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

Updated Jun 18, 2009 by ajaksu

The SoC framework, and Melange web applications built upon it, are implemented in Python (since that is the only programming language currently supported by Google App Engine). This style guide is 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. pychecker: Recommended
2. Module and package imports: Yes, **except for wildcard imports**
3. Packages: Yes
4. Exceptions: Yes
5. Global variables: Use sparingly
6. Nested/Local/Inner Classes and Functions: Yes
7. List Comprehensions: Yes, **if simple**
8. Default Iterators and Operators: Yes
9. Generators: Yes
10. Using apply, filter, map, reduce: Yes, **for one-liners**
11. Lambda functions: Yes, **for one-liners**
12. Default Argument Values: Yes
13. Properties: Yes
14. True/False evaluations: Yes
15. Boolean built-in type: Yes
16. String Methods: Yes
17. Lexical Scoping: Yes
18. Function and Method Decorators: Yes, **in moderation**
19. Threading: **do not use** (not available in Google App Engine)
20. Power features: **No**

Python Style Rules

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

1. Semicolons: **Avoid**
2. Line length: **80** columns maximum
3. Parentheses: **Use sparingly**
4. Indentation: **2** spaces (**no tabs**), **differs from PEP8**
5. Blank lines: **2** for functions and classes, **1** for methods
6. Whitespace: Sparingly within a line
7. Python Interpreter: the one supported by Google App Engine: `#!/usr/bin/python2.5`
8. Comments: `__doc__` strings, block, inline
   - `__doc__` Strings
   - Modules
   - Functions and Methods
   - Classes
   - Block and Inline Comments
9. Classes: Inherit from object
10. Strings: Avoid repeated use of + and +=

10. Strings: Avoid repeated use of + and +=
11. TODO style: `TODO(username):` use a consistent style
12. Imports grouping, order and sorting: One per line, grouped and ordered by packages, sorted alphabetically
13. Statements: One per line, avoid Semicolons
14. Access control: `foo` if lightweight, `GetFoo()` and `SetFoo()` otherwise
15. Naming: `foo_bar.py` not `foo-bar.py`, **some differ from PEP8**
16. Main: `if __name__ == '__main__':`
17. Conclusion: Look at what's around you!

Python Language Rules

pychecker

What it is: `pychecker` is a tool for finding bugs in Python source code. It finds problems that are typically caught by a compiler for less dynamic languages like C and C++. It is similar to lint. Because of the dynamic nature of Python, some warnings may be incorrect; however, spurious warnings should be fairly infrequent.

Pros: Catches easy-to-miss errors like typos, use-vars-before-assignment, etc.

Cons: `pychecker` isn't perfect. To take advantage of it, we'll need to sometimes: a) Write around it b) Suppress its warnings c) Improve it or d) Ignore it.

Decision: Make sure you run `pychecker` on your code.

To suppress warnings, you can set a module-level variable named `__pychecker__` to suppress appropriate warnings. For example:

```
__pychecker__ = 'no-callinit no-classattr'
```

Suppressing in this way has the advantage that we can easily search for suppressions and revisit them.

You can get a list of `pychecker` warnings by doing `pychecker --help`.

Unused argument warnings can be suppressed by using _ as the identifier for the unused argument or prefixing the argument name with `unused_`. In situations where changing the argument name is infeasible, you can mention them at the begining of the function. For example:

```
def Foo(a, unused_b, unused_c, d=None, e=None):
  d, e = (d, e)  # silence pychecker
  return a
```

Ideally, `pychecker` would be extended to ensure that such 'unused declarations' were true.

You can find more information on the PyChecker homepage.

Module and package imports

What it is: Reusability mechanism for sharing code from one module to another.

Pros: Simplest way of sharing things and the most commonly used one, too.

