Table of Contents

## 0.1 FunctionDef count\_vowels(text)

# 1 Function: count\_vowels(text: str) -> int

## 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 a text string.

The logic begins by defining a set named vowels which contains all lowercase and uppercase English vowels ("aeiouAEIOU"). Using a set is highly efficient for checking if a character exists within it.

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

Finally, the built-in sum() function is called on this generator. It calculates the total sum of all the 1s generated, which corresponds to the total count of vowels in the string. This final sum is then returned.

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

## 1.4 Usage Notes

* The vowel counting is case-insensitive. For example, ‘a’ and ‘A’ are both counted as vowels.
* Characters that are not vowels, including consonants, numbers, whitespace, and symbols, are ignored.
* The function is designed to work with string inputs. Providing a non-string type may lead to a TypeError.

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

## 1.5 Example

# Example usage  
input\_sentence = "Hello World! This is a test."  
vowel\_count = count\_vowels(input\_sentence)  
print(f"The number of vowels is: {vowel\_count}")

**Output:**

The number of vowels is: 8

## 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 a sequence of numbers using a numerically stable algorithm to minimize floating-point errors.

## 2.2 parameters

* numbers: An iterable (e.g., a list, tuple) of numeric values (integers or floats) to be summed.

## 2.3 Description

This function provides a more accurate method for summing floating-point numbers compared to the standard built-in sum() function. It implements the Kahan summation algorithm, which is designed to reduce the accumulation of numerical errors that can occur when adding numbers of different magnitudes or when summing a large set of values.

The algorithm works by maintaining a running total and a compensation variable, which tracks the “lost” low-order bits from previous additions.

The logic proceeds as follows: 1. Initialize total and compensation to 0.0. 2. For each value in the input numbers iterable: a. The value is first corrected by subtracting the compensation from the previous iteration. This corrected value is stored in y. b. The 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. c. The new compensation value is calculated as (t - total) - y. This formula precisely captures the numerical error (the part of y that was lost) during the addition of total + y. d. The total is updated with the value of t. 3. After iterating through all numbers, the final total is returned, which is a more accurate representation of the true sum.

# Kahan summation for improved precision  
total = 0.0  
compensation = 0.0  
for value in numbers:  
 y = float(value) - compensation  
 t = total + y  
 compensation = (t - total) - y  
 total = t  
return total

## 2.4 Usage Notes

* This function is particularly useful when summing a large number of floating-point values or when the values have widely different magnitudes, as these are scenarios where standard summation is prone to significant precision loss.
* The input numbers can be any iterable, such as a list, tuple, or generator.
* All elements within the iterable will be cast to float before being added.

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

## 2.5 Example

The following example demonstrates a scenario where a naive summation fails due to floating-point limitations, but pairwise\_sum produces the correct result.

# Example usage with numbers of different magnitudes  
data = [1e10, 1.0, -1e10]  
  
# A naive sum might incorrectly evaluate to 0.0  
# (1e10 + 1.0) can be rounded to 1e10, then subtracting 1e10 results in 0.0  
naive\_result = sum(data)  
print(f"Naive sum result: {naive\_result}")  
  
# The pairwise\_sum function correctly computes the sum  
accurate\_result = pairwise\_sum(data)  
print(f"Accurate sum result: {accurate\_result}")

**Output:**

Naive sum result: 0.0  
Accurate sum result: 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, 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 zero or a negative number, it raises a ValueError, as chunking a string into non-positive lengths is not a valid operation.

If the size is valid, the function proceeds to iterate through the input text using a generator expression. It uses range(0, len(text), size) to generate the starting indices for each chunk. For each starting index i, it slices the string from that index up to i + size. This process continues until the entire string has been processed. The resulting substrings are then collected into a tuple.

It is important to note that if the length of the input text is not perfectly divisible by size, the final chunk in the returned tuple will be shorter than the specified size.

# The core logic uses a generator expression and tuple conversion  
return 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 string in the returned tuple may have a length less than size if the total string length is not a multiple of size.

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

## 3.5 Example

# Example usage  
long\_string = "abcdefghijklmnopqrstuvwxyz"  
chunk\_size = 5  
  
# Split the string into chunks of 5 characters  
result = split\_into\_chunks(long\_string, chunk\_size)  
print(result)  
  
# Example with a string length that is a multiple of the chunk size  
another\_string = "1234567890"  
result\_even = split\_into\_chunks(another\_string, 5)  
print(result\_even)

**Output:**

('abcde', 'fghij', 'klmno', 'pqrst', 'uvwxy', 'z')  
('12345', '67890')