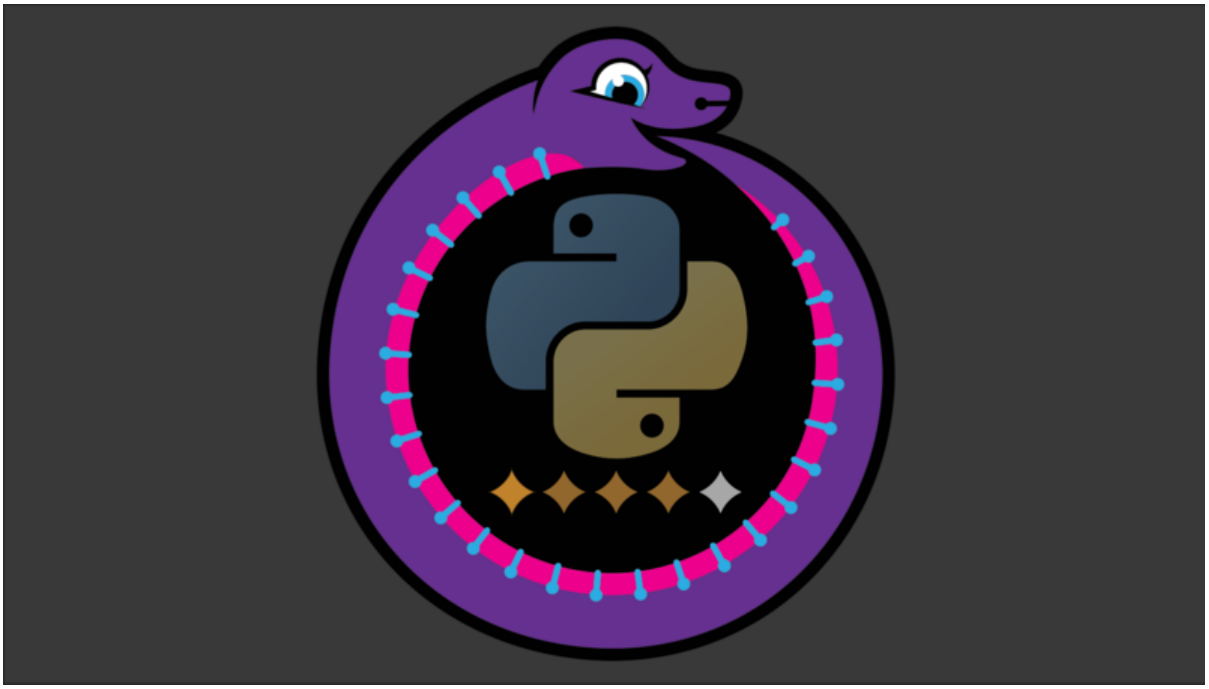
Improve Your Code with Pylint and Black

Created by Kattni Rembor



https://learn.adafruit.com/improve-your-code-with-pylint

Last updated on 2023-08-29 04:14:07 PM EDT

Table of Contents

Overview

Pylint is a tool for linting code. Linting is the process of checking your code for basic stylistic mistakes. It doesn't verify that your code works, it simply checks that it looks as good as possible and is readable by others.

Black is a really useful code formatting tool maintained by the Python Software Foundation. It reformats files to improve readability.

In order to have the best quality code, it's good to follow some rules that are established for Python coding. Many of these rules are in PEP-8 (). Some are simply good coding practice, such as using explicit names and including documentation for class functions, while others are syntax related. There are a significant number of Pylint checks (). Since it's hard to remember all of them, we use Pylint.

This guide covers getting started with Pylint. You'll learn how to install Pylint, how to configure it to check for only what you want it to, and how to run it. Then you'll see it in action with an example program containing some common errors, and you'll learn how to read the output and fix the errors in your code. This guide also covers getting started with Black. You'll also learn how to verify your code using the Black formatter, and apply Black formatting to your code.

This guide assumes you have installed and worked with Python 3 and pip. If you do not have both installed, please take the time to do so before continuing.

