

AI ASSISTED CODING LAB EXAM-2

NAME: SIRIVELLA SANJANA

ROLLNO: 2403A510D4

BATCH NO: 05

SUBGROUP-D

TASK:1

Scenario (sports analytics):

Context:

File versioning in the sports analytics data pipeline uses a `_vNN` suffix before the extension.

Your Task:

Create tests and implement `bump_version(name)` that adds or increments `_vNN` with zero-padding.

Data & Edge Cases:

Handle names with and without existing suffix; preserve original extension.

AI Assistance Expectation:

Use AI to propose regex and test cases for edge names like `report_v9.csv`, `summary.csv`.

Constraints & Notes:

Preserve original extension and base name.

Sample Input

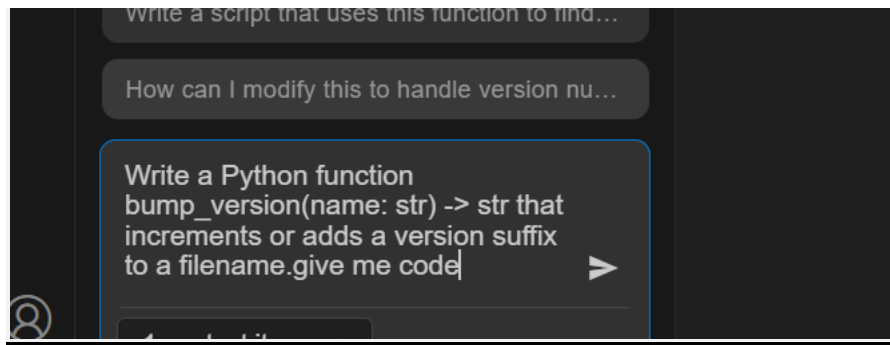
`['report_v1.csv', 'summary.csv', 'log_v09.txt']`

Sample Output

`['report_v02.csv', 'summary_v01.csv', 'log_v10.txt']`

Acceptance Criteria: Correct zero-padding; extension preserved

PROMPT:



CODE:

```
ai_exam.py > bump_version
1 import os
2 import re
3
4 def bump_version(name: str) -> str:
5     """
6     Increments or adds a version suffix to a filename.
7
8     If the filename has a version suffix like '_v1', it increments it (_v2).
9     If it doesn't have a version suffix, it adds '_v1'.
10
11     Args:
12         name: The input filename (e.g., 'report.txt', 'image_v2.jpg').
13
14     Returns:
15         The new filename with the bumped or added version.
16
17     Examples:
18         >>> bump_version("file.txt")
19         'file_v1.txt'
20         >>> bump_version("file_v1.txt")
21         'file_v2.txt'
22         >>> bump_version("archive_v10.zip")
23         'archive_v11.zip'
24         >>> bump_version("no_extension_v3")
25         'no_extension_v4'
26         >>> bump_version("another.file.with.dots.txt")
27         'another.file.with.dots_v1.txt'
28     """
29     # Split the filename into the part before the extension and the extension itself
30     base, ext = os.path.splitext(name)
```

```
ai_exam.py > bump_version
4 def bump_version(name: str) -> str:
31
32 # Regex to find a version suffix like '_v1', '_v99', etc., at the end of the base name
33 match = re.search(r'_v(\d+)$', base)
34
35 if match:
36     # If a version suffix is found, increment it
37     version = int(match.group(1))
38     new_version = version + 1
39     # Reconstruct the base name with the new version
40     new_base = base[:match.start()] + f'_v{new_version}'
41 else:
42     # If no version suffix is found, add '_v1'
43     new_base = f'{base}_v1'
44
45 # Return the new base name combined with the original extension
46 return f'{new_base}{ext}'
47
48 # --- Example Usage ---
49 if __name__ == '__main__':
50     filenames = [
51         "report.docx",
52         "photo_v1.jpg",
53         "archive_v9.zip",
54         "document.with.dots.pdf",
55         "script_v12.py",
56         "data_file", # No extension
57         "old_data_file_v5" # No extension
58     ]
59
```

```
ai_exam.py > bump_version
59
60 print("Bumping versions for a list of files:")
61 for filename in filenames:
62     new_filename = bump_version(filename)
63     print(f'{filename}' -> '{new_filename}')
64
65 # You can also run doctests to verify the examples in the docstring
66 import doctest
67 print("\nRunning doctests...")
68 doctest.testmod()
69
70
```

