Lecture 6: Profiling and Documentation

CEE 690

“the code works” ->“the code is efficient”

Profiling

Code development: Ranked priorities

1. **Unit testing** - Speed is irrelevant if the answer is wrong

2. **Profiling** - If there is a fast and efficient way to increase the speed of your code by an order of magnitude… do it.

3. **Refactoring to “prettify”** - Don’t do any of this until it is necessary

So what is profiling?

The process of measuring the time and memory complexity of software as it is running.

Why should we care?

Why should we care?

1. **Feasibility** - We need to determine how much we can accomplish with a program.

Why should we care?

1. **Feasibility** - We need to determine how much we can accomplish with a program.
2. **Computing cost** - Asking for 20 hours of computing vs 2 hours on HPC matters.

Why should we care?

1. **Feasibility** - We need to determine how much we can accomplish with a program.
2. **Computing cost** - Asking for 20 hours of computing vs 2 hours on HPC matters.
3. **Resource allocation** - Determine if you are CPU/GPU limited or I/O limited.

Use profiling to optimize everything???

Use profiling to optimize everything???

Profiling priorities

Profiling priorities

1. **Algorithmic efficiency** - Is this really the best way to write this algorithm?

Profiling priorities

1. **Algorithmic efficiency** - Is this really the best way to write this algorithm?

2. **Data structures and I/O** - How we write and read matters a lot.

Profiling priorities

1. **Algorithmic efficiency** - Is this really the best way to write this algorithm?

2. **Data structures and I/O** - How we write and read matters a lot.

3. **Vectorization** - Get rid of “for loops” as much as you can.

Profiling priorities

1. **Algorithmic efficiency** - Is this really the best way to write this algorithm?

2. **Data structures and I/O** - How we write and read matters a lot.

3. **Vectorization** - Get rid of “for loops” as much as you can.

4. **Micro-optimization** - Tinkering is “fun” but mostly an inefficient use of time.

So what do we actually measure?

So what do we actually measure?

1. **Wall-clock time** - The total you spend waiting for a script/function to finish

So what do we actually measure?

1. **Wall-clock time** - The total you spend waiting for a script/function to finish
2. **CPU time** - The actual processor time spent doing the work

So what do we actually measure?

1. **Wall-clock time** - The total you spend waiting for a script/function to finish
2. **CPU time** - The actual processor time spent doing the work
3. **Memory footprint** - The peak amount of RAM used to run your program

Profiling tools in Python

Back to the nested for loops

```
14 def calculate_spatial_mean(data, config):
15 
16     # Define the final variable as a list
17     temporal_spatial_mean = []
18 
19     # Calculate temporally varying spatial mean
20     for t in range(data.shape[0]):
21         if ((t < config['TIME_MIN']) | (t >= config['TIME_MAX'])):
22             continue
23 
24         pixel_count = 0
25         total_data = 0
26 
27         for y in range(data.shape[1]):
28             if ((y < config['LAT_MIN']) | (y >= config['LAT_MAX'])):
29                 continue
30 
31             for x in range(data.shape[2]):
32                 if ((x < config['LON_MIN']) | (x >= config['LON_MAX'])):
33                     continue
34 
35                 pixel_count = pixel_count + 1
36                 total_data = total_data + data[t][y][x]
37 
38         temporal_spatial_mean.append(total_data / pixel_count)
39 
40     return np.array(temporal_spatial_mean)
```

