Table of Contents

## 0.1 FunctionDef fibonacci(n)

# 1 Function: fibonacci

## 1.1 Overview

The fibonacci Function computes the nth Fibonacci number using an efficient iterative approach.

## 1.2 parameters

* n (int): The index of the Fibonacci number to compute (0-indexed).

## 1.3 Description

This function calculates a number in the Fibonacci sequence based on its index n. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.

The function first validates the input n. If n is a negative number, it raises a ValueError because the Fibonacci sequence is not defined for negative indices.

It initializes two variables, a and b, to 0 and 1 respectively. These represent the first two numbers in the sequence, F(0) and F(1).

The function then enters a for loop that iterates n times. In each iteration, the values of a and b are updated using tuple assignment: a, b = b, a + b. This simultaneously sets a to the current value of b and b to the sum of the old a and b. This process effectively walks through the Fibonacci sequence.

After the loop completes, the variable a holds the value of the nth Fibonacci number, which is then returned.

# For n = 5, the loop runs 5 times:  
# Initial: a = 0, b = 1  
# 1. a = 1, b = 1 (0 + 1)  
# 2. a = 1, b = 2 (1 + 1)  
# 3. a = 2, b = 3 (1 + 2)  
# 4. a = 3, b = 5 (2 + 3)  
# 5. a = 5, b = 8 (3 + 5)  
# The loop ends, and the function returns a, which is 5.

## 1.4 Usage Notes

* The input n must be a non-negative integer. Providing a negative value will result in a ValueError.
* The function is 0-indexed, meaning fibonacci(0) returns the first number in the sequence, which is 0.
* This iterative implementation is memory-efficient and avoids the recursion depth limits that can be an issue with naive recursive solutions for large values of n.

**Output Example**: The function returns a single integer representing the Fibonacci number at the specified index.

## 1.5 Example

# Example usage  
# Calculate the 9th Fibonacci number (0-indexed)  
result = fibonacci(9)  
print(result)  
  
# Example with an edge case  
result\_zero = fibonacci(0)  
print(result\_zero)

**Output:**

34  
0

## 1.6 FunctionDef invert\_dictionary(mapping)

# 2 Function: invert\_dictionary

## 2.1 Overview

The invert\_dictionary function inverts a given dictionary by swapping its keys and values, provided that all values in the original dictionary are unique.

## 2.2 parameters

* **mapping** (Dict[str, int]): A dictionary mapping string keys to integer values. The values within this dictionary must be unique for the inversion to be successful.

## 2.3 Description

This function takes a dictionary, mapping, as input and returns a new dictionary where the keys and values have been swapped.

Before performing the inversion, the function first validates that all values in the input dictionary are unique. It achieves this by comparing the number of values (len(mapping.values())) with the number of unique values (len(set(mapping.values()))). If these counts do not match, it indicates the presence of duplicate values.

In the case of duplicate values, a ValueError is raised with the message “Values must be unique to invert dictionary”, as a one-to-one mapping cannot be guaranteed for the inverted dictionary.

If all values are unique, the function proceeds to create the inverted dictionary using a dictionary comprehension: {value: key for key, value in mapping.items()}. This iterates through the key-value pairs of the original dictionary and constructs a new dictionary with the values as keys and the keys as values.

## 2.4 Usage Notes

* The primary constraint of this function is that the input dictionary’s values must be unique. Attempting to invert a dictionary with duplicate values will result in a ValueError.
* The function is non-destructive; it does not modify the original mapping dictionary. It returns a new, inverted dictionary.

**Output Example**: A successfully inverted dictionary might look like this:

{1: 'apple', 2: 'banana', 3: 'cherry'}

## 2.5 Example

**Example 1: Successful Inversion**

The following example demonstrates a successful inversion where all values in the input dictionary are unique.

# Example usage with unique values  
original\_map = {'user\_id': 101, 'session\_id': 202, 'request\_id': 303}  
inverted\_map = invert\_dictionary(original\_map)  
print(inverted\_map)

**Output:**

{101: 'user\_id', 202: 'session\_id', 303: 'request\_id'}

**Example 2: Failed Inversion with Duplicate Values**

This example shows how the function raises a ValueError when the input dictionary contains duplicate values.

# Example usage with non-unique values  
non\_unique\_map = {'alpha': 1, 'beta': 2, 'gamma': 1}  
  
try:  
 invert\_dictionary(non\_unique\_map)  
except ValueError as e:  
 print(e)

**Output:**

Values must be unique to invert dictionary

## 2.6 FunctionDef is\_palindrome(text)

# 3 Function: is\_palindrome

## 3.1 Overview

The is\_palindrome function determines if a given string is a palindrome, ignoring letter casing and whitespace characters.

## 3.2 parameters

* text (str): The input string to be checked.

## 3.3 Description

The is\_palindrome function evaluates whether a string reads the same forwards as it does backward. To achieve this, it first performs a normalization process on the input text.

The normalization is handled in this line: normalized = ''.join(ch.lower() for ch in text if not ch.isspace())

This process involves three steps: 1. It iterates through each character (ch) in the input text. 2. It filters out any whitespace characters using if not ch.isspace(). 3. Each remaining character is converted to lowercase using ch.lower().

The resulting characters are then joined together to form a new normalized string.

Finally, the function compares the normalized string with its reverse. The reversal is achieved using slice notation [::-1]. If the normalized string is identical to its reversed version, the function returns True; otherwise, it returns False.

# Internal logic breakdown  
text = "Taco Cat"  
# 1. Iterate and filter: ['T', 'a', 'c', 'o', 'C', 'a', 't']  
# 2. Convert to lowercase: ['t', 'a', 'c', 'o', 'c', 'a', 't']  
# 3. Join to create normalized string: "tacocat"  
normalized = "tacocat"  
# 4. Reverse the normalized string: "tacocat"  
reversed\_normalized = normalized[::-1]  
# 5. Compare: "tacocat" == "tacocat" -> True  
return True

## 3.4 Usage Notes

* The function is case-insensitive. For example, Racecar and racecar will both be correctly identified as palindromes.
* All standard whitespace characters (spaces, tabs, newlines, etc.) are ignored during the comparison.
* Punctuation marks and other special characters are **not** ignored and will be part of the comparison. For instance, is\_palindrome("A man, a plan, a canal: Panama") will return False because the commas and colon are included in the check.

**Output Example**: The function returns a boolean value.

True

## 3.5 Example

# Example 1: A classic palindrome with mixed casing and spaces  
result1 = is\_palindrome("A nut for a jar of tuna")  
print(f"'A nut for a jar of tuna' is a palindrome: {result1}")  
  
# Example 2: A simple palindrome  
result2 = is\_palindrome("Taco cat")  
print(f"'Taco cat' is a palindrome: {result2}")  
  
# Example 3: A non-palindrome string  
result3 = is\_palindrome("hello world")  
print(f"'hello world' is a palindrome: {result3}")

**Output:**

'A nut for a jar of tuna' is a palindrome: True  
'Taco cat' is a palindrome: True  
'hello world' is a palindrome: False