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 vowels.

## 1.3 Description

This function provides a straightforward way to determine the vowel count in any given text.

The core logic begins by defining a set named vowels which contains all lowercase and uppercase English vowels (a, e, i, o, u). Using a set is highly efficient for membership testing, which is the primary operation performed.

The function then iterates through each character (ch) of the input text using a generator expression: (1 for ch in text if ch in vowels). For every character, it checks if the character exists within the vowels set. If a character is found in the set, 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 the total count of vowels found in the string. This final sum is the integer value returned by the function.

# Internal logic breakdown  
vowels = set("aeiouAEIOU")  
# For a text like "Hi", the generator would be (1 for 'H' if 'H' in vowels, 1 for 'i' if 'i' in vowels)  
# This evaluates to a generator that yields just one value: 1  
# sum() then calculates the total, which is 1.

## 1.4 Usage Notes

* The function is case-insensitive. It will count both ‘a’ and ‘A’ as vowels.
* Characters that are not vowels, including consonants, numbers, whitespace, and symbols, are ignored.
* The letter ‘y’ is not considered a vowel by this function.

**Output Example**: The function returns an integer representing the total count of vowels. For the input "Hello World", the return value would be 3.

## 1.5 Example

# Example usage  
input\_string = "Programming is fun and rewarding!"  
vowel\_count = count\_vowels(input\_string)  
print(f"The number of vowels in '{input\_string}' is: {vowel\_count}")

**Output:**

The number of vowels in 'Programming is fun and rewarding!' is: 8

## 1.6 FunctionDef pairwise\_sum(numbers)

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

## 2.1 Overview

The pairwise\_sum function computes the sum of a sequence of numbers using a numerically stable algorithm to minimize floating-point errors.

## 2.2 parameters

* **numbers** (Iterable[float]): An iterable (like a list, tuple, or generator) containing the numbers to be summed. The elements can be floats or integers, as they will be converted to floats.

## 2.3 Description

This function provides a more precise method for summing floating-point numbers compared to a standard iterative addition. It implements the Kahan summation algorithm, which is designed to reduce the accumulation of rounding errors that can occur in floating-point arithmetic. This is especially important when summing many numbers or when the numbers vary greatly in magnitude.

The algorithm works by maintaining a running compensation variable that accumulates the error from each addition.

1. The function initializes a total sum and a compensation value to 0.0.
2. It iterates through each value in the input numbers.
3. For each value, it first creates a corrected value y by subtracting the compensation from the previous step. This reintroduces the “lost” part of the previous number into the current calculation. y = float(value) - compensation
4. It then adds this corrected value y to the running total, storing it in a temporary variable t. t = total + y
5. The core of the algorithm is calculating the new compensation value. This is the numerical error that occurred in the previous step. It’s calculated as (t - total) - y. This isolates the low-order bits that were lost during the addition of y to total.
6. The total is updated to the new sum t.
7. This process repeats for all numbers, ensuring that the error from each step is carried over and corrected in the next, leading to a significantly more accurate final sum.

## 2.4 Usage Notes

* This function is highly recommended over the standard sum() when precision is critical, such as in scientific, financial, or statistical computations.
* It is particularly effective when summing a large list of numbers or when adding very small numbers to a very large running total.
* The input iterable can contain integers; they will be automatically cast to float during the calculation.

**Output Example**: The function returns a single float value representing the precise sum.

1.0

## 2.5 Example

The following example demonstrates the precision difference between Python’s built-in sum() and pairwise\_sum when summing a list of floating-point numbers that are subject to rounding errors.

# A list where standard summation can introduce floating-point errors  
data = [0.1] \* 10 # Represents [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]  
  
# Using the built-in sum() function  
standard\_sum = sum(data)  
print(f"Result with built-in sum(): {standard\_sum}")  
  
# Using the numerically stable pairwise\_sum function  
precise\_sum = pairwise\_sum(data)  
print(f"Result with pairwise\_sum(): {precise\_sum}")

**Output:**

Result with built-in sum(): 0.9999999999999999  
Result with pairwise\_sum(): 1.0

## 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 or “chunks”.

## 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 into non-positive lengths is not a valid operation.

If the size is valid, the function proceeds to iterate through the input text. It uses a range object that starts at index 0 and increments by the given size until it reaches the end of the string. In each iteration, it uses Python’s string slicing text[i : i + size] to extract a substring of length size. These substrings are collected into a generator, which is then converted into a tuple.

If the length of the input text is not perfectly divisible by size, the final chunk in the resulting tuple will contain the remaining characters and will be shorter than the specified size.

# The core logic uses a generator expression within a tuple constructor  
# For text="abcdefg" and size=3, the range generates indices 0, 3, 6  
# Slices will be text[0:3], text[3:6], text[6:9]  
# This results in ('abc', 'def', 'g')  
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 (> 0). Providing 0 or a negative number will result in a ValueError.
* The function returns a tuple of strings. Tuples are immutable, meaning they cannot be changed after creation.
* The last chunk in the returned tuple may be shorter than size if the total string length is not a multiple of size.

**Output Example**: A typical return value is a tuple of strings.

('chunk1', 'chunk2', 'chnk3')

## 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)

**Output:**

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