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

Decision: Use `import x` for importing packages and modules. Use `from x import y` only when `x` is a package and `y` is a module. This allows the importer to refer to the module without specifying the full package prefix. For example the module `sound.effects.echo` may be imported as follows:

```
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.

The following exception applies: Use of `from foo import Bar as foo_Bar` is allowed, iff Bar is a top-level singleton in foo and foo_Bar is a descriptive name of Bar in relation to foo.

For example, the following is allowed:

```
from soc.logic.models.user import logic as user_logic
...
user_logic.getForCurrentAccount()
```

This is favored over:

```
from soc.logic import models
...
import soc.logic.models.user
...
```

```
models.logic.user.logic.getForCurrentAccount()
```

Packages

What it is: Each module is imported and referred to using the full pathname location of that module.

Pros: Avoids conflicts in module names. Makes it easier to find modules.

Cons: Makes it harder to deploy code because you have to replicate the package hierarchy.

Decision: All new code should refer to modules based on their package name. Do not modify `sys.path` or `PYTHONPATH` to import modules from other directories. (There may be some situations in which path modification is still needed, but this should be avoided if at all possible.)

Imports should be as follows:

```
# Reference in code with complete name.
import soc.logging

# Reference in code with just module name.
from soc import logging
```

Some module names might be the same as common local variable names. In those cases, to avoid confusion, it sometimes makes sense to only import the containing package, and then explicitly import the module. The result looks something like this:

```
from soc import models
import soc.models.user

...
  user = models.user.User()
```

It probably best to avoid module names that lead to this confusion in the first place, but sometimes using the obvious module name (such as in the `soc.models` package example above) will result in the need for workarounds like the one above.

Exceptions

What they are: Exceptions are a means of breaking out of the normal flow of control of a code block to handle errors or other exceptional conditions.

Pros: The control flow of normal operation code is not cluttered by error-handling code. It also allows the control flow to skip multiple frames when a certain condition occurs, e.g., returning from N nested functions in one step instead of having to carry-through error codes.

Cons: May cause the control flow to be confusing. Easy to miss error cases when making library calls.

Decision: Exceptions are very Pythonic so we allow them, too, but only as long as certain conditions are met:

- Raise exceptions like this: `raise MyException("Error message")` or `raise MyException`. Do not use the two-argument form (`raise MyException, "Error message"`) or deprecated string-based exceptions (`raise "Error message"`).
- Modules or packages should define their own domain-specific base exception class, which should inherit from the built-in `Exception` class. The base exception for a module should be called `Error`.

```
class Error(Exception):
  """Base exception for all exceptions raised in module Foo."""
  pass
```

- Never use catch-all `except:` statements, or catch `Exception` or `StandardError`, unless you are re-raising the exception or in the outermost block in your thread (and printing an error message). Python is very tolerant in this regard and `except:` will really catch everything including Python syntax errors. It is easy to hide real bugs using `except:`.
- Minimize the amount of code in a `try/except` block. The larger the body of the `try`, the more likely that an exception will be raised by a line of code that you didn't expect to raise an exception. In those cases, the `try/except` block hides a real error.
- Use the `finally` clause to execute code whether or not an exception is raised in the `try` block. This is often useful for cleanup, i.e., closing a file.

Global variables

What it is: Variables that are declared at the module level.

Pros: Occasionally useful.

Cons: Has the potential to change module behavior during the import, because assignments to module-level variables are done when the module is imported.

Decision: Avoid global variables in favor of class variables. Some exceptions are:

- Default options for scripts.
- Module-level constants. For example: `PI = 3.14159`. Constants should be named using all- caps with underscores; see Naming below.
- It is sometimes useful for globals to cache values needed or returned by functions.
- If needed, globals should be made internal to the module and accessed through public module level functions; see Naming

below.

Nested Local Inner Classes and Functions

What they are: A class can be defined inside of a function or class. A function can be defined inside a function. Nested functions have read-only access to variables defined in enclosing scopes.

Pros: Allows definition of utility classes and functions that are only used inside of a very limited scope. Very ADT-y.

Cons: Instances of nested or local classes cannot be pickled.

Decision: They are fine.

List Comprehensions

What they are: List comprehensions and generator expressions provide a concise and efficient way to create lists and iterators without resorting to the use of `map()`, `filter()`, or `lambda`.

Pros: 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.

Cons: Complicated list comprehensions or generator expressions can be hard to read.

Decision: OK to use for one-liners, or when the `x for y in z` fits on one line and a conditional on another (but see the example with the very-long `x` below, which is a three-line case that is acceptable). Use loops instead when things get more complicated. This is a judgment call.

No:

```
# too complicated, use nested for loops instead
result = [(x, y) for x in range(10)
                 for y in range(5)
                   if x * y > 10]
```

Yes:

```
# this case is complicated enough *not* to use a list comprehension
result = []
for x in range(10):
  for y in range(5):
    if x * y > 10:
      result.append((x, y))

# a nice, simple one-liner
squares = [x * x for x in range(10)]

# a two-line generator (not list comprehension) case that is also OK
Eat(jelly_bean for jelly_bean in jelly_beans
    if jelly_bean.color == 'black')

# three-line case that is still debatably more readable than the equivalent loop
result = [someReallyLongWayToEatAJellyBean(jelly_bean)
          for jelly_bean in jelly_beans
          if jelly_bean != 'black']
```

Default Iterators and Operators

What they are: Container types, like dictionaries and lists, define default iterators and membership test operators (e.g `in` and `not in`).

Pros: The default iterators and operators are simple and efficient. They express the operation directly, without extra method calls. A function that uses default operators is generic. It can be used with any type that supports the operation.

Cons: You can't tell the type of objects by reading the method names (e.g. `has_key()` means a dictionary). This is also an advantage.

Decision: Use default iterators and operators for types that support them, like lists, dictionaries, and files. The built-in types define iterator methods, too. Prefer these methods to methods that return lists, except that you should not mutate a container while iterating over it.

Yes:

```
for key in adict: ...

if key not in adict: ...

if obj in alist: ...

for line in afile: ...

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

No:

```
for key in adict.keys(): ...

if not adict.has_key(key): ...

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

Generators

What they are: A generator function returns an iterator that yields a value each time it executes a `yield` statement. After it yields a value, the runtime state of the generator function is suspended until the next value is needed.

Pros: Simpler code, because the state of local variables and control flow are preserved for each call. A generator uses less memory than a function that creates an entire list of values at once.

Cons: None.

Decision: Fine. Use `"Yields:"` rather than `"Returns:"` in the `__doc__` string for generator functions.

Using apply filter map reduce

What they are: Built-in functions useful for manipulating lists. Commonly used in conjunction with `lambda` functions.

Pros: Code is compact.

Cons: Higher-order functional programming tends to be harder to understand.

Decision: Use list comprehensions when possible and limit use of these built-in functions to simple code and one-liners. In general, if such code gets anywhere longer than 60-80 characters or if it uses multi-level function calls (e.g., `map(lambda x: x[1], filter(...))`), that's a signal that you are better off writing a regular loop instead. Compare:

map/filter:

```
map(lambda x:x[1], filter(lambda x:x[2] == 5, my_list))
```

list comprehensions:

```
[x[1] for x in my_list if x[2] == 5]
```

Lambda functions

What they are: Lambdas define anonymous functions in an expression, as opposed to a statement. They are often used to define callbacks or operators for higher-order functions like `map()` and `filter()`.

Pros: Convenient.

Cons: Harder to read and debug than local functions. The lack of names means stack traces are more difficult to understand. Expressiveness is limited because the function may only contain an expression.

Decision: OK to use them for one-liners. If the code inside the `lambda` function is any longer than 60-80 characters, it's probably better to define it as a regular (nested) function.

For common operations like addition, use the functions from the operator module instead of `lambda` functions. For example, prefer `operator.add` to `lambda x, y: x + y`. (The built-in `sum()` function would be better than either alternative.)

Default Argument Values

What they are: You can specify values for variables at the end of a function's parameter list, e.g., `def foo(a, b=0):`. If `foo` is called with only one argument, `b` is set to `0`. If it is called with two arguments, `b` has the value of the second argument.

Pros: Often you have a function that uses lots of default values, but rarely you want to override the defaults. Default argument values allow an easy way to do this, without having to define lots of functions for the rare exceptions. Also, Python does not support overloaded methods/functions and default arguments are an easy way of "faking" the overloading behavior.

Cons: Default arguments are evaluated once at module load time. This may cause problems if the argument is a mutable object such as a list or a dictionary. If the function modifies the object (e.g., by appending an item to a list), the default value is modified.

Decision: OK to use with the following caveats:

- Do not use mutable objects as default values in the function or method definition.

Yes:

```
def foo(a, b=None):
  if b is None:
    b = []
```

No:

```
def foo(a, b=[]):
  ...
```

Calling code must use named values for the default args. This helps document the code somewhat and helps prevent and detect interface breakage when more arguments are added.

