Style Guides Documentation

Release 0.1

Oscar Engineering

Jun 16, 2023

Contents

Contents:

CHAPTER 1

Python Style Guide

1.1 Goal

The goal of this style-guide is to standardize the motif of Python at Oscar in a way that reduces the risk of bugs while increasing our ability to utilize the richer features of Python when necessary. In addition, it reduces the risk exposure from the many third party libraries we use. Finally, if it is true that reading code is 90% of programming, then we should favor readability over writability.

Example: *Import packages and modules only, do not import objects or functions directly* may seem constrictive or verbose, but it allows objects in module namespaces to be mutated during execution without unexpected behavior due to object duplication. Examples of this in our current code-base are sqlalchemy-continuum and tagging.

1.2 Code of Conduct

- Be polite and respectful.
- Don't make assumptions—be clear and concise.
- Your co workers deserve a response to comments, questions or concerns, even if it is to respectfully decline or dissent.
- Boy Scout Rule: Always check a module in cleaner than you checked it out. Clean-up and compliance to the style guide is the responsibility of whomever touches the code.

1.3 Language Rules

1.3.1 Lint

Lint is important in Python due to the dynamic nature of the language. However, spurious warnings and errors do happen.

Disabling lint rules

- When disabling warnings, prefer the symbolic name to the alpha-numeric code. Pylint messages.

Bad

```
# pylint: disable=C0302
```

Good

```
# pylint: disable=max-module-lines
```

1.3.2 Imports

Import packages and modules only, do not import objects or functions directly

- Once an object has been imported into another module's namespace, a reference to that object lives in two places. Changing either reference will result in two separate objects in each namespace. See: https://docs.python.org/2/howto/doanddont.html#from-module-import-name1-name2
- Anything in a module's namespace may be accessed by importing that module, including any imported objects and modules.
- Readability can be enhanced.

Bad

```
from geordi.services.base import OscarServiceBase

class ExampleService(OscarServiceBase):
    pass
```

```
from utils.iterables import chunk

for c in chunk([1, 2, 3, 4, 5], 2):
    pass
```

Good

```
import math
math.abs(-10)
```

```
import geordi.services.base as geordi_base

class ExampleService(geordi_base.OscarServiceBase):
    pass
```

Do not use wildcard imports `from foo import *`

- This is completely invalid if it is not done at the top of a module (e.g. in a function body). See: https://docs.python.org/2/howto/doanddont.html#from-module-import
- This can clutter a namespace in a way that is completely out of the control of the importer. Imagine redefining `list` or `dict` in the imported module.

Prefer importing at the top of a module, and only at the top of a module

Do not import in function bodies

- Often this is done to circumvent circular imports. Refactor these instead.
- Rarely this may be done to avoid side effects in imported third party modules. This is an acceptable exception.
- Rarely this may be done to avoid loading modules. This may be acceptable if the system is otherwise not used or imported anywhere else. Example: debug middleware.

Only use absolute imports

- Import modules using their fully qualified name. The only downside to this is difficulty in deployment, which is solved by deploying a pex.

Deliberately order imports

- Organize imports so they are easy to find. Use three sections separated by a new line. The three sections (in order) are:
  - Standard Library Imports
  - Third Party Imports
  - Project Imports
- Within each section, imports should be sorted lexicographically, ignoring case, according to each module's full package path. Import statements of the form `import module` should always precede import statements of the form `from module import identifier`.

1.3.3 Modules

Avoid global variables

Exceptions

- Constants which should be denoted by UPPER_SNAKE_CASE.
- If absolutely necessary, internalize and provide access through functions or accessors.

Avoid excessive side-effects

- Module side-effects should be limited to mutating values in that module only.
- Side-effects at import should be as limited as possible, and should also not interact with other modules or do anything that can fail (e.g. network IO).

1.3.4 Exceptions

Use `raise MyException("message")` syntax only

- Do not use the deprecated forms:

```
raise MyException, "message"
raise "message"
```

Do not catch-all without re-raising

- Consider catching specific Exception classes in these cases, as not all Exceptions are program errors (e.g. `StopIteration`, `KeyboardInterrupt`). See: https://docs.python.org/2/library/exceptions.html#exception-hierarchy

Use naked asserts except in unit tests

- Use naked asserts in code:

```
assert isinstance(foo, Foo)
```

This plays nicely with Jedi and PyCharm for type hinting, and while it is a deviation from the above restriction, it is very common in Python.

- Use unittest.TestCase provided `assert*` methods over naked asserts in tests.

1.3.5 Nested classes and functions

Nested classes and functions are ok and useful

- Be aware that they cannot typically be serialized.

Nested functions cannot write to values in an enclosing scope

- Workarounds to do so (such as mutating a dictionary in enclosing scope) should be avoided.

