**Sustainable City Management Coding Standard**

Language Used:

* Python
* JavaScript
* HTML5

Major Library Used:

* D3 (Javascript)
* Django

**Coding Standard**

We would be using Django Coding Style for the projects:

* Imports and Indentation
  + Use isort to automate import sorting
  + On each line, alphabetize the items with the upper case items grouped before the lowercase items.
  + Break long lines using parentheses and indent continuation lines by 4 spaces. Include a trailing comma after the last import and put the closing parenthesis on its own line.
  + Use a single blank line between the last import and any module level code, and use two blank lines above the first function or class.
  + Use convenience imports whenever available
* Template style
  + In Django template code, put one (and only one) space between the curly brackets and the tag contents.
* View style
  + In Django views, the first parameter in a view function should be called request.
* Model style
  + Field names should be all lowercase, using underscores instead of camelCase.
  + The class Meta should appear after the fields are defined, with a single blank line separating the fields and the class definition.
  + The order of model inner classes and standard methods should be as follows (noting that these are not all required):
    - All database fields
    - Custom manager attributes
    - class Meta
    - def \_\_str\_\_()
    - def save()
    - def get\_absolute\_url()
    - Any custom method()
* Please conform to the indentation style dictated in the .editorconfig file. We recommend using a text editor with EditorConfig support to avoid indentation and whitespace issues. The Python files use 4 spaces for indentation and the HTML files use 2 spaces.
* Unless otherwise specified, follow PEP 8.
* Use flake8 to check for problems in this area. Note that our setup.cfg file contains some excluded files (deprecated modules we don’t care about cleaning up and some third-party code that Django vendors) as well as some excluded errors that we don’t consider as gross violations. Remember that PEP 8 is only a guide, so respect the style of the surrounding code as a primary goal.
* An exception to PEP 8 is our rules on line lengths. Don’t limit lines of code to 79 characters if it means the code looks significantly uglier or is harder to read. We allow up to 119 characters as this is the width of the GitHub code review; anything longer requires horizontal scrolling which makes the review more difficult. This check is included when you run flake8. Documentation, comments, and doc strings should be wrapped at 79 characters, even though [PEP 8](https://www.python.org/dev/peps/pep-0008) suggests 72.
* Use four spaces for indentation.
* Use four space hanging indentation rather than vertical alignment:
  + **raise** **AttributeError**(  
     'Here is a multiline error message '  
     'shortened for clarity.'  
    )
* Instead of:
  + **raise** **AttributeError**('Here is a multiline error message '  
     'shortened for clarity.')
* This makes better use of space and avoids having to realign strings if the length of the first line changes.
* Use single quotes for strings, or a double quote if the string contains a single quote. Don’t waste time doing unrelated refactoring of existing code to conform to this style.
* Avoid use of “we” in comments, e.g. “Loop over” rather than “We loop over”.
* Use underscores, not camelCase, for variable, function and method names (i.e. poll.get\_unique\_voters(), not poll.getUniqueVoters()).
* Use InitialCaps for class names (or for factory functions that return classes).
* In docstrings, follow the style of existing docstrings and [PEP 257](https://www.python.org/dev/peps/pep-0257).
* In tests, use [assertRaisesMessage()](https://docs.djangoproject.com/en/dev/topics/testing/tools/#django.test.SimpleTestCase.assertRaisesMessage) and [assertWarnsMessage()](https://docs.djangoproject.com/en/dev/topics/testing/tools/#django.test.SimpleTestCase.assertWarnsMessage) instead of [assertRaises()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaises) and[assertWarns()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertWarns) so you can check the exception or warning message. Use [assertRaisesRegex()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaisesRegex) and [assertWarnsRegex()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertWarnsRegex) only if you need regular expression matching.
* In test docstrings, state the expected behavior that each test demonstrates. Don’t include preambles such as “Tests that” or “Ensures that”.
* Reserve ticket references for obscure issues where the ticket has additional details that can’t be easily described in docstrings or comments. Include the ticket number at the end of a sentence like this:
  + **def** test\_foo():  
     *"""*  
     *A test docstring looks like this (#123456).*  
     *"""*  
     ...

**JavaScript Coding Style**

* Indentation Style to be confirmed in the .editorconfig file
* Naming variables, use camelCase instead of underscore\_case. Different JavaScript files sometimes use a different code style. Please try to conform to the code style of each file
* Use of JSHint code linter to check your code for bugs and style errors.
* Where possible, write code that will work even if the page structure is later changed with JavaScript.
* Django’s admin system leverages the jQuery framework to increase the capabilities of the admin interface. In conjunction, there is an emphasis on admin JavaScript performance and minimizing overall admin media file size. Serving compressed or “minified” versions of JavaScript files is considered best practice in this regard.
* To that end, patches for JavaScript files should include both the original code for future development (e.g. foo.js), and a compressed version for production use (e.g. foo.min.js). Any links to the file in the codebase should point to the compressed version.