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 to be scanned for 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 operates in two main steps: 1. A set named vowels is initialized with both lowercase and uppercase vowels: set("aeiouAEIOU"). Using a set allows for highly efficient, constant-time lookups (average O(1)) to check if a character is a vowel. 2. The function then iterates through each character (ch) in the input text. It uses a generator expression (1 for ch in text if ch in vowels) which yields the number 1 for every character that is found within the vowels set. 3. Finally, the built-in sum() function is used to add up all the 1s produced by the generator, resulting in the total count of vowels.

# Internal logic breakdown  
vowels = set("aeiouAEIOU")  
# For a text "Example", the generator would yield 1 for 'E', 1 for 'a', and 1 for 'e'.  
# sum() would then calculate 1 + 1 + 1 = 3.  
return sum(1 for ch in text if ch in vowels)

## 1.4 Usage Notes

* The function is case-insensitive. It will correctly count both ‘a’ and ‘A’ as vowels.
* Characters that are not vowels, such as consonants, numbers, whitespace, and symbols, are ignored and not included in the count.
* The function returns an integer 0 if the input string is empty or contains no vowels.

**Output Example**: A non-negative integer representing the total vowel count.

## 1.5 Example

# Example usage  
sample\_string = "This is a Test String for Counting Vowels!"  
vowel\_count = count\_vowels(sample\_string)  
print(f"The string is: '{sample\_string}'")  
print(f"The number of vowels is: {vowel\_count}")

**Output:**

The string is: 'This is a Test String for Counting Vowels!'  
The number of vowels is: 11

## 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 maintain high numerical precision.

## 2.2 parameters

* **numbers** (Iterable[float]): An iterable (like a list, tuple, or generator) containing the numbers to be summed. The function can handle both floats and integers, as it internally casts all values to float.

## 2.3 Description

This function provides a numerically stable method for summing a sequence of floating-point numbers. Standard summation, like total += value, can suffer from precision loss, especially when adding numbers of vastly different magnitudes or when summing a large quantity of numbers. This is because the lower-order bits of smaller numbers can be lost when they are added to a much larger running total.

The pairwise\_sum function implements the Kahan summation algorithm to mitigate this issue. It works by maintaining a separate variable, compensation, to accumulate the round-off error from each addition.

The logic proceeds as follows for each value in the input numbers: 1. y = float(value) - compensation: The error from the previous addition (compensation) is subtracted from the current value. This creates a corrected value y. 2. t = total + y: The corrected value y is added to the running total. This addition may still introduce a small floating-point error. 3. compensation = (t - total) - y: This is the core of the algorithm. It calculates the new round-off error. In an ideal scenario, (t - total) would be exactly equal to y, making compensation zero. However, due to floating-point limitations, (t - total) is what was *actually* added. The difference between what was actually added and what was *intended* to be added (y) is the new error, which is stored in compensation. 4. total = t: The running total is updated.

By carrying the “lost” part of the sum from one iteration to the next, the algorithm ensures that the final total is significantly more accurate than a naive summation.

## 2.4 Usage Notes

* This function is more computationally intensive than Python’s built-in sum(). It is best used in scenarios where numerical precision is critical, such as in scientific or financial calculations involving a wide range of number magnitudes.
* The function internally casts all input values to float, so an iterable containing integers or a mix of integers and floats is acceptable.
* For maximum precision with floating-point numbers in Python, consider also math.fsum(), which implements a similar algorithm.

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

6.14

## 2.5 Example

# Example demonstrating precision with large and small numbers  
# A naive sum might lose precision when the small numbers are added to the large one.  
numbers\_to\_sum = [1e10, 1, -1e10, 2, 3.14]  
  
# Using the pairwise\_sum function  
accurate\_sum = pairwise\_sum(numbers\_to\_sum)  
print(f"Accurate sum: {accurate\_sum}")  
  
# For comparison, a naive sum  
naive\_sum = sum(numbers\_to\_sum)  
print(f"Naive sum: {naive\_sum}")

**Output:**

Accurate sum: 6.14  
Naive sum: 6.140000000000114

## 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. If size is zero or a negative number, it raises a ValueError because a chunk length must be a positive value.

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 by using range(0, len(text), size), which creates a sequence of numbers starting from 0 up to the length of the text, incrementing by the size value at each step.

For each start index i, a substring is extracted from the text using a slice: text[i : i + size]. This slice creates a chunk of length size. If the remaining part of the string is shorter than size, the final slice will simply include all remaining characters, resulting in a final chunk that is shorter than the specified size.

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

# The core logic uses a generator expression and tuple conversion  
tuple(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 integer will result in a ValueError.
* The function returns a tuple of strings, not a list.
* The last chunk in the returned tuple may be shorter than the specified size if the total length of the input text is not an even multiple of size.

**Output Example**: A possible return value for a string of length 25 and a chunk size of 10.

('0123456789', '0123456789', '01234')

## 3.5 Example

# Example usage  
long\_string = "This is a sample string for demonstrating the chunking function."  
chunk\_size = 10  
  
# Split the string into chunks of 10 characters  
chunks = split\_into\_chunks(long\_string, chunk\_size)  
print(chunks)

**Output:**

('This is a ', 'sample str', 'ing for de', 'monstratin', 'g the chun', 'king funct', 'ion.')