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 simple 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 and uppercase versions of all five vowels: aeiouAEIOU. 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 a 1 for every character that is a vowel. Finally, the built-in sum() function is used to add up all the 1s, producing the total count of vowels. Characters that are not vowels (such as consonants, numbers, whitespace, or symbols) are ignored.

# The function defines a set of all vowels for efficient, case-insensitive lookup.  
vowels = set("aeiouAEIOU")  
  
# It then sums a generator expression that yields 1 for each character found in the vowels set.  
return sum(1 for ch in text if ch in vowels)

## 1.4 Usage Notes

* The function is case-insensitive. It will count both ‘a’ and ‘A’ as vowels.
* Any characters that are not vowels, including consonants, numbers, punctuation, and whitespace, 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 possible return value for a given string.

7

## 1.5 Example

# Example usage of the count\_vowels function  
  
sample\_text = "This is a Sample String with VOWELS and numbers 123!"  
vowel\_count = count\_vowels(sample\_text)  
  
print(f"The string is: '{sample\_text}'")  
print(f"The number of vowels is: {vowel\_count}")

**Output:**

The string is: 'This is a Sample String with VOWELS and numbers 123!'  
The number of vowels is: 12

## 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** (Iterable[float]): An iterable collection of numbers (e.g., a list, tuple, or generator) to be summed. Non-float values like integers will be converted to floats during the computation.

## 2.3 Description

This function implements the Kahan summation algorithm, a technique designed to improve the precision of summing a sequence of floating-point numbers. Standard summation can suffer from significant round-off errors, especially when adding numbers of widely different magnitudes or when summing a very large quantity of numbers.

The algorithm maintains two key variables: 1. total: The running sum, initialized to 0.0. 2. compensation: A variable to track the accumulated error (the “lost” low-order bits) from previous additions, also initialized to 0.0.

For each value in the input numbers: 1. The value is first cast to a float. 2. A corrected value y is calculated by subtracting the compensation from the current value. This step reintroduces the error from the previous summation into the current calculation. 3. A temporary sum t is computed by adding the corrected value y to the current total. 4. The new compensation value is calculated as (t - total) - y. In perfect arithmetic, this would be zero. However, in floating-point arithmetic, (t - total) might not be exactly equal to y. This expression captures the round-off error that occurred when y was added to total. 5. The total is updated to the new temporary sum t.

# Inside the loop:  
y = float(value) - compensation  
t = total + y  
compensation = (t - total) - y  
total = t

After iterating through all numbers, the final total is returned, which is a more accurate representation of the true sum than what a simple sum() might produce.

## 2.4 Usage Notes

* This function is particularly useful when high precision is required for summations, such as in scientific or financial calculations.
* It provides a more accurate result than Python’s built-in sum() function when dealing with floating-point numbers that could lead to significant precision loss.
* The input numbers can be any iterable, including lists, tuples, and generators. This allows for memory-efficient processing of large datasets.

**Output Example**: The function returns a single floating-point number representing the sum. For an input of [0.1] \* 10, the return value is 1.0, whereas a standard sum might yield 0.9999999999999999.

## 2.5 Example

# Example usage demonstrating precision improvement over standard sum.  
# Summing a small fractional number multiple times can introduce errors.  
data = [0.1] \* 10  
  
# Using pairwise\_sum for better accuracy  
accurate\_result = pairwise\_sum(data)  
print(f"Result with pairwise\_sum: {accurate\_result}")  
  
# Using standard built-in sum() for comparison  
standard\_result = sum(data)  
print(f"Result with standard sum(): {standard\_result}")

**Output:**

Result with pairwise\_sum: 1.0  
Result with standard sum(): 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.

## 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 split the text. It uses a generator expression that iterates through the input text with a step equal to the size. In each step, it slices the string from the current index i up to i + size. This creates a sequence of substrings. The final substring in the sequence may be shorter than size if the total length of the text is not perfectly divisible by size.

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

# The core logic uses a generator expression and range with a step  
# For text="abcdefg" and size=3  
# range(0, 7, 3) will produce indices 0, 3, 6  
# Slices will be text[0:3], text[3:6], text[6:9]  
# Resulting in "abc", "def", "g"  
tuple(text[i : i + size] for i in range(0, len(text), size))

## 3.4 Usage Notes

* The function will raise a ValueError if the size parameter is not a positive integer (i.e., if size <= 0).
* The returned value is a tuple of strings, which is an immutable sequence.
* The last chunk in the returned tuple may be shorter than the specified size if the length of the input text is not a multiple of size.
* If an empty string is passed as text, the function will return an empty tuple ().

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

('This is a ', 'sample tex', 't.')

## 3.5 Example

# Example usage  
long\_string = "This is a sample text for the chunking function."  
chunk\_size = 10  
  
try:  
 result = split\_into\_chunks(long\_string, chunk\_size)  
 print(result)  
except ValueError as e:  
 print(e)  
  
# 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:**

('This is a ', 'sample tex', 't for the ', 'chunking f', 'unction.')  
('12345', '67890')