The Project Exist/Watercress Studios Somewhat Decent Ren'Py/Pygame/Python-Specific Style Guide for Modules, Libraries, Functions, Screens, and Other Incredibly Awkward Objects (IAOs)
~ NintendoToad/Nolan (Version 1.2)

(Please read the entire document before asking questions. Google's style guide is far shorter than this.)

General Rules – *These do not apply to specific objects.*

```
            # Show second image if available
            # Clear text because correct image
            if(os.path.isfile(loc2)):
                setimage = self.img2
                setloc = loc2
                text = ""
            # First image otherwise
            # Generate text if wrong image
            else:
                setimage = self.img1
                setloc = loc1
                text = self.pretext + " ".join(self.name) + ": " + self.img2
```

1. Examples of implementations may be found in the examples folder included with this style guide.
    1. An indentation of four SPACE characters is standard. Please do not use tab '\t' characters. Each operating system/text editor/code monkey handles those differently.
    2. **Reminder**: Here's a good rule of thumb. If you do anything that is genius, clever, weird, obfuscated, dumb, memory-intensive, confusing, or at the point that you don't know why it works... comment!

```
#######
# File name: doublespeak.rpy
#
# Description: Allows multiple Characters to speak at once
#
# Original author: delta, Shiz
# Modifications: Nolan/NintendoToad
#
# Type: Library, Screen
#
# Usage:
#     doublespeak Character() Character() String
# ##or##
#     doublespeak Character() Character() String String
#######
```

2. Absolutely every file must have a file header.
    1. **Exception**: If is a script that uses a Character-Support file, ignore the remainder of the file header guidelines guidelines. The used Character-Support files, possible previous script files, and possible next script files are sufficient.
    2. The file header must include the intended file name.
    3. A description of the file's general purpose is to be written by either a coder or writer. It must summarize the purpose of most, if not all, objects in the file. Exceptions: helper functions.
    4. Except for the first and last lines of the header, each line should begin with a pound symbol '#' and a SPACE character.
    5. No line within the file header is to exceed 80 characters.
    6. The original author's legal first name (if public), or at the very least TeamSpeak/reddit handle, are to be included.
    7. A module that implements two good ideas is still a bad idea. Please use two files!
    8. Usage should also be specified.
    9. A file should specify one or two (no more; either leave off third+ types or split your code into multiple files) of the following types: [Library], [Screen], [Character-Support]

```
#####
# Function name: doublespeak_parse()
#
# Descripiton: Parses a lexer, separating it into parts.
#
# Parameters:
# lexer - arguments from the renpy statement
#
# Returns: The requested characters and messages as a dict
#####
```

3. Every function must have a function header, even if you are borrowing the function. Every function you “borrowed” from somewhere else must specify the source.
    1. **Exception**: Strictly, neither inits for classes nor callbacks are truly functions. If it makes more sense to only header these with the callback's purpose, do so.
    2. **Exception**: A function that only serves as a predict function (such as a function that is only a return statement) need not be commented. Ditto for functions that serve only to return magic numbers based on other conditions. (See the dyslexic_support.rpy example.)
    3. **Reminder**: A Character-Support function (these normally return Strings based on Character points) should specify the script in which it is called. Additional notes are redundant. They should check store.* rather than take an argument.
    4. The header description should specify what the function's purpose is – especially if it is a helper function.
    5. The header must specify arguments taken – even if there are none, and return values – even if there are none.
    6. It is good style to ensure that every function returns, even if there is no returned value at all. This prevents us from believing that functions within functions are a good idea. **Exception**: immediately adding a function to an array of callback functions serves as a return statement for this purpose.
    7. **Reminder:** Generally speaking, your function is not self-documenting. As soon as you call

something along the lines of "renpy.*", your code probably just became not self-documenting.