```
def foo(a, b=1):
  ...
```

Yes:

```
foo(1)
foo(1, b=2)
```

No:

```
foo(1, 2)
```

Properties

What they are: A way to wrap method calls for getting and setting an attribute as a standard attribute access when the computation is lightweight.

Pros: Readability is increased by eliminating explicit `get` and `set` method calls for simple attribute access. Allows calculations to be lazy. Considered the Pythonic way to maintain the interface of a class. In terms of performance, allowing properties bypasses needing trivial accessor methods when a direct variable access is reasonable while allowing future accessor methods to be possible without breaking the interface.

Cons: Properties are specified after the getter and setter methods are declared, requiring one to notice they are used for properties farther down in the code (except for read-only properties created with the `@property` decorator - see below). Must inherit from object. Can hide side-effects much like operator overloading. Can be confusing for subclasses.

Decision: Use properties in code where you are accessing or setting data where you would normally have used simple, lightweight accessor or setter methods. Read-only properties should be created with the `@property` decorator.

Inheritance with properties can be non-obvious if the property itself is not overridden. Thus one must make sure that accessor methods are called indirectly to ensure methods overridden in subclasses are called by the property (using the Template Method design pattern).

Yes:

```
import math

class Square(object):
  """Basic square with writable 'area' property, read-only 'perimeter' property.

  To use:
  >>> sq = Square(3)
  >>> sq.area
  9
```

```
>>> sq.perimeter
12
>>> sq.area = 16
>>> sq.side
4
>>> sq.perimeter
16
"""

def __init__(self, side):
  self.side = side

def _getArea(self):
  """Calculation for 'area' property"""
  return self.side ** 2

def __getArea(self):
  """Indirect accessor for 'area' property"""
  return self._getArea()

def _setArea(self, area):
  """Setter for 'area' property"""
  self.side = math.sqrt(area)

def __setArea(self, area):
  """Indirect setter for 'area' property"""
  self._setArea(area)

area = property(__getArea, __setArea,
                doc="""Get or set the area of the square""")

@property
def perimeter(self):
  return self.side * 4
```

True False evaluations

What it is: Python evaluates certain values as "false" when in a boolean context. A quick "rule of thumb" is that all "empty" values are considered "false" so `0`, `None`, `[]`, `{}`, `""` all evaluate as "false" in a boolean context.

Pros: Conditions using Python booleans are easier to read and less error-prone. In most cases, it's also faster.

Cons: May look strange to C/C++ developers.

Decision: Use the "implicit" false if at all possible, e.g., `if foo:` rather than `if foo != []:`. There are a few caveats that you should keep in mind though:

- Comparisons to singletons like `None` should always be done with `is` or `is not`. 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 be a value that's false in a boolean context!
- 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):`.
- Note that `'0'` (i.e., 0 as string) evaluates to true.

Boolean built in type

What it is: A boolean type has been available in Python since version 2.3. Two new built-in constants were added: `True` and `False`.

Pros: It makes code easier to read and is backward-compatible with the use of integers as boolean in previous versions.

Cons: None.

Decision: Use booleans.

String Methods

What they are: String objects include methods for functions that used to be in the `string` module.

Pros: No need to import the `string` module; methods work with both regular byte-strings and unicode-strings.

Cons: None.

Decision: Use them. The `string` module is deprecated in favor of string methods.

No: `words = string.split(foo, ':')`

Yes: `words = foo.split(':')`

Lexical Scoping

What it is: A nested Python function can refer to variables defined in enclosing functions, but can not assign to them. Variable bindings are resolved using lexical scoping, that is, based on the static program text. Any assignment to a name in a block will cause Python to treat all references to that name as a local variable, even if the use precedes the assignment. If a global declaration occurs, the name is treated as a global variable.

An example of the use of this feature is:

```
def getAdder(summand1):
  """getAdder returns a function that adds numbers to a given number."""
  def anAdder(summand2):
    return summand1 + summand2

  return anAdder
```

Pros: Often results in clearer, more elegant code. Especially comforting to experienced Lisp and Scheme (and Haskell and ML and ...) programmers.

Cons: None.

Decision: Okay to use.

Function and Method Decorators

What they are: Decorators for Functions and Methods, added in Python 2.4 (a.k.a "the `@` notation"). The most common decorators are `@classmethod` and `@staticmethod`, for converting ordinary methods to class or static methods. However, the decorator syntax allows for user-defined decorators as well. Specifically, for some function `myDecorator`, this:

```
class C(object):
  @myDecorator
  def aMethod(self):
    # method body ...
```

is equivalent to:

```
class C(object):
  def aMethod(self):
    # method body ...
  aMethod = myDecorator(aMethod)
```

Pros: Elegantly specifies some transformation on a method; the transformation might eliminate some repetitive code, enforce invariants, etc.

Cons: Decorators can perform arbitrary operations on a function's arguments or return values, resulting in surprising implicit behavior. Additionally, decorators execute at import time. Failures in decorator code are pretty much impossible to recover from.

Decision: Use decorators judiciously when there is a clear advantage. Decorators should follow the same import and naming guidelines as functions. Decorator `__doc__` string should clearly state that the function is a decorator. Write unit tests for decorators.

Avoid external dependencies in the decorator itself (e.g. don't rely on files, sockets, database connections, etc.), since they might not be available when the decorator runs (at import time, perhaps from `pychecker` or other tools). A decorator that is called with valid parameters should (as much as possible) be guaranteed to succeed in all cases.

Decorators are a special case of "top level code" - see Main for more discussion.

Threading

Threading is not available in Google App Engine, so do not use it in the SoC framework or Melange web applications.

Power features

What it is: Python is an extremely flexible language and gives you many fancy features such as metaclasses, access to bytecode, on-the-fly compilation, dynamic inheritance, object reparenting, import hacks, reflection, modification of system internals, etc.

Pros: These are powerful language features. They can make your code more compact.

Cons: It's very tempting to use these "cool" features when they're not absolutely necessary. It's harder to read, understand, and debug code that's using unusual features underneath. It doesn't seem that way at first (to the original author), but when revisiting the code, it tends to be more difficult than code that is longer but is straightforward.

Decision: Avoid these features in Melange code. Discuss exceptions on the developer mailing list.

Python Style Rules

Semicolons

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

Line length

Maximum line length is 80 characters.

Exception: lines importing modules may end up longer than 80 characters.

Make use of Python's implicit line joining inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression.

Yes:

```
fooBar(self, width, height, color='black', design=None, x='foo',
       emphasis=None, highlight=0)


if ((width == 0) and (height == 0)
    and (color == 'red') and (emphasis == 'strong')):
```

When a literal string won't fit on a single line, use parentheses for implicit line joining.

```
x = ('This will build a very long long '
     'long long long long long long string')
```

Make note of the indentation of the elements in the line continuation examples above; see the indentation section for explanation.

Parentheses

Use parentheses sparingly. Do not use them:

- in `return` statements
- in conditional statements unless using parentheses for implied line continuation (see above)
- around tuples, unless they are necessary syntactically or for clarity

It is, however, fine to use parentheses:

- for implied line continuation
- around sub-expressions in a larger expression (including the sub-expressions that are part of a conditional statement)

In fact, parentheses around sub-expressions is preferred to relying solely on operator precedence.

Yes:

```
if foo:

while x:

if x and y:

if not x:

if (x < 3) and (not y):

return foo

for x, y in dict.items():

x, (y, z) = funcThatReturnsNestedTuples()
```

No:

```
if (x):

while (x):

if not(x):

if ((x < 3) and (not y)):

return (foo)

for (x, y) in dict.items():

(x, (y, z)) = funcThatReturnsNestedTuples()
```

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)
```

Blank lines

Two blank lines between top-level definitions, be they function or class definitions. One blank line between method definitions and between the class line and the first method. One blank line between `__doc__` strings and the code that follows them. Use single blank lines as you judge appropriate within functions or methods. Always have a single blank line at the end of the file, this suppresses the "\ No newline at end of file" message that is generated by most diff tools.

Whitespace

No whitespace inside parentheses, brackets or braces.

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

No: `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.