Let's get linting and formatting!

Check your code with pre-commit

Maintaining code is quite a challenge over the long-term. Over the lifetime of a library, many people read and edit it. Each person has a different background in coding and also a different goal in mind. These variations can lead to inconsistencies throughout the code that makes it a bit harder for the next person to understand. In CircuitPython libraries, we use tools like Pylint and Black to ensure consistency in new code.

As we've added more automated checks, we've changed to a system called pre-commit to manage the checks overall. Once installed properly, you can run pre-commit locally, before committing new code into Git. It also runs remotely on GitHub when a pull request has been proposed. pre-commit is set up to remotely run on all existing libraries. It will automatically run remotely on a new library thanks to cookiecutter.

However, it won't run locally unless you install it into your local directory. We highly recommend doing this because it will both check and fix your code locally.

One-time initial install of pre-commit

If you've never used `pre-commit` on your computer before, you'll need to install it gl obally (there is a second "install" for each repository.) The easiest way to install it is with `pip`.

```
pip install pre-commit
```

Workaround for pre-commit issues on Ubuntu and Debian

In ubuntu 22.04 or the analogous Debian release, you may see the error "expected environment for python to be healthy immediately after install" when trying to use `pre-commit`. To fix this, add this line to your `.bashrc` or `.bash_aliases` file, or other shell startup file. Restart your shell as necessary to pick up this setting.

```
export SETUPTOOLS_USE_DISTUTILS=stdlib
```

This `export` must be present before `pre-commit` sets up its `virtualenv` environment, which happens the first time you do `pre-commit run` or you try to push a commit. If the `virtualenv` is already set up, do `pre-commit clean`, which removes the existing `virtualenv`.

See the instructions from the pre-commit project for installation for alternative ways of installing `pre-commit`.

Per-repository installation

For every new repository, you'll need to perform an pre-commit installation. This installs the specific versions of checks that the repository specifies. From within the repository do:

```
pre-commit install
```

After running this command, pre-commit will automatically run when you do `git commit`.

However, if you don't do this, you can still run pre-commit manually.

Running `pre-commit`

`pre-commit` will run each check every commit for all of the modified files and either pass or fail. Most checks that fail will also modify the source file to make it pass (like removing extra spaces). Once that happens, you'll see newly modified files in `git status`. `git add` them and then try the commit again.

Manually

You can run the pre-commit checks on every file whenever you like with:

```
pre-commit run --all-files
```

More Info

For more info on `pre-commit` see pre-commit.com.

Install Pylint

If you are using pre-commit, you do not need to follow these steps. pre-commit installs Pylint for you. We highly recommend using pre-commit. For more information, check out this page: https://learn.adafruit.com/creating-and-sharing-a-circuitpython-library/check-your-code

Installing Pylint is quite easy. Simply run the install command and it will install Pylint and all of the necessary dependencies.

Install Pylint on MacOS

Open a terminal program, and verify you have the necessary tools installed:

- Type `python --version`. If it returns Python 3.x.x, type `pip --version` to verify you have `pip` installed.
- If `--version` returns `Python 2.x.x`, type `python3 --version` to verify you have Python 3 installed. Then, type `pip3 --version` to verify you have it installed.

To install Pylint, run the following command:

- `pip3 install pylint`
- If `python --version` returned `3.x.x`, run: `pip install pylint`

Install Pylint on Windows

Open the Command Prompt application. (You can search for "command" to find it.) Consider pinning it to the task bar as you'll be using it often!

Open Command Prompt, and verify you have the necessary tools installed:

- Type `python --version` to verify you have `3.x.x` installed. Then type `pip --version` to verify you have `pip` installed.

If you find you do not have Python installed, you can find various versions on the official Python.Org website at https://www.python.org/downloads/windows/

To install Pylint, run the following command:

- `pip install pylint`

Now that you have Pylint installed, you'll want to know how to configure what it checks for. If this is the first time you've installed Pylint, you can skip the next section on this page.

