**1. Naming**

Function names, variable names, and filenames should be descriptive; avoid abbreviation. In particular, do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

Always use a .py filename extension. Never use dashes.

**1.1 Names to Avoid**

* single character names, except for specifically allowed cases:
  + counters or iterators (e.g. i, j, k, v, et al.)
  + e as an exception identifier in try/except statements.
  + f as a file handle in with statements
  + private [TypeVars](https://google.github.io/styleguide/pyguide.html#typing-type-var) with no constraints (e.g. \_T, \_U, \_V)
* Please be mindful not to abuse single-character naming. Generally speaking, descriptiveness should be proportional to the name’s scope of visibility. For example, i might be a fine name for 5-line code block but within multiple nested scopes, it is likely too vague.
* dashes (-) in any package/module name
* \_\_double\_leading\_and\_trailing\_underscore\_\_ names (reserved by Python)
* offensive terms
* names that needlessly include the type of the variable (for example: id\_to\_name\_dict)

#### **1.2 Naming Conventions**

* “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 (linters will flag protected member access).
* Prepending a double underscore (\_\_ aka “dunder”) to an instance variable or method effectively makes the variable or method private to its class (using name mangling); we discourage its use as it impacts readability and testability, and isn’t *really* private. Prefer a single underscore.
* 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.
* Use CapWords for class names, but lower\_with\_under.py for module names. Although there are some old modules named CapWords.py, this is now discouraged because it’s confusing when the module happens to be named after a class. (“wait – did I write import StringIO or from StringIO import StringIO?”)
* Underscores may appear in *unittest* method names starting with test to separate logical components of the name, even if those components use CapWords. One possible pattern is test<MethodUnderTest>\_<state>; for example testPop\_EmptyStack is okay. There is no One Correct Way to name test methods.

#### **1.3 File Naming**

Python filenames must have a .py extension and must not contain dashes (-). This allows them to be imported and unittested. If you want an executable to be accessible without the extension, use a symbolic link or a simple bash wrapper containing exec "$0.py" "$@".

#### **1.4 Guidelines derived from** [**Guido**](https://en.wikipedia.org/wiki/Guido_van_Rossum)**’s Recommendations**

|  |  |  |
| --- | --- | --- |
| **Type** | **Public** | **Internal** |
| Packages | lower\_with\_under |  |
| Modules | lower\_with\_under | \_lower\_with\_under |
| Classes | CapWords | \_CapWords |
| Exceptions | CapWords |  |
| Functions | lower\_with\_under() | \_lower\_with\_under() |
| 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) |
| Method Names | lower\_with\_under() | \_lower\_with\_under() (protected) |
| Function/Method Parameters | lower\_with\_under |  |
| Local Variables | lower\_with\_under |  |

#### **1.5 Docstrings**

Python uses *docstrings* to document code. A docstring 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.) Always use the three double-quote """ format for docstrings (per [PEP 257](https://www.google.com/url?sa=D&q=http://www.python.org/dev/peps/pep-0257/)). A docstring should be organized as a summary line (one physical line not exceeding 80 characters) terminated by a period, question mark, or exclamation point. When writing more (encouraged), this must be followed by a blank line, followed by the rest of the docstring starting at the same cursor position as the first quote of the first line. There are more formatting guidelines for docstrings below.

#### **1.6 Functions and Methods**

In this section, “function” means a method, function, generator, or property.

A docstring is mandatory for every function that has one or more of the following properties:

* being part of the public API
* nontrivial size
* non-obvious logic

A docstring should give enough information to write a call to the function without reading the function’s code. The docstring should describe the function’s calling syntax and its semantics, but generally not its implementation details, unless those details are relevant to how the function is to be used. For example, a function that mutates one of its arguments as a side effect should note that in its docstring. Otherwise, subtle but important details of a function’s implementation that are not relevant to the caller are better expressed as comments alongside the code than within the function’s docstring.