What you probably will do

```
114     tic0 = time.time()
115
116     # Gather arguments from the terminal and convert to dictionary
117     config = vars(get_args())
118
119     # Override with json info if present
120     if config.get('JSON_FILE'):
121         with open(config['JSON_FILE'], 'r') as f:
122             json_config = json.load(f)
123             # This one line replaces all the manual overwriting
124             config.update(json_config)
125
126     # Load dataset
127     print("Loading the dataset")
128     t2m_data = load_dataset(config)
129
130     # Compute temporal series of spatial mean and spatial standard deviation
131     print("Computing the statistics")
132     tic1 = time.time()
133     temporal_spatial_mean = calculate_spatial_mean(t2m_data, config)
134     print("time - calculate_spatial_mean:",time.time()-tic1)
135     tic1 = time.time()
136     temporal_spatial_variance = calculate_spatial_variance(t2m_data, config, temporal_spatial_mean)
137     print("time - calculate_spatial_variance:",time.time()-tic1)
138
139     #Visualize the data
140     print("Visualizing the data")
141     visualize_data(temporal_spatial_mean, temporal_spatial_variance, config)
142
143     #Output the data to netcdf
144     print("Saving the computed statistics to netcdf")
145     output_data_to_netcdf(config['OUTPUT_FILE'], temporal_spatial_mean, temporal_spatial_variance)
146     print("time - main:",time.time()-tic0)
```

```
(cee690) nc153 $ python spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
time - calculate_spatial_mean: 4.7502734661102295
time - calculate_spatial_variance: 4.909358263015747
Visualizing the data
Saving the computed statistics to netcdf
time - main: 9.788025140762329
```

What you probably will do

```
    tic0 = time.time()

    # Gather arguments from the terminal and convert to dictionary
    config = vars(get_args())

    # Override with json info if present
    if config.get('JSON_FILE'):
        with open(config['JSON_FILE'], 'r') as f:
            json_config = json.load(f)
            # This one line replaces all the manual overwriting
            config.update(json_config)

    # Load dataset
    print("Loading the dataset")
```

Not very precise and prone to a lot of “edits” but it is easy to use

```
    #Visualize the data
    print("Visualizing the data")
    visualize_data(temporal_spatial_mean, temporal_spatial_variance, config)

    #Output the data to netcdf
    print("Saving the computed statistics to netcdf")
    output_data_to_netcdf(config['OUTPUT_FILE'], temporal_spatial_mean, temporal_spatial_variance)
    print("time - main:",time.time()-tic0)
```

```
(cee690) nc153 $ python spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
time - calculate_spatial_mean: 4.7502734661102295
time - calculate_spatial_variance: 4.909358263015747
Visualizing the data
Saving the computed statistics to netcdf
time - main: 9.788025140762329
```

Time it at the command line

```
(cee690) nc153 $ time python spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf

real    0m11.343s
user    0m10.060s
sys     0m0.090s
```

- Real - Wall-clock time (actual time)
- User - CPU time (performing calculations)
- Sys - System calls (talking to hardware)

User ~ real (what does that mean?)

cProfile:
Deterministic profiler

cProfile will record every single function call. It is the ideal candidate for finding the function or functions that are consuming most all of our wall-clock time

cProfile

```
(cee690) nc153 $ python -m cProfile -s cumulative spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf
         156070827 function calls (156048881 primitive calls) in 46.996 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    339/1    0.005    0.000   46.996   46.996 {built-in method builtins.exec}
        1    0.000    0.000   46.996   46.996 spatialstats.py:1(<module>)
        1    0.000    0.000   44.862   44.862 spatialstats.py:107(main)
  4506600    6.218    0.000   43.125    0.000 core.py:3283(__getitem__)
        1    0.732    0.732   22.378   22.378 spatialstats.py:14(calculate_spatial_mean)
        1    0.808    0.808   22.290   22.290 spatialstats.py:42(calculate_spatial_variance)
  6008803   12.837    0.000   21.606    0.000 core.py:3015(_update_from)
12017605/12017604    2.692    0.000   20.725    0.000 {method 'view' of 'numpy.ndarray' objects}
  3004403    3.369    0.000   18.033    0.000 core.py:3040(__array_finalize__)
48149816/48149811    4.355    0.000    4.357    0.000 {built-in method builtins.getattr}
       44    0.001    0.000    4.202    0.095 __init__.py:1(<module>)
27041577/27041397    3.614    0.000    3.614    0.000 {method 'update' of 'dict' objects}
  9013200    1.987    0.000    3.475    0.000 core.py:3761(_get_data)
 27121495    2.568    0.000    2.571    0.000 {built-in method builtins.isinstance}
   395/10    0.002    0.000    2.167    0.217 <frozen importlib._bootstrap>:1360(_find_and_load)
   391/10    0.002    0.000    2.166    0.217 <frozen importlib._bootstrap>:1308(_find_and_load_unlocked)
```

```
pixel_count = pixel_count + 1
total_data = total_data + data[t][y][x]
```

