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 efficient iterative approach.

## 1.2 parameters

* n (int): The 0-indexed position in the Fibonacci sequence for which to find the value.

## 1.3 Description

This function calculates a Fibonacci number based on its index n. The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones (0, 1, 1, 2, 3, 5, …).

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. The function then enters a for loop that iterates n times. In each iteration, it performs a simultaneous assignment: a takes the value of b, and b is updated to the sum of the old a and b. This process effectively steps through the Fibonacci sequence.

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

# Initialization for the sequence  
a, b = 0, 1  
  
# For n=3, the loop runs 3 times:  
# 1. a becomes 1, b becomes 0 + 1 = 1  
# 2. a becomes 1, b becomes 1 + 1 = 2  
# 3. a becomes 2, b becomes 1 + 2 = 3  
  
# The function returns a, which is 2.  
# Sequence: 0, 1, 1, 2. The number at index 3 is 2.

## 1.4 Usage Notes

* The function uses a 0-indexed sequence. For example, fibonacci(0) returns 0, and fibonacci(1) returns 1.
* The input n must be a non-negative integer. Providing a negative integer will result in a ValueError.
* This iterative implementation is memory-efficient and avoids the recursion depth limits that can affect recursive solutions for large values of n.

**Output Example**: The function returns a single integer value.

34

## 1.5 Example

# Example usage: Find the 9th Fibonacci number (0-indexed)  
n\_index = 9  
result = fibonacci(n\_index)  
print(f"The Fibonacci number at index {n\_index} is: {result}")

**Output:**

The Fibonacci number at index 9 is: 34

## 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 essential that all values in this dictionary are unique to ensure a valid inversion.

## 2.3 Description

This function provides a safe way to invert a dictionary where string keys are mapped to integer values. The core logic is implemented in two main steps:

1. **Validation**: Before attempting to invert the dictionary, 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. The expression len(mapping.values()) != len(set(mapping.values())) evaluates to True if any duplicates exist, as converting a list of values to a set removes duplicates. If duplicates are found, a ValueError is raised with the message “Values must be unique to invert dictionary” to prevent data loss in the inverted dictionary.
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 output dictionary where the original value becomes the new key and the original key becomes the new value.

This process ensures a one-to-one mapping in the resulting dictionary, which maps integers back to their corresponding strings.

## 2.4 Usage Notes

* The function requires that all values in the input mapping dictionary are unique. Attempting to invert a dictionary with duplicate values will result in a ValueError.
* The function is not an in-place operation. It returns a new dictionary and does not modify the original mapping dictionary.
* The type hints indicate a mapping from str to int, but the logic will work for any dictionary with hashable keys and unique, hashable values.

**Output Example**: A dictionary with integer keys and string values.

{1: 'one', 2: 'two', 3: 'three'}

## 2.5 Example

# Example 1: Successful inversion with unique values  
original\_map = {'alpha': 10, 'beta': 20, 'gamma': 30}  
inverted\_map = invert\_dictionary(original\_map)  
print(f"Original dictionary: {original\_map}")  
print(f"Inverted dictionary: {inverted\_map}")  
  
# Example 2: Attempted inversion with duplicate values  
invalid\_map = {'apple': 1, 'banana': 2, 'apricot': 1}  
try:  
 invert\_dictionary(invalid\_map)  
except ValueError as e:  
 print(f"\nAttempting to invert: {invalid\_map}")  
 print(f"Error: {e}")

**Output:**

Original dictionary: {'alpha': 10, 'beta': 20, 'gamma': 30}  
Inverted dictionary: {10: 'alpha', 20: 'beta', 30: 'gamma'}  
  
Attempting to invert: {'apple': 1, 'banana': 2, 'apricot': 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, after ignoring character casing and whitespace.

## 3.2 parameters

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

## 3.3 Description

This function provides a straightforward way to validate if a string is a palindrome by performing a two-step process: normalization and comparison.

1. **Normalization**: The function first processes the input text to create a “normalized” version. It iterates through each character of the string. During this iteration, it discards any whitespace characters (like spaces, tabs, or newlines) using ch.isspace(). All other characters are converted to their lowercase equivalents using ch.lower(). These processed characters are then joined together to form a new string, normalized.

* normalized = ''.join(ch.lower() for ch in text if not ch.isspace())

1. **Comparison**: Once the normalized string is created, the function compares it with its own reverse. The reversal is achieved using the slice notation [::-1], which creates a reversed copy of the string.

* return normalized == normalized[::-1]

If the normalized string is identical to its reversed version, the function returns True; otherwise, it returns False.

## 3.4 Usage Notes

* The function is case-insensitive. For example, “Racecar” and “racecar” will both be evaluated as palindromes.
* All whitespace characters are completely ignored and removed before the palindrome check.
* Punctuation and special characters are **not** removed. They are included in the check and may cause a string that is otherwise a palindrome to fail. For instance, is\_palindrome("A man, a plan, a canal: Panama") would return False because the commas and colon are not ignored.

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

True

## 3.5 Example

# Example 1: A classic palindrome with mixed casing and spaces  
palindrome\_string = "Taco cat"  
result1 = is\_palindrome(palindrome\_string)  
print(f"Is '{palindrome\_string}' a palindrome? {result1}")  
  
# Example 2: A non-palindrome string  
non\_palindrome\_string = "Hello World"  
result2 = is\_palindrome(non\_palindrome\_string)  
print(f"Is '{non\_palindrome\_string}' a palindrome? {result2}")  
  
# Example 3: A palindrome with numbers  
numeric\_palindrome = "123 321"  
result3 = is\_palindrome(numeric\_palindrome)  
print(f"Is '{numeric\_palindrome}' a palindrome? {result3}")

**Output:**

Is 'Taco cat' a palindrome? True  
Is 'Hello World' a palindrome? False  
Is '123 321' a palindrome? True