Upgrading Pylint

If you've had Pylint installed for a while, you may want to consider upgrading it. First, to check the version of Pylint you're running, run the following command:

- `pylint --version`

To upgrade to the absolute latest version, run the following command:

- `pip install pylint --upgrade`

If you want to install a specific version, run the following command with the version you wish to install. For example, to install version 2.7.1, run the following:

- `pip install --force-reinstall pylint==2.7.1`

That's all there is to upgrading Pylint. Now, it's time to learn about configuration.

.pylintrc

Even if you are using pre-commit, you MUST have our .pylintrc file in place. We highly recommend using pre-commit. For more information, check out this page: https://learn.adafruit.com/creating-and-sharing-a-circuitpython-library/check-your-code

Pylint checks for many things. Some of them may be more important than others for your application. You can create a file that allows you to tell Pylint to ignore certain checks. This file is called .pylintrc ().

Adafruit uses two different .pylintrc files: one for library code (), and one for Learn guide code examples (). The library code .pylintrc is much stricter than the one for code examples. This was done to ensure readability over strict compliance.

For this guide, you're going to use the Learn code .pylintrc. First, download the file from here () and save it to the folder that will contain your example. The easiest way to ensure that Pylint uses the desired .pylintrc file is to place it in the same working directory as your code.

Open the file into an editor to take a look at the configuration options. The part you'll be most likely to configure is the second section under `[MESSAGES CONTROL]`, beginning with `# Disable the message, report, category or checker with the given id(s).` This section allows you to disable specific checks. As you can see, for Learn examples a significant number of them have been disabled:

```
disable=
    too-many-instance-attributes,
    len-as-condition,
    too-few-public-methods,
    anomalous-backslash-in-string,
    no-else-return,
    simplifiable-if-statement,
    too-many-arguments,
    duplicate-code,
    no-name-in-module,
    no-member,
    print-statement,
    parameter-unpacking,
    unpacking-in-except,
    old-raise-syntax,
    backtick,
    long-suffix,
    old-ne-operator,
    old-octal-literal,
    import-star-module-level,
    raw-checker-failed,
    bad-inline-option,
    locally-disabled,
    locally-enabled,
    file-ignored,
    suppressed-message,
    useless-suppression,
    deprecated-pragma,
    apply-builtin,
    basestring-builtin,
    buffer-builtin,
    cmp-builtin,
    coerce-builtin,
    execfile-builtin,
    file-builtin,
    long-builtin,
    raw_input-builtin,
    reduce-builtin,
    standarderror-builtin,
    unicode-builtin,
    xrange-builtin,
    coerce-method,
    delslice-method,
    getslice-method,
```

```
    setslice-method,
    no-absolute-import,
    old-division,
    dict-iter-method,
    dict-view-method,
    next-method-called,
    metaclass-assignment,
    indexing-exception,
    raising-string,
    reload-builtin,
    oct-method,
    hex-method,
    nonzero-method,
    cmp-method,
    input-builtin,
    round-builtin,
    intern-builtin,
    unichr-builtin,
    map-builtin-not-iterating,
    zip-builtin-not-iterating,
    range-builtin-not-iterating,
    filter-builtin-not-iterating,
    using-cmp-argument,
    eq-without-hash,
    div-method,
    idiv-method,
    rdiv-method,
    exception-message-attribute,
    invalid-str-codec,
    sys-max-int,
    bad-python3-import,
    deprecated-string-function,
    deprecated-str-translate-call,
    import-error,
    missing-docstring,
    invalid-name,
    bad-whitespace,
    consider-using-enumerate
```

Note: In the file you downloaded, the `disable=` checks are all on a single line. The list provided above is formatted onto separate lines to make it easier to read. Both are valid, functional formats for a .pylintrc `disable=` configuration.

If you'd like to know what each of these does, check out the Pylint documentation ().

