Table of Contents

## 1 FunctionDef count\_vowels(text)

**count\_vowels**: The function of count\_vowels is to count the total number of vowels within a given string in a case-insensitive manner. **parameters**: · text: The input string to be scanned for vowels. This parameter is of type str. **Code Description**: The function begins by defining a set named vowels which contains all the English vowels in both lowercase (‘a’, ‘e’, ‘i’, ‘o’, ‘u’) and uppercase (‘A’, ‘E’, ‘I’, ‘O’, ‘U’). Using a set data structure is highly efficient for checking the existence of an element, as it provides an average time complexity of O(1) for membership tests. The function then iterates through each character (ch) in the input text string. For each character, it checks if ch is present in the vowels set. If the character is a vowel, a generator expression yields the value 1. The built-in sum() function is then called on this generator. It aggregates all the yielded 1s, effectively summing them up to produce the total count of vowels found in the string. The function returns this final sum as an integer. **Note**: The vowel counting is case-insensitive because the vowels set includes both uppercase and lowercase letters. Any characters in the input string that are not vowels (such as consonants, numbers, punctuation, or whitespace) are simply ignored and do not contribute to the count. **Output Example**:

vowel\_count = count\_vowels("This is an Example String.")  
print(vowel\_count)  
  
# Expected output:  
# 7

## 2 FunctionDef pairwise\_sum(numbers)

**pairwise\_sum**: The function of pairwise\_sum is to compute the arithmetic sum of a sequence of numbers using a numerically stable algorithm to maintain precision.

**parameters**: · numbers: An iterable collection of numbers, such as floats or integers, that are to be summed.

**Code Description**: The function implements the Kahan summation algorithm, a method designed to minimize the accumulation of floating-point errors when adding a sequence of numbers. It initializes two floating-point variables: total to store the running sum, and compensation to track the accumulated error from previous operations. The function iterates through each value in the input numbers iterable. Inside the loop, the current value is first converted to a float. An adjusted value y is calculated by subtracting the compensation from the previous iteration. This step corrects the current number with the error lost in the prior addition. Next, this adjusted value y is added to the current total, and the result is stored in a temporary variable t. The compensation for the next iteration is then calculated as (t - total) - y. This expression precisely captures the low-order bits that were lost during the addition of total + y. Finally, the total is updated to the new sum t. After iterating through all the numbers, the function returns the final total.

**Note**: This function is particularly useful for financial or scientific calculations where high precision is critical. It provides a more accurate result than a simple iterative addition or the standard sum() function when dealing with floating-point numbers, as it mitigates the problem of round-off error accumulation, especially for datasets with a large number of values or values of widely different magnitudes.

**Output Example**: 10005.85987 ## FunctionDef split\_into\_chunks(text, size) **split\_into\_chunks**: The function of split\_into\_chunks is to divide a given string into a sequence of smaller strings of a specified fixed length.

**parameters**: · text: A string (str) that needs to be split into chunks. · size: An integer (int) representing the desired length of each chunk. This value must be a positive number.

**Code Description**: The function split\_into\_chunks is designed to segment a string into multiple substrings, or “chunks”. It accepts two arguments: the text to be processed and the size for each chunk.

Initially, the function performs a validation check on the size parameter. If the provided size is zero or a negative number, it raises a ValueError with the message “size must be positive”, preventing invalid operations.

If the size is valid (i.e., positive), the function proceeds to slice the string. It uses a generator expression combined with the range() function to iterate through the text string at intervals equal to the size. For each step, it extracts a slice of the string from the current index i up to i + size. This process continues until the entire string has been traversed. The slicing mechanism inherently handles cases where the total length of the string is not perfectly divisible by size; in such scenarios, the final chunk will simply contain the remaining characters and will be shorter than the specified size.

Finally, all the generated string chunks are collected and returned as a tuple of strings.

**Note**: The size parameter must be a positive integer greater than zero. Providing a non-positive value will result in a ValueError. The last element in the returned tuple may have a length less than the specified size if the length of the input text is not a multiple of size.

**Output Example**: (‘Hel’, ‘loW’, ‘orl’, ‘d’)