8. **Exception**: In the case of functions borrowed from the Ren'Py library, specifying the file name is sufficient. If the function is modified in any way, a full header is required.
9. **Reminder**: If, due to obfuscation, inaccessibility of documentation, or some other reason, the code does not inherently make sense while doing what you think it does, you do not need to comment borrowed functions asides from the header.

```
    import collections #OrderedDict()
    from get_image_size import get_image_size, UnknownImageFormat, BadFile
```

4. Every import statement is to be commented.
    1. The members used from the import statement need be listed. Additional data is redundant.
    2. From statements are generally self-documenting and need not be commented.

```
        # Other screens should behave differently or not appear if we are in
        # countdown.
    $ in_countdown = False
```

5. Every global/Ren'Py variable is to be commented.
    1. If the comment is above the variable in question, the comment should be indented. A comment should not be added to the right of a line if the line would be – or already is – 80 characters.
    2. **Exception/Reminder**: Styles tend to be global. You only need to comment the style.create function of a style declaration, assuming that the children of the style are directly below the style.create call. (If they don't, you are insane.)
    3. **Reminder**: This includes characters. Character() objects are only global once they are declared. Therefore, characters may be unique to a route, but should still be commented.

```
# Shows the bar as it depletes, and the timer. Recolors if necessary. Forces a
# jump if necessary.
screen countdown:
    #...
```

6. A Ren'Py block (especially screens that took more than 10 minutes to write) should really have a header as well.
    1. Only specifying its purpose is necessary for a header; there should be additional comments in the screen itself.

```
# declare styles for red/white countdown bar
init -1:
    $ cdw_color = (255, 0, 0, 255) # the color red
```

7. Every Ren'Py init block is to be commented.
    1. **Reminder**: these are in the form of “init int” without 'python'. Pythonic blocks do not need a comment.
    2. **Exception**: init blocks holding only global variables

```
screen countdown:
    tag countdown_tag
        # move timer every 0.1 seconds
    timer 0.1 repeat True action If(time > 0,
        true=SetVariable('time', time - 0.1),
        false=[SetVariable('in_countdown', False),
            Hide('countdown'),
            Jump(timer_jump)])
    if time > 3:
        # white bar for lots of time left
        # -- code truncated --
    if time < 0:
        # redundantly set in_countdown to False just in case
        $ in_countdown = False
```

8. Every Ren'Py sub-block is to be commented.
    1. Most members of labels, screens, etc. Comments should probably be under the more “obvious” objects, such as tag members.
    2. **Only if**: … if you are drawing a UI element that is a member of a separate object, or are declaring a label that is to be used as a function. (Well, this should be most of the time.) In the latter case, give it a header as you would a function. *Why is this a rule?* Most members of the ui class do not have an inherently obvious purpose. And if you are using a Ren'Py label as a function, there had best be a reason why.

9. Pythonic sub-blocks generally should be commented, but this is not as necessary as the above rules.

10. Every image should be self documenting by the name alone.
    1. The name of the image object should still be less than 80 characters. If it is longer, please get help from another coder, or your assigned writer..

11. Having a copypaste text file for each header type is a good idea.

Label Rules *– These apply to Ren'Py label definitions.*

1. All non-story labels need a label header.
    1. Again, labels that serve as functions rather than an actual label should be headered as a function rather than a label. The main difference is that most labels are not expected to return an actual value, or serve as a menu.
    2. Story labels require no header, but should have a patently-obvious name.

2. If a label's purpose cannot be mostly garnered from its name, something is wrong.

3. There are no other rules governing Labels.

Screen Rules *– These apply to Ren'Py screen definitions.*

1. If you think you need to define a screen specifically for your assigned route, then one of the following are true: (A): **You are doing something wrong.** … or (B): **You are doing something incredibly right – so right, that other coders should consider using this new screen for their routes.**
    1. Please contact a coding coordinator if you feel the need to make a new screen.

Character-Support Library Rules *– Collections of functions for routes*

1. All file names should be in the form of CharacterName_##.
    1. NolanLemahn_01.rpy for instance.

2. Most functions need not be commented, should only check a variable in store.*, and return a String.
    1. You will still be SuperSorry™ if you need to make edits to these functions later, but have a had time finding where the function's name is, or where it is used.

Library Rules *– When you actually add functionality*

1. Each library should either implement only one additional functionality, or should be a collection of relatable functions.
    1. For usage, the method of invoking the functionality should be specified.
    2. Contact a coding coordinator before writing a library that implements more than one significant functionality.