1 Background

WikiTrust will use Python 3 for backend processing and JavaScript for the frontend development. Python is the main dynamic language for cloud engineering and data engineering processes. This style guide is a list of *dos and don'ts* for both Python and JavaScript programs.

2 Python Language Rules

2.1 Lint

Run `pylint` over your code.

2.2 Global variables

Avoid global variables.

2.2.1 Definition

Variables that are declared at the module level or as class attributes.

2.2.2 Pros

Occasionally useful.

2.2.3 Cons

Has the potential to change module behavior during the import, because assignments to global variables are done when the module is first imported.

2.2.4 Decision

Avoid global variables.

3 Python Style Rules

3.1 Semicolons

Do not terminate your lines with semicolons, and do not use semicolons to put two statements on the same line.

3.2 Parentheses

Use parentheses only when required.

It is recommended but not required around tuples. Do not use them in return statements or conditional statements unless using parentheses for implied line continuation or to indicate a tuple.

```
Yes: if foo:
         bar()
     while x:
         x = bar()
     if x and y:
         bar()
     if not x:
         bar()
     # For a 1 item tuple the ()s are more visually obvious than the comma.
     onesie = (foo,)
     return foo
     return spam, beans
     return (spam, beans)
     for (x, y) in dict.items(): ...
```

```
No:  if (x):
         bar()
     if not(x):
         bar()
     return (foo)
```

3.3 Indentation

Indent your code blocks with *4 spaces*.

Never use tabs or mix tabs and spaces. In cases of implied line continuation, you should align wrapped elements either vertically; or using a hanging indent of 4 spaces, in which case there should be nothing after the open parenthesis or bracket on the first line.

3.4 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. No blank line following a `def` line. Use single blank lines as you judge appropriate within functions or methods.

3.5 Whitespace

No whitespace inside parentheses, brackets or braces.
No whitespace before a comma, semicolon or colon. Do use whitespace after a comma, semicolon or colon, except at the end of the line.
No whitespace before the open parenthesis/bracket that starts an argument list, indexing or slicing.

3.6 Comments and Docstrings

3.6.1 Functions and Methods

A function must have a docstring, unless it meets all of the following criteria:

- not externally visible
- very short
- obvious

A docstring should give enough information to write a call to the function without reading the function's code.

Certain aspects of a function should be documented in special sections, listed below.

Args:

List each parameter by name. A description should follow the name, and be separated by a colon and a space. If the description is too long to fit on a single 80-character line, use a hanging indent of 2 or 4 spaces (be consistent with the rest of the file).

The description should include required type(s) if the code does not contain a corresponding type annotation. If a function accepts `*foo` (variable length argument lists) and/or `**bar` (arbitrary keyword arguments), they should be listed as `*foo` and `**bar`.

Returns: (or Yields: for generators)

Describe the type and semantics of the return value. If the function only returns None, this section is not required. It may also be omitted if the docstring starts with Returns or Yields (e.g. `"""Returns row from Bigtable as a tuple of strings."""`) and the opening sentence is sufficient to describe return value.

3.6.2 Block and Inline Comments

Complicated operations get a few lines of comments before the operations commence. Non-obvious ones get comments at the end of the line.

To improve legibility, these comments should start at least 2 spaces away from the code with the comment character `#`, followed by at least one space before the text of the comment itself.

4 Javascript Style Guide

Formatting

Terminology Note: *block-like construct* refers to the body of a class, function, method, or brace-delimited block of code. Note that any array or object literal may optionally be treated as if it were a block-like construct.

> Tip: Use `clang-format`. The JavaScript community has invested effort to make sure clang-format does the right thing on JavaScript files. `clang-format` has integration with several popular editors.

4.1 Braces

4.1.1 Braces are used for all control structures

Braces are required for all control structures (i.e. `if`, `else`, `for`, `do`, `while`, as well as any others), even if the body contains only a single statement. The first statement of a non-empty block must begin on its own line.

Disallowed:

```
if (someVeryLongCondition())

  doSomething();



for (let i = 0; i < foo.length; i++) bar(foo[i]);
```

Exception: A simple if statement that can fit entirely on a single line with no wrapping (and that doesn't have an else) may be kept on a single line with no braces when it improves readability. This is the only case in which a control structure may omit braces and newlines.

```
if (shortCondition()) foo();
```

4.2 Block indentation: +2 spaces

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.

4.2.1 Array literals: optionally block-like

Any array literal may optionally be formatted as if it were a “block-like construct.” For example, the following are all valid (**not** an exhaustive list):

```
const a = [
  0,
  1,
  2,
];

const b =
    [0, 1, 2];


const c = [0, 1, 2];

someMethod(foo, [
  0, 1, 2,

], bar);
```

Other combinations are allowed, particularly when emphasizing semantic groupings between elements, but should not be used only to reduce the vertical size of larger arrays.

4.2.2 Object literals: optionally block-like

Any object literal may optionally be formatted as if it were a “block-like construct.” The same examples apply as 4.2.1 Array literals: optionally block-like. For example, the following are all valid (**not** an exhaustive list):

```
const a = {
  a: 0,
  b: 1,
};

const b =
```

```
    {a: 0, b: 1};

const c = {a: 0, b: 1};

someMethod(foo, {
  a: 0, b: 1,

}, bar);
```

4.2.3 Class literals

Class literals (whether declarations or expressions) are indented as blocks. Do not add semicolons after methods, or after the closing brace of a class *declaration* (statements—such as assignments—that contain class *expressions* are still terminated with a semicolon). Use the `extends` keyword, but not the `@extends` JSDoc annotation unless the class extends a templatized type.

Example:

```
class Foo {
  constructor() {
    /** @type {number} */
    this.x = 42;
  }

  /** @return {number} */
  method() {
    return this.x;
  }
}

Foo.Empty = class {};

/** @extends {Foo<string>} */
foo.Bar = class extends Foo {
  /** @override */
  method() {
    return super.method() / 2;
  }
};

/** @interface */
class Frobnicator {
  /** @param {string} message */
  frobnicate(message) {}
```

```
}
```

4.2.4 Function expressions

When declaring an anonymous function in the list of arguments for a function call, the body of the function is indented two spaces more than the preceding indentation depth.

Example:

```
prefix.something.reallyLongFunctionName('whatever', (a1, a2) => {
  // Indent the function body +2 relative to indentation depth
  // of the 'prefix' statement one line above.
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
  }
});

some.reallyLongFunctionCall(arg1, arg2, arg3)
    .thatsWrapped()
    .then((result) => {
      // Indent the function body +2 relative to the indentation depth
      // of the '.then()' call.
      if (result) {
        result.use();
      }

    });
```