructured Text, which is defined in PEP-287 at <https://www.python.org/dev/peps/pep-0287/>.

A good resource to help you understand how the docstring -> reStructuredText -> html page generation works is:

<http://www.tele3.cz/jbar/rest/rest.html>

A great resource to test your docstrings and provide feeback is:

<http://www.tele3.cz/jbar/rest/rest.html>  Just paste your docstring (from “”” to “””) in the box and click render. The page will attempt to render out the docstring as ReST and provide inline warnings/errors were problems occur.

### Block and In-Line Comments

Use block and in-line comments to explain what’s happening in your code, where it’s happening. Try to do this as you write code, and don’t rely on some “document new system” phase later, because it won’t happen, my friend.

If you're going to have to explain it at the next [code review](http://en.wikipedia.org/wiki/Code_review), you should comment it now. Complicated operations get a few lines of comments before the operations commence. Add more comments at the end of lines, if you feel each step needs more explanation.

# GOOD

# We use a weighted dictionary search to find out where i is in

# the array. We extrapolate position based on the largest num

# in the array and the array size and then do binary search to

# get the exact number.

**if** i & ( i – 1 ) == 0: # true if i is a power of 2

To improve legibility, these comments should be at least 2 spaces or a tabstop away from the code.

On the other hand, never describe the code. Assume the person reading the code knows Python (though not what you're trying to do) better than you do.

# BAD:

# Now go through the b array and make sure whenever i occurs

# the next element is i+1

### Todo Comments

Use TODO comments in code that is temporary, hacky, or where there’s missing functionality that will be needed in the future. The standardized doc string headers already have a \*Todo:\* section, please use it! The person that ends up doing that item may be someone else.

\*Todo:\*

\* Add support for flushing the cache

\* Replace the in-code HTML stuff with string.Template

## Main

Even a file meant to be used as a script should be importable, and the act of importing it should not execute the script's main functionality. The main functionality should be in a main( ) function, or one of similar/obvious naming.

Your files run as scripts should always check if \_\_name\_\_ == '\_\_main\_\_' before executing your main program so that the main program is not executed when the module is imported.

# BAD

foo = Bar( )

This next one is closer, but still not great. This puts foo into the module’s global namespace:

# BAD

**if** \_\_name\_\_ == '\_\_main\_\_':  
 foo = Bar( )

The following is best, allowing the script to be imported, and leaving nothing extraneous at the global level:

# GOOD

def main( ):

foo = Bar( )

**if** \_\_name\_\_ == '\_\_main\_\_':  
 main( )

All code at the top level will be executed when the module is imported. Be careful not to call functions, create objects, or perform other operations that should not be executed when the file is being pychecked, pydoced, or run through unit tests.

## Parentheses

Use parens sparingly. Do not use them in return statements or conditional statements unless using parentheses for implied line continuation. It is fine, however, to use parentheses around tuples.

# GOOD

**if** foo:

bar( )

**if** this **and** not that:

bar( )

**while** not\_done:

do\_stuff( )

**return** foo

( keys, values ) = bar( )

my\_tuple = ( this, that )

# BAD

**if** ( foo ):

bar( )

**if** ( this ) **and not**( that ):

bar( )

**while** ( not\_done ):

do\_stuff( )

**return** ( foo )

TODO: Add something about spaces around indexes.

## Whitespace

### Vertical

Give your code some room to breathe.

# GOOD

\_resourcelib.init( 'resourcelib.py' )

# See if new category already exists

**try**:

existing\_cat = \_resourcelib.get\_category( system\_name )

**except** RuntimeError:

existing\_cat = None

**if** existing\_cat:

**raise** Category\_Error( 'Category "{0}" already exists in this workspace'.format( system\_name ) )

# Verify this user has the workspace config file checked out.

workspace\_file = os.path.join( \_resourcelib.get\_path( \_resourcelib.PATH\_ROOT ), 'project\_workspace.ctg\_config' )

result = vlib.perforce.fstat( workspace\_file )[ 0 ]

This is too cramped, making it difficult to read:

# BAD

\_resourcelib.init( 'resourcelib.py' )

# See if new category already exists

**try**:

existing\_cat = \_resourcelib.get\_category( system\_name )

**except** RuntimeError:

existing\_cat = None

**if** existing\_cat:

**raise** Category\_Error( 'Category "{0}" already exists in this workspace'.format( system\_name ) )

# Verify this user has the workspace config file checked out.

workspace\_file = os.path.join( \_resourcelib.get\_path( \_resourcelib.PATH\_ROOT ), 'project\_workspace.ctg\_config' )

result = vlib.perforce.fstat( workspace\_file )[ 0 ]

Please use **three** blank lines between top-level definitions (functions or classes). **Two** blank lines between method definitions. **One** blank line between the class's docstring and its first method, and between a function’s docstring and first line of code.

Use single blank lines as you judge appropriate within functions or methods.

