Problem (BigCodeBench/737)

Calculate the median of all elements in a nested list 'L'. The function should output with: median (float): The median. You should write self-contained code starting with: `import numpy as np`, `import math`, `def task_func(L):`.

Model's raw reasoning steps).
$< step1 >$ The problem requires calculating the median of all numerical elements within a potentially nested list. This means the first step is to extract all numbers from the nested structure into a single, flat list or array. –**State the goal**
$< step2 >$ I considered different ways to flatten a nested list, including recursion and iteration. An iterative approach using a stack is generally more robust against deep recursion limits compared to a recursive function call approach. I decided to implement an iterative flattening process. –**Choose a method**
$< step3 >$ The iterative flattening works by initializing a stack with the input list. While the stack is not empty, I pop an item. If the item is a list, I push its elements onto the stack (in reverse order to –**Outline the stack pass, but stopped**
Problematic pattern: Incomplete Reasoning The reasoning chains end at step 3.

Figure 1: Example for Incomplete Reasoning of BigCodeBench/737 with raw reasoning contents

Problem (BigCodeBench/1082)

Compute the Pearson correlation between numeric `score` and categorical `grade`: (1) convert string scores to floats; (2) encode grades by rank order; (3) return the correlation coefficient as a `float`. If the DataFrame has $<$ 2 rows, return `NaN`. Starter stub: `import pandas as pd`, `from scipy.stats import pearsonr`, `def task_func(data):`.

Model's raw reasoning steps $< step1 >$ First, we need to validate the input DataFrame. Since the Pearson correlation coefficient cannot be computed with fewer than 2 rows, we immediately check for this case and return NaN if the DataFrame's length is less than 2. - **Input Validation**
$< step2 >$ Next, we convert the "score" column (which is provided as a string) to floats. This is done by using the astype method (or pd.to_numeric) to ensure we have valid numerical data to compute the correlation. -**Convert scores from string to float**
$< step3 >$ Then, we need to encode the categorical "grade" column. The task requires encoding based on the rank order. We interpret this by sorting the unique grade values (e.g., if the grades are A, B, C, they will be sorted alphabetically) and then mapping each grade to an integer according to its position in the sorted list. -**Encode categorical grades by rank order**
$< step4 >$ After converting scores and encoding the grades, we compute the Pearson correlation coefficient using the pearsonr function from the scipy.stats module. This function returns a tuple where the first element is the correlation coefficient. -**Compute Pearson correlation using scipy** $< step5 >$ Finally, the function returns the correlation coefficient as a float if everything is computed successfully. -**Return correlation coefficient**
Problematic pattern: Lacking Edge Cases Handling The model did not check for missing columns, non-numeric scores, or dropped NaNs in any reasoning steps.

Figure 2: Example for lacking egde cases handling: BigCodeBench/1082.

Problem (BigCodeBench/458)

Load a JSON string into a dictionary, double numerical values (including within lists and numeric substrings in strings via regex), do not process nested dictionaries, and return a one-row `pandas.DataFrame`; return an empty `DataFrame` for empty/invalid input.

Model's raw reasoning steps $< step1 >$ The task requires converting a JSON string into a normalized DataFrame. First, the JSON is parsed into a Python dictionary. JSON parsing errors or an empty input should return an empty DataFrame. -**Understand the problem and overall approach**
$< step2 >$ To handle numerical values, create a helper function process_value to recursively process each value. Numerics and numeric strings are doubled, lists are processed element-wise, and non-convertible strings are retained as-is. –**Handle numeric values: double numerics and numeric strings, process lists element-wise**
$< step3 >$ The keys of the parsed JSON form the DataFrame columns. Values must be normalized using process_value, ensuring scalars become floats and lists remain lists of numerics or mixed values where applicable. –**Use original keys as columns and normalize values**
$< step4 >$After normalization, construct the DataFrame using pd.DataFrame([normalized_dict]) for a single row format. This ensures each column corresponds to a key, storing processed values as scalars, lists, or strings appropriately. –**Build Dataframe**
$< step5 >$ Check edge cases like invalid JSON, empty inputs, or non-numeric strings to ensure graceful failure by returning an empty DataFrame. Valid inputs with strings convertible to numerics must be transformed and included as numeric entries. **Handle edge cases**

Problematic pattern: Missing Covering Requirements Across the model's reasoning steps, there are no steps that involve extracting and doubling numeric substrings within strings via regex, as required in the problem description. It handles only fully numeric strings in step 2.

Figure 3: Example for Missing-coverage illustration for `BigCodeBench/458`.

