Table of Contents

[Coding Standards 2](#_Toc521578467)

[PEP8 2](#_Toc521578468)

[Python Coding Standards 3](#_Toc521578469)

[Palo Alto Header 3](#_Toc521578470)

[Code Comments 4](#_Toc521578471)

[Imports 5](#_Toc521578472)

[Naming Conventions 6](#_Toc521578473)

[String Formatting 7](#_Toc521578474)

[Logging 7](#_Toc521578475)

[Indention and Line Length 9](#_Toc521578476)

[Line Spacing 10](#_Toc521578477)

[Whitespace 11](#_Toc521578478)

[Single Quotes 12](#_Toc521578479)

[IDEs a Auto-Formatting 12](#_Toc521578480)

[Shebang Lines 13](#_Toc521578481)

[Linting 13](#_Toc521578482)

# Coding Standards

This set of standards is for the main language (Python) used by the Palo Alto Networks WWSE for automation and operational scripts. This style guide is a list of dos and don'ts for Python programs that will be written for others to use and/or modify. These standards are only guidelines and can be “bent” if there is a good reason. “Because that’s the way I want to do it” is not a good reason.

As stated in the Languages section, going forward we will be using Python 3.6 or later.

## PEP8

Before we start on the Palo Alto Networks WWSE coding standards for Python, it is a good idea to familiarize yourself with [PEP8](https://www.python.org/dev/peps/pep-0008/). PEP8 is the style guide for Python code. The style guide is used as set of *recommendations* for making your code more readable and consistent. It is not intended as a steadfast set of strict rules that you must adhere to. Hence the mantra, “A foolish consistency is the hobgoblin of little minds.” Make sure that you read it and keep it in the back of your mind while writing your code. This helps everyone, including yourself, when the code has to be read sometime in the future.

## Python Coding Standards

For Python code style follow [PEP8](https://www.python.org/dev/peps/pep-0008/) ***plus*** the guidelines below.

### Palo Alto Header

All scripts must have the following header at the top, before all imports, and is a requirement for any script to be accepted for publishing to a public site.

Any items within the pipes ( | ) need to be changed for your purposes.

# Copyright (c) 2018, Palo Alto Networks

#

# Permission to use, copy, modify, and/or distribute this software for any

# purpose with or without fee is hereby granted, provided that the above

# copyright notice and this permission notice appear in all copies.

#

# THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES

# WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF

# MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR

# ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES

# WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN

# ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF

# OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

# Author: |YOUR\_NAME| <|your\_name|@paloaltonetworks.com>

'''

Palo Alto Networks |python script name|

|Brief Description of the script|

|Instructions/Usage statement including parameters|

This software is provided without support, warranty, or guarantee.

Use at your own risk.

'''

### Code Comments

Code commenting is discussed in more detail in the [Documentation](#_Documentation) section of this document. However, no code will be accepted that is not properly documented. Basic comments as to what/why the code is doing what is are what is needed. It doesn’t have to be a dissertation, just a brief line describing the several lines below and what their purpose is.

Classes, functions and methods must all have descriptive headers, formatted as docstrings ([PEP257](http://www.python.org/dev/peps/pep-0257/#multi-line-docstrings)), underneath the declaration that include a brief synopsis of the reason/function of the class/func/method and any pertinent information such as passed in parameters and computed results.

* No inline comments, comments should be above the line they comment.
* Write in whole sentences
* Keep comments up to date. Incorrect comments are worse than no comments.
* Docstrings will start and end with ‘’’ (triple single quotes)
* Docstrings with one line can all be on the same line
* In docstrings, list each argument on a separate line
* It can be useful to precede comments with FIXME: TODO: or NOTE: if it clarifies the intent of the comment.

def my\_function():

''' A one-line docstring '''

# TODO: Write some code after this block comment

def my\_other\_function(parameter=False):

'''

A multiline docstring.

Keyword arguments:

parameter -- an example parameter (default False)

'''

### Imports

Imports should be at the top of the file, just after any module comments and docstrings. They should also come before and definitions of constants or any module globals.

