Table of Contents

## 0.1 FunctionDef count\_vowels(text)

# 1 Function: count\_vowels

## 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 in a string. The core logic operates as follows:

1. A set named vowels is initialized with all lowercase and uppercase English vowels ("aeiouAEIOU"). Using a set allows for highly efficient, constant-time average complexity (O(1)) checks to see if a character is a vowel.
2. The function then iterates through each character (ch) in the input text.
3. For each character, it checks if ch is present in the vowels set.
4. A generator expression, (1 for ch in text if ch in vowels), yields the number 1 for every character that is found in the vowels set.
5. 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 in the string.

# 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

* The vowel count is case-insensitive. For example, ‘A’ and ‘a’ are both counted as vowels.
* The function only considers the five standard English vowels: a, e, i, o, u.
* Characters that are not vowels, including consonants, numbers, whitespace, and symbols, are ignored and not included in the count.

**Output Example**: The function returns a single integer representing the total number of vowels found.

## 1.5 Example

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

**Output:**

The input string is: 'Hello World! This is a test.'  
The number of vowels is: 7

## 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 ensure high numerical precision.

## 2.2 parameters

* **numbers**: Iterable[float]
  + An iterable collection of numbers (e.g., a list, tuple) containing floats or integers. Each element will be converted to a float before summation.

## 2.3 Description

This function provides a numerically stable method for summing floating-point numbers, which is crucial for avoiding precision errors that can occur with standard summation, especially when dealing with a large set of numbers or numbers of widely varying magnitudes.

Instead of a simple iterative addition, the function implements the Kahan summation algorithm. This algorithm maintains a running compensation variable to correct for the low-order bits that are lost during each addition step.

The logic proceeds as follows: 1. A total sum and a compensation value are initialized to 0.0. 2. The function iterates through each value in the input numbers. 3. For each value, it is first corrected by subtracting the compensation from the previous iteration. This corrected value is stored in y. 4. A temporary sum t is calculated by adding the current total and the corrected value y. 5. The core of the algorithm is calculating the next compensation. This is done by evaluating (t - total) - y. In perfect arithmetic, this would be zero. However, in floating-point arithmetic, this expression captures the round-off error from the addition total + y. 6. The main total is updated with the value of t.

By repeatedly carrying over the round-off error into the next iteration, the algorithm minimizes cumulative error, yielding a final sum that is significantly more accurate than a naive summation.

# 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

## 2.4 Usage Notes

* This function is more computationally intensive than the built-in sum() function but provides higher accuracy. It is recommended for scientific and financial calculations where precision is critical.
* It is particularly effective when summing a long series of numbers or when the numbers have very different magnitudes (e.g., adding a very small number to a very large one repeatedly).
* The function internally converts all input numbers to float, so an iterable of integers is also a valid input.

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

1000.0

## 2.5 Example

The following example demonstrates the precision advantage of pairwise\_sum over the standard built-in sum(). Adding 0.1 ten times with standard floating-point arithmetic results in a small precision error, which pairwise\_sum corrects.

# A list where standard summation can introduce floating-point errors  
numbers\_to\_sum = [0.1] \* 10  
  
# Using the standard sum() function  
standard\_sum = sum(numbers\_to\_sum)  
print(f"Standard sum: {standard\_sum}")  
  
# Using the numerically stable pairwise\_sum function  
stable\_sum = pairwise\_sum(numbers\_to\_sum)  
print(f"Pairwise (Kahan) sum: {stable\_sum}")

**Output:**

Standard sum: 0.9999999999999999  
Pairwise (Kahan) sum: 1.0

## 2.6 FunctionDef split\_into\_chunks(text, size)

# 3 Function: split\_into\_chunks

## 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 split 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 slice the string. It uses a generator expression combined with the range() function to iterate through the input text. The range() function is configured with a step equal to size, which effectively creates the starting index for each new chunk (0, size, 2\*size, etc.).

For each starting index i, a slice text[i : i + size] is taken. This slice extracts a substring of length size. If the final slice extends beyond the end of the string, Python’s slicing mechanism automatically handles this by taking all characters until the end. This results in the last chunk potentially being shorter than the specified size.

Finally, all generated chunks are collected into a tuple which is then returned.

# Internal logic breakdown  
def split\_into\_chunks(text: str, size: int) -> Tuple[str, ...]:  
 # 1. Validate the size parameter  
 if size <= 0:  
 raise ValueError("size must be positive")  
   
 # 2. Generate chunks using a generator expression and string slicing  
 # The range function creates indices: 0, size, 2\*size, ...  
 # The slice text[i : i + size] extracts each chunk  
 chunks\_generator = (text[i : i + size] for i in range(0, len(text), size))  
   
 # 3. Convert the generator to a tuple and return  
 return tuple(chunks\_generator)

## 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.
* The last string in the returned tuple may be shorter than size if the total length of the input text is not a multiple of size.

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

('chunk1', 'chunk2', 'chu')

## 3.5 Example

# Example usage  
my\_text = "This is a sample text to be split."  
chunk\_size = 10  
result = split\_into\_chunks(my\_text, chunk\_size)  
print(result)  
  
# Example with a string length not divisible by chunk size  
another\_text = "abcdefghij"  
short\_chunk\_size = 3  
result\_short = split\_into\_chunks(another\_text, short\_chunk\_size)  
print(result\_short)

**Output:**

('This is a ', 'sample tex', 't to be sp', 'lit.')  
('abc', 'def', 'ghi', 'j')