Problem (BigCodeBench/797)

Count the total number of brackets across all *values* in a `pandas.DataFrame`. Brackets are `( ) { } [ ]`; use the regex pattern `[(){}[\\]]` to identify them. Non-`DataFrame` input → raise `TypeError`. Return an `int` (e.g., `['(a)', 'b', '[[[[))c']` → `8`).

Model's behavior (observed). The model vacillates between multiple strategies, and fixates on regex escaping minutiae with several competing patterns before eventually settling on patterns before eventually settling on `[(){}[\\]]`. The plan gets repeatedly re-stated, delaying a clear, single implementation path.

Model's Raw Reasoning Steps: We are given a pandas DataFrame and we need to count the total number of brackets in the entire DataFrame. The brackets are defined as: '(', ')', '', '', '[', ']' We note that the example uses a pattern '[()[
]]' to identify brackets. However, note that in a regex pattern, we have to escape appropriately. Also, note that inside a character class (square brackets for regex), some characters lose their special meaning, but we must be cautious. Steps: 1. Check that the input is a pandas DataFrame. If not, raise a TypeError. 2. We need to traverse every cell in the DataFrame. Since the DataFrame can have multiple columns and rows, we can convert the entire DataFrame to a string and then count? But note: converting to string might include the DataFrame structure (like indices and column names) and we don't want that. Alternatively, we can iterate over each cell and count the brackets in each cell.However, the problem says: C̈ount the total number of brackets in a pandas DataFrame̦ meaning in the data (the values) only. Approach: We can use:¿
total_brackets = 0 for each column in df:for each cell in that column: count the brackets in that string (if the cell is a string) and add to total_brackets. But note: the DataFrame might have non-string values. We must convert non-string values to string? Or only consider string columns? The example: ['(a)', 'b', '[[[[))c'] are all strings. The problem does not specify non-string columns. But to be safe, we can convert every cell to string and then count. Alternatively, we can use the regex pattern to find all bracket characters in a string. We can use ‘re.findall‘ with pattern r'[()[
]]'. However, note: the pattern in the note is '[()[
]]'. In regex, the inner [and] must be escaped? Actually, inside a character class, the only special characters are: $^{(}ifatthebeginning), -, andtheclosingbracket].Wecanwritethepatternas: r'[()[$
]]' but that is redundant. Actually, the pattern: [()[
]]. This pattern means: match any one of: '(', ')', '', '', '[', ']'. But note that the last part [
]]: the first [is the start of the class, then we have the characters: '(', ')', '', '', and then we have [and then we have to include ']'? But wait, the pattern inside is: ()[
] But the backslash is escaping the]? Actually, in a regex character class, to include a literal ']', we can put it at the beginning (or escape it). Alternatively, we can write: r'[
()]'.So let's use: pattern = r'[
()]'.However, the note says: the function uses a specific pattern '[()[
]]'. This pattern in a regex would be: [()[
]] -¿ this is a character class that matches: '(' , ')', '', '', '[', and then because of the backslash, it escapes the next character which is ']'? But actually, in a character class, the backslash is not needed for the ']' if we place it appropriately. Alternatively, we can write: r'[()[
]]' is redundant? But note: the example pattern in the problem is given as a string: '[()[
]]'. If we break it down: It starts with [and then the characters: ( ) [and then
] which becomes] because the backslash escapes the] and then the class ends? Actually, the pattern as a regex would be: [()[
]] -¿ this is valid and matches the six bracket characters. However, in Python, to represent this pattern in a raw string we can write: r'[()[
]]' Why? Because inside a character class, the backslash is a literal backslash? Actually, no: the backslash is still an escape. But in this case, the pattern '[()[
]]' is interpreted as: [...] is the class. Inside, we have: ( ) [
] But the
] is interpreted as the character ']' (because the backslash escapes the bracket). So the pattern matches: '(', ')', '', '', '[', and ']'. Therefore, we can use: pattern = r'[()[
]]' Steps in code: 1. Check if df is a pandas DataFrame. If not, raise TypeError. 2. Initialize a counter to 0. 3. Iterate over each column in the DataFrame. 4. For each column, iterate over each element (cell) in the column. 5. Convert the cell to string (in case it's not a string) because we are going to search for brackets in the string representation. 6. Use re.findall with the pattern to find all bracket characters in the string of that cell. 7. Add the count (length of the list) to the total counter. Alternatively, we can avoid iterating by using .applymap and then .sum(). But .applymap might be inefficient for large DataFrames, but the problem doesn't specify size. However, note: the problem does not specify the size, so we can choose either. Let's do: total = 0 pattern = r'[()[
]]' for col in df.columns: for cell in df[col]: s = str(cell)3convert to string to handle non-string and also NaN (which becomes string 'nan') matches = re.findall(pattern, s) total += len(matches) return total But wait: what about NaN? If there is a NaN, converting to string gives 'nan'. We don't have brackets in 'nan', so that's safe. However, we can also do without regex? We could use a list of bracket characters and then count the occurrence of each bracket in the string? But the regex approach is straightforward and matches the

Problem (BigCodeBench/473)

Generate and plot `n_walks` random walks of `n_steps` steps with NumPy; validate inputs; plot each walk on a Matplotlib `Axes` while cycling colors through `['b','g','r','c','m','y','k']`; return the `Axes`.