The docstring may be descriptive-style ("""Fetches rows from a Bigtable.""") or imperative-style ("""Fetch rows from a Bigtable."""), but the style should be consistent within a file. The docstring for a @property data descriptor should use the same style as the docstring for an attribute or a [function argument](https://google.github.io/styleguide/pyguide.html#doc-function-args) ("""The Bigtable path.""", rather than """Returns the Bigtable path.""").

A method that overrides a method from a base class may have a simple docstring sending the reader to its overridden method’s docstring, such as """See base class.""". The rationale is that there is no need to repeat in many places documentation that is already present in the base method’s docstring. However, if the overriding method’s behavior is substantially different from the overridden method, or details need to be provided (e.g., documenting additional side effects), a docstring with at least those differences is required on the overriding method.

Certain aspects of a function should be documented in special sections, listed below. Each section begins with a heading line, which ends with a colon. All sections other than the heading should maintain a hanging indent of two or four spaces (be consistent within a file). These sections can be omitted in cases where the function’s name and signature are informative enough that it can be aptly described using a one-line docstring.

[***Args:***](https://google.github.io/styleguide/pyguide.html#doc-function-args)

List each parameter by name. A description should follow the name, and be separated by a colon followed by either a space or newline. If the description is too long to fit on a single 80-character line, use a hanging indent of 2 or 4 spaces more than the parameter name (be consistent with the rest of the docstrings in 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)***](https://google.github.io/styleguide/pyguide.html#doc-function-returns)

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 the return value. Do not imitate ‘NumPy style’ ([example](http://numpy.org/doc/stable/reference/generated/numpy.linalg.qr.html)), which frequently documents a tuple return value as if it were multiple return values with individual names (never mentioning the tuple). Instead, describe such a return value as: “Returns: A tuple (mat\_a, mat\_b), where mat\_a is …, and …”. The auxiliary names in the docstring need not necessarily correspond to any internal names used in the function body (as those are not part of the API).

[***Raises:***](https://google.github.io/styleguide/pyguide.html#doc-function-raises)

List all exceptions that are relevant to the interface followed by a description. Use a similar exception name + colon + space or newline and hanging indent style as described in *Args:*. You should not document exceptions that get raised if the API specified in the docstring is violated (because this would paradoxically make behavior under violation of the API part of the API).

**def** **fetch\_smalltable\_rows**(table\_handle: smalltable.Table,  
 keys: Sequence[Union[bytes, str]],  
 require\_all\_keys: bool **=** False,  
) **->** Mapping[bytes, tuple[str, ...]]:  
 """Fetches rows from a Smalltable.  
  
 Retrieves rows pertaining to the given keys from the Table instance  
 represented by table\_handle. String keys will be UTF-8 encoded.  
  
 Args:  
 table\_handle: An open smalltable.Table instance.  
 keys: A sequence of strings representing the key of each table  
 row to fetch. String keys will be UTF-8 encoded.  
 require\_all\_keys: If True only rows with values set for all keys will be  
 returned.  
  
 Returns:  
 A dict mapping keys to the corresponding table row data  
 fetched. Each row is represented as a tuple of strings. For  
 example:  
  
 {b'Serak': ('Rigel VII', 'Preparer'),  
 b'Zim': ('Irk', 'Invader'),  
 b'Lrrr': ('Omicron Persei 8', 'Emperor')}  
  
 Returned keys are always bytes. If a key from the keys argument is  
 missing from the dictionary, then that row was not found in the  
 table (and require\_all\_keys must have been False).  
  
 Raises:  
 IOError: An error occurred accessing the smalltable.  
 """

Similarly, this variation on Args: with a line break is also allowed:

**def** **fetch\_smalltable\_rows**(table\_handle: smalltable.Table,  
 keys: Sequence[Union[bytes, str]],  
 require\_all\_keys: bool **=** False,  
) **->** Mapping[bytes, tuple[str, ...]]:  
 """Fetches rows from a Smalltable.  
  
 Retrieves rows pertaining to the given keys from the Table instance  
 represented by table\_handle. String keys will be UTF-8 encoded.  
  
