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: An integer representing the 0-indexed position in the Fibonacci sequence for which to find the value.

## 1.3 Description

This function provides an efficient, iterative method to calculate a Fibonacci number. The logic proceeds as follows:

1. **Input Validation**: The function first checks if the input n is less than 0. If it is, a ValueError is raised, as the Fibonacci sequence is not defined for negative indices.
2. **Initialization**: Two variables, a and b, are initialized to 0 and 1 respectively. These represent the first two numbers in the Fibonacci sequence, F(0) and F(1).
3. **Iteration**: 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 sequence.
   * For n=0, the loop does not run, and the initial value of a (0) is returned.
   * For n=1, the loop runs once, setting a to 1, which is then returned.
4. **Return Value**: After the loop completes, the variable a holds the nth Fibonacci number, which is then returned.

# Inside the loop for n=3  
# Initial state: a = 0, b = 1  
# Iteration 1: a becomes 1, b becomes 0 + 1 = 1  
# Iteration 2: a becomes 1, b becomes 1 + 1 = 2  
# Iteration 3: a becomes 2, b becomes 1 + 2 = 3  
# Loop ends. The function returns a, which is 2.

## 1.4 Usage Notes

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

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

## 1.5 Example

# Example usage  
# Find 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 swaps the keys and values of a given dictionary, creating a new dictionary where the original values become the keys and the original keys become the values.

## 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.

## 2.3 Description

This function provides a safe way to invert a dictionary. 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 are found, as converting a list of values to a set removes duplicates. If duplicates exist, a ValueError is raised to prevent data loss, as multiple keys would otherwise map to the same new key in the inverted dictionary.
2. **Inversion**: If all values are unique, the function proceeds to create a new dictionary using a dictionary comprehension: {value: key for key, value in mapping.items()}. This expression iterates through each key-value pair of the original mapping and constructs a new dictionary where each original value is now a key, and each original key is its corresponding value.

The function returns this newly created inverted dictionary.

## 2.4 Usage Notes

* The function will raise a ValueError if the input dictionary contains duplicate values. This is a critical safeguard to ensure the inversion is lossless.
* The original dictionary passed as the mapping argument is not modified. The function returns a new dictionary object.
* While the type hints specify Dict[str, int], the function’s logic will work with any dictionary as long as its values are unique and hashable (e.g., strings, numbers, tuples), as they will become keys in the new dictionary.

**Output Example**: A dictionary where keys are integers and values are strings.

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

## 2.5 Example

# Example 1: Successful inversion with unique values  
original\_dict = {"alpha": 1, "beta": 2, "gamma": 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:  
 faulty\_dict = {"a": 1, "b": 2, "c": 1}  
 print(f"\nAttempting to invert: {faulty\_dict}")  
 invert\_dictionary(faulty\_dict)  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

Original: {'alpha': 1, 'beta': 2, 'gamma': 3}  
Inverted: {1: 'alpha', 2: 'beta', 3: 'gamma'}  
  
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, after ignoring character casing and spaces.

## 3.2 parameters

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

## 3.3 Description

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

First, the function normalizes the input text. It iterates through each character of the string, converts it to lowercase, and discards any whitespace characters. The resulting characters are joined to form a new, clean string. This step ensures that the comparison is case-insensitive and unaffected by spacing.

For example, the input 'Taco Cat' would be normalized to 'tacocat'.

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

Second, the function compares the normalized string with its own reverse. The reversal is efficiently created using Python’s slice notation [::-1]. If the normalized string is identical to its reversed version, the function returns True; otherwise, it returns False.

return normalized == normalized[::-1]

## 3.4 Usage Notes

* The function is case-insensitive. For example, is\_palindrome("Racecar") and is\_palindrome("racecar") both return True.
* All whitespace characters (spaces, tabs, newlines, etc.) are ignored during the check.
* Punctuation and symbols are **not** ignored and will be included in the comparison. For instance, is\_palindrome("A man, a plan...") will evaluate to False because the commas and periods are part of the string being checked.

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

True

## 3.5 Example

# Example 1: A classic palindrome  
result1 = is\_palindrome("A man a plan a canal Panama")  
print(f"'A man a plan a canal Panama' is a palindrome: {result1}")  
  
# Example 2: A simple palindrome with different casing  
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}")  
  
# Example 4: A string with punctuation that fails the check  
result4 = is\_palindrome("Eva, can I see bees in a cave?")  
print(f"'Eva, can I see bees in a cave?' is a palindrome: {result4}")

**Output:**

'A man a plan a canal Panama' is a palindrome: True  
'Taco Cat' is a palindrome: True  
'Hello World' is a palindrome: False  
'Eva, can I see bees in a cave?' is a palindrome: False