Section	Unacceptable	Below Average	Average	Good	Excellent	Weight	Max Points	100
I. Functionality & Originality	Trivial functionality (e.g., a simple wrapper for existing packages).	Essentially the same task done conceptually in class, minor variation.	Moderate extension to work done in class (e.g., combining two learned techniques in a new way).	Function shows original work, demonstrating novel application, but is limited in scope or general utility.	Completely original work, solving a novel problem; code is structured as a significant contribution to the community (highly robust and generic).	2	8	
II. Mechanics & Modularity	Function uses the anti-pattern input() instead of parameters; Hard-coded values severely restrict applicability; Function fails or errors when called.	Function is written but its modular use is not demonstrated effectively; Module imports are poorly structured.	Module is not modular (e. g., file reading, scaling, plotting, and core analysis occur within the same function).	Code is mostly modular; Key constants are passed as variables or optional arguments (keywords).	Function(s) are broken down into their "most basic building blocks"; Returns complex data efficiently using containers like SimpleNamespace.	3	12	
III. Adaptability & Modifiability	Module only works with a single, specific file; Values are hardcoded inside the function/module that should be variables.	Variables are defined in inside the module but are not accepted as an input parameters, severely limiting runtime flexibility.	Variables are passed correctly as a required argument/parameter (e.g., def read_file(file))	File path and critical domain-specific constants (e.g., min/max scaling inputs or fixed boundaries) are passed as keyword arguments.	Flexibility is maximized: all file paths, scaling parameters (min/max), and user-defined constraints are controlled via keyword arguments, demonstrating maximum adherence to the Modifiable tenet.	4	16	
IV. Efficiency & Error Handling	Extensive use of explicit Python for loops to operate on individual array elements, leading to "horribly inefficient" code; Logic uses highly discouraged constructs (e. g., goto logic).	Code uses for loops when vectorization (array operations) would be feasible; Compound conditional indexing on NumPy arrays uses logical and/or keywords incorrectly instead of bitwise operators.	Code is mostly vectorized (Efficient); Uses LBYL (Look Before You Leap) approach to anticipate simple errors (e.g., checking condition before computing).	Implements effective error handling using EAFP (Easier to Ask Forgiveness Than Permission) or LBYL; Exceptions are caught and handled gracefully (e.g., try/except ValueError).	Code is maximized for efficiency; Includes Elegant Outs by checking inputs for validity and returning nonsensical values (like np.nan or None) upon critical failure.	3	12	
V. Comments & Documentation Quality	Poorly commented; Docstring is missing entirely. Comments are missing the "why".	Documentation is present but missing the simple, self-contained Example section; Comments/Docstrings are poorly formatted.	Fair comments, but rationale ("why") is unclear; Fails the rule of thumb that half the routine should be comments.	Documentation exists and generally adheres to the NumPy/SciPy style; Includes all sections (Summary, Parameters, Returns, Description) but with minor formatting issues.	Well commented and documented; Docstring strictly adheres to the NumPy/SciPy style, including self-contained Examples, detailed parameter descriptions, and return values.	2	8	
VI. Style and Naming	Inconsistent capitalization and formatting; Excessive use of single-letter variables when descriptive names are needed.	Adherence to Python style conventions (PEP8) for naming and syntax.		Naming follows general style (e.g., lower_case_with_underscore) but there are exceptions; Constants use ALL_CAPS.	Strict adherence to PEP8 style for variable, function, and module names; Variable names are descriptive and avoid problematic single-letters (e.g., 'l').	1	4	

Section	**Unacceptable**	**Below Average**	**Average**	**Good**	**Excellent**	**Weight**	**Max Points**	100
VII. Example	Missing if __name__ == '__main__': block OR testing code is completely missing from documentation.	Includes only basic testing (e.g., simple scalars/dummy data) within the main block; Does not test edge cases or optional parameters.		Includes a fully functional if __name__ == '__main__': test block. Testing uses complex dummy data.	Testing code uses both dummy data (e.g., made up data) AND real data (e.g., reading external file). Complex usage is also demonstrated.	3	12	
VIII. Proficiency	Only uses basic Python constructs (lists, simple loops); Fails to demonstrate use of any core scientific package (e.g., NumPy, SciPy, Matplotlib).*****		Uses 1-2 core scientific packages (e.g., basic NumPy/Matplotlib); Demonstrates basic control flow (if/then); Uses Python loops for iteration/analysis.	Integrates 3+ scientific packages; Reads basic scientific data (e.g., ASCII/CSV); Uses arrays effectively (slicing/differencing).	Routinely integrates 4+ advanced scientific packages; Handles complex data types (netCDF/HDF) including scaling/offsets; Applies advanced array techniques (broadcasting, reshaping, interpolation).	7	28	
	*** Note this is an especially heavy block. If your code falls here, it pulls your entire grade down to a zero.							