Yes:

```
if x == 4:

print x, y

x, y = y, x
```

No:

```
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.

Yes: `spam(1)`

No: `spam (1)`

Yes: `dict['key'] = list[index]`

No: `dict ['key'] = list [index]`

Surround binary operators with a single space on either side for assignment (=), comparisons (==, <, >, !=, <>, <=, >=, `in`, `not in`, `is`, `is not`), and booleans (`and`, `or`, `not`). Use your better judgment for the insertion of spaces around arithmetic operators but always be consistent about whitespace on either side of a binary operator.

Yes: `x == 1`

No: x<1

Don't use spaces around the '=' sign when used to indicate a keyword argument or a default parameter value.

Yes: `def Complex(real, imag=0.0): return Magic(r=real, i=imag)`

No: `def Complex(real, imag = 0.0): return Magic(r = real, i = imag)`

Python Interpreter

Modules should begin with a "shebang" line specifying the Python interpreter used to execute the program:

```
#!/usr/bin/python2.5
```

Google App Engine requires Python 2.5.

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. (Try running pydoc on your module to see how it looks.) Our convention for __doc__ strings is to use the three double-quote format for strings. A __doc__ string should be organized as a summary line (one physical line, 80 characters or less) terminated by a period, followed by a blank line, followed by the rest of the __doc__ string starting at the same cursor position as the first quote of the first line. There are more formatting guidelines for __doc__ strings below.

Modules

Every file should contain a block comment with a copyright notice and license statement at the top of the file.

Copyright and license notices

```
#!/usr/bin/python2.5
#
# 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.
```

This should be followed by a __doc__ string describing the contents of the module. Author information is then included after this string. If an optional email is provided, then the entire author string added to the `__authors__` list needs to be an RFC 2821 compliant email address.

Module header and authors

```
"""A one line summary of the module or script, terminated by a period.

Leave one blank line.  The rest of this __doc__ string should contain an
overall description of the module or script.  Optionally, it may also
contain a brief description of exported classes and functions.

  ClassFoo: One line summary.
  functionBar(): One line summary.
"""

__authors__ = [
  # alphabetical order by last name, please
  '"John Smith" <johnsmith@example.com>',
  '"Joe Paranoid" <joeisgone@example.com>',  # email address is required
]
```

If a new contributor is not yet present in the AUTHORS file, that contributor should be added to that file as part of the first commit from that contributor.

Functions and Methods

Any function or method which is not both obvious and very short needs a __doc__ string. Additionally, any externally accessible function or method regardless of length or simplicity needs a __doc__ string. The __doc__ string should include what the function does and have detailed descriptions of the input ("`Args:`") and output ("`Returns:`", "`Raises:`", or "`Yields:`"). The __doc__ string should give enough information to write a call to the function without looking at a single line of the function's code. The __doc__ string should specify the expected types where specific types are required, and it should mention any default argument values. A "`Raises:`" section should list all exceptions that can be raised by the function. The __doc__ string for generator functions should use "`Yields:`" rather than "`Returns:`".

The function or method __doc__ string should not, generally, describe implementation details unless it's some complicated algorithm. For tricky code, block/inline comments within the code are more appropriate.

```
def fetchRows(table, keys):
  """Fetch rows from a table.

  Args:
    table: An open table.Table instance.
    keys: A sequence of strings representing the key of each table
      row to fetch.

  Returns:
    A dict mapping keys to the corresponding table row data
    fetched. Each row is represented as a tuple of strings. For
    example:

    {'Serak': ('Rigel VII', 'Preparer'),
     'Zim': ('Irk', 'Invader'),
     'Lrrr': ('Omicron Persei 8', 'Emperor')}

    If a key from the keys argument is missing from the dictionary,
    then that row was not found in the table.

  Raises:
    IOError: An error occurred accessing the table.Table object.
  """
  pass
```

Classes

Classes should have a __doc__ string below the class definition describing the class. If your class has public attributes they should be documented here in an `Attributes:` section.

```
class SampleClass(object):

  """Summary of class here.

  Longer class information....
  Longer class information....

  Attributes:
    likes_spam: A boolean indicating if we like SPAM or not.
    eggs: An integer count of the eggs we have laid.
  """

  def __init__(self, likes_spam=False):
    """Inits SampleClass with blah.

    Args:
      likes_spam: Initial indication of if the SampleClass
        instance likes SPAM or not (default is False)
    """
    self.likes_spam = likes_spam
    self.eggs = 0

  def publicMethod(self):
    """Perform operation blah."""
    pass
```

Block and Inline Comments

The final place to have comments is in tricky parts of the code. If you're going to have to explain it at the next code review, you should comment it now. Complicated operations get a few lines of comments before the operations commence. Non-obvious ones get comments at the end of the line.

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest number
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:        # true iff i is a power of 2
```

These comments should be separated from the code to improve legibility. Block comments should be preceded by a blank line. In general, end-of-line comments should be at least 2 spaces away from the code. These end-of-line comments can be lined up if there are many in a row (or in a function), but this is not required.