“__getitem__”

cProfile

```
(cee690) nc153 $ python -m cProfile -s cumulative spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf
```

Generally, you will only conclude that calculate_spatial_mean and calculate_spatial_variance are the main issues

```
  9013200    1.987    0.000    3.475    0.000 core.py:3761(_get_data)
 27121495    2.568    0.000    2.571    0.000 {built-in method builtins.isinstance}
    395/10    0.002    0.000    2.167    0.217 <frozen importlib._bootstrap>:1360(_find_and_load)
    391/10    0.002    0.000    2.166    0.217 <frozen importlib._bootstrap>:1308(_find_and_load_unlocked)
```

```
pixel_count = pixel_count + 1
total_data = total_data + data[t][y][x]
```

“__getitem__”

line_profiler

```python
@profile
def calculate_spatial_mean(data, config):

    # Define the final variable as a list
    temporal_spatial_mean = []

    # Calculate temporally varying spatial mean
    for t in range(data.shape[0]):
        if ((t < config['TIME_MIN']) | (t >= config['TIME_MAX'])):
            continue

        pixel_count = 0
        total_data = 0

        for y in range(data.shape[1]):
            if ((y < config['LAT_MIN']) | (y >= config['LAT_MAX'])):
                continue

            for x in range(data.shape[2]):
                if ((x < config['LON_MIN']) | (x >= config['LON_MAX'])):
                    continue

                pixel_count = pixel_count + 1
                total_data = total_data + data[t][y][x]

        temporal_spatial_mean.append(total_data / pixel_count)

    return np.array(temporal_spatial_mean)
```

Timing of each line of a given function. The function is defined by adding a decorator at the top.

line_profiler

```
(cee690) nc153 $ kernprof -l -v spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf
Wrote profile results to 'spatialstats.py.lprof'
Timer unit: 1e-06 s

Total time: 42.1304 s
File: spatialstats.py
Function: calculate_spatial_mean at line 13

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    13                                           @profile
    14                                           def calculate_spatial_mean(data, config):
    15
    16                                               # Define the final variable as a list
    17         1          1.0      1.0      0.0      temporal_spatial_mean = []
    18
    19                                               # Calculate temporally varying spatial mean
    20        38         20.5      0.5      0.0      for t in range(data.shape[0]):
    21        37         26.6      0.7      0.0          if ((t < config['TIME_MIN']) | (t >= config['TIME_MAX'])):
    22                                                       continue
    23
    24        37         12.5      0.3      0.0          pixel_count = 0
    25        37         12.2      0.3      0.0          total_data = 0
    26
    27      6697       2247.8      0.3      0.0          for y in range(data.shape[1]):
    28      6660       2383.8      0.4      0.0              if ((y < config['LAT_MIN']) | (y >= config['LAT_MAX'])):
    29      1295        384.3      0.3      0.0                  continue
    30
    31   1550485     492197.1      0.3      1.2              for x in range(data.shape[2]):
    32   1545120     524752.4      0.3      1.2                  if ((x < config['LON_MIN']) | (x >= config['LON_MAX'])):
    33    794020     226571.2      0.3      0.5                      continue
    34
    35    751100     239076.3      0.3      0.6                  pixel_count = pixel_count + 1
    36    751100   40642638.7     54.1     96.5                  total_data = total_data + data[t][y][x]
    37
    38        37        102.7      2.8      0.0          temporal_spatial_mean.append(total_data / pixel_count)
    39
    40         1          9.6      9.6      0.0      return np.array(temporal_spatial_mean)
```

