

## EDS 220 - Working with Environmental Datasets



Appendices &gt; Commenting code

This wesbite is under construction.



## Commenting code

In this section, we'll explore how to write clear and helpful comments in your code. Good comments make your code easier to understand and maintain, for both you and others. By following a few guidelines, you can write comments that are professional, consistent, and aligned with best practices.

As you read through this, remember: **Your comments will evolve hand-in-hand with your programming. Don't get caught up on "writing the perfect comments"!** Start with correctly formatting your comments and thinking about their content and move on from there.

## The basics

- Comment should start with a pound sign `#` followed by a single space, then the text.
- The first word in the comment should be capitalized [\[1\]](#).
- Always use proper spelling.
- Periods at the end of short comments are optional, but you should be consistent across your code.

### Example

Typos, inconsistent capitalization, spacing, and punctuation

```
#calculate teh average temp from dataset.  
average_temp = sum(temperatures) / len(temperatures)  
  
##      Apply the TEMPERATURE correction  
corrected_temp = average_temp + correction_factor
```

Comments follow all the basic standards

```
# Calculate average temperature from dataset  
average_temp = sum(temperatures) / len(temperatures)  
  
# Apply temperature correction  
corrected_temp = average_temp + correction_factor
```

# Content

## Make sure comments are consistent with the code

As stated in the Python PEP 8 [\[1\]](#) guide:

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Outdated comments and comments that contradict the code can cause confusion and lead to bugs.

### Example

❌ Comment is outdated and contradicts the code

```
# Convert temperature from Fahrenheit to Celsius
df['temp_f'] = df['temp_c'] * 9/5 + 32
```

✅ Comments must be consistent with the code

```
# Convert temperature from Celsius to Fahrenheit
df['temp_f'] = df['temp_c'] * 9/5 + 32
```

## Keep comments short and clear

Make comments concise and easy to understand. Avoid long-winded explanations. If the code is self-explanatory (which may vary by audience) then minimal commenting is needed.

### Example

❌ Too long and redundant

```
# In this part of the code, we are filtering the DataFrame 'df' to keep only
# the rows where the value in the 'region_type' column is equal to 'wetland'.
# This will give us data specifically for wetland regions.
wetland_data = df[df['region_type'] == 'wetland']
```

✅ Concise and clear comments

```
# Filter rows where the region is a wetland
wetland_data = df[df['region_type'] == 'wetland']
```

### Want a longer explanation? Use a markdown cell!

Keeping comments short does not mean there's no place for in-depth, detailed explanations while learning programming. If you are using a Jupyter notebook (or another file format that supports combining markdown with code), use a markdown cell for longer explanations instead of adding them as comments to your code.

## Keep it professional

Comments should be professional and avoid jokes, personal remarks, or irrelevant information.

### Example

🚫 Unprofessional comment with casual remarks

```
# Time to crunch some numbers and save the planet! 🌍  
total_emissions = df['emissions'].sum()
```

✅ Professional and relevant comment

```
# Calculate the total carbon emissions for the region  
total_emissions = df['emissions'].sum()
```

## Add comments!

While long or redundant comments can clutter our code and decrease readability, under-commenting can make our code obscure and difficult to share with others (including our future selves!)

### Example

🚫 No comments make it unclear what the code is doing

```
year = '2017'  
bbox = [-112.826843, 32.974108, -111.184387, 33.863574]  
collection = 'io-biodiversity'  
  
search = catalog.search(collections=[collection],  
                        bbox=bbox,  
                        datetime=year)
```

✅ Comments clarify code

```
# Parameters for search in cloud catalog  
year = '2017'  
bbox = [-112.826843, 32.974108, -111.184387, 33.863574] # Phoenix bounding box  
collection = 'io-biodiversity' # Biodiversity Intactness data  
  
search = catalog.search(collections=[collection],  
                        bbox=bbox,  
                        datetime=year)
```

## Special types of comments

### In-line comments

In-line comments are comments on the same line as the code. Use in-line comments sparingly and keep them short.

## Example

❌ In-line comments are overused and don't follow regular spacing and capitalization

```
years = [2000, 2005, 2010, 2015, 2020]#list of years for x-axis
co2_levels = [370, 380, 390, 400, 410] #CO2 levels for y-axis

plt.plot(years, co2_levels) # Plot years against CO2 levels
plt.title('CO2 Levels Over Time') # Add a title
plt.xlabel('Year') # x-axis = Year
plt.ylabel('CO2 Levels (ppm)')# y-axis = CO2 Levels (ppm)
plt.grid(True) # Enable grid on the plot
plt.axhline(y=400, color='r', linestyle='--', label='400 ppm Threshold')
plt.show() #display the plot
```

✅ In-line comments are sparingly used and focused

```
years = [2000, 2005, 2010, 2015, 2020]
co2_levels = [370, 380, 390, 400, 410]

# Plot to see trend in atmospheric CO2 concentrations
plt.plot(years, co2_levels)

plt.title('CO2 Levels Over Time')
plt.xlabel('Year')
plt.ylabel('CO2 Levels (ppm)')

plt.grid(True) # Show grid for better readability
plt.axhline(y=400, # Add a threshold line for 400 ppm
            color='r',
            linestyle='--',
            label='400 ppm Threshold')
plt.show()
```

## Block comments

For more complex explanations, use block comments spanning multiple lines. Each line should start with a # and be aligned with the code it describes.

