Table of Contents

## 0.1 FunctionDef num(a, b)

# 1 Function: def num(a, b)

## 1.1 Overview

The num function calculates and returns the sum of two provided numbers.

## 1.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| a | int/float | The first number to add. |
| b | int/float | The second number to add. |

## 1.3 Description

This function provides a straightforward way to perform addition. It accepts two arguments, a and b, which are expected to be numerical values (integers or floating-point numbers). The core logic of the function is to use the + operator to compute the sum of a and b. The resulting value is then returned to the caller.

For example, if a is 5 and b is 10, the function will compute 5 + 10 and return 15.

# The function simply returns the result of a + b  
return a + b

## 1.4 Usage Notes

* This function is designed for numeric types like int and float.
* If string values are passed as arguments, the function will perform string concatenation instead of mathematical addition (e.g., num("hello", "world") would return "helloworld").
* The type of the returned value will depend on the types of the input parameters (e.g., adding two integers returns an integer; adding an integer and a float returns a float).

**Output Example**: A numeric value representing the sum. 15

## 1.5 Example

# Example usage with two integers  
result = num(5, 10)  
print(result)  
  
# Example usage with a float and an integer  
result\_float = num(7.5, 3)  
print(result\_float)

**Output:**

15  
10.5

## 1.6 FunctionDef generate\_random\_integers(count, start, end)

# 2 Function: generate\_random\_integers(count: int, start: int = 0, end: int = 100) -> List[int]

## 2.1 Overview

The generate\_random\_integers function generates a list of a specified number of pseudo-random integers within a given inclusive range.

## 2.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| count | int | The number of random integers to generate. This value must be non-negative. |
| start | int | The inclusive lower bound for the random numbers. Defaults to 0. |
| end | int | The inclusive upper bound for the random numbers. Defaults to 100. |

## 2.3 Description

This function provides a convenient way to create a list of pseudo-random integers. Its operation can be broken down into three main steps:

1. **Input Validation**: The function first checks if the count parameter is a non-negative number. If count is less than zero, it is considered invalid for generating a list, and the function will raise a ValueError.
2. **Boundary Correction**: It then compares the start and end parameters. If start is found to be greater than end, the function automatically swaps their values. This ensures that the range is always valid, preventing errors in the random number generation step. For example, a call with start=50 and end=10 will be treated as start=10 and end=50.
3. **Random Number Generation**: Using a list comprehension, the function iterates count times. In each iteration, it calls random.randint(start, end) to produce a single integer that is uniformly sampled from the inclusive range [start, end]. These integers are collected into a list.

# The core generation logic  
return [random.randint(start, end) for \_ in range(count)]

Finally, the function returns the newly created list of random integers.

## 2.4 Usage Notes

* This function requires the random module to be imported in the script.
* If the start value is greater than the end value, they will be swapped internally to ensure a valid range.
* Providing a negative value for the count parameter will raise a ValueError.
* The default behavior, if no start or end is provided, is to generate numbers in the range [0, 100].

**Output Example**: The function returns a list of integers. The length of the list will be equal to the count parameter.

## 2.5 Example

import random # This import is necessary for the function to work  
  
# Example 1: Generate 5 random integers between 10 and 20.  
result = generate\_random\_integers(5, 10, 20)  
print(result)  
  
# Example 2: Generate 3 random integers using default bounds (0 to 100).  
result\_default = generate\_random\_integers(3)  
print(result\_default)

**Output:**

(Note: The actual output will vary with each execution due to its random nature)

[12, 18, 11, 20, 15]  
[87, 2, 54]

## 2.6 FunctionDef fibonacci(n)

# 3 Function: fibonacci(n: int)

## 3.1 Overview

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

## 3.2 parameters

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

## 3.3 Description

This function provides a memory-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 a negative number. Since the Fibonacci sequence is not defined for negative indices, it raises a ValueError if this condition is met.
2. **Initialization**: Two variables, a and b, are initialized to 0 and 1 respectively. These represent the first two numbers in the sequence, F₀ and F₁.
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 previous a and b, effectively advancing one step in the sequence.
4. **Return Value**: After the loop completes, the variable a holds the nth Fibonacci number. If n is 0, the loop does not run, and the initial value of a (0) is returned, which is correct.

For example, to compute fibonacci(3): - Initial state: a = 0, b = 1 - Loop 1: a becomes 1, b becomes 0 + 1 = 1 - Loop 2: a becomes 1, b becomes 1 + 1 = 2 - Loop 3: a becomes 2, b becomes 1 + 2 = 3 - The loop finishes, and the function returns the final value of a, which is 2.