line_profiler

```
(cee690) nc153 $ kernprof -l -v spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf
Wrote profile results to 'spatialstats.py.lprof'
Timer unit: 1e-06 s

Total time: 42.1304 s
File: spatialstats.py
Function: calculate_spatial_mean at line 13

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    13                                           @profile
    14                                           def calculate_spatial_mean(data, config):
    15
    16                                               # Define the final variable as a list
    17         1          1.0      1.0      0.0      temporal_spatial_mean = []
    18
    19                                               # Calculate temporally varying spatial mean
    20        38         20.5      0.5      0.0      for t in range(data.shape[0]):
    21        37         26.6      0.7      0.0          if ((t < config['TIME_MIN']) | (t >= config['TIME_MAX'])):
    22                                                       continue
    23
    24        37         12.5      0.3      0.0          pixel_count = 0
    25        37         12.2      0.3      0.0          total_data = 0
    26
    27      6697       2247.8      0.3      0.0          for y in range(data.shape[1]):
    28      6660       2383.8      0.4      0.0              if ((y < config['LAT_MIN']) | (y >= config['LAT_MAX'])):
    29      1295        384.3      0.3      0.0                  continue
    30
    31   1550485     492197.1      0.3      1.2              for x in range(data.shape[2]):
    32   1545120     524752.4      0.3      1.2                  if ((x < config['LON_MIN']) | (x >= config['LON_MAX'])):
    33    794020     226571.2      0.3      0.5                      continue
    34
    35    751100     239076.3      0.3      0.6                  pixel_count = pixel_count + 1
    36    751100   40642638.7     54.1     96.5                  total_data = total_data + data[t][y][x]
    37
    38        37        102.7      2.8      0.0          temporal_spatial_mean.append(total_data / pixel_count)
    39
    40         1          9.6      9.6      0.0      return np.array(temporal_spatial_mean)
```

Types of profilers

Types of profilers

1. **Deterministic** - The profiler records every function call. It is very accurate; however, while measuring, it slows down the script considerably (e.g., cProfile).

Types of profilers

1. **Deterministic** - The profiler records every function call. It is very accurate; however, while measuring, it slows down the script considerably (e.g., cProfile).

2. **Statistical** - The profiler looks at the code every every few milliseconds. It will miss “fast functions” (e.g., pyspy)

Types of profilers

1. **Deterministic** - The profiler records every function call. It is very accurate; however, while measuring, it slows down the script

Lots of mature and professional profilers out there. And yet…

2. **Statistical** - The profiler looks at the code every every few milliseconds. It will miss “fast functions” (e.g., pyspy)

What you probably will do

```
114     tic0 = time.time()
115
116     # Gather arguments from the terminal and convert to dictionary
117     config = vars(get_args())
118
119     # Override with json info if present
120     if config.get('JSON_FILE'):
121         with open(config['JSON_FILE'], 'r') as f:
122             json_config = json.load(f)
123             # This one line replaces all the manual overwriting
124             config.update(json_config)
125
126     # Load dataset
127     print("Loading the dataset")
128     t2m_data = load_dataset(config)
129
130     # Compute temporal series of spatial mean and spatial standard deviation
131     print("Computing the statistics")
132     tic1 = time.time()
133     temporal_spatial_mean = calculate_spatial_mean(t2m_data, config)
134     print("time - calculate_spatial_mean:",time.time()-tic1)
135     tic1 = time.time()
136     temporal_spatial_variance = calculate_spatial_variance(t2m_data, config, temporal_spatial_mean)
137     print("time - calculate_spatial_variance:",time.time()-tic1)
138
139     #Visualize the data
140     print("Visualizing the data")
141     visualize_data(temporal_spatial_mean, temporal_spatial_variance, config)
142
143     #Output the data to netcdf
144     print("Saving the computed statistics to netcdf")
145     output_data_to_netcdf(config['OUTPUT_FILE'], temporal_spatial_mean, temporal_spatial_variance)
146     print("time - main:",time.time()-tic0)
```

```
(cee690) nc153 $ python spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
time - calculate_spatial_mean: 4.7502734661102295
time - calculate_spatial_variance: 4.909358263015747
Visualizing the data
Saving the computed statistics to netcdf
time - main: 9.788025140762329
```

