Google Style Guidelines

short summary

Christian Heusel(christian@heusel.eu)

`Source:` https://google.github.io/styleguide/pyguide.html

Python Language Rules

- Use the program `pylint` for code correction and take its output serious.
  Sometimes it makes sense to suppress its output, but be ready to justify why it is needed.
  (see https://pylint.readthedocs.io/en/latest/tutorial.html)
- Imports should be used in the following manner:
  - Use `import x` for importing packages and modules.
  - Use `from x import y` where `x` is the package prefix and `y` is the module name with no prefix.
  - Use `from x import y as z` if two modules named `y` are to be imported or if `y` is a long name.
  - Use `import y as z` only when `z` is a standard abbreviation (e.g., np for numpy).
- Use full path names for `import` statements: ~~`import jodie`~~ ⇒ `from doctor.who import jodie`
- Dont overuse exceptions and use clear error messages: `raise MyError("Alice hates Bob")`
- Dont use global variables. `Python` ≠ `C`
- If possible, use the built-in iterators and test operators ("`in`" & "`not in`").
  - Bad: (`C`-style loops)

    ```
    list = ["1","2","THREE"]
    for i in range(len(list)):
        print(list[i])
    ```

  - Good:

    ```
    list = ["1","2","THREE"]
    for str in list:
        print(str)
    ```

- If possible, use the implicit boolean value of objects;
  - Bad:

    ```
    if string == "":
    ```

    ```
    if len(list) != 0:
    ```

  - Good:

    ```
    if not string:
    if list:
    if somevalue is None:
    ```

- Do NOT use deprecated language features.
- Do not rely on the thread safety of the built in types.
- Don't use "Power Features" such as metaclasses, dynamic inheritance or object reparenting.
  They often overcomplicate simple use-cases.

Python Style Rules

Semicolons:

- Do not terminate your lines with semicolons. Do not use them to put two statements in one line.

Line length

- The maximum line length is *80 characters*.
- Except for:
  - ... long `import` statements
  - ... URLs, pathnames or long flags
  - ... module level constants
  - ... Pylint disable comments (e.g.: `# pylint: disable=invalid-name`)

Indentation

- Indent you code blocks with *4 spaces*.
- Never mix tabs and spaces.

Blank lines

- Two blank lines between top-level definitions (functions or `class` definitions).
- No blank line following a `def` line.

Whitespace

Good:

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

Bad:

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

Good:

```
if x == 1 and y != 2:
```

Bad:

```
if x==1 and y!= 2 :
```

Comments & Docstrings

- Every file should contain license boilerplate with the appropiate license for the project.
- Every function must have a docstring unless it meets one of the following criteria:
  - internal only
  - very short
  - obvious
- Docstrings should use descriptive grammar instead of imperative grammar:
  `"""Fetches rows from a Bigtable."""` ⇔ ~~`"""Fetch rows from a Bigtable."""`~~
- A docstring should declare its functionality, arguments, return value and possibly thrown exceptions.

Example:

```
def datetime_to_str(date):
    """Converts a datetime object to a string

    Args:
        date: A datetime instance

    Returns:
        A string representing the value in german time format
        example:
```

```
        "30. January 2019, 02:06 Uhr"

    Raises:
        AttributeError: wrong type of instance given to the function.
    """
    return date.strftime("%d. %B %Y, %H:%M Uhr")
```

(click here for more examples)

- Use comments when you expect that you would need to explain functionality in a code review.
- Don't write code-descriptive comments:

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

Strings

- Use the `format` or the `%`-operator for string-formatting.

  Good:

```
x = a + b
x = '%s, %s!' % (imperative, expletive)
x = '{}, {}'.format(first, second)
x = 'name: %s; score: %d' % (name, n)
x = 'name: {}; score: {}'.format(name, n)
x = f'name: {name}; score: {n}'  # Python 3.6+
```

  Bad:

```
x = '%s%s' % (a, b)  # use + in this case
x = '{}{}'.format(a, b)  # use + in this case
x = first + ', ' + second
x = 'name: ' + name + '; score: ' + str(n)
```

- Have consistent usage of the string quote character, so either use `"` or `'`.
- Use `"""` for multi-line strings.

"`TODO`"-Comments

- Use `TODO` comments for code that is temporary or obviously improvable, especially because `pylint` marks them for you.
  Example:

```
try:
    self.db.commit()
except exc.SQLAlchemyError:
    # TODO: better exception handling
    self.db.rollback()
```

- Don't use `TODO` for issue tracking.

Naming

- In general, use Snake-Case-Naming:
  `module_name`,`package_name`,`ClassName`,`method_name`,`ExceptionName`,`function_name`,

`GLOBAL_CONSTANT_NAME`,`global_var_name`,`instance_var_name`,`function_parameter_name`, `local_var_name`.
- Dont use single-character name. (Except for counters/iterators/exception-identifier)

Main

- Even if a file is meant to be an executable, it should be importable.
- The main functionality should be in the `main()` function.
- You should always check if the module is currently imported before execution:

```
def main():
    [...]

if __name__ == '__main__':
    main()
```

Parting Words

BE CONSISTENT.

If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around all their arithmetic operators, you should too. If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you're saying rather than on how you're 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. Avoid this.