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 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 a straightforward way to determine the number of vowels (a, e, i, o, u) in any given text.

The core logic begins by defining a set named vowels which contains both lowercase ("aeiou") and uppercase ("AEIOU") vowels. Using a set is highly efficient for checking if a character is a vowel, as membership tests are very fast.

The function then iterates through each character (ch) of the input text. For each character, it checks if ch is present in the vowels set. A generator expression, (1 for ch in text if ch in vowels), yields the number 1 for every character that is a vowel.

Finally, the built-in sum() function is used to add up all the 1s generated, effectively tallying the total count of vowels. The resulting integer sum is then returned.

# 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 handles both uppercase and lowercase vowels equally, so ‘A’ is counted the same as ‘a’.
* **Non-Vowel Characters**: Any characters that are not vowels, including consonants, numbers, whitespace, and symbols, are ignored and not included in the count.
* **Efficiency**: The use of a set for vowel lookup provides excellent performance, with an average time complexity of O(1) for each character check.

**Output Example**: A possible return value for an input string.

11

## 1.5 Example

# Example usage  
text\_to\_scan = "This is an Example of the Vowel Counter!"  
vowel\_count = count\_vowels(text\_to\_scan)  
print(f"The text is: '{text\_to\_scan}'")  
print(f"The number of vowels is: {vowel\_count}")

**Output:**

The text is: 'This is an Example of the Vowel Counter!'  
The number of vowels is: 13

## 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 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, to be summed. Elements can be floats or integers, as they are internally cast to float.

## 2.3 Description

This function is designed to minimize the floating-point errors that can accumulate when summing a sequence of numbers, especially when the numbers vary greatly in magnitude. Standard summation can lose precision when a small number is added to a large running total.

The function implements the Kahan summation algorithm to counteract this effect. It maintains a running total and a compensation variable, which tracks the “lost” low-order bits from previous additions.

The process for each number in the input numbers is as follows: 1. The current value is first corrected by subtracting the compensation from the previous iteration. This creates a corrected value y. 2. This corrected value y is added to the running total, and the result is stored in a temporary variable t. This is the step where precision loss can occur. 3. The error from the addition is calculated and stored in the compensation variable for the next iteration. The calculation (t - total) - y isolates the low-order bits that were lost during the total + y operation. 4. The main total is updated with the value of t.

By carrying the round-off error from each step into the next, the algorithm produces a final sum that is significantly more accurate than a naive summation.

# Kahan summation algorithm implementation  
total = 0.0  
compensation = 0.0  
for value in numbers:  
 y = float(value) - compensation  
 t = total + y  
 compensation = (t - total) - y  
 total = t

## 2.4 Usage Notes

* This function is highly recommended when summing a large quantity of floating-point numbers or when the dataset contains values with a wide range of magnitudes.
* It provides greater precision compared to Python’s built-in sum() function for floating-point arithmetic in edge cases.
* While the type hint is Iterable[float], the function will correctly handle iterables containing integers by casting them to float.

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

5.85987

## 2.5 Example

# Example demonstrating precision with large and small numbers  
# The expected sum is 3.14159 + 2.71828 = 5.85987  
# A naive sum might result in floating-point inaccuracies.  
data = [1e10, 3.14159, -1e10, 2.71828]  
result = pairwise\_sum(data)  
print(result)  
  
# Another example with many small numbers  
small\_numbers = [0.1] \* 10  
result\_small = pairwise\_sum(small\_numbers)  
print(result\_small)

**Output:**

5.85987  
0.9999999999999999

## 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 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 zero or a negative number, it raises a ValueError because it’s impossible to create chunks of non-positive length.

If the size is valid, the function proceeds to iterate through the input text using a generator expression. It generates start indices for each chunk using range(0, len(text), size), which steps through the string’s indices by the given size. For each start index i, it slices the string from that index to i + size.

tuple(text[i : i + size] for i in range(0, len(text), size))

This slicing mechanism automatically handles the last chunk; if the remaining string length is less than size, the slice will simply include all remaining characters. The resulting substrings are then collected into a tuple and returned.

## 3.4 Usage Notes

* The size parameter must be a positive integer. Providing a value of 0 or less will result in a ValueError.
* The final chunk in the returned tuple may be shorter than the specified size if the length of the input text is not perfectly divisible by size.
* The function returns a tuple of strings. If the input text is empty, an empty tuple () is returned.

**Output Example**: A tuple containing string chunks.

('This is a ', 'sample str', 'ing.')

## 3.5 Example

# Example usage  
input\_string = "This is a sample string for demonstration."  
chunk\_size = 10  
  
try:  
 result = split\_into\_chunks(input\_string, chunk\_size)  
 print(result)  
except ValueError as e:  
 print(e)  
  
# Example with a string length not divisible by chunk size  
short\_string = "abcdefg"  
result\_short = split\_into\_chunks(short\_string, 3)  
print(result\_short)  
  
# Example of an invalid size  
try:  
 split\_into\_chunks("some text", 0)  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

('This is a ', 'sample str', 'ing for de', 'monstratio', 'n.')  
('abc', 'def', 'g')  
Error: size must be positive