PythonStyleGuide

Style guide for Python code contributed to Melange
Contents-Complete, Phase-Guidelines, Importance-Featured, Featured

Updated Sep 28, 2013 by nathaniel@google.com

Melange follows the Google Python Style Guide; this document describes only those respects in which Melange's style differs from what is laid out in that document.

The SoC framework, and Melange web applications built upon it, are implemented in Python (it is one of the programming language besides Java and Go which are currently supported by Google App Engine). The Google Python Style Guide and these amendments to it are a list of dos and don'ts for Python contributions to the Melange project.

The rules below are not guidelines or recommendations, but strict rules. You may not disregard the rules we list below except as approved on a need-to-use basis. But note also the advice on consistency at the end of the guide.

If you have questions about these guidelines, you can send mail to the developer mailing list. Please put **Python style:** in the subject line to make it easier to locate these discussions in the mailing list archives.

Since this style guide is documented in the Melange wiki, it can be changed via the existing documentation review process?. However, changes to the style guide should not be made lightly, since consistency is one of the key goals of this style guide, and old code will not necessarily be updated for new style guide changes.

Summary

Python Language Rules

1. Imports: *Import only modules and only by name* ***(no wildcard imports)***
2. Function Arguments: *Pass positional arguments positionally and keyword arguments by keyword.*
3. Threading: ***For now, do not use*** *(but it is on its way as part of issue 1596)*

Python Style Rules

Programs are much easier to maintain when all files have a consistent style. Here is the canonical Melange Python style.

1. Indentation: ***2*** *spaces (****no tabs****)*, **differs from PEP8**
2. Python Interpreter: *Variable, either omitted, `#!/usr/bin/python`, or `#!/usr/bin/python2.7`*
3. Copyright And License Notices: *Include in every file.*
4. Imports grouping, order and sorting: *One per line, grouped and ordered by packages, sorted alphabetically*
5. Naming: *`foo_bar.py` not `foo-bar.py`*, **some differ from PEP8**
6. Conclusion: *Look at what's around you!*

Python Language Rules

Imports

What it is: Mechanism for making use of code between modules.

Pros: Very flexible relative to other languages. Packages, modules, classes, functions, and fields all may be imported.

Cons: `from foo import *` or `import bar.baz.Spar` is very nasty and can lead to serious maintenance issues because it makes it hard to find module dependencies.

Decision: Import only modules. Use `as` to resolve name conflicts if necessary. Use `from` for all imports below top-level. Use only absolute imports; do not use relative imports. Do not import packages, classes, functions, fields, or anything that is not a module.

```
import os
import sys


from models.graphics import views as graphics_views
from models.sounds import views as sounds_views
from sound.effects import echo
...
echo.echofilter(input, output, delay=0.7, atten=4)
```

Even if the module is in the same package, do not directly import the module without the full package name. This might cause the package to be imported twice and have unintended side effects.

Function Arguments

What they are: Arguments passed to functions either by position or by name.

Pros: Passing arguments always by name can help code be self-documenting.

Cons: Passing positional arguments by name can make a call site appear as though all arguments are optional. Passing keyword arguments by position can make a call site appear as though all arguments are required.

Decision: Always pass positional arguments positionally and keyword arguments by keyword.

Threading

Melange as of January 2013 is not thread-safe. Work to support thread safety is happening as part of [https://code.google.com/p/soc/issues/detail?id=1596 ~~issue 1596~~], but for now avoid threading.

Python Style Rules

Indentation

Note that this differs from PEP8 and instead follows the original Google Python Style guide from which this style guide originated.

Indent your code blocks with 2 spaces. Never use tabs or mix tabs and spaces. In cases of implied line continuation, you should align wrapped elements either vertically, as per the examples in the line length section; or using a hanging indent of **4** spaces (not 2, so as not to be confused with an immediately-following nested block), in which case there should be no argument on the first line.

Yes:

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

# 4-space hanging indent; nothing on first line
foo = longFunctionName(
    var_one, var_two, var_three,
    var_four)
```

No:

```
# Stuff on first line forbidden
foo = longFunctionName(var_one, var_two,
    var_three, var_four)

# 2-space hanging indent forbidden
foo = longFunctionName(
  var_one, var_two, var_three,
  var_four)
```

Python Interpreter

Modules which are not intended for direct execution (and which have no effect if executed) should not include a shebang line.

Modules intended for direct execution should begin with a shebang line specifying the Python interpreter used to execute the program, most likely `#!/usr/bin/python` or `#!/usr/bin/python2.7`.

Copyright And License Notices

Every module should feature the following either at the top or just below the module's shebang and blank-comment-following-shebang lines.

```
# Copyright [current year] the Melange authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Imports grouping and order

Imports should be on separate lines, e.g.:

Yes:

```
import os
import sys
```

No:

```
import sys, os
```

Imports are always put at the top of the file, just after any module comments and __doc__ strings and before module globals and constants. Imports should be grouped with the order being most generic to least generic:

- standard library imports
- third-party library imports

- Google App Engine imports
- django framework imports
- SoC framework imports
- SoC-based module imports
- application specific imports
- imports from "tests/" - test utilities and Melange-specific test modules

Sorting should be done alphabetically.

```
import a_standard
import b_standard

import a_third_party
import b_third_party

from a_soc import f
from a_soc import g
from b_soc import d
```

Resolution of clashing names between soc-framework and soc-module modules should be done by prefixing the soc-module module with the name of the soc-module from which it was imported:

```
from soc.views.helper import url_patterns
from soc.modules.gsoc.views.helper import url_patterns as gsoc_url_patterns
```

Naming

Names to avoid

- single character names, except for counters or iterators
- dashes (`-`) in any package/module name
- `__double_leading_and_trailing_underscore__` names (reserved by Python)

Naming convention

Note that some naming conventions differ from PEP8 and instead follow the original Google Python Style guide from which this style guide originated.

- "Internal" means internal to a module or protected or private within a class.
- Prepending a single underscore (_) has some support for protecting module variables and functions (not included with `import * from`).
- Prepending a double underscore (__) to an instance variable or method effectively serves to make the variable or method private to its class (using name mangling).
- Place related classes and top-level functions together in a module. Unlike Java, there is no need to limit yourself to one class per module. However, make sure the classes and top-level functions in the same module have high cohesion.
- Use `CapWords` for class names, but `lower_with_under.py` for module names.

Naming examples

Type	**Public**	**Internal**
Packages	`lower_with_under`	
Modules	`lower_with_under`	`_lower_with_under`
Classes	`CapWords`	`_CapWords`
Exceptions	`CapWords`	
Functions	`firstLowerCapWords()`	`_firstLowerCapWords()`
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` *(protected)* or `__lower_with_under` *(private)*
Method Names[*]	`firstLowerCapWords()`	`_firstLowerCapWords()` *(protected)* or `__firstLowerCapWords()` *(private)*
Function/Method Parameters	`lower_with_under`	
Local Variables	`lower_with_under`	

[*] Consider just using direct access to public attributes in preference to getters and setters, as function calls are expensive in Python, and `property` can be used later to turn attribute access into a function call without changing the access syntax.

Conclusion

BE CONSISTENT.

If you are editing code, take a few minutes to look at the code around you and determine its style. If spaces are used around the `if` clauses, you should, too. If the comments have little boxes of stars around them, make your comments have little boxes of stars around them, too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying rather than on how you are saying it. We present 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 go to read it. Try to avoid this.
