Table of Contents

## 0.1 FunctionDef fibonacci(n)

# 1 Function: fibonacci(n: int)

## 1.1 Overview

The fibonacci function computes the nth number in the Fibonacci sequence using an iterative approach.

## 1.2 parameters

* n: int - The 0-indexed position in the Fibonacci sequence for which to find the value. This must be a non-negative integer.

## 1.3 Description

This function calculates a Fibonacci number based on its index n. The logic begins by validating the input. If n is a negative number, a ValueError is raised, as the Fibonacci sequence is not defined for negative indices.

The function initializes two variables, a and b, to 0 and 1 respectively. These represent the first two numbers in the 0-indexed Fibonacci sequence (F(0) = 0, F(1) = 1).

It then enters a for loop that iterates n times. In each iteration, the values of a and b are updated simultaneously. The current value of b is assigned to a, and the sum of the old a and b is assigned to b. This process effectively walks through the sequence: - Iteration 1: a becomes 1, b becomes 1 (0+1) - Iteration 2: a becomes 1, b becomes 2 (1+1) - Iteration 3: a becomes 2, b becomes 3 (1+2) - and so on…

After the loop completes, the variable a holds the nth Fibonacci number, which is then returned. If n is 0, the loop does not execute, and the initial value of a (0) is correctly returned.

# Initialization for the sequence  
a, b = 0, 1  
# Loop n times to find the nth number  
for \_ in range(n):  
 # Update a and b to the next numbers in the sequence  
 a, b = b, a + b  
# Return the nth number  
return a

## 1.4 Usage Notes

* The input n must be a non-negative integer. Providing a negative value will result in a ValueError.
* The function uses a 0-indexed sequence, meaning fibonacci(0) returns 0, fibonacci(1) returns 1, and so on.
* This iterative implementation is efficient in terms of memory and performance for large values of n compared to a simple recursive approach, as it avoids redundant calculations and deep recursion stacks.

**Output Example**: The function returns a single integer representing the Fibonacci number at the specified index. For an input of 10, the output would be 55.

## 1.5 Example

# Example usage  
# Calculate the 9th Fibonacci number (0-indexed)  
result = fibonacci(9)  
print(f"The 9th Fibonacci number is: {result}")  
  
# Example with an edge case  
result\_zero = fibonacci(0)  
print(f"The 0th Fibonacci number is: {result\_zero}")

**Output:**

The 9th Fibonacci number is: 34  
The 0th Fibonacci number is: 0

## 1.6 FunctionDef invert\_dictionary(mapping)

# 2 Function: invert\_dictionary(mapping: Dict[str, int]) -> Dict[int, str]

## 2.1 Overview

The invert\_dictionary function inverts a given dictionary by swapping its keys and values, returning a new dictionary.

## 2.2 parameters

* mapping (Dict[str, int]): A dictionary mapping string keys to integer values. It is crucial that all values in this dictionary are unique.

## 2.3 Description

This function provides a safe way to invert a dictionary. It operates in two main steps:

1. **Validation**: Before performing the inversion, the function first validates that all values in the input mapping are unique. It does this by comparing the number of values with the number of unique values. Specifically, it compares len(mapping.values()) with len(set(mapping.values())). If these lengths are not equal, it indicates the presence of duplicate values, and the function raises a ValueError.
2. **Inversion**: If all values are unique, the function proceeds to create and return a new dictionary. It uses a dictionary comprehension, {value: key for key, value in mapping.items()}, to iterate through each key-value pair of the original mapping. For each pair, it creates a new entry in the new dictionary where the original value becomes the key and the original key becomes the value.

# Internal logic for inversion  
# This is executed only if values are unique  
inverted\_dict = {value: key for key, value in mapping.items()}

## 2.4 Usage Notes

* The function will raise a ValueError with the message “Values must be unique to invert dictionary” if the input dictionary contains duplicate values.
* This function is not an in-place operation. It returns a new dictionary and does not modify the original mapping dictionary.
* The type hints indicate an input of Dict[str, int] and an output of Dict[int, str], but the logic will work for any dictionary with hashable keys and unique, hashable values.

**Output Example**: A dictionary where the keys are the values from the input dictionary, and the values are the keys from the input dictionary.

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

## 2.5 Example

# Example 1: Successful inversion with unique values  
original\_dict = {'apple': 1, 'banana': 2, 'cherry': 3}  
inverted\_dict = invert\_dictionary(original\_dict)  
print(f"Original: {original\_dict}")  
print(f"Inverted: {inverted\_dict}")  
  
# Example 2: Attempted inversion with duplicate values  
try:  
 duplicate\_value\_dict = {'a': 1, 'b': 2, 'c': 1}  
 print(f"\nAttempting to invert: {duplicate\_value\_dict}")  
 invert\_dictionary(duplicate\_value\_dict)  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

Original: {'apple': 1, 'banana': 2, 'cherry': 3}  
Inverted: {1: 'apple', 2: 'banana', 3: 'cherry'}  
  
Attempting to invert: {'a': 1, 'b': 2, 'c': 1}  
Error: Values must be unique to invert dictionary

## 2.6 FunctionDef is\_palindrome(text)

# 3 Function: is\_palindrome(text: str)

## 3.1 Overview

The is\_palindrome function determines if a given string is a palindrome, meaning it reads the same forwards and backwards, while ignoring letter casing and all whitespace characters.

## 3.2 parameters

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

## 3.3 Description

This function evaluates whether a string is a palindrome through a two-step process: normalization and comparison.

First, it normalizes the input text. This is achieved by iterating through each character of the string. Any character that is a whitespace character (as identified by the isspace() method) is discarded. All remaining characters are converted to their lowercase equivalents. These processed characters are then joined together to form a new, “normalized” string.

For example, the input "A man a plan a canal Panama" would be normalized to "amanaplanacanalpanama".

# Normalization logic  
normalized = ''.join(ch.lower() for ch in text if not ch.isspace())

Second, the function compares the normalized string with its reversed version. The reversal is accomplished using slice notation [::-1]. If the normalized string is identical to its reversed counterpart, the function returns True, indicating the original text is a palindrome. Otherwise, it returns False.

# Comparison logic  
return normalized == normalized[::-1]

## 3.4 Usage Notes

* The function is case-insensitive. For example, Racecar and racecar are both treated as palindromes.
* All whitespace characters (spaces, tabs, newlines, etc.) are ignored during the check.
* Punctuation, numbers, and other symbols are **not** ignored and are included in the palindrome check. For example, "A man, a plan..." will evaluate to False because the comma is not symmetrical.

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

True

## 3.5 Example

# Example 1: A classic palindrome with mixed casing and spaces  
phrase = "A man a plan a canal Panama"  
result = is\_palindrome(phrase)  
print(f"Is '{phrase}' a palindrome? {result}")  
  
# Example 2: A simple non-palindrome  
word = "hello"  
result\_non\_palindrome = is\_palindrome(word)  
print(f"Is '{word}' a palindrome? {result\_non\_palindrome}")  
  
# Example 3: A palindrome with numbers (which are not ignored)  
numeric\_palindrome = "12321"  
result\_numeric = is\_palindrome(numeric\_palindrome)  
print(f"Is '{numeric\_palindrome}' a palindrome? {result\_numeric}")

**Output:**

Is 'A man a plan a canal Panama' a palindrome? True  
Is 'hello' a palindrome? False  
Is '12321' a palindrome? True