1.3.6 List, generator and dict comprehensions

Keep it simple

- Complicated comprehensions are difficult to read and understand. Each component (mapping, for, filter) should fit on a single line. Do not use multiple for and filter clauses.

Prefer newer syntax, use generators where possible

- Use the newer dict comprehension syntax `{x:y for x, y in foo}` versus the old style syntax `dict(x, y for x, y in foo)`.
- Prefer generator comprehensions to list comprehension when possible.

1.3.7 Lambda

Keep it simple

- Generally lambdas should fit on a single line.

Beware the binding

- If you need to bind to a variable in an outer scope, you probably need to use the form `lambda x=x: f(x)`.
- See discussion here: http://markmail.org/message/fypalne4rp5curta or here: http://docs.python-guide.org/en/latest/writing/gotchas/

1.3.8 Conditionals

Keep it simple

- Should be simple and fit on a single line.
- Should be limited to assignment and avoid side-effects.

Prefer if/else ternary to and/or ternary

- Prefer the syntax `a = b if c else d` to `a = c and b or d`
- Using simply `or` with truth-value testing is ok, e.g. `a = a or b`

1.3.9 Default Arguments

Never use mutable default arguments

- Default argument values are global values. Mutable objects as defaults are almost never desired.

Bad

```
# The default value for a will be shared across all calls to foo.
def foo(a=[]):
    a.append(1)
```

Good

```
# Use None in these cases, and test using is None:
def foo(a=None):
    a = a or []
    a.append(1)
```

1.3.10 Properties

Use @property versus getter/setter methods

- Use `@property` to override property access.
- Do not use java-style property accessors, e.g. `get_foo` or `set_foo`.
- Do not use `@property` for attributes that require heavy computation (ie: parsing json). Let attribute access signal to a developer that accessing this value is essentially free.

Prefer instance variables to properties

- Use instance variables if there is no need to capture property access. The mantra from Java to always use accessors is not valid in Python, since property access can be overridden after the fact.

Avoid mutable class properties except where explicitly needed

- Setting properties on a class can be used as a default value for instances which is overwritten on the instance when set by an instance, but mutable values may be mutated class-wide.
- Beware of accessing class properties through an instance handle (e.g. self). Instance properties shadow class properties.
- Class properties are very nearly module globals, and should be treated as such.

1.3.11 Implicit True/False

Use the implicit True/False provided

- Prefer testing for implicit True/False versus tests such as `len(foo) == 0`.
- Implement `__len__` or `__nonzero__` when appropriate.
- See: https://docs.python.org/2/library/stdtypes.html#truth-value-testing

Use `is` for comparing against singletons

- Most notably: `is None`.
- Useful to test for sentinels.

1.3.12 Magic methods and values

Do not access magic values directly if possible

- Use `type()` to retrieve an object's class/type versus `__class__`.
- Not all classes contain a `__dict__`.
- If there is no built-in for accessing a magic value, it may be accessed directly, though care should be taken to understand the full implications (e.g. `__file__` does not exist on objects created in an interactive interpreter).

Do not call magic methods directly

- Invoke magic methods via their syntax or built-ins:
  - `__repr__`: `repr(foo)`
  - `__lt__`: `a < b`
  - `__str__`: `str(foo)`
  - `__nonzero__`: `bool(foo); if foo:`
  - `__len__`: `len(foo)`
- It may sometimes be necessary to call magic methods directly, such as `__init__` in a subclass.

1.3.13 Functional programming built-ins

Avoid map and filter when the argument would be a lambda

- If the argument to map or filter is a lambda, use a list comprehension or for loop instead.

Avoid reduce

- Use a for loop to reduce instead of the built-in function.

1.3.14 Decorators

Use sparingly

- Errors in decorators are nearly impossible to recover from.
- Decorators execute at module load time, making them a module import side-effect.
- Decorators can change anything about the decorated class/function/method.
- Decorators should be thoroughly tested and robust.
- A decorator provided with valid inputs should always succeed.

Avoid external dependencies in decorators

- Because decorators evaluate at module load time, they should not rely on the existence of external resources which may not exist.

1.3.15 Threading/Concurrency

Never rely on the atomicity of builtin types and functions

- Some access in Python is guaranteed to be synchronized or atomic, but not all. Locks and semaphores are very cheap, use them instead of relying on built-in atomicity.
- Atomicity assumptions and guarantees change from platform to platform.
- Furthermore, assume nothing in the standard library is thread-safe unless it is clearly documented as synchronized or atomic (e.g. Queue).

Share memory by communicating

- Use Queue to communicate versus locking and sharing memory. Queue is synchronized and thread-safe.
- If sharing memory is necessary, try to use threading.Condition.