* Try to use absolute imports over relative ones
* Don’t import multiple packages per line
* Multiple modules from a single package are allowed per line
* When using relative imports be explicit (with .)
* Try to use absolute imports over relative ones

Use **import x** for importing packages and modules.   
Use **from x import y** where **x** is the package prefix and **y**is the module name with no prefix.   
Use **from x import y as z** if two modules named **y** are to be imported or if y is a long name.

***Never Use from x import \****

**(**Tools like pylint can help warn you about unused imports. See [Linting](#_Linting))

Imports should be grouped in the following order with a blank line between each group of imports:

– Standard library imports

– Third party imports

– Your library imports

**NOTE: import x** lines should come before **from x import y** lines in each grouping

BAD

from pymongo import Mongo Client # 3rd party pkgs go below the standard imports

import os, sys # multiple packages

import sibling # local module without "."

from mypkg import \* # wildcards

GOOD

import os

import sys

from subprocess import Popen, PIPE  # Acceptable

from pymongo import Mongo Client

from flask import Flask, render template, redirect # Acceptable

from mypkg.sibling import example

from .sibling import example # Acceptable

### Naming Conventions

Consistency here is the key. For the most part, you can follow PEP8 conventions and be fine.

* Module names should be lowercase with underscores instead of spaces. (And should be valid module names for importing.)
* Class names should be CamelCase (uppercase letter to start with, words run together, each starting with an uppercase letter).
* Function/method names should be consistently lowercase with underscores to separate words or camelCase with the first word *not* capitalized.
* Module constants should be all uppercase.
* Variables should either be all lowercase with underscores to separate words or camelCase with the first word *not* capitalized.
* Variables also should have descriptive names. Things like i, j and x are no longer needed to save valuable memory space.

Module names in CamelCase with a main class name identical to the module name are annoying. (e.g. ConfigParser.ConfigParser, which should *always* be spelled as configobj.ConfigObj anyway.)

Variables, functions, methods and classes which *aren't* part of a public API, should begin with a single underscore.

class MyClass:

''' A purely illustrative class '''

\_property = None

def \_\_init\_\_(self, property\_value):

self.\_property = property\_value

def get\_property(self):

''' A simple getter for "property" '''

return self.\_property

@classmethod

def default(cls):

instance = MyClass(‘default value’)

return instance

### String Formatting

***Using old style string formatting –*** “My name is %s” my\_name ***– will not be accepted as newer formatting styles not only make the code more readable, they are also helpful for Internationalization.***

While the .format() method can be used, the preference is to use [f-strings](https://realpython.com/python-f-strings/#f-strings-a-new-and-improved-way-to-format-strings-in-python) and give meaningful names to each replacement field, for example:

# Define variables

osVer = '8.1.2'

fwName = 'panw\_ctl'

# Print variables out using f-strings

print(f"The NGFW is {fwName} and the OS version is {osVer}”)

### Logging

Logging may not be a requirement for simple, task oriented scripts (though it does help) but for anything that runs as a state-machine or is intended to run continuously, logging is required. Part of the Python standard library is a [logging module](https://docs.python.org/3/howto/logging.html).

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.debug('some debug message')

When logging:

* Keep messages brief
* Don’t include objects in the log message. See above about “brief”
* Choose an appropriate log level (see the above link on HOW-TO)

Expanding the logging module is allowed, and log names, sizes, rotating file handlers are all encouraged for troubleshooting purposes when the logs are used.

NOTE: Logging is ***much*** better than using a print statement to know what is going on in your code.