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 COMMENT: Now go through the b array and make sure whenever i occurs
# the next element is i+1
```

Classes

If a class inherits from no other base classes, explicitly inherit from `object`. This also applies to nested classes.

No:

```
class SampleClass:
  pass

class OuterClass:
  class InnerClass:
    pass
```

Yes:

```
class SampleClass(object):
  pass

class OuterClass(object):
  class InnerClass(object):
    pass

class ChildClass(ParentClass):
  """Explicitly inherits from another class already."""
  pass
```

Inheriting from `object` is needed to make properties work properly, and it will protect our code from one particular flavor of things that might break once we switch to Python 3000. It also defines special methods that implement the default semantics of objects including `__new__`, `__init__`, `__delattr__`, `__getattribute__`, `__setattr__`, `__hash__`, `__repr__`, and `__str__`.

Strings

Use the `%` operator for formatting strings, even when the parameters are all strings. Use your best judgement to decide between `+` and `%` though.

No:

```
x = '%s%s' % (a, b)  # use + in this case
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
```

Yes:

```
x = a + b
x = '%s, %s!' % (imperative, expletive)
x = 'name: %s; score: %d' % (name, n)
```

Avoid using the `+` and `+=` operators to accumulate a string within a loop. Since strings are immutable, this creates unnecessary temporary objects and results in quadratic rather than linear running time. Instead, add each sub-string to a `list` and `''.join()` that list after the loop terminates.

No:

```
employee_table = '<table>'
for last_name, first_name in employee_list:
  employee_table += '<tr><td>%s, %s</td></tr>' % (last_name, first_name)
employee_table += '</table>'
```

Yes:

```
items = ['<table>']
for last_name, first_name in employee_list:
  items.append('<tr><td>%s, %s</td></tr>' % (last_name, first_name))
items.append('</table>')
employee_table = ''.join(items)
```

Use `"""` for multi-line strings rather than `'''`. Note, however, that it is often cleaner to use implicit line joining since multi-line strings do not flow with the indentation of the rest of the program:

No:

```
    print """This is pretty ugly.
Don't do this.
"""
```

Yes:

```
    print ("This is much nicer.\n"
           "Do it this way.\n")
```

TODO style

Use `TODO` comments for code that is temporary, a short-term solution, or good-enough but not perfect.

`TODO`s should include the string `TODO` in all caps, followed by your `username` in parentheses: `TODO(username)`. A colon is optional. The main purpose is to have a consistent `TODO` format searchable by `username`.

```
# TODO(someuser): Use a "*" here for concatenation operator.
# TODO(anotheruser) change this to use relations.
```

If your `TODO` is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2008") or a very specific event ("Remove this code after the Foo entities in the datastore have all added the new fubar property.").

Imports grouping and order

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

Yes:

```
import sys
import os
```

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

Sorting should be done alphabetically, with the exception that all lines starting with 'from ...' should come first, an empty line, then all lines starting with 'import ...'. Standard library and third-party library imports that start with 'import ...' should be placed first and separately from each generic group:

```
import a_standard
import b_standard
import a_third_party
import b_third_party

from a_soc import f
from a_soc import g

import a_soc
import b_soc
```

Within the import/from lines the statements should be sorted alphabetically:

```
from a import f
from a import g
from a.b import h
from a.d import e

import a.b
import a.b.c
import a.d.e
```

Statements

Generally only one statement per line. However, you may put the result of a test on the same line as the test only if the entire statement fits on one line. In particular, you can never do so with `try/except` since the `try` and `except` can't both fit on the same line, and you can only do so with an `if` if there is no `else`.

Yes:

```
if foo: fuBar(foo)
```

No:

