Gaming Automation Team
Coding Guidelines

Standard coding for any piece of code that is written in Python should follow these guidelines which are following PEP8 – Style Guide for Python Code, slightly extended. PEP8 was adapted from Guido van Rossum original Python Style Guide essay. One of Guido's key insight is that **code is read much more often than it is written.**
The guidelines provided are intended to improve readability of code and make it consistent. Consistency with this style is important. Consistency within a project is more important. Consistency within one module or function is the most important.

1. Code Lay-out:

1.1. Indentation:

Use 4 spaces for indentation level. You can change your editor to insert 4 spaces when Tab key is pressed.

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

or

```
# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

When the conditional part of an if-statement is too long and require to be written across multiple line, it's worth the combination of a two character keyword (i.e. *if*), plus a single space, plus an opening bracket to create a 4-space indent for the next lines of the multiline conditional.

```
# No extra indentation.
if (this_is_one_condition and
    that_is_another_condition):
    do_something()
```

1.2. Tab and Spaces

Spaces are the preferred indentation method.
Tab should be used only to remain consistent with an existing code that is already indented with tabs.
Keep in mind that python3 disallow mixing the use of tabs and spaces for indentation.
Do not add spaces or tabs at the end of a line.
Avoid extraneous whitespaces:

- immediately inside parentheses, brackets or braces,

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

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

- between a trailing comma and a following close parenthesis,

```
foo = (0,) # Correct

foo = (0, ) # Wrong:
```

- immediately before a comma, semicolon, or colon

```
if x == 4: print x, y; x, y = y, x # Correct

if x == 4 : print x , y; x , y = y , x # Wrong
```

- immediately before the open parenthesis that starts the argument list of a function call

```
spam(1) # Correct

spam (1) # Wrong
```

- immediately before the open parenthesis that starts an indexing or slicing

```
dct['key'] = lst[index] # Correct

dct ['key'] = lst [index] # Wrong
```

- more than one space around an assignment (or other) operator to align it with another

```
# Correct:

x = 1

long_variable = 3

# Wrong:

x             = 1

long_variable = 3
```

1.3. Other recommendations:

Avoid trailing whitespaces anywhere, it can be confusing (e.g., a backslash followed by a space and a new line does not count as a line continuation maker.

Always surround these binary operations with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not)

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority. Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Compound statements (multiple statements on the same line) are generally discouraged:

```
# Correct:
if foo == 'something':
    do_something()
do_one()
do_two()
do_three()

# Wrong
if foo == 'something': do_something()
do_one(); do_two(); do_three()
```

While sometimes it is okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
# Wrong:
if foo == 'something': do_something()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()
```

```
do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

1.4. Maximum Line Length

Limit all lines to a maximum of 79 characters.
For long multiple with-statements that exceed maximum 79 characters use backslashes and continue to the next line (backslash should be in 79 characters, and do not add spaces after the backslash)

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

Break a line before binary operators to keep each operator close to its operand as shown below:

```
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - student_loan_interest)
```

1.5. Imports

Imports are always at the top of the file, just after any module comments and docstring, and before global and constants variable. Imports should usually be on separate lines:

```
import os
import sys
```

Rather then:

```
import sys, os
```

If import from a specific package it is okay to do this:

```
from subprocess import Popen, PIPE
```

They should be grouped in the following order:

- Standard library imports
- Related third party imports.
- Local application/library specific imports

2. Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up to date when the code changes!
Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower-case letter (never alter the case of identifiers!).
Ensure that your comments are clear and easily understandable to other.
Inline comments are unnecessary and in fact distracting if they state the obvious. Do not do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

2.1. Documentation Strings

Write docstrings for all public modules, functions, classes, and methods. This comment should appear after the *def* line.
For a docstring on multiple lines:

```
"""
Runs a sequence of apps from each app list. A list can contain multiple apps.
Returns:
       dict : Result of running apps.
"""
```

Note that most importantly, the """ that ends a multiline docstring should be on a line by itself.
For one liner docstrings, you can keep everything on the same line:

```
"""Return an ex-parrot."""
```

3. Naming Convention

New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

3.1. Variable Names

Make sure the given name to the variables are properly choose in order to describe the meaning that is used for.

- **Constants:** are usually defined on a module level and written in all capital letters with underscores separating words (ALL_CAPS_WITH_UNDERSCORES). Example: MAX_OVERFLOW and TOTAL.
- **Global and local variables:** use lower case words separated by underscore: this_is_a_variable. When possible, a good practice is to try to indicate the type of the variable name inside the name as follow:
    - Integer or Long: for integer variables other than "count", "i", "n", "x" that clearly indicate that is an integer type, it is good to use an "int" separated by underscore at the beginning or at the end of the variable: "int_enhanced_val", "enhanced_val_int". This will indicate that enhanced_val is an integer type.
    - Float: "float_enhanced_val". This will indicate that enhanced_val is a float type
    - Boolean: "bool_enhanced_val"
    - String: is the variable is not obvious (such as name, address, cmd, etc) that is a string add a "str" at the beginning or at the end of the variable: str_args, ret_val_str
    - List: "args_list" Adding a list at the end of args will make anyone aware that args it is a list
    - Dictionary: "args_dic"
    - Tuple: "args_tuple"
- **Using underscores:**
    - _before_a_variable : Use one leading underscore only for non-public methods and instance variables
    - __this_is_a_private_var : double _ → private variable. If a class Bar has an attribute name __bar, it cannot be accede by Bar.__bar. (an insistent user could still gain access by calling Bar._Bar__bar)
    - class_ : single underscore after should be used if using a Python keyword as a variable. For eg: return class_ is better than return klass or return clss

3.2. Classes

Class names should normally use the CapitalizeWords convention. Preferable to use one class per file.

- Classes not requiring an identifier. For eg: AppSequencer, AppListener
- Classes requiring identifiers: For eg: App_OCLBandwidthTest ("App_" indicates that this is an application plugin)

3.3. Functions and Methods Arguments

General functions: def this_is_a_function()
Always use self for the first argument to instance methods.
Always use cls for the first argument to class methods.
Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it is easier to make it public later than to make a public attribute non-public.

4. Things To Avoid

- Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero.
- Never use python key words as variable names such as: str, int, list, dict, self, __file__, __name__, etc.
- Comparisons to singletons like None should always be done with "*is*" or "*is not*", never the equality operators. Also, 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 have a type (such as a container) that could be false in a Boolean context!
- Use "*is not*" operator rather than "*not*" ... "*is*".

```
# Correct:
if foo is not None:


# Wrong
if not foo is None:
```

- Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier

```
# Correct:
def f(x): return 2*x


# Wrong
F = lambda x: 2*x
```

- When catching exceptions, mention specific exceptions whenever possible instead of using a bare *except*: clause:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

- For all try/except clauses, limit the try clause to the absolute minimum amount of code necessary, to avoid masking bugs:

```
# Correct:
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

```
# Wrong:
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

5. The Zen of Python

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!

If you really want to learn more about style guide for Python code, you can visit PEP8 and PEP257.

Happy coding!