What you probably will do

```
114     tic0 = time.time()
115
116     # Gather arguments from the terminal and convert to dictionary
117     config = vars(get_args())
118
119     # Override with json info if present
120     if config.get('JSON_FILE'):
121         with open(config['JSON_FILE'], 'r') as f:
122             json_config = json.load(f)
123             # This one line replaces all the manual overwriting
124             config.update(json_config)
125
126     # Load dataset
127     print("Loading the dataset")
```

Not very precise and prone to a lot of “edits” but it is easy to use

```
139     #Visualize the data
140     print("Visualizing the data")
141     visualize_data(temporal_spatial_mean, temporal_spatial_variance, config)
142
143     #Output the data to netcdf
144     print("Saving the computed statistics to netcdf")
145     output_data_to_netcdf(config['OUTPUT_FILE'], temporal_spatial_mean, temporal_spatial_variance)
146     print("time - main:",time.time()-tic0)
```

```
(cee690) nc153 $ python spatialstats.py --JSON_FILE example.json
Loading the dataset
Computing the statistics
time - calculate_spatial_mean: 4.7502734661102295
time - calculate_spatial_variance: 4.909358263015747
Visualizing the data
Saving the computed statistics to netcdf
time - main: 9.788025140762329
```

Let’s revisit our vectorized code

```
46      def run_analysis(self):
47          """Orchestrates the computation of statistics."""
48          print("Computing the statistics")
49          t_start, t_end = self.config['TIME_MIN'], self.config['TIME_MAX']
50
51          # Slicing the subset once to be used by both mean and variance
52          subset = self.data[t_start:t_end,
53                             self.config['LAT_MIN']:self.config['LAT_MAX'],
54                             self.config['LON_MIN']:self.config['LON_MAX']]
55
56          self.means = np.mean(subset, axis=(1, 2))
57          self.variances = np.var(subset, axis=(1, 2))
```

```
(cee690) nc153 $ python -m cProfile -s cumulative spatialstats.py --JSON_FILE example.json
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf
         1333930 function calls (1311984 primitive calls) in 2.863 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       44    0.001    0.000    4.897    0.111 __init__.py:1(<module>)
    339/1    0.005    0.000    2.864    2.864 {built-in method builtins.exec}
        1    0.000    0.000    2.864    2.864 spatialstats.py:1(<module>)
   395/10    0.002    0.000    2.694    0.269 <frozen importlib._bootstrap>:1360(_find_and_load)
   391/10    0.002    0.000    2.693    0.269 <frozen importlib._bootstrap>:1308(_find_and_load_unlocked)
   375/10    0.001    0.000    2.681    0.268 <frozen importlib._bootstrap>:914(_load_unlocked)
   329/10    0.001    0.000    2.681    0.268 <frozen importlib._bootstrap_external>:753(exec_module)
  1083/16    0.001    0.000    2.659    0.166 <frozen importlib._bootstrap>:483(_call_with_frames_removed)
   637/56    0.001    0.000    1.795    0.032 {built-in method builtins.__import__}
   387/89    0.001    0.000    1.767    0.020 <frozen importlib._bootstrap>:1401(_handle_fromlist)
    37/19    0.000    0.000    1.361    0.072 <frozen importlib._bootstrap_external>:1059(exec_module)
    37/19    0.013    0.000    1.361    0.072 {built-in method _imp.exec_dynamic}
      329    0.003    0.000    1.227    0.004 <frozen importlib._bootstrap_external>:826(get_code)
      416    0.001    0.000    1.168    0.003 <frozen importlib._bootstrap_external>:947(get_data)
        1    0.000    0.000    0.754    0.754 pyplot.py:1(<module>)
      389    0.002    0.000    0.745    0.002 <frozen importlib._bootstrap>:1243(_find_spec)
      382    0.000    0.000    0.741    0.002 <frozen importlib._bootstrap_external>:1284(find_spec)
      382    0.001    0.000    0.741    0.002 <frozen importlib._bootstrap_external>:1255(_get_spec)
```