The documentation also includes the rest of the available checks. If you'd like to configure your own .pylintrc, you would add to or remove from the list any of the checks you'd rather have included or excluded. However, you will want to use the two provided as-is when working with example code intended for Adafruit libraries or Learn guides.

For this guide, the important checks are the ones that haven't been disabled above. Those are the rules that you still have to follow for the Pylint checks to pass. Before continuing, make sure you saved the .pylintrc you downloaded () to same directory as your code. Next you'll learn how to run Pylint on your code.

Run Pylint

If you are using pre-commit, you do not need to follow these steps. pre-commit runs Pylint for you. We highly recommend using pre-commit. For more information, check out this page: https://learn.adafruit.com/creating-and-sharing-a-circuitpython-library/check-your-code

Now that you've installed Pylint and downloaded the .pylintrc configuration file, you're ready to start linting. First thing we need is an example to check.

Download pylint_example.py using the "pylint_example.py" link below. Then, place the file in the same location as your recently downloaded .pylintrc file.

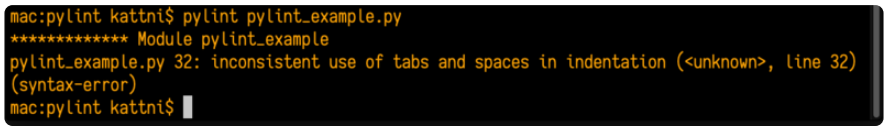
```
import board
import digitalio
import adafruit_lis3dh
import touchio
import time
import neopixel
import adafruit_thermistor

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness=0.2)

i2c = board.I2C()
int1 = digitalio.DigitalInOut(board.ACCELEROMETER_UNTERRUPT)
lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, int1=int1)

circuit_playground_temperature = adafruit_thermistor.Thermistor(board.TEMPERATURE,
10000, 10000, 25, 3950)

touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

button_A = digitalio.DigitalInOut(board.BUTTON_A)
button_A.direction = digitalio.Direction.INPUT
button_A.pull = digitalio.Pull.DOWN

while True:
    x, y, z = lis3dh.acceleration

    if button_A.value:
        led.value = True
    else:
        led.value = False

    print("Temperature:", circuit_playground_temperature.temperature)
    print("Acceleration:", x, y, z)

    if touch_A1.value:
        pixels.fill((255, 0, 0))
    if touch_A2.value:
        pixels.fill((0, 0, 255))
    else:
        pixels.fill((0, 0, 0))
```

```
    time.sleep(0.01)
```

Pylint looks in a series of locations () in order to find the configuration file. The first place it looks is the current working directory. This is the easiest way to ensure you're using the right configuration file.

Return to your terminal program or command line. From the command line, navigate to the folder containing pylint_example.py and .pylintrc. Then, run the following command:

```
pylint pylint_example.py
```



Alright! Your first error! Consider it a badge of honor. And don't worry! The next section will walk through how to read the Pylint output with a series of common errors. Time to start linting!

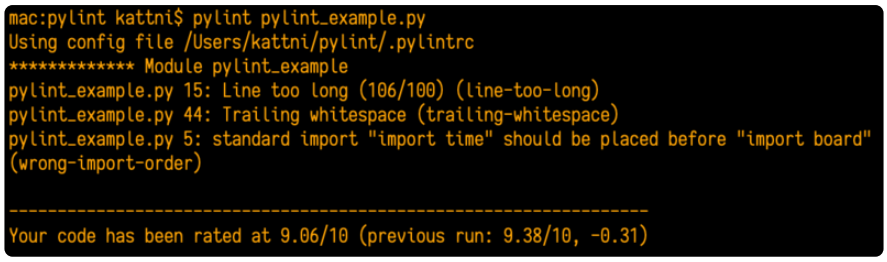
Pylint Errors

If you are using pre-commit, the Pylint errors will be the same as if using Pylint alone. We highly recommend using pre-commit. For more information, check out this page: https://learn.adafruit.com/creating-and-sharing-a-circuitpython-library/check-your-code

Pylint has a significant number of checks it can perform. Many are disabled for the purposes of Adafruit Learn guide code. The important checks for this guide are the ones Adafruit does not disable. You've already downloaded the Pylint configuration file and the example. You've completed the first run of Pylint and received an error. Now you'll learn how to read the Pylint output and resolve a few common errors.