Here is an example that you can use in your code for logging as well as how to use it:

class MyLogFormatter(logging.Formatter):

'''

Format of the log message to contain the module name, function we are in,

the line number in the file and the thread identifier. Then append the text

sent to the logger and send back the msg to be logged

extends Class: logging.Formatter

returns: String to be logged

'''

width = 45

datefmt='%Y-%m-%d %H:%M:%S'

# Add pretermined data for the message to be logged

def format(self, record):

cpath = (

f'{record.module}:{record.funcName}:'

f'[{record.lineno}]:{record.thread}')

cpath = cpath[-self.width:].ljust(self.width)

levelName = f"[{record.levelname}]"

outputString = (f"{levelName:<10}: "

f"{self.formatTime(record, self.datefmt)} : {cpath} : "

f"{record.getMessage()}")

if record.exc\_info:

# Cache the traceback text to avoid converting it multiple times

if not record.exc\_text:

record.exc\_text = self.formatException(record.exc\_info)

if record.exc\_text:

if outputString[-1:] != "\n":

outputString = outputString + "\n"

outputString = outputString + record.exc\_text

return outputString

# Set up logging for the app, rotate through 10 files each 10MB in size

logLevel = 'DEBUG'

maxSize = 10000000

numFiles = 10

handler = RotatingFileHandler('myApp.log',maxBytes=maxSize,backupCount=numFiles)

logFormatter = MyLogFormatter()

handler.setLevel(logLevel)

handler.setFormatter(logFormatter)

logger.addHandler(handler)

logger.info(f"INIT - Application initializing with log level of {logLevel}")

logger.debug(f"App init with log values: "

f"maxBytes - {maxSize} and backupCount - {numFiles}")

### Indention and Line Length

**Indentions:**

Use 4 spaces for indentation.

In Python 3, the interpreter will ***not*** allow both tabs and spaces in the same file. Earlier versions of Python would allow mixed indentation characters as long as they all matched. Well, no more. Use the 4 spaces.

**Line Length:**

Over long lines hurt readability, but so does breaking lines purely because of setting an arbitrary maximum line length. Maximum of 79 characters for code is a good guideline, but readability must come first - Longer lines should be wrapped by surrounding expressions in parentheses (rather than using “\”). Long strings can be split across several lines using parentheses:

# This comment is exactly, not yet, need some chars, 72 characters long

(f”This is an excessively long {stringType}. So, what we can do is split it “

f” across, several {numLines}. Instead of one long one. Make sense ?”)

Comments should always be limited to 72 characters in length.

Where expressions go more than the length of the line, it is helpful to indent the lines after first one. This shows that they belong to each other:

result = (question1() + question2() + question3() +

question4() + question5() + 7)

If possible, all of the indented lines should match up. For multiline containers the following style is fine:

values = [

value1,

value2,

value3

]

Example of long code lines using the above line length suggestions:

# Check all vars and make sure we have everything for the request else

# log the message with the values

if (('hash' in afVar) and (initCall is True or jobType is 'reset') and

(indexSet is 'multiple')):

queryResponse = requests.post(url=searchURL,headers=headers,

data=json.dumps(data))

else:

app.logger.info(f"Unable to make request - afVar is {afVar}, "

f"initCall is {initCall}, jobType is {jobType}, "

f"indexSet is {indexSet})

### Line Spacing

Functions and methods should be separated with two blank lines. Class definitions with three blank lines.

Normally variable definitions don't need a blank line between them, unless they are logically distinct, and you want to separate them. Logical chunks of code within function/method definitions can have a blank line if you want to visually separate them. With these rules the blank lines provide a visual guide to the structure of the code.

Use blank lines in functions sparingly

Examples of line spacing:

'''Module docstring.'''

var1 = 'Some Value'

var2 = "Another value."

CONST1 = 'Important constant'

CONST2 = 'Another constant'

class ClassName(object):

'''Class docstring.'''

class\_attribute = 3

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

'''Method docstring.'''

pass

def another\_method(self):

'''Another docstring. '''

print 'A method.'

#

x = 3

class AnotherClass(object):

'''Class docstring. '''

...

### Whitespace

* Avoid extraneous whitespace
* Don’t use whitespace to line up assignment operatiors (=,:)
* Spaces around = for assignments
* No spaces around = for default parameter values
* Spaces around mathematical operators, but group them sensibly.
* Multiple statements on the same line are discouraged (a LOT)
* Lines at the end of files should end with a newline, just like every other line in the file.

