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.

## 1.3 Description

This function provides a memory-efficient way to calculate Fibonacci numbers. The logic begins by validating the input n. 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 sequence, F(0) and F(1).

It then enters a for loop that iterates n times. Inside the loop, the core logic a, b = b, a + b is executed. This is a tuple assignment that simultaneously updates the values: 1. The current value of b is assigned to a. 2. The sum of the old a and b is assigned to b.

This process effectively shifts the sequence forward one step in each iteration. For example, if a=3 and b=5, the next iteration will set a=5 and b=8. After the loop completes n iterations, the variable a will hold 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 function returns a, which is 5.

## 1.4 Usage Notes

* The input n must be a non-negative integer. The function will raise a ValueError for negative inputs.
* The function uses a 0-indexed sequence, meaning fibonacci(0) returns the first number of the sequence, which is 0.
* This iterative implementation is efficient in terms of memory and performance for large values of n compared to a naive recursive approach, as it avoids redundant calculations and deep recursion stacks.

**Output Example**: A single integer representing the calculated Fibonacci number.

55

## 1.5 Example

# Example usage  
# Calculate the 10th Fibonacci number (0-indexed)  
result = fibonacci(10)  
print(result)

**Output:**

55

## 1.6 FunctionDef invert\_dictionary(mapping)

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

## 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 keys and the original keys become values.

## 2.2 parameters

* **mapping** (Dict[str, int]): A dictionary mapping string keys to integer values. It is critical that all values in this dictionary are unique for the inversion to be successful.

## 2.3 Description

This function provides a safe way to invert a dictionary by first ensuring the integrity of the operation.

The function’s logic proceeds as follows: 1. **Uniqueness Validation**: It first checks if all values in the input mapping are unique. This is accomplished by comparing the total number of values (len(mapping.values())) with the number of unique values, which is found by converting the values to a set and checking its length (len(set(mapping.values()))). 2. **Error Handling**: If the counts do not match, it signifies that duplicate values exist. In this case, a ValueError is raised with the message “Values must be unique to invert dictionary”. This prevents an ambiguous or lossy inversion where multiple keys would map to the same new key. 3. **Inversion**: If all values are unique, the function uses a dictionary comprehension, {value: key for key, value in mapping.items()}, to iterate over each key-value pair in the original mapping. It constructs and returns a new dictionary where each original value is now a key, and its corresponding key is the value.

## 2.4 Usage Notes

Important points to consider when using this function:

* The primary requirement is that the values of the input mapping dictionary must be unique. The function will raise a ValueError if this condition is not met.
* The function is non-destructive; it returns a new inverted dictionary and does not modify the original dictionary provided in the mapping parameter.
* The type hints specify a Dict[str, int] input and Dict[int, str] output, but the function will work with any dictionary where both keys and values are hashable types.

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

{10: 'apple', 20: 'banana', 30: 'cherry'}

## 2.5 Example

# Example usage of a valid dictionary  
from typing import Dict  
  
def invert\_dictionary(mapping: Dict[str, int]) -> Dict[int, str]:  
 """Invert a dictionary with unique values.  
  
 Parameters:  
 mapping: Dictionary mapping strings to integers with unique values.  
  
 Returns:  
 A new dictionary mapping integers to strings.  
 """  
 if len(mapping.values()) != len(set(mapping.values())):  
 raise ValueError("Values must be unique to invert dictionary")  
 return {value: key for key, value in mapping.items()}  
  
original\_dict = {'user\_one': 101, 'user\_two': 202, 'user\_three': 303}  
inverted\_dict = invert\_dictionary(original\_dict)  
print(f"Original Dictionary: {original\_dict}")  
print(f"Inverted Dictionary: {inverted\_dict}")  
  
# Example that raises a ValueError  
try:  
 invalid\_dict = {'a': 1, 'b': 2, 'c': 1}  
 invert\_dictionary(invalid\_dict)  
except ValueError as e:  
 print(f"\nError with invalid dictionary: {e}")

**Output:**

Original Dictionary: {'user\_one': 101, 'user\_two': 202, 'user\_three': 303}  
Inverted Dictionary: {101: 'user\_one', 202: 'user\_two', 303: 'user\_three'}  
  
Error with invalid dictionary: 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 letter casing and all whitespace characters.

## 3.2 parameters

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

## 3.3 Description

This function provides a straightforward way to check for palindromes by first normalizing the input string and then comparing it to its reverse.

The core logic operates in two main steps:

1. **Normalization**: The function first creates a “normalized” version of the input text. It iterates through every character (ch) in the string. Using a generator expression, it filters out any character that is a whitespace character (like spaces, tabs, or newlines) via the ch.isspace() method. For all remaining characters, it converts them to lowercase using ch.lower(). The ''.join() method then combines these filtered, lowercase characters into a new string, normalized.
2. **Comparison**: The function then compares the normalized string with its reversed version. The slice notation normalized[::-1] efficiently creates a reversed copy of the string. If the normalized string is identical to its reversed counterpart, the expression evaluates to True, indicating a palindrome. Otherwise, it evaluates to False.

For instance, if the input text is "Taco Cat", the normalized string becomes "tacocat". The function then checks if "tacocat" is equal to "tacocat"[::-1], which is also "tacocat". Since they are equal, the function returns True.

## 3.4 Usage Notes

* The function is **case-insensitive**. For example, is\_palindrome("Racecar") and is\_palindrome("racecar") will both return True.
* 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, is\_palindrome("A man, a plan, a canal: Panama") will return False because the commas and colon are part of the comparison.

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

True

## 3.5 Example

# Example usage with mixed casing and spaces  
text\_to\_check = "No lemon no melon"  
result = is\_palindrome(text\_to\_check)  
print(f"Is '{text\_to\_check}' a palindrome? {result}")  
  
# Example with a non-palindrome  
another\_text = "hello world"  
result\_two = is\_palindrome(another\_text)  
print(f"Is '{another\_text}' a palindrome? {result\_two}")

**Output:**

Is 'No lemon no melon' a palindrome? True  
Is 'hello world' a palindrome? False