Never wait on a thread during import

- Imports are guarded by an import lock (this is not the GIL, and it exists on platforms where the GIL does not) and can result in deadlock.
- See: https://docs.python.org/2/library/threading.html#importing-in-threaded-code

Beware of mixing synchronization primitives

- Tornado, gevent, Twisted, etc all provide their own synchronization primitives for use on their event-driven platforms. Using primitives from the threading module in this case will cause a deadlock in the event loop.
- Mixing synchronization primitives may be necessary in some rare situations, such as mixing threaded and asynchronous code.

Synchronization is cheap

- Locks are cheap. It's easier to remove locks later than to debug a synchronization issue.

Signals and interrupts

- Beware of the issues around sending signals to a multi-threaded Python application: http://snakesthatbite.blogspot.com/2010/09/cpython-threading-interrupting.html

1.3.16 Thrift

Avoid wrapping generated clients

- Abstractions around thrift clients are always going to leak.
- Additional functionality should be approached through free functions and collaborators.

Use binary, framed transport

- Mixing and matching transports will increase the complexity of any standardized helpers (such as client context managers) by supporting many transports.
- Mixing and matching transports will require users to find out which transport to use in each case.
- Framed transport can avoid accidental memory overflows (through setting maximum frame sizes).
- Framed transport is available for all language/framework combinations we have in use (e.g. Python+tornado, Python+threading, Go).

Do not use TSimpleServer

- TSimpleServer is single-threaded and may not respond to requests when a connection is reset.

Use `auster.server`

- `auster.server` uses Twisted for robust connection and protocol management.

1.3.17 Power Features

Python is a very rich and powerful language that attempts to toe the line between something like Ruby and something like Java. Power features should be used sparingly. It might be easier to write, but it can end up being hard to understand. Readability should always win over writability.

Metaclasses

- Avoid writing metaclasses. If you feel that you absolutely must use a metaclass, consider a class decorator (with the caveats and warnings mentioned above). If you still feel you must use a metaclass, please get a second opinion.
- Use metaclasses sparingly. abc.ABCMeta is probably the only metaclass that should ever be used directly.
- If providing a metaclass for use, consider hiding the metaclass from users and placing it on a base class which is public.

Descriptor Protocol

- Understand the implications of a non-data descriptor versus a data-descriptor before setting out.
- Descriptors are useful and powerful, but also difficult to debug. Each possible invocation should be thoroughly tested and understood. See: https://docs.python.org/2/howto/descriptor.html#invoking-descriptors

Decorators

- Decorators are tricky to get right across a variety of uses (e.g. free functions versus bound methods). Consider the wrapt package.
- Class decorators are typically easier to understand than metaclasses, and can often solve the same problems.

Monkey Patching

- Monkey Patching should be considered a last ditch-effort. Monkey patching may have unintended consequences with other modules. It is almost certainly better to fork and modify code that needs monkey patching.

Mixins

Bad

- Implementing methods through a class's public interface may decrease encapsulation. If a method can be implemented purely through a class's public interface, consider a free function, which keeps the class interface minimal.
- Mixins are harder to extend and change later, as modifying a mixin's internal interface modifies every class that uses it. Explicit composition relies only on the public interface of the composed objects, and the internals are free to change.

Good

- Because of magic methods in Python, mixins may be very beneficial to adapt a class to a specific interface that interacts with Python's syntax. Examples of this are the collections abstract base classes. Generally, a purely functional mixin which adapts one well-known interface to another well-known interface is an acceptable use of the mixin pattern.

1.4 Style Rules

1.4.1 PEP 8

- Follow the style recommendations in PEP 8.

1.4.2 Semicolons

- Do not use semicolons.

1.4.3 Line Length

- Maximum line length is 120 characters.

1.4.4 Documentation

Docstrings

- When in doubt, follow *PEP 257*
- The first line of the docstring should be a summary that fits on a single line. This may be sufficient for simple cases.
- The rest of the docstring should follow, separated from the summary by a blank line.

- Parameters, exceptions, yielded values and returned values should be type hinted in a way that Jedi and PyDev understand.

Example method docstring - note the use of type hinting, as well as descriptions:

```
def send_message(sender, recipient, message_body, priority=None):
    """Send a message to a recipient.

    :param str sender: The person sending the message
    :param str recipient: The recipient of the message
    :param str message_body: The body of the message
    :param priority: The priority of the message, can be a number 1-5
    :type priority: int or None
    :return: the message id
    :rtype: int
    :raises ValueError: if the message_body exceeds 160 characters
    :raises TypeError: if the message_body is not a basestring
    """
    pass
```

README

- Supply a README.md (markdown format) or README.rst (restructuredText format) to document any oddities, usage or gotchas. A readme is not strictly required.