Pylint Output

Pylint has a standard format to its output. When there is a syntax error, it will not show a code rating.



Depending on your version of Pylint, you may or may not see the first line informing you what .pylintrc configuration file you're using. This allows you to verify that you're using the one you intended to. Pylint will first look in the current working directory for a .pylintrc to use. That is why it's easiest to copy the file into your working directory when running Pylint.

The next line, `************ Module pylint_example`, begins with a series of asterisks and then identifies which file (`Module`) you're linting. In this case, you're linting pylint_example.py, so the line ends with `pylint_example`. If you were linting more than one file at once and more than one contained errors, there would be multiple instances of this line followed by any errors found in each file.

The next section is the most important part: the error list. This is where Pylint lists all the checks that have failed. The errors all follow a standard format. This is the first of three errors:

- `pylint_example.py 15: Line too long (120/100) (line-too-long)`.

The error lines are composed of the same four key parts:

1. The file name: `pylint_example.py`
2. The line containing the error: `15:`
3. The details of the error: `Line too long`
4. The Pylint check name: `(line-too-long)`

Note that the details and the Pylint check name will not always be the same (as in the third error shown in the image above). Some error details tell you the issue, but not necessarily how to resolve it, e.g. the first two errors shown in the image above. Others explicitly tell you how to resolve the error, e.g. the third error shown in the image above. The next section will explain how to work through each type of error.

Once syntax errors have been resolved, your code will receive a rating out of 10. The error is followed by a line of `-` dashes to separate the rating from the errors. The last line is the code rating:

- `Your code has been rated at 9.06/10`

Each error is worth a negative number of "points" which are added up and subtracted from 10 to provide the rating. Sometimes the code rating will be negative. Don't worry! It's happened to everyone. Start at the beginning and work your way through it regardless of the rating.

Each time you run Pylint after the first time, it compares the current score to the previous score. Sometimes your score will go up, sometimes your score will go down. Each successive run's score line will look something like this:

- `Your code has been rated at 9.69/10 (previous run: 9.06/10, +0.62)`

The score is irrelevant to the actual process of linting your code. It is not the important part of the Pylint output. The important part is the list of errors for you to resolve. Focus on that list and you'll soon have your code linted perfectly.

Working Through Pylint Errors

Now that you know how to read the output, it's finally time to start working through the Pylint errors.

Be aware that Pylint may not show you all the errors at once. It will cascade fail by returning one error and then, once that error is resolved, returning another series of errors. This is why it's important to run Pylint locally, otherwise you'll be waiting for the remote linter to run each time before you can start working on the next set of errors.

Let's get started!

Error: `inconsistent use of tabs and spaces`

```
mac:pylint kattni$ pylint pylint_example.py
************* Module pylint_example
pylint_example.py 32: inconsistent use of tabs and spaces in indentation (<unknown>, line
32) (syntax-error)
```

The first error is a syntax error on line 32: `inconsistent use of tabs and spaces in indentation (, line 32) (syntax-error)`. The example has a mix of tabs and spaces in the same `if`/`else` statement. Knowing spaces were used for the majority of the code, it's clear here that a tab managed to sneak in. There is a tab before the `else` on line 32. Replace it with four spaces to match the rest of the code.

Errors: `line too long`, `trailing whitespace`, `wrong import order`

```
mac:pylint kattni$ pylint pylint_example.py
************ Module pylint_example
pylint_example.py 15: Line too long (106/100) (line-too-long)
pylint_example.py 44: Trailing whitespace (trailing-whitespace)
pylint_example.py 5: standard import "import time" should be placed before "import board"
(wrong-import-order)

--------------------------------
Your code has been rated at 9.06/10
```

Now that you've resolved the syntax error, Pylint has found three linting errors.