## 3.4 Usage Notes

* The input n must be a non-negative integer. The function will raise a ValueError for negative inputs.
* The sequence is 0-indexed, meaning fibonacci(0) returns the first number in 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 deep recursion stacks.

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

34

## 3.5 Example

# Example usage to find the 9th Fibonacci number (0-indexed)  
result = fibonacci(9)  
print(result)

**Output:**

34

## 3.6 FunctionDef choose\_random\_item(items)

# 4 Function: choose\_random\_item(items: List[str])

## 4.1 Overview

The choose\_random\_item function selects and returns a single random item from a given non-empty list of strings.

## 4.2 parameters

* items (List[str]): A list of strings to choose from. This list must not be empty.

## 4.3 Description

This function provides a safe way to select a random element from a list of strings.

The function first performs a validation check on the input items list. It uses the condition if not items to determine if the list is empty. If the list has no elements, the condition is True, and the function raises a ValueError with the message “items must not be empty”. This prevents runtime errors that would occur if trying to select an item from an empty sequence.

If the list is not empty, the function proceeds to use the random.choice() method. This method, from Python’s built-in random module, takes a sequence as an argument and returns a single item chosen uniformly at random. The item selected from the items list is then returned as the result of the choose\_random\_item function.

# Internally, the function first checks the list  
if not items:  
 raise ValueError("items must not be empty")  
# Then, it returns a random choice  
return random.choice(items)

## 4.4 Usage Notes

* The input list items must contain at least one element. Providing an empty list will result in a ValueError.
* This function requires the random module to be imported in the execution environment.
* The selection is uniformly random, meaning every item in the list has an equal probability of being chosen.

**Output Example**: A single string from the input list, such as "cherry".

## 4.5 Example

import random  
from typing import List  
  
# Definition of the function  
def choose\_random\_item(items: List[str]) -> str:  
 """Choose a single random item from a non-empty sequence."""  
 if not items:  
 raise ValueError("items must not be empty")  
 return random.choice(items)  
  
# Example usage  
options = ["apple", "banana", "cherry", "date"]  
random\_fruit = choose\_random\_item(options)  
print(f"A random fruit was chosen: {random\_fruit}")  
  
# Example of error handling  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error caught: {e}")

**Output:**

A random fruit was chosen: banana  
Error caught: items must not be empty

(Note: The chosen fruit in the first line of the output is random and will vary with each execution.)

## 4.6 FunctionDef shuffle\_copy(items)

# 5 Function: shuffle\_copy(items: List[int])

## 5.1 Overview

The shuffle\_copy function returns a new, randomly shuffled copy of a given list, ensuring the original list remains unchanged.

## 5.2 parameters

* items (List[int]): A list of integers that will be copied and then shuffled.

## 5.3 Description

This function provides a safe way to shuffle a list without altering the original data structure. The process is executed in three main steps:

1. A shallow copy of the input items list is created using the list() constructor. This new list is stored in a variable named copy. This step is crucial for preserving the original list.
2. The random.shuffle() function is then called on the copy. random.shuffle() shuffles the elements of a sequence *in-place*, meaning it directly modifies the copy list by rearranging its elements into a random order.
3. Finally, the function returns the modified copy list, which now contains the same elements as the original items list but in a new, random sequence.

# Internal logic breakdown  
copy = list(items) # Create a new list object with the same elements  
random.shuffle(copy) # Shuffle the new list in-place  
return copy # Return the shuffled new list

## 5.4 Usage Notes

* This function is non-mutating. It will not change the order of the original list passed as the items argument.
* The function depends on Python’s built-in random module. Ensure that import random is present in the file where this function is used.
* The returned value is a new list object, not a reference to the original.
* While the type hint specifies List[int], the function will work correctly with lists containing any type of element (e.g., strings, floats, or mixed types).

**Output Example**: A possible return value for an input of [1, 2, 3, 4, 5].

[4, 1, 5, 3, 2]

## 5.5 Example

The following example demonstrates how to use shuffle\_copy and confirms that the original list remains unmodified.

import random  
  
# Assume shuffle\_copy is defined in the same scope  
def shuffle\_copy(items: list) -> list:  
 copy = list(items)  
 random.shuffle(copy)  
 return copy  
  
# Define an original list of numbers  
original\_list = [10, 20, 30, 40, 50]  
print(f"Original list before shuffling: {original\_list}")  
  
# Get a shuffled copy of the list  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Shuffled copy: {shuffled\_list}")  
print(f"Original list after shuffling: {original\_list}")

**Output:**

Original list before shuffling: [10, 20, 30, 40, 50]  
Shuffled copy: [30, 50, 10, 20, 40] # Note: The order is random and will vary with each execution.  
Original list after shuffling: [10, 20, 30, 40, 50]