```
if foo: fuBar(foo)
else:   fuBaz(foo)

try:               fuBar(foo)
except ValueError: fuBaz(foo)

try:
  fuBar(foo)
except ValueError: fuBaz(foo)
```

Access control

If an accessor function would be trivial you should use public variables instead of accessor functions to avoid the extra cost of function calls in Python. When more functionality is added you can use property to keep the syntax consistent.

On the other hand, if access is more complex, or the cost of accessing the variable is significant, you should use function calls (following the Naming guidelines) such as `getFoo()` and `setFoo()`. If the past behavior allowed access through a property, do not bind the new accessor functions to the property. Any code still attempting to access the variable by the old method should break visibly so callers are made aware of the change in complexity.

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.

Main

Every Python source file should be importable. In Python, `pychecker`, `pydoc`, and unit tests require modules to be importable. Your code should always check `if __name__ == '__main__':` before executing your main program so that the main program is not executed when the module is imported. The capitalization of `main()` is intentionally inconsistent with the rest of the naming conventions, which would suggest `Main()`.

```
if __name__ == '__main__':
  # parse arguments
```

```
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` or `pydoced`.

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.

Comment by burc...@gmail.com, Aug 13, 2008

You've got a misprint: area = property(setArea, setArea, ... should be: area = property(getArea, setArea, ...

Comment by c...@online.de, Aug 14, 2008

It would be nice if you could highlight the rules that deviate from PEP8 since most people are using the PEP8 rules already, and most of your rules comply with PEP8. Some deviations I noticed are using 2 instead of 4 spaces for indentation of blocks, and using firstLowerCapWords for functions and methods. It would be also interesting to know the reasons why you deviate from the standard here.

Comment by c...@online.de, Aug 14, 2008

The second ("bad") example for decorators is messed up: The last line should not be indented, and it should be "aMethod" instead of "method".

Comment by andrew.f...@gmail.com, Sep 12, 2008

This style guide evolved from something written before PEP8, hence the differences.

Comment by todd.larsen, Sep 12, 2008

This style guide was derived from the Google Python Style guide, which is pre-Guido-joining-Google era. The only thing we changed from that style guide is to use Java-style method names (and functions), with an initial lowercase followed by camel-case. The Google Python Style Guide uses the same style (full camel-case) for both class names and method names (and functions).

Comment by todd.larsen, Sep 12, 2008

I have tried to correct the mistakes pointed out so far and also note the places pointed out that are different from PEP8. Thanks for the comments.

Comment by douglasjanderson, Nov 18, 2008

Found a few small errors. The closing) should be] in this listcomp:

```
result = [someReallyLongWayToEatAJellyBean(jelly_bean)
          for jelly_bean in jelly_beans
          if jelly_bean != 'black')
```

Small typo in 'iff' here:

```
if i & (i-1) == 0:        # true iff i is a power of 2
```

Very useful resource. Thanks!

Comment by todd.larsen, Nov 18, 2008

Fixed the list comprehension example. "iff" is a mathematical shorthand that means "if and only if".

Comment by alexkon, Jan 26, 2009

And what about conditional expressions which appeared in Python 2.5?

```
# conditional_expression (see the link above)
status = "OK" if (2 * 2 == 4) else "WTF?!"

# classical equivalent
if 2 * 2 == 4:
  status = "OK"
else:
  status = "WTF?!"
```

Comment by tlar...@google.com, Jan 26, 2009

I'm OK with allowing conditional expressions, iff: **they fit on a single line (no line continuation)** Google App Engine is using Python 2.5 (which I believe it is)

Comment by alturin, Jan 26, 2009

We are using conditional expressions already, and yes we are running 2.5.

Comment by todd.larsen, Jan 26, 2009

alturin: So, you are volunteering to update the style guide, then? :)

Comment by alexkon, Jan 26, 2009

By the way, do you merge updates to Google Python Style guide into this document periodically? Or are they diverging further and further with time?

Comment by todd.larsen, Jan 26, 2009

alexkon: I do not keep track of Google Python Style Guide changes. So, no, they do not get merged into this one until someone points them out.

Comment by bre...@gmail.com, Sep 9, 2009

Nooo.. why do you use just two spaces for indentation? That is so wrong! :)

Comment by project member SRabbelier, Sep 9, 2009

@brejoc: because that's Google's style, and this started out as a Google project.

Comment by the...@google.com, Mar 10, 2010

Maybe this should be replaced by or at least reference http://google-styleguide.googlecode.com/svn/trunk/pyguide.html

Comment by project member SRabbelier, Mar 11, 2010

We've diverged somewhat from the original Google styleguide, in particular when it comes to imports. But you are right, they should at least be mentioned :).