You'll start with the first error:

- `pylint_example.py 15: Line too long (106/100) (line-too-long)`

Line 15 is too long. In this case, there doesn't need to be much more explanation, so Pylint keeps it simple. The fix is easy enough. After the first `10000,` separate the rest onto a new line, as follows:

```
circuit_playground_temperature = adafruit_thermistor.Thermistor(board.TEMPERATURE,
10000,
10000, 25, 3950)
```

Now, look at the second error:

- `pylint_example.py 44: Trailing whitespace (trailing-whitespace)`

There is trailing whitespace on line 44. Again, Pylint keeps the details simple. If you look at the code, you'll see that there are four spaces on line 44 where there should be none. Delete those spaces.

Now, you'll address the last error:

- `pylint_example.py 5: standard import "import time" should be placed before "import board" (wrong-import-order)`

The libraries are imported in the wrong order. In this error, the details explicitly state what needs to happen to resolve the error. Currently found on line 5, `import time` should be placed before `import board`. Rearrange the import list as follows to resolve this error:

```
import time
import board
import digitalio
import adafruit_lis3dh
import touchio
import neopixel
import adafruit_thermistor
```

Ok! You've addressed all of the errors Pylint found on this run. It's time to run Pylint again.

Error: `bad continuation`

```
mac:pylint kattni$ pylint pylint_example.py
************* Module pylint_example
pylint_example.py 16: Wrong continued indentation (add 64 spaces).
10000, 25, 3950)
^                                                    | (bad-continuation)

--------------------------------------------------------------------
Your code has been rated at 9.69/10 (previous run: 9.06/10, +0.62)
```

Pylint has found one more error:

- `pylint_example.py 16: Wrong continued indentation (add 64 spaces).`
  `10000, 25, 3950)`
  `^ | (bad-continuation)`

This error is on line 16. That's the line we split off from line 15 to resolve the line too long error. The issue is that the indentation used on line 16 is incorrect. This error's details are quite clear on how to resolve it: you need to indent the line by 64 spaces. Lines 15 and 16 should look like the following when the error is addressed:

```
circuit_playground_temperature = adafruit_thermistor.Thermistor(board.TEMPERATURE,
10000,
                                                                10000, 25, 3950)
```

Once that line is fixed up, it's time to run Pylint again.

```
mac:pylint kattni$ pylint pylint_example.py

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 9.69/10, +0.31)
```

Excellent! We've worked through all the errors in our code, and Pylint is happy! 10 out of 10!

Black

If you are using pre-commit, you do not need to follow these steps. pre-commit runs Black for you. We highly recommend using pre-commit. For more information, check out this page: https://learn.adafruit.com/creating-and-sharing-a-circuitpython-library/check-your-code

Black is a really useful code formatting tool maintained by the Python Software Foundation. It reformats files to improve readability.

It can be run in two ways, the first just checks to see if there is any code it would reformat. This is the way we use on all of our CircuitPython repositories. The second way actually reformats the files. This is the way you'll be wanting to use locally.

This page explains how to install and run black, the different ways to run it, and some cases when you may not want to adhere to black's suggestions.

Installing Black

Installing black is super easy. Simply run:

```
pip install black
```

Note: if you also have a version of python2 installed, you may have to run:

```
pip3 install black
```

Using Black

As I mentioned above, there are two ways to run black. The first way is just checking the code. This is what GitHub actions runs to test commits and pull requests.

This is run by typing in Linux:

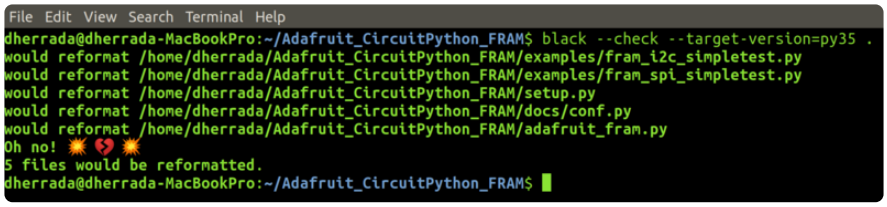
```
black --check --target-version=py35 .
```

And for Windows command line (Python 3 must be installed):

```
python -m black .
```

You can replace the `.` with whatever files you want it to check if you don't want it to check every .py file in your current directory.