Model's Raw Reasoning Contents $< step1 >$ The first step is to understand the problem requirements. We need to create a Python function task_func that takes n_walks (number of random walks) and n_steps (number of steps per walk) as input, along with an optional seed for reproducibility. The function should generate these random walks using numpy, plot them on a single matplotlib Axes using different colors from a predefined list, and return the Axes object. It also needs to perform input validation. -**Problem understanding**
$< step2 >$ Next, I will handle the input validation. The problem implies that n_walks and n_steps should be positive integers. I will add checks at the beginning of the function to ensure they are of type int and are greater than 0. If not, I will raise a ValueError with a descriptive message. –**input validation**
$< step3 >$ I will set the random seed if provided. numpy.random.seed() is the standard way to do this with numpy. This ensures that the generated walks are the same each time the function is called with the same seed. –**Seed handling**
$< step4 >$Now, I will implement the random walk generation using numpy. A 2D random walk starts at (0, 0) and at each step moves one unit in one of four cardinal directions (up, down, left, or right).To generate n_walks walks of n_steps each simultaneously: First, generate a (n_walks, n_steps) array of random integers representing the direction of each step for each walk. Integers 0, 1, 2, 3 can represent the four directions. Second, define the corresponding displacement vectors for each direction: (0, 1), (0, -1), (-1, 0), (1, 0). Third, use fancy indexing to map the random direction integers to the displacement vectors, resulting in a (n_walks, n_steps, 2) array where the last dimension contains the (dx, dy) displacement for each step. Fourth, calculate the cumulative sum of these displacements along the step axis (axis=1). This gives the (x, y) coordinates of the end point after each step, relative to the start. The result will be a (n_walks, n_steps, 2) array. Finally, since the walk starts at (0, 0), we need to include this point. Prepend a (n_walks, 1, 2) array of zeros to the cumulative sum array along the step axis. The final shape of the generated paths will be (n_walks, n_steps + 1, 2), where the first element along axis 1 is the starting point (0,0). -**random walk generation (misinterprets 1D requirement as 2D cardinal walk)**
$< step5 >$ I will set up the plotting environment. This involves creating a figure and an axes object using matplotlib.pyplot.subplots(). – **Creating figure and axes**
$< step6 >$ I need to plot each walk with a different color, cycling through the list ['b', 'g', 'r', 'c', 'm', 'y', 'k']. I will define this color list. To cycle through the colors repeatedly, I can use itertools.cycle. –**Setting up color cycle**
$< step7 >$ I will iterate through the generated walks. For each walk (which is a sequence of (x,y) points), I will extract the x-coordinates and y-coordinates. I will get the next color from the color cycle and use ax.plot() to draw the walk on the axes object. –**Plotting each walk**
$< step8 >$ Add optional plot enhancements like labels (xlabel, ylabel) and a title (title) for better readability, although not strictly required by the prompt. –**Optional labels and title**
$< step9 >$ Finally, return the ax object as required by the problem description. I must ensure that plt.show() is not called inside the function, as the user calling the function is responsible for displaying the plot. –**Return axes object**
$< step10 >$ Review the code to ensure it meets all requirements: takes n_walks, n_steps, seed; validates inputs; uses numpy for generation; plots n_walks on one axes; uses specified color cycle; returns ax; handles seed

– **Review code**
Problematic pattern: Beyond Requirements / Hallucination The model overreach occurs in step 4, where the reasoning imposes a specific two-dimensional, cardinal-direction random-walk model and explicitly preends the origin $(0, 0)$. In addition, in step 8, where it adds axis labels and a title. These choices exceed the specification that the problem does not require a 2D lattice or a fixed starting point; a simple 1D walk plotted as value vs. step index would satisfy the requirement.

Figure 5: Example for Beyond-requirements illustration of `BigCodeBench/473`.

Problem (BigCodeBench/1006)

Compute the Pearson correlation between numeric `score` and categorical `grade`: (1) convert string scores to floats; (2) encode grades by rank order; (3) return the correlation coefficient as a `float`. If the DataFrame has < 2 rows, return `NaN`. Starter stub: `import pandas as pd`, `from scipy.stats import pearsonr`, `def task\_func(data):`.

