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 (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 vowels ("aeiouAEIOU"). Using a set allows for highly efficient membership checking, which is faster than searching through a list or string repeatedly.

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 the number 1 every time a character is identified as a vowel.

Finally, the built-in sum() function is used to add up all the 1s generated, effectively tallying the total count of vowels. The final integer sum is then returned.

# 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.

## 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, symbols, and whitespace, are ignored and not included in the count.
* The letter ‘y’ is not considered a vowel by this function.

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

3

## 1.5 Example

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

**Output:**

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 sum of a sequence of numbers using a numerically stable approach to minimize floating-point errors.

## 2.2 parameters

* **numbers** (Iterable[float]): An iterable collection of numbers (e.g., a list, tuple, or generator) that can be converted to floats.

## 2.3 Description

This function provides a more precise method for summing floating-point numbers compared to the standard built-in sum() function. Standard summation can suffer from a loss of precision, especially when adding numbers of vastly different magnitudes or when summing a large quantity of numbers.

The pairwise\_sum function implements the Kahan summation algorithm to counteract this problem. The algorithm works by maintaining a running compensation variable that accumulates the small errors lost in 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 to the corrected value y. 5. The core of the algorithm is calculating the error from the previous addition: compensation = (t - total) - y. In perfect arithmetic, this would be zero. However, due to floating-point limitations, (t - total) may not be exactly equal to y, and the difference represents the “lost” part of the sum, which is captured in compensation. 6. The total is updated with the value of t. 7. This process repeats, carrying the error from one step to the next, ensuring that small values are not lost when added to large running totals.

The final total returned is a more accurate representation of the true arithmetic sum.

## 2.4 Usage Notes

* This function is more computationally intensive than a simple sum() but is recommended for financial calculations, scientific computing, or any scenario where high precision is critical.
* It is particularly effective for datasets containing a large number of values or values with a wide range of magnitudes.
* The input iterable can contain integers, as each value is explicitly cast to a float during the calculation.

**Output Example**: The function returns a single floating-point number.

2.0

## 2.5 Example

The following example demonstrates a classic case where naive summation can fail, but pairwise\_sum produces the correct result.

# A list where naive summation can lead to precision loss.  
# (1.0 + 1e100) results in 1e100, so the initial 1.0 is lost.  
# Then, adding 1.0 again and subtracting 1e100 results in 0.0.  
numbers\_list = [1.0, 1e100, 1.0, -1e100]  
  
# Using the standard sum() function  
simple\_sum = sum(numbers\_list)  
print(f"Standard sum: {simple\_sum}")  
  
# Using the numerically stable pairwise\_sum function  
stable\_sum = pairwise\_sum(numbers\_list)  
print(f"Pairwise sum: {stable\_sum}")

**Output:**

Standard sum: 0.0  
Pairwise sum: 2.0

## 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 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. It checks if size is a positive number. If size is zero or negative, the function raises a ValueError to prevent invalid slicing operations.

If the size is valid, the function proceeds to slice the string. It uses a generator expression that iterates through the input text with a step equal to size. In each iteration, it creates a substring of length size using Python’s slicing mechanism text[i : i + size]. This process continues until the entire string has been covered.

Finally, all the generated substrings are collected into a tuple, which is the return value of the function. Note that if the total length of the text is not a multiple of size, the last substring in the tuple will be shorter than the specified size.

# Internal logic for a text "abcdefg" and size 3  
# 1. Check if size (3) > 0. It is.  
# 2. range(0, len("abcdefg"), 3) generates indices 0, 3, 6.  
# 3. First chunk: text[0:3] -> "abc"  
# 4. Second chunk: text[3:6] -> "def"  
# 5. Third chunk: text[6:9] -> "g" (shorter than size)  
# 6. Return tuple: ("abc", "def", "g")

## 3.4 Usage Notes

* The size parameter must be a positive integer. Providing a value of 0 or less will result in a ValueError.
* The last chunk in the returned tuple may be shorter than the specified size if the length of the input text is not perfectly divisible by size.
* The function returns a tuple of strings, not a list.

**Output Example**: A possible return value for a given string and size.

('This ', 'is a ', 'sampl', 'e tex', 't.')

## 3.5 Example

# Example usage  
long\_string = "This is a sample text for splitting into chunks."  
chunk\_length = 10  
  
try:  
 chunks = split\_into\_chunks(long\_string, chunk\_length)  
 print(chunks)  
except ValueError as e:  
 print(e)  
  
# Example with an invalid size  
try:  
 invalid\_chunks = split\_into\_chunks("some text", -5)  
 print(invalid\_chunks)  
except ValueError as e:  
 print(e)

**Output:**

('This is a ', 'sample tex', 't for spli', 'tting into', ' chunks.')  
size must be positive