Comments

- If a block of code is probably going to be discussed in a code review, explain it in a comment.
- Assume the next person knows Python. Don't describe code.
- Mark code that is less-than-desirable or needing some update with a comment using `TODO(ldap): description`. This allows the code base to be searched by TODO and filtered by user. E.g. `grep -rnH 'TODO(waldo)' *`

1.4.5 Calling functions

Readability

It is recommended to use keyword args when calling a function with three or more args.

Bad

```
foo(bar.baz(), some_function(), blah, x, y)
```

Good

```
foo(baz=bar.baz(),
    some_result=some_function(),
    blah=blah,
```

(continues on next page)

(continued from previous page)

```
    x=x,
    y=y)
```

Exceptions

- While this is generally a good idea, it is not a hard and fast rule. For example, well named args may be readable enough.

```
move_to(x_coordinate, y_coordinate, speed_per_second)
```

1.4.6 Classes

- Use Python "new-style" classes (must inherit from something, at least `object`) only. Do not use old-style classes.

1.4.7 Strings

- Use utf-8 characters directly instead of their byte representations or html entity tags. Don't forget to add '# -- *coding: utf-8* --' at the top of your file. ex: u'Put é instead of xe9'
- Use your best judgement with regard to readability when putting together strings. Simple concatenation is ok when it is very simple.
- When concatenating a large number of strings, either add strings to a list and use `''.join(...)` or use `io.BytesIO`. Strings are immutable in Python; concatenation always allocates a new string.

1.4.8 Resources

- Explicitly clean up resources such as files, transports, connections and sockets. Use `try/finally`, or use `with` and contextlib to simplify management.

1.4.9 Inversion of Control and Dependency Injection

- Inject objects and resources versus creating them. This will simplify testing (injectable mocks versus patching) and increase flexibility (injected objects and resources need only meet an interface).
- It's a trivial one-liner to add object creation to a function that accepts an object as an argument, the converse requires rewriting the function.

1.4.10 Accessors

- Accessors allow property access to be caught and modified later without changing access sites. However, if a significant amount of work (in particular, resource allocation or io) is added around property access, move that functionality to a method.

1.4.11 Constructors

- Limit the amount of "real work" done in a constructor. Dependency injection is a tremendous help here.
- If an object requires expensive initialization (e.g. the creation of a zookeeper session, communication over the network, file IO, concurrency) use a separate method to initialize the object. Also consider the thread/concurrency safety of this initializer function. Remember that an object may be created elsewhere as a side-effect of module import.

1.4.12 Naming

- Use a single underscore prefix to denote protected access.
- Use a double underscore prefix to denote private access (and effect name mangling).
- Avoid stutters: `foo.FooThing`, `bar.bar_function`.
- Avoid smurf-naming - when almost everything shares some similar prefix.

1.4.13 Main

Every "main" should be an importable Python module. Importing that module should never cause it to execute itself as a script. Python files that are scripts should use the execution guard `if __name__ == "__main__":`. Not only does this allow "mains" to be imported and used elsewhere, many tools require modules to be importable (documentation tools, test frameworks, some refactoring and analysis tools).

1.5 BUILD Guidelines

BUILD files related to the pants build system.

1.5.1 PEP 8

BUILD files are Python and should follow pep8 style. Use the build-deps goal if you want to get BUILD files right without fussing over the details.

1.5.2 Dependencies

Depend on all direct dependencies

- Do not rely on transitive dependencies to satisfy module requirements. For example, we have many wrappers around SQLAlchemy, but any target depending on these wrappers which uses SQLAlchemy should also directly depend on SQLAlchemy.
- Generally any import in any file in a target should be backed by a dependency unless it is standard library.

Organize to minimize dependency overlap

- Users of a library should not be unknowingly bundling entire frameworks that are not used. If you find yourself depending on several large, unrelated dependencies that are not strictly necessary, you might need to split your modules and targets.

1.5.3 Tests

- Tests should live next to the targets they test and be suffixed with "_test".

1.5.4 Sources

- Avoid `globs` and `rglobs`. There are exceptions (such as generated code and templates), but do not use globs as a shortcut to include files as sources.

1.6 Exceptions to the Style Guide

There are bound to be exceptions born of necessity.

1.6.1 Exceptions must be reviewed

- Any violation of best practice and style should not escape code review, and should be explicitly reviewed based on its necessity to break the rules. Style and language rules are meant to reduce gotchas and corner cases while increasing readability through consistency, but they are most effective in aggregate.

1.6.2 Exceptions should be isolated

- E.g. a common module designed to be used as a wildcard import would proliferate bad style, while a case for mixins could probably be made if they were isolated to a specific application.

Documentation lives at Read the Docs, the markup on GitHub.

CHAPTER 2

License

CHAPTER 3

Indices and tables

- genindex
- search