 Args:  
 table\_handle:  
 An open smalltable.Table instance.  
 keys:  
 A sequence of strings representing the key of each table row to  
 fetch. String keys will be UTF-8 encoded.  
 require\_all\_keys:  
 If True only rows with values set for all keys will be returned.  
  
 Returns:  
 A dict mapping keys to the corresponding table row data  
 fetched. Each row is represented as a tuple of strings. For  
 example:  
  
 {b'Serak': ('Rigel VII', 'Preparer'),  
 b'Zim': ('Irk', 'Invader'),  
 b'Lrrr': ('Omicron Persei 8', 'Emperor')}  
  
 Returned keys are always bytes. If a key from the keys argument is  
 missing from the dictionary, then that row was not found in the  
 table (and require\_all\_keys must have been False).  
  
 Raises:  
 IOError: An error occurred accessing the smalltable.  
 """

#### **1.7 Classes**

Classes should have a docstring below the class definition describing the class. If your class has public attributes, they should be documented here in an Attributes section and follow the same formatting as a [function’s Args](https://google.github.io/styleguide/pyguide.html#doc-function-args) section.

**class** **SampleClass**:  
 """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: bool **=** False):  
 """Inits SampleClass with blah."""  
 self.likes\_spam **=** likes\_spam  
 self.eggs **=** 0  
  
 **def** **public\_method**(self):  
 """Performs operation blah."""

All class docstrings should start with a one-line summary that describes what the class instance represents. This implies that subclasses of Exception should also describe what the exception represents, and not the context in which it might occur. The class docstring should not repeat unnecessary information, such as that the class is a class.

*# Yes:*  
*class* **CheeseShopAddress**:  
 """The address of a cheese shop.  
  
 ...  
 """  
  
class **OutOfCheeseError**(Exception):  
 """No more cheese is available."""

*# No:*  
*class* **CheeseShopAddress**:  
 """Class that describes the address of a cheese shop.  
  
 ...  
 """  
  
class **OutOfCheeseError**(Exception):  
 """Raised when no more cheese is available."""

#### **1.8 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](http://en.wikipedia.org/wiki/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 num*  
*# in the array and the array size and then do binary search to*  
*# get the exact number.*  
  
*if* i **&** (i**-**1) **==** 0: *# True if i is 0 or a power of 2.*

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.

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*

#### **1.9 Punctuation, Spelling, and Grammar**

Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

**2. Programming Conventions**

### **2.1 Indentation**

Indent your code blocks with *4 spaces*.

Never use tabs. Implied line continuation should align wrapped elements vertically or use a hanging 4-space indent. Closing (round, square or curly) brackets can be placed at the end of the expression, or on separate lines, but then should be indented the same as the line with the corresponding opening bracket.

### **2.2 Exceptions**

Exceptions are allowed but must be used carefully.

#### **2.2.1 Definition**

Exceptions are a means of breaking out of normal control flow to handle errors or other exceptional conditions.

#### **2.2.2 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 plumb error codes through.

#### **2.2.3 Cons**

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

#### **2.2.4 Decision**

Exceptions must follow certain conditions:

* Make use of built-in exception classes when it makes sense. For example, raise a ValueError to indicate a programming mistake like a violated precondition (such as if you were passed a negative number but required a positive one). Do not use assert statements for validating argument values of a public API. assert is used to ensure internal correctness, not to enforce correct usage nor to indicate that some unexpected event occurred. If an exception is desired in the latter cases, use a raise statement. FLibraries or packages may define their own exceptions. When doing so they must inherit from an existing exception class. Exception names should end in Error and should not introduce repetition (foo.FooError).
* Never use catch-all except: statements, or catch Exception or StandardError, unless you are
  + re-raising the exception, or
  + creating an isolation point in the program where exceptions are not propagated but are recorded and suppressed instead, such as protecting a thread from crashing by guarding its outermost block.
* Python is very tolerant in this regard and except: will really catch everything including misspelled names, sys.exit() calls, Ctrl+C interrupts, unittest failures and all kinds of other exceptions that you simply don’t want to catch.
* 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.

**3. Citations**

1. <https://google.github.io/styleguide/>