Model's Raw Reasining. $< step1 >$ The problem requires downloading a file from a URL, verifying it's a ZIP file, extracting its contents, and handling various errors with specific return messages. The function signature ‘task_func(url, download_path=m̈nt/data/downloads/)̈‘ is provided. I need to use the ‘requests‘ library for downloading and the ‘zipfile‘ library for handling ZIP files. The ‘os‘ library will be needed for path manipulation and directory creation. – **Identify the core problem and needed libraries**
$< step2 >$ I need to define a temporary path where the downloaded ZIP file will be saved before extraction. This path should be within the specified ‘download_path‘. A fixed name like ‘downloaded_file.zip‘ inside ‘download_path‘ is a simple approach. I will use ‘os.path.join‘ to create the full path. –**Define temporary download path using os.path.join**
$< step3 >$ I need to ensure the target directory (‘download_path‘) exists before attempting to save the file or extract. I will use ‘os.makedirs(download_path, exist_ok=True)‘ to create the directory if it doesn't exist. This should be done early in the function. –**Ensure download directory exists using os.makedirs**
$< step4 >$The core download process using ‘requests.get‘ needs to be wrapped in a ‘try...except‘ block to catch network-related errors or invalid URLs. ‘requests.exceptions.RequestException‘ is a broad exception that covers most issues during a request. If this exception occurs, I must return Ërror: Unable to download the file from the provided URL.̈ I should also check the HTTP status code after the request; ‘response.raise_for_status()‘ is a convenient way to raise an ‘HTTPError‘ for bad responses (like 404 or 500), which is a subclass of ‘RequestException‘, so it will be caught by the same ‘except‘ block. Using ‘stream=True‘ is good practice for potentially large downloads, and writing the content in chunks is necessary. **Describe downloading with requests and handling RequestException**
$< step5 >$After a successful download response is received, I must check the ‘Content-Type‘ header to verify it's a ZIP file. The expected content type for ZIP is ‘application/zip‘. It's safer to check if the header *starts with* ‘application/zip‘ to account for potential parameters like ‘; charset=utf-8‘. If the content type is not ‘application/zip‘, I should return Ërror: The URL does not point to a ZIP file.änd make sure to close the response connection. **Verifiy content type is ZIP**
$< step6 >$ If the content type is correct, I need to save the downloaded content to the temporary file path defined in step 2. I'll open the temporary file in binary write mode (‘'wb'‘) and iterate through the response content using ‘response.iter_content(chunk_size=8192)‘, writing each chunk to the file. I must close the response after saving the content. –**Save content to file using iter_content, binary writ**
$< step7 >$ Once the file is downloaded, I need to open it as a ZIP file and extract its contents. This process should also be wrapped in a ‘try...except‘ block. The ‘zipfile.ZipFile‘ constructor or the ‘extractall‘ method can raise ‘zipfile.BadZipFile‘ if the file is not a valid ZIP archive. If this exception occurs, I must return Ërror: The downloaded file is not a valid ZIP file.̈ I will use a ‘with ZipFile(...)‘ statement for proper resource management. –**Open and extracts ZIP using ZipFile, handles BadZipFile**
$< step8 >$ The extraction process (‘zip_ref.extractall‘) could potentially raise other exceptions (e.g., permission errors, disk full). The problem requires catching G̈eneral Exceptionsänd reporting them as Ërror: [exception message].̈ I will add a general ‘except Exception as e:‘ block after the specific ‘BadZipFile‘ exception handler within the ZIP processing block. –**Catches general exceptions during extraction and formats message** $< step9 >$ A temporary file was created during the download. This file should be removed after the process is complete, regardless of whether extraction succeeded or failed *after* the download. A ‘finally‘ block is suitable for this cleanup. Inside the ‘finally‘ block, I will check if the temporary file exists using ‘os.path.exists‘ and, if it does, attempt to remove it using ‘os.remove‘. I should handle potential errors during cleanup (like permission errors) gracefully, perhaps by ignoring them, as they shouldn't prevent the function from returning the primary result (success path or the main error message). **Cleans up temporary file with finally, handles potential removal errors** $< step10 >$ If all steps complete without raising an exception that results in an early return, the function was successful. In this case, I should return the ‘download_path‘. –**Returns download_path on success** $< step11 >$ I will structure the ‘try...except‘ blocks to handle errors in the required order: first download/network errors, then content type, then ZIP-specific errors, and finally general exceptions. The cleanup in the ‘finally‘ block will execute last.", –REDUNDANCY

Problematic pattern: Redundancy/Verbosity
In the last step, the model's reasoning sescribes overall structure of exception handling (ordering, nesting) which was already described above, so here is a redundancy step.

Figure 6: Example for Redundancy: BigCodeBench/1006.