BAD

spam ( ham[ 1 ], { eggs: 2 } ) # spaces inside brackets

if x == 4 : print x , y ; x , y = y , x # inline statements, space before commas

dict ['key'] = list [index] # space before dictionary key

y = 2 # Using spaces to line up operators

long\_variable = 3

hypot2 = x \* x + y \* y # Too much space around operators

c = (a + b) \* (a - b) # Too much space around operators

def complex(real, imag = 0.0):

return magic(r = real, i = imag) # Spaces in default values

GOOD

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

if x == 4:

print x, y

x, y = y, x

dict['key'] = list[index]

y = 2

long\_variable = 3

hypot2 = x\*x + y\*y

c = (a+b) \* (a-b)

def complex(real, imag=0.0):

return magic(r=real, i=imag)

### Single Quotes

Single quotes are easier to type and to read, but if a string contains single quotes then double-quotes are better than escaping the single quotes or wrapping the string in double single quotes – which you can do but shouldn’t.

Use single-quotes for string literals, e.g.  'my-variable', but use double-quotes for strings that are likely to contain single-quote characters as part of the string (such as error messages or any strings containing natural language), e.g. "You've got an error!"

A good rule of thumb is to use the single quotes for everything except f-strings. For the most part, any string literals that will be in the f-string will have the single quotes around it and will not mess up the f-string. There will be caveats to this, so use good judgement and try to use single quotes whenever possible.

Triple single quotes are used for docstrings, see Docstrings.

### IDEs a Auto-Formatting

Most IDE’s have an option to reformat your code, either on-demand, or during a save operation. Using this feature will generally take care of all the PEP8 style issues for you. Learn to use this feature and your code will adhere to these guidelines and be much more readable.

To help you format code correctly, it is suggested to invest your time in installing and learning an Integrated Development Environment (IDE) that will automatically help with the formatting. Some of the more popular ones are:

[PyCharm](https://www.jetbrains.com/pycharm/)

[Atom](https://atom.io/)

[Sublime Text](https://www.sublimetext.com/)

[Visual Studio Code](https://code.visualstudio.com/)

and even [Vim](https://www.vim.org/).

Notice that Notepad is not on this list.

There really is no “best IDE” as each one has its pros and cons. Just pick one and read up on how to customize it for python coding. You can also use an auto-formatter, such as [AutoPep8](https://github.com/hhatto/autopep8) or [Yapf](https://github.com/google/yapf/), in most of the IDEs out there that will automatically format your code when you save it.

### Shebang Lines

For those that have no idea what a shebang line is, it is the first line in a script that tells the kernel where to find the Python interpreter.

***#!/usr/bin/env python*** & ***#!/usr/bin/python*** are two of the most common that you will se at the top of a file.

Since this directive is ignored by Python when importing modules, it is only necessary on a file that will be executed directly. And even then, it is recommended to guide the user to use a virtualenv (more on this later) and provide them with a requirements.txt file so that they do not need to change the OS installed interpreter. This will alleviate future errors with library incompatibilities, versioning numbers and the like too.

### Linting

Optional, but highly suggested.

pylint is a tool for finding bugs and style problems in Python source code. It finds problems that are typically caught by a compiler for less dynamic languages like C and C++. Using pylint helps with catching easy-to-miss errors like typos, using-vars-before-assignment, etc. However, it isn’t perfect and you must determine from the output if you really need to worry about the messages it is generating. You can [configure pylint to either conform to your style of coding or suppress warnings if you wish.](http://pylint.pycqa.org/en/latest/user_guide/options.html)

Running pylint over your code is always a good idea. This can be done either by hand on the command line:

pylint <python file>

or by installing a plugin to your IDE that runs pylint as you type. This option, however, can get hella annoying with pop-ups if you don’t spend some time [configuring pylint.](http://pylint.pycqa.org/en/latest/user_guide/options.html)