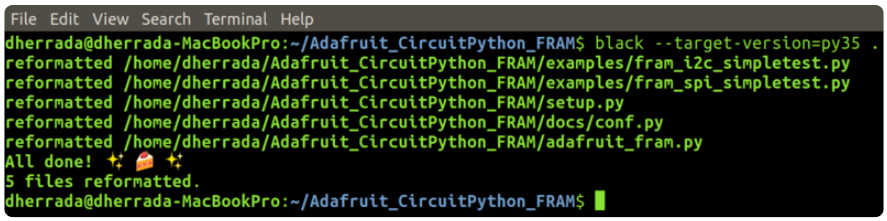
Here's what that output looks like:



However, most of the time, you're going to want Black to actually reformat the code. This is accomplished by running:

```
black --target-version=py35 .
```

Here's what running that looks like:



Black isn't always right

Sometimes, Black will make a change that just looks really bad. We've mostly encountered this with longer lists of numbers or short strings.

For example: Black would make each element of this list have it's own line.

Here's what the list looked like originally:

```
heatmap_gp = bytes([
    0, 255, 255, 255, # White
    64, 255, 255, 0,  # Yellow
    128, 255, 0, 0,   # Red
    255, 0, 0, 0])    # Black
```

Here's what the same list looked like after being reformatted by Black:

```
heatmap_gp = bytes([
    0,
    255,
    255,
    255, # White
    64,
    255,
    255,
    0,  # Yellow
    128,
    255,
    0,
    0,   # Red
    255,
    0,
    0,
    0,
])    # Black
```

You can disable black In that section by adding `# fmt: off` at the start of the section you don't want Black to reformat and `# fmt: on` at the end of said section to re-enable it.

Here's how we disabled Black for the list above:

```
# fmt: off
heatmap_gp = bytes([
    0, 255, 255, 255, # White
    64, 255, 255, 0,  # Yellow
    128, 255, 0, 0,   # Red
    255, 0, 0, 0])    # Black
# fmt: on
```

Lint Black's changes

Make sure that after you run Black, you re-run Pylint. They don't always agree, and their major point of disagreement (Pylint's `bad-continuation` check) has been dealt with for all of our CircuitPython repositories.

Documentation and Contributing

Check out Black's ReadTheDocs page ().

Also, Black's source code () is hosted on a public GitHub repository.

Resources

A list of Pylint Error Messages

Here are some web resources on looking up error messages that may be unfamiliar:

- https://pylint.readthedocs.io/ ()
- Messages alphabetically ()
- Messages by number ()

Message Control Options

Message control options are specially formatted lines in the source code, prefixed by a "#" to appear as a comment to running code. The options will turn of (or on) certain checks. For a list of options, see:

- Message control options ()

In the Adafruit Learning system, putting pylint message control options is not recommended. During the guide moderation process, the moderators will look at any pylint message control options in the code to see is suggesting changes in the code are the best option.

If you believe your code requires pylint message control options, contact the Learn moderator at the address provided when you were accepted to author in the Adafruit Learning System to reach a consensus on the best course of action.

General syntax:

```
#pylint: disable=message
```

where `message` is one of the options found in the message control options. Not all that easy to match things up, yes. Well the goal is to not have errors in the first place. And the Learn repo is less strict that the CircuitPython library and main code.

Examples for disabling three specific errors:

```
#pylint: disable=too-many-nested-blocks
#pylint: disable=too-many-branches
#pylint: disable=too-many-statements
```

While the syntax `#pylint: disable=check-one,check-two` is valid, please put them on separate lines like above as we're really trying to enforce NOT using any disable statements.