Timing at command line: revisited

```
(cee690) nc153 $ vi spatialstats.py
(cee690) nc153 $ time python spatialstats.py --JSON_FILE example.json
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf

real    0m2.506s
user    0m0.587s
sys     0m0.092s
```

Timing at command line: revisited

```
(cee690) nc153 $ vi spatialstats.py
(cee690) nc153 $ time python spatialstats.py --JSON_FILE example.json
Computing the statistics
Visualizing the data
Saving the computed statistics to netcdf

real    0m2.506s
user    0m0.587s
sys     0m0.092s
```

What do we learn here?

Documentation

Layers of documentation

1. **Comments** - Explain why not what
2. **Docstrings** - Define inputs, output, and behavior (what is supposed to do)
3. **README** - High-level how to install and run

Documented code

```
"""
Spatial Statistics Analyzer
===========================

This module provides an Object-Oriented interface for processing weather and climate data
stored in NetCDF format. It handles data loading, statistical computation
(spatial mean and variance), visualization, and data export.

classes
-------
SpatialAnalyzer
    The main driver class for the load, transform, save, and plot pipeline.
"""

import argparse
import json
import netCDF4 as nc
import numpy as np
import matplotlib
import os
import sys

# use 'Agg' backend to allow plotting on headless servers (HPC clusters)
# without requiring an X11 window system.
matplotlib.use('Agg')
import matplotlib.pyplot as plt

class SpatialAnalyzer:
    """
    A class to handle the loading, processing, and visualization of spatial
    climate data.

    Attributes
    ----------
    config : dict
        Configuration dictionary containing file paths and slice indices.
    data : numpy.ndarray
        The loaded 3D climate and weather dataset (Time x Latitude x Longitude).
    means : numpy.ndarray or None
        The computed spatial means over time.
    variances : numpy.ndarray or None
        The computed spatial variances over time.
    """
```

README

README.md

Spatial Statistics Analyzer

A Python tool for efficient processing of climate data stored in NetCDF format. This script calculates spatial means and variances over user-defined temporal and spatial subsets using vectorized NumPy operations.

Features

- **Vectorized Performance:** Uses NumPy optimization to process large datasets without slow Python loops.
- **HPC Ready:** Uses the `Agg` matplotlib backend, allowing plot generation on headless servers.
- **Flexible Configuration:** Supports both Command Line Arguments and JSON configuration files.
- **Standardized Output:** Produces both a visualization (PNG) and a data file (NetCDF).

Requirements

- Python 3.8+
- `netCDF4`
- `numpy`
- `matplotlib`

Install dependencies via pip:

```
pip install netCDF4 numpy matplotlib
```

README

README.md

Spatial Statistics Analyzer

A Python tool for efficient processing of climate data stored in NetCDF format. This script calculates spatial means and variances over user-defined temporal and spatial subsets using vectorized NumPy operations.

This is now ready for Sphinx

- **HPC Ready:** Uses the `Agg` matplotlib backend, allowing plot generation on headless servers.
- **Flexible Configuration:** Supports both Command Line Arguments and JSON configuration files.
- **Standardized Output:** Produces both a visualization (PNG) and a data file (NetCDF).

Requirements

- Python 3.8+
- `netCDF4`
- `numpy`
- `matplotlib`

Install dependencies via pip:

```
pip install netCDF4 numpy matplotlib
```

Example: Basic Sphinx project for Read the Docs

This example shows a basic Sphinx project with Read the Docs. You're encouraged to view it to get inspiration and copy & paste from the files in the source code. If you are using Read the Docs for the first time, have a look at the official Read the Docs Tutorial.

