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 (a, e, i, o, u) within a given string, ignoring case.

## 1.2 parameters

* text: str - The input string to be scanned for vowels.

## 1.3 Description

This function provides an efficient way to determine the number of vowels in a text string.

The core logic begins by defining a set named vowels containing both lowercase and uppercase vowels ("aeiouAEIOU"). Using a set is highly efficient for membership testing, as it provides an average time complexity of O(1) for checking if a character exists within it.

The function then employs a generator expression, (1 for ch in text if ch in vowels), to iterate through each character (ch) of the input text. For each character, it checks if ch is present in the vowels set. If it is, the generator yields the number 1.

Finally, the built-in sum() function is called on this generator. It consumes all the yielded 1s and adds them together, producing a final integer count of all the vowels found in the string. This sum is then returned as the result.

# Internal logic breakdown  
vowels = set("aeiouAEIOU")  
# For an input "Hello", the generator would yield 1 for 'e' and 1 for 'o'.  
# sum() would then calculate 1 + 1 = 2.  
return sum(1 for ch in text if ch in vowels)

## 1.4 Usage Notes

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

**Output Example**: A possible return value for a given string.

5

## 1.5 Example

# Example usage  
input\_sentence = "This is a simple Test Sentence."  
vowel\_count = count\_vowels(input\_sentence)  
print(f"The input sentence is: '{input\_sentence}'")  
print(f"The number of vowels is: {vowel\_count}")

**Output:**

The input sentence is: 'This is a simple Test Sentence.'  
The number of vowels is: 8

## 1.6 FunctionDef pairwise\_sum(numbers)

# 2 Function: pairwise\_sum

## 2.1 Overview

The pairwise\_sum function computes the arithmetic sum of an iterable of numbers using the Kahan summation algorithm to provide a more numerically stable and precise result.

## 2.2 parameters

* numbers (Iterable[float]): An iterable collection of numbers, such as a list or tuple of floats or integers, that will be summed.

## 2.3 Description

This function provides a robust method for summing floating-point numbers, significantly reducing the numerical error that can accumulate with standard summation methods. Standard addition of floating-point numbers can lead to a loss of precision, especially when adding a very small number to a very large one.

The pairwise\_sum function implements the Kahan summation algorithm to counteract this issue. It maintains a running compensation variable to account for the low-order bits that are typically lost during addition.

The process for each value in the input numbers is as follows: 1. The value is first corrected by subtracting the compensation from the previous iteration. This corrected value is stored in y. 2. The corrected value y is added to the running total. This operation may still result in some precision loss. 3. The error from the addition in the previous step is calculated as (t - total) - y and stored in the compensation variable for the next iteration. 4. The total is updated with the new sum t.

By repeatedly carrying over the “lost” part of the sum, the algorithm ensures that the final total is much more accurate than a naive summation.

# Inside the function  
total = 0.0  
compensation = 0.0  
for value in numbers:  
 y = float(value) - compensation # Correct the value  
 t = total + y # Add to the running total  
 compensation = (t - total) - y # Calculate the error (what was lost)  
 total = t # Update the total

## 2.4 Usage Notes

* This function is particularly useful when summing a large set of floating-point numbers or when the numbers have widely different magnitudes, as it minimizes cumulative floating-point errors.
* It is more accurate than Python’s built-in sum() for floating-point arithmetic under such conditions.
* The input numbers must be an iterable.
* Any integers in the input iterable will be automatically cast to floats during the computation.

**Output Example**: The function returns a single floating-point number representing the accurate sum.

## 2.5 Example

# A standard sum can lose precision when adding very small numbers to a large one.  
# The small numbers get rounded away.  
data = [1.0] + [1e-16] \* 1000000  
  
# Using the built-in sum()  
naive\_sum = sum(data)  
print(f"Naive Sum: {naive\_sum}")  
  
# Using pairwise\_sum for better precision  
accurate\_sum = pairwise\_sum(data)  
print(f"Pairwise (Kahan) Sum: {accurate\_sum}")

**Output:**

Naive Sum: 1.0  
Pairwise (Kahan) Sum: 1.0000000000001

## 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, returned as a tuple.

## 3.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| text | str | The input string that needs to be divided into chunks. |
| size | int | The desired maximum 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. If size is less than or equal to zero, it raises a ValueError, as chunking is not possible with a non-positive length.

The core logic uses a generator expression with range() to iterate through the input text. The range is configured to start at index 0, end at the length of the string (len(text)), and step by the given size. This process generates the starting index i for each chunk.

For each starting index i, a slice of the string text[i : i + size] is created. This slice represents one chunk. Because the slicing operation handles boundaries gracefully, the final chunk will automatically be shorter than size if the total string length is not evenly divisible by size.

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

# The core logic is equivalent to this generator expression  
(text[i : i + size] for i in range(0, len(text), size))

## 3.4 Usage Notes

* The size parameter must be a positive integer. Providing 0 or a negative number will result in a ValueError.
* The last chunk in the returned tuple may be shorter than the specified size if the length of the input text is not a multiple of size.
* If the input text is an empty string, the function will return an empty tuple ().
* The function always returns a tuple of strings.

**Output Example**: A typical return value for a string “hello world” with a size of 4 would look like this: ('hell', 'o wo', 'rld')

## 3.5 Example

# Example usage  
long\_string = "This is a sample string to demonstrate the chunking functionality."  
chunk\_size = 10  
  
# Split the string into chunks of 10 characters  
chunks = split\_into\_chunks(long\_string, chunk\_size)  
print(chunks)  
  
# Example with a string length not divisible by the chunk size  
short\_string = "abcdefgh"  
short\_chunk\_size = 3  
short\_chunks = split\_into\_chunks(short\_string, short\_chunk\_size)  
print(short\_chunks)

**Output:**

('This is a ', 'sample str', 'ing to dem', 'onstrate t', 'he chunkin', 'g function', 'ality.')  
('abc', 'def', 'gh')