## Example

❌ Wordy block comment with inconsistent spacing

```
# In this code I:
    # calculate the avg temp from a list of temperatures and then adjust the result.
    # We first sum the temperatures, divide by the number of entries,
```

```
# and finally add the correction factor to the calculated average.
```

```
temperatures = [20.5, 21.0, 19.8, 22.3]
average_temp = sum(temperatures) / len(temperatures)
corrected_temp = average_temp + 1.2
```

✓ Concise block comments focusing on the code purpose

```
# Calculate the average temperature from a list of temperature readings.
# Then, apply a correction factor to account for measurement adjustments.

temperatures = [20.5, 21.0, 19.8, 22.3]
average_temp = sum(temperatures) / len(temperatures)
corrected_temp = average_temp + 1.2
```

## Next level

As you advance in your programming journey and code looks more familiar, you can take next steps to improve your code via comments. Take some time too to learn about coding style and best practices, this [resource by the UCSB Code Carpentries can help you!](#)

## Use comments to explain why, not what

When we are learning to code, **it can be super useful to comment on ‘what’ each line of code does. That’s ok!** Over time, as you become more comfortable with the syntax, try to focus your comments on ‘why’ certain choices were made or when something might not be immediately obvious to others. **Use comments to explain why a piece of code exists, what it’s doing at a high level, or to describe a complex algorithm.** Great comments add value to the code, going beyond describing what is clear from the function and variable names.

Your comments will become more streamlined and naturally shift from technical to conceptual as your coding skills improve!

### Example

⚠ Comment restates in plain language what the code does

```
# Assign the maximum value of the array to x
x = find_max_value(array)
```

✓ Comment explains the rationale behind the code

```
# Find the largest value to normalize the data
x = find_max_value(array)
```

## Avoid over-commenting

It's ok to err on over-commenting when you are beginning your coding journey. As you advance, focus on not over-commenting obvious code, as this can clutter your code and reduce readability.

## Example

⚠ Redundant comments clutter the code

```
temperatures = [20.5, 21.0, 19.8, 22.3, 24.1] # List of temperatures

# Initialize total to 0
total = 0

# Loop through each temperature in the list
for temp in temperatures:
    total += temp # Add the current temperature to the total

# Calculate the average by dividing the total by the number of temperatures
average_temp = total / len(temperatures)

print(average_temp) # Print the average temperature
```

✅ Comments focus on important information

```
temperatures = [20.5, 21.0, 19.8, 22.3, 24.1]

# Calculate the total and average temperature
total = 0
for temp in temperatures:
    total += temp

average_temp = total / len(temperatures)
print(average_temp)
```

## Don't use “comments as deodorant”

The phrase “[code smells](#)” refers to symptoms in our code that may indicate deeper problems with its design or structure. Sometimes comments are used to explain overly-complicated, instead of making the code as self-explanatory as possible. If that happens, then comments are being used as deodorant for smelly code [2]. Avoid comments as deodorant and, instead, **work on making your code simple and understandable**. Of course, you will learn better techniques to improve your code with time!

## Example

⚠ Comments compensate for obscure variable names and overly-complicated code

```
# List of pollutant concentrations (in ppm)
a = [50, 120, 85, 30, 95, 110, 70] # 'a' stands for air quality measurements

# Filter out pollutant readings that are greater than 100 ppm (considered outliers)
```

```
b = [] # 'b' stands for valid pollutant readings
for x in a: # 'x' is the individual pollutant reading
    if x <= 100:
        b.append(x) # Add valid readings to 'b'

print(b)
```

✅ Self-explanatory code needs less comments

```
# List of pollutant concentrations (in ppm)
pollutant_readings = [50, 120, 85, 30, 95, 110, 70]

# Filter out pollutant readings that are greater than 100 ppm (considered outliers)
valid_readings = [reading for reading in pollutant_readings if reading <= 100]

print(valid_readings)
```

## Takeaways

- **Keep it clean and consistent:** Consistently use proper capitalization, spacing, spelling, and indentation.
- **Keep comments up-to-date:** Make sure your comments always match what the code is doing.
- **Be clear and concise:** Write concise comments. Keep them focused, and avoid adding personal notes or jokes. When possible, write comments that explain why your code is doing something, not just how.
- **Write clean code first:** Focus on making your code clear and easy to understand. Use comments to make things even clearer, not to explain complicated or messy code.
- **Commenting is a learning process!** Don't stress about writing "perfect comments". Focus on the basics, and let your commenting style evolve naturally as you improve your coding skills!



Happy Commenting image created with Dall-E 4.

## References

- [1] G. van Rossum, B. Warsaw, and N. Coghlan, “Style guide for Python code,” PEP 8, 2001. Available: <https://www.python.org/dev/peps/pep-0008/>
- [2] M. Fowler, *Refactoring: Improving the design of existing code*. Reading, MA: Addison-Wesley, 1999.

This work is licensed under [CC Attribution-NonCommercial 4.0 International \(CC BY-NC 4.0\) License](#)

This website is built with [Quarto](#) and [GitHub pages](#).