OUTPUT:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\DELL\Desktop\vs code\.vscode> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/ai_exam.py"
Bumping versions for a list of files:
'report.docx' -> 'report_v1.docx'
'photo_v1.jpg' -> 'photo_v2.jpg'
'archive_v9.zip' -> 'archive_v10.zip'
'document.with.dots.pdf' -> 'document.with.dots_v1.pdf'
'script_v12.py' -> 'script_v13.py'
'data_file' -> 'data_file_v1'
'old_data_file_v5' -> 'old_data_file_v6'
```

OBSERVATIONS:

Of course! As Gemini Code Assist, I'd be happy to provide my observations on the Python code you've shared.

The provided `bump_version` function is very well-written. It's clear, robust, and follows good Python practices. Here are my detailed observations:

Strengths

- 1. Robustness with `os.path.splitext`:** The use of `os.path.splitext` is excellent. It correctly handles filenames with multiple dots (e.g., `archive.tar.gz`) and those with no extension at all, which is a common edge case that is often missed.
- 2. Precise Regular Expression:** The regex `r'_v(\d+)'` is perfectly crafted for this task.
 - The `$` anchor is crucial as it ensures the version suffix is matched only at the very end of the filename's base. This prevents incorrect matches in filenames like `my_v2_archive_final.zip`.
 - Capturing the digits with `(\d+)` is efficient for extraction and conversion to an integer.
- 3. Clarity and Readability:** The code is easy to follow. The variable names (`base`, `ext`, `match`, `new_version`) are descriptive, and the logic flows naturally through the `if/else` block.
- 4. Excellent Documentation:** The docstring is a model of good documentation. It clearly explains what the function does, its arguments, and what it returns. The inclusion of `Examples` that can be run with `doctest` is a fantastic practice for ensuring the function remains correct over time.
- 5. Self-Contained and Testable:** The `if __name__ == '__main__':` block makes the script directly runnable. It provides clear example usage and

Prompts to try

- 4. Excellent Documentation:** The docstring is a model of good documentation. It clearly explains what the function does, its arguments, and what it returns. The inclusion of `Examples` that can be run with `doctest` is a fantastic practice for ensuring the function remains correct over time.
- 5. Self-Contained and Testable:** The `if __name__ == '__main__':` block makes the script directly runnable. It provides clear example usage and automatically runs the embedded `doctest` examples, which is a great way to combine demonstration and testing.

TASK:2

Scenario (sports analytics):

Context:

Data analysts in sports analytics normalize metrics to [0,1] for comparability.

Your Task:

Add Google-style docstrings and handle the edge-case where all scores are equal (avoid divide-by-zero).

Data & Edge Cases:

Empty lists return empty; if `max==min`, return zeros of the same length.

AI Assistance Expectation:

Use AI to draft docstrings with Args/Returns/Examples and generate unit tests for edge-cases.

Constraints & Notes:

Add tests demonstrating the `m==n` case.

Sample Input

`def normalize(scores):`

`m = max(scores); n = min(scores)`

`return [(x-n)/(m-n) for x in scores]`

Sample Output

Docstring includes Args/Returns/Examples; guard for `m==n`

Acceptance Criteria: Doc quality and guard confirmed by tests.

PROMPT:

Write a Python function `normalize(scores: list[float]) -> list[float]` that normalizes a list of numbers into the range [0, 1].

Requirements:

Use Google-style docstrings with Args, Returns, and Examples.

Edge-case handling:

If `scores` is empty, return an empty list.

If all values are equal (`max == min`), return a list of zeros of the same length.

Otherwise, normalize using:

`normalized`

```
x
)
=
x
-
min
max
-
min
normalized(x)=
max-min
x-min
```

1 context item

Agent Preview

CODE:

```
ai_exam.py > normalize
1  from typing import List
2
3  def normalize(scores: List[float]) -> List[float]:
4      """Normalizes a list of numbers to a scale from 0 to 1.
5
6      This is also known as min-max scaling. The formula used is:
7      normalized(x) = (x - min) / (max - min)
8
9      Args:
10         scores: A list of floating-point numbers to be normalized.
11
12      Returns:
13         A new list containing the normalized scores. Returns an empty
14         list if the input is empty. Returns a list of zeros if all
15         input scores are the same.
16
17      Examples:
18         >>> normalize([1, 2, 3, 4, 5])
19         [0.0, 0.25, 0.5, 0.75, 1.0]
20         >>> normalize([10, 20, 30])
21         [0.0, 0.5, 1.0]
22         >>> normalize([5, 5, 5])
23         [0.0, 0.0, 0.0]
24         >>> normalize([])
25         []
26         >>> normalize([-5, 0, 5])
27         [0.0, 0.5, 1.0]
28
29         # Edge case: Handle empty list
30         if not scores:
```

```

ai_exam.py > normalize
3 def normalize(scores: List[float]) -> List[float]:
30     if not scores:
31         return []
32
33     min_score = min(scores)
34     max_score = max(scores)
35
36     # Edge case: All values are the same, avoid division by zero
37     if min_score == max_score:
38         return [0.0] * len(scores)
39
40     # Normalize the scores using a list comprehension
41     denominator = max_score - min_score
42     return [(score - min_score) / denominator for score in scores]
43
44 # --- Example Usage ---
45 if __name__ == '__main__':
46     # Standard case
47     my_scores = [100, 150, 200, 250, 300]
48     normalized_scores = normalize(my_scores)
49     print(f"Original: {my_scores}")
50     print(f"Normalized: {normalized_scores}\n")
51
52     # Case with all equal values
53     equal_scores = [75, 75, 75]
54     normalized_equal = normalize(equal_scores)
55     print(f"Original: {equal_scores}")
56     print(f"Normalized: {normalized_equal}\n")
57
58     # Empty list case

```

```

ai_exam.py > normalize
57
58     # Empty list case
59     empty_list = []
60     normalized_empty = normalize(empty_list)
61     print(f"Original: {empty_list}")
62     print(f"Normalized: {normalized_empty}\n")
63
64     # Run doctests to verify examples in the docstring
65     import doctest
66     print("Running doctests...")
67     doctest.testmod()
68
69

```

OUTPUT:

```

PS C:\Users\DELL\Desktop\vs_code\.vscode>
> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs
code/.vscode/ai_exam.py"
Original: [100, 150, 200, 250, 300]
Normalized: [0.0, 0.25, 0.5, 0.75, 1.0]

Original: [75, 75, 75]
Normalized: [0.0, 0.0, 0.0]

Original: []
Normalized: []

```

OBSERVATIONS:

This function is a high-quality implementation of min-max scaling. It adheres to best practices and handles important edge cases gracefully.

Strengths

1. **Correctness and Robustness:** The function correctly implements the min-max normalization formula. Crucially, it handles the two most important edge cases:
 - **Empty List:** The `if not scores:` check is an efficient and Pythonic way to handle empty inputs, preventing errors in subsequent lines.
 - **Uniform Data:** The `if min_score == max_score:` check correctly identifies lists where all elements are the same. This prevents a `ZeroDivisionError` and correctly returns a list of zeros, which is the logical outcome for data with no variance.
2. **Readability and Style:** The code is very clean and easy to understand.
 - Variable names like `min_score`, `max_score`, and `denominator` are descriptive and unambiguous.
 - The use of a list comprehension (`[...] for score in scores`) is a concise and efficient way to perform the transformation.
3. **Excellent Documentation:** The Google-style docstring is exemplary. It clearly defines the function's purpose, arguments, and return value. The inclusion of `Examples` that can be verified with `doctest` is a fantastic practice that makes the function self-testing and easy to understand.
4. **Type Hinting:** The use of `List[float]` from the `typing` module makes the function's contract explicit, improving code clarity and allowing static analysis tools to catch potential errors.

Prompts to try

analysis tools to catch potential errors.

5. **Demonstration and Testing:** The `if __name__ == '__main__':` block is perfect. It not only demonstrates how to use the function with various inputs but also automatically runs the doctests, ensuring the code works as documented.

-----THE END-----