📚 docs/

A basic Sphinx project lives in `docs/`. All the `*.rst` make up sections in the documentation.

⚙️ .readthedocs.yaml

Read the Docs Build configuration is stored in `.readthedocs.yaml`.

⚙️ docs/conf.py

Both the configuration and the folder layout follow Sphinx default conventions. You can change the Sphinx configuration values in this file

📍 docs/requirements.txt and docs/requirements.in

Python dependencies are pinned (uses pip-tools). Make sure to add your Python dependencies to `requirements.txt` or if you choose pip-tools, edit `docs/requirements.in` and remember to run `pip-compile docs/requirements.in`.

But why should I use Sphinx?

1. **Living documentation** - As you update and new functions (and docstrings), it provides fast documentation updates.

2. **Native math support** - It will render Latex that you included in your docstrings.

3. **Hyperlinked cross-referencing** - Jump between the different functions

4. **Professional standard** (The cool kids are doing it) - Numpy, Pandas, Spicy, etc… are all documented with Sphinx.

But why should I use Sphinx?

1. **Living documentation** - As you update and new functions (and docstrings), it provides fast documentation updates.

2. **Like it or not, it instantaneously adds credibility to the software** ed

3. **Hyperlinked cross-referencing** - Jump between the different functions

4. **Professional standard** (The cool kids are doing it) - Numpy, Pandas, Spicy, etc… are all documented with Sphinx.

Go take the tutorial (or ask a LLM) on how to run Sphinx

Build your first project

In this tutorial you will build a simple documentation project using Sphinx, and view it in your browser as HTML. The project will include narrative, handwritten documentation, as well as autogenerated API documentation.

The tutorial is aimed towards Sphinx newcomers willing to learn the fundamentals of how projects are created and structured. You will create a fictional software library to generate random food recipes that will serve as a guide throughout the process, with the objective of properly documenting it.

To showcase Sphinx capabilities for code documentation you will use Python, which also supports *automatic* documentation generation.

> **Note**
>
> Several other languages are natively supported in Sphinx for *manual* code documentation, however they require extensions for *automatic* code documentation, like Breathe.

To follow the instructions you will need access to a Linux-like command line and a basic understanding of how it works, as well as a working Python installation for development, since you will use *Python virtual environments* to create the project.

Go take the tutorial (or ask a LLM) on how to run Sphinx

Build your first project

In this tutorial you will build a simple documentation project using Sphinx, and view it in your browser as HTML. The project will include narrative, handwritten documentation, as well as autogenerated API documentation.

The tutorial is aimed towards Sphinx newcomers willing to learn the fundamentals of how projects are created and structured. You will create a fictional software library to generate random food recipes that will serve as a guide throughout the process, with the objective of properly documenting it.

To showcase Sphinx capabilities for code documentation you will use Python, which also supports *automatic* documentation generation.

> **Note**
>
> Several other languages are natively supported in Sphinx for *manual* code documentation, however they require extensions for *automatic* code documentation, like Breathe.

To follow the instructions you will need access to a Linux-like command line and a basic understanding of how it works, as well as a working Python installation for development, since you will use *Python virtual environments* to create the project.

- Getting started
  - Setting up your project and development environment
  - Creating the documentation layout
- First steps to document your project using Sphinx

Hint: It’s very easy to implement/learn

Sphinx magic

Spatial Statistics Analyzer

This module provides an Object-Oriented interface for processing weather and climate data stored in NetCDF format. It handles data loading, statistical computation (spatial mean and variance), visualization, and data export.

classes

SpatialAnalyzer

The main driver class for the load, transform, save, and plot pipeline.

class **spatialstats.SpatialAnalyzer**(*config*) [source]

Bases: `object`

A class to handle the loading, processing, and visualization of spatial climate data.

config

Configuration dictionary containing file paths and slice indices.

Type: dict

data

The loaded 3D climate and weather dataset (Time x Latitude x Longitude).

Type: numpy.ndarray