Table of Contents

## 0.1 FunctionDef count\_vowels(text)

# 1 Function: count\_vowels(text: str)

## 1.1 Overview

The count\_vowels function counts the total number of vowels within a given input string in a case-insensitive manner.

## 1.2 parameters

* text: (str) The input string in which to count the vowels.

## 1.3 Description

This function provides an efficient way to determine the number of vowels (a, e, i, o, u) in a string.

The logic operates as follows: 1. A set named vowels is initialized with both lowercase and uppercase vowels: set("aeiouAEIOU"). Using a set allows for very fast membership checking (average O(1) time complexity). 2. The function then iterates through each character (ch) of the input text string. 3. A generator expression, (1 for ch in text if ch in vowels), is used. For each character ch in the text, it checks if the character is present in the vowels set. If it is, the expression yields the number 1. 4. The built-in sum() function is called on this generator. It consumes the generated 1s and adds them together, producing a final integer total that represents the count of all vowels found. 5. This final sum is returned as the result.

# Internal logic breakdown  
vowels = set("aeiouAEIOU")  
text = "Example"  
# The generator will yield 1 for 'E', 'a', and 'e'  
# sum() will calculate 1 + 1 + 1  
result = sum(1 for ch in text if ch in vowels)  
# result will be 3

## 1.4 Usage Notes

* **Case-Insensitive**: The function is case-insensitive by design, as the vowels set contains both ‘a’ and ‘A’, ‘e’ and ‘E’, and so on.
* **Non-Vowel Characters**: Any characters that are not vowels, including consonants, numbers, whitespace, and symbols, are ignored and not included in the count.
* **Return Value**: The function always returns an integer (int). If no vowels are found or the input string is empty, it will return 0.

**Output Example**: The function returns a single integer representing the total count of vowels.

## 1.5 Example

# Example usage  
input\_string = "Hello World! This is a test."  
vowel\_count = count\_vowels(input\_string)  
print(vowel\_count)

**Output:**

7

## 1.6 FunctionDef pairwise\_sum(numbers)

# 2 Function: pairwise\_sum(numbers: Iterable[float])

## 2.1 Overview

The pairwise\_sum function computes the arithmetic sum of an iterable of numbers using the Kahan summation algorithm to minimize numerical errors that can occur with standard floating-point arithmetic.

## 2.2 parameters

* **numbers** (Iterable[float]): An iterable collection of numbers (floats or integers) to be summed.

## 2.3 Description

This function provides a numerically stable method for summing floating-point numbers, which is crucial when the numbers vary widely in magnitude or when summing a large quantity of values. Standard summation can lead to a loss of precision as small values are added to a large running total.

The pairwise\_sum function implements the Kahan summation algorithm to counteract this problem. It maintains a running total and a compensation variable that accumulates the error from previous additions.

The process for each value in the input numbers is as follows: 1. The input value is first cast to a float. 2. A corrected value y is calculated by subtracting the compensation (the error from the previous step) from the current value. 3. This corrected y is added to the running total, and the result is stored in a temporary variable t. This addition may still suffer from precision loss. 4. The new error is calculated as (t - total) - y. This captures the low-order bits that were lost during the summation of total + y. This new error is stored in the compensation variable for the next iteration. 5. The total is updated to the new sum t.

By repeatedly carrying forward the rounding error, the algorithm ensures that the final sum is significantly more accurate than a naive summation.

## 2.4 Usage Notes

* This function is more computationally intensive than Python’s built-in sum() but offers superior precision.
* It is highly recommended for scientific and financial calculations where accuracy is critical, especially with datasets containing a large number of floating-point values.
* The function internally converts all numbers to float, so it can accept iterables containing integers as well.

**Output Example**: A single floating-point number representing the accurate sum.

1000.0

## 2.5 Example

The following example demonstrates the precision advantage of pairwise\_sum over the standard sum() function. When summing a large number, a small number, and the negation of the large number, a naive sum can lose the small number due to floating-point limitations.

# A list where a naive sum would likely result in 0.0 due to precision loss  
data = [1e10, 1, -1e10] \* 1000  
  
# Using the standard sum() function  
naive\_sum = sum(data)  
print(f"Naive Sum: {naive\_sum}")  
  
# Using the pairwise\_sum for a more accurate result  
accurate\_sum = pairwise\_sum(data)  
print(f"Pairwise (Kahan) Sum: {accurate\_sum}")

**Output:**

Naive Sum: 0.0  
Pairwise (Kahan) Sum: 1000.0

## 2.6 FunctionDef split\_into\_chunks(text, size)

# 3 Function: split\_into\_chunks(text: str, size: int)

## 3.1 Overview

The split\_into\_chunks function divides a given string into a series of smaller, fixed-size substrings.

## 3.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| text | str | The input string that needs to be divided into chunks. |
| size | int | The desired length for each chunk. This value must be a positive integer. |

## 3.3 Description

This function provides a straightforward way to segment a string into multiple parts of a specified length.

The function first validates the size parameter. It checks if size is a positive integer. If size is less than or equal to zero, a ValueError is raised, as it’s impossible to split a string into chunks of non-positive length.

If the size is valid, the function proceeds to split the text. It uses a generator expression that iterates through the string with a step equal to size. The range(0, len(text), size) call generates the starting indices for each chunk (e.g., 0, size, 2\*size, etc.).

For each starting index i, a substring is extracted using Python’s slice notation: text[i : i + size]. This creates a chunk of length size. If the final slice extends beyond the string’s length, Python handles it gracefully by including all remaining characters, which may result in a final chunk that is shorter than size.

Finally, all the generated string chunks are collected into a tuple and returned.

# Internal logic for splitting 'abcdefg' with size 3  
# 1. range(0, 7, 3) produces indices 0, 3, 6  
# 2. Slice at index 0: text[0:3] -> 'abc'  
# 3. Slice at index 3: text[3:6] -> 'def'  
# 4. Slice at index 6: text[6:9] -> 'g' (shorter than size)  
# 5. Result is collected into a tuple: ('abc', 'def', 'g')

## 3.4 Usage Notes

* The size parameter must be a positive integer. Providing 0 or a negative number will raise a ValueError.
* The last chunk in the returned tuple may be shorter than the specified size if the total length of the text is not an exact multiple of size.
* The function returns a tuple of strings, not a list.

**Output Example**: A tuple containing string chunks. ('chunk1', 'chunk2', 'chnk3')

## 3.5 Example

# Example usage  
input\_string = "This is a sample string to be split."  
chunk\_size = 10  
result = split\_into\_chunks(input\_string, chunk\_size)  
print(result)  
  
# Example with a string length that is not a multiple of the chunk size  
another\_string = "HelloWorld"  
short\_chunk\_size = 4  
result\_short = split\_into\_chunks(another\_string, short\_chunk\_size)  
print(result\_short)

**Output:**

('This is a ', 'sample str', 'ing to be ', 'split.')  
('Hell', 'oWor', 'ld')