# GOOD

**import** **os**

**class** **Spam**( Food ):

"""

Standard template header here

"""

**def** \_\_init\_\_( self ):

**pass**

**def** foo( self ):

**pass**

**def** like\_spam( ):

"""

Standard template header here

"""

**pass**

### Surrounding

Parens and brackets should have leading/trailing spaces.

GOOD: spam( ham[ 1 ], { eggs: 2 }, [ ] )  
BAD: spam(ham[1], {eggs: 2}, [])

No whitespace before a comma, semicolon, or colon. Do use whitespace after a comma, semicolon, or colon except at the end of the line:

GOOD: **if** x == 4:

**print** x, y

x, y = y, x

BAD: **if** x == 4 :

**print** x , y

x , y = y , x

No whitespace before the open paren/bracket that starts an argument list, indexing or slicing.

GOOD: spam( 1 )

BAD: spam ( 1 )

GOOD: dict['key'] = list[index]

BAD: dict ['key'] = list [index]

Surround operators with a single space on either side for assignment.

GOOD: x == 1  
 y > 3

BAD: x==1  
 y>3

Also use spaces around the '=' sign when used to indicate a keyword argument or a default parameter value.

GOOD: def complex( real, imag = 0.0 ): return magic( r = real, I = imag )

BAD: def complex( real, imag=0.0 ): return magic( r=real, i=imag )

## Semicolons

Do not terminate lines with semi-colons and do not use semi-colons to put multiple on the same line.

## Working with XML

Reading and writing XML is a common thing to do in Python. There's many avenues you can take (too many, IMO), so here's some guidelines we've created. This is based on experience and tests we've run over the years.

* Use xml.etree.cElementTree to do all your XML parsing.  It’s in the standard library.  Easy to use and very fast.  Please don’t use minidom, xmlobjects, regular expressions or even regular file reading.  Those are either slower, fragile for XML, or both.
* Also use cElementTree for XML writing.  If you want it to use newlines and indentation (as we often do), write your cElementTree root node object to disk using the vlib.prettyxml module instead.

If you would like help working with cElementTree, there are many examples in the CTG Editor codebase, or you can just email CTGSupport or the TechArt aliases.

## Perforce API

When using the Python API for Perforce, it’s important that you’re aware of how your tool deals with Perforce connections. In Python terms, we often use connections like this:

p4\_api = P4.P4( )

p4\_api.connect( )

<do stuff>

p4\_api.disconnect( )

That is generally fine, as long as the stuff in between is fairly quick, and doesn’t sit indefinitely in a UI loop or something.

What’s not so great is doing the above but neglecting to do a disconnect( ) when done. Actually this often works out anyway when the code is inside a function. When the execution of that function is finished the p4\_api object will fall out of scope, get dereferenced and destroyed (fairly soon), which causes it to disconnect. I think it’s still best to explicitly disconnect when your tool is done, however.

One convenient way to do this is using connect() as a context manager:

with p4\_api.connect( ):

<do stuff>

This automatically disconnects when execution leaves that block of code *for any reason*. That includes cases where that code throws an exception or otherwise exits the block abnormally. On the other hand, we also don’t want tools rapidly connecting and disconnecting. Doing this can also put some strain on the server. It’s up to you to strike the right balance in your tools code. A good rule of thumb could be: A tool shouldn’t connect/disconnect more than once every few seconds, but also not leave an unused connection open more than a minute or so.

What we *definitely* want to avoid is code like this:

<top of module>

import P4

p4\_api = P4.P4( )

p4\_api.connect( )

<functions, classes, etc.>

This creates an API object at the top-level of a module, opens a connection and never disconnects it. Since that object is on an imported module it never falls out of scope or gets destroyed, at least until the entire Python process exits.

We used to not care about this, but every connection we have consumes one from a finite list, and may prevent another user from connecting. We can keep bumping the max connections but that still costs something on the server. Also, every open connection will eventually time-out anyway. It may take hours/days, but when it does that the tool you wrote in this manner is probably not designed to reconnect gracefully.

vDeployer is an interesting example. It often remains open on people’s workstations for hours/days. The code used to keep one Perforce connection open and would eventually barf in that manner, since the code assumed the connection was still valid. Especially for off-site workers, whose connections in general tend to be somewhat more tenuous. When I added the sync progress bars to vDeployer I also changed it to only open a connection when a sync was starting, then close it when finished. Since it only does 3-4 syncs per “get”, and each one takes at least several seconds, this is a good balance.

# Parting Words

BE CONSISTENT.

If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around all their arithmetic operators, you should too. If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.

On the other hand, if the code you’re working with is hard to follow and doesn’t fit the above standards, consider taking the time to change it.

The point of these guidelines is to have a common vocabulary of coding so we can concentrate on what you're saying rather than on how you're saying it. These global style rules here so people know the vocabulary, but local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they read it. Avoid this.