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 input numbers.

## 1.2 parameters

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

## 1.3 Description

This function provides a straightforward way to perform addition. It accepts two arguments, a and b, which are expected to be numeric types such as integers or floating-point numbers. The core logic of the function is the addition operator (+), which computes the sum of a and b. The resulting value is then returned by the function.

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

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

## 1.4 Usage Notes

* This function is designed for numeric types. While the + operator also works for string concatenation, using this function for that purpose is not its intended design and may lead to unexpected behavior.
* Passing incompatible types (e.g., an integer and a string) will result in a TypeError.

**Output Example**: A numeric value representing the sum of the inputs. For instance: 15

## 1.5 Example

# Example usage with two integers  
result = num(5, 10)  
print(result)  
  
# Example usage with two floats  
float\_result = num(3.14, 2.71)  
print(float\_result)

**Output:**

15  
5.85

## 1.6 FunctionDef generate\_random\_integers

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

## 2.1 Overview

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

## 2.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| count | int | The total number of integers to generate in the list. |
| start | int | The inclusive lower bound for the random values. Defaults to 0. |
| end | int | The inclusive upper bound for the random values. Defaults to 100. |

## 2.3 Description

This function provides a straightforward way to generate a list of random integers. The process is as follows:

1. **Input Validation**: The function first validates the count parameter. If count is a negative number, it raises a ValueError because it’s impossible to generate a negative quantity of items.
2. **Range Correction**: It checks if the start value is greater than the end value. If this condition is true, it swaps the two values. This ensures that the range is always valid (start <= end) for the random number generation logic, making the function more robust and user-friendly.
3. **Generation**: The function then uses a list comprehension to build the final list. It iterates count times, and in each iteration, it calls random.randint(start, end). The random.randint method returns a single integer sampled uniformly from the inclusive range [start, end].

The resulting list, containing count random integers, is then returned.

# Internal logic for generating 5 numbers between 10 and 20  
# Assumes the 'random' module is imported  
count = 5  
start = 10  
end = 20  
random\_list = [random.randint(start, end) for \_ in range(count)]  
# random\_list might be [12, 19, 10, 15, 11]

## 2.4 Usage Notes

* This function depends on Python’s built-in random module. Ensure it is imported (import random) before calling this function.
* The count parameter must be a non-negative integer.
* Both the start and end boundaries are inclusive, meaning they can appear in the output list.
* If start is provided as a larger number than end, the function will automatically swap them to create a valid range.

**Output Example**: A call to generate\_random\_integers(5, 1, 100) might produce a list similar to this:

[42, 88, 15, 97, 23]

## 2.5 Example

import random  
from typing import List  
  
def generate\_random\_integers(count: int, start: int = 0, end: int = 100) -> List[int]:  
 """Return a list of pseudo-random integers.  
  
 Parameters:  
 count: Number of integers to generate.  
 start: Inclusive lower bound for values.  
 end: Inclusive upper bound for values.  
  
 Returns:  
 A list containing `count` integers sampled uniformly in [start, end].  
 """  
 if count < 0:  
 raise ValueError("count must be non-negative")  
 if start > end:  
 start, end = end, start  
 return [random.randint(start, end) for \_ in range(count)]  
  
# Example usage: Generate 5 random integers between 10 and 20.  
result = generate\_random\_integers(5, 10, 20)  
print(result)

**Output:**

[15, 11, 20, 18, 13]

## 2.6 FunctionDef fibonacci

# 3 Function: fibonacci(n: int)

## 3.1 Overview

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

## 3.2 parameters

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

## 3.3 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 (e.g., 0, 1, 1, 2, 3, 5, 8…).

The function begins by validating the input. 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 core logic is within 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.

For example: - Start: 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

After the loop completes, the variable a holds the nth Fibonacci number, which is then returned.

# Initialization for n > 0  
a, b = 0, 1  
# After one loop iteration  
a, b = b, a + b # a becomes 1, b becomes 1

## 3.4 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, 0.
* This iterative implementation is highly efficient in terms of memory and performance, avoiding the overhead and potential stack overflow of naive recursive solutions for large n.

**Output Example**: The function returns a single integer representing the Fibonacci number at the specified index. For fibonacci(9), the returned value is 34.

## 3.5 Example

# Example usage  
# Calculate the 9th Fibonacci number (0-indexed)  
result = fibonacci(9)  
print(f"The 9th Fibonacci number is: {result}")  
  
# Example for 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

## 3.6 FunctionDef choose\_random\_item

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

## 4.1 Overview

The choose\_random\_item function selects and returns a single, random element from a given list of strings.

## 4.2 parameters

* items (List[str]): A list of strings from which a random item will be selected. This list must not be empty.

## 4.3 Description

This function provides a simple and safe way to get a random item from a list. The logic operates in two main steps:

1. **Validation**: The function first validates the input items list. It checks if the list is empty using the if not items: condition. If the list is found to be empty, it raises a ValueError with the message “items must not be empty”. This prevents potential errors from the underlying random.choice function and enforces the contract that the input must be a non-empty sequence.
2. **Random Selection**: If the validation passes (i.e., the list is not empty), the function proceeds to call random.choice(items). This standard library function selects a single item from the list with a uniform probability distribution, meaning every item has an equal chance of being chosen. The selected string is then returned as the result.

## 4.4 Usage Notes

* The input items list must contain at least one element. Providing an empty list will raise a ValueError.
* This function depends on Python’s random module. Ensure it is imported in the scope where this function is defined or used.
* The selection is uniformly random, which means each item in the list has an equal probability of being chosen on any given call.

**Output Example**: A single string chosen from the input list. For an input of ['cat', 'dog', 'bird'], a possible return value is 'dog'.

## 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 = ["rock", "paper", "scissors"]  
result = choose\_random\_item(options)  
print(result)

**Output:**

paper

## 4.6 FunctionDef shuffle\_copy

# 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] - The input list of integers to be shuffled.

## 5.3 Description

This function provides a safe way to get a randomized version of a list without altering the original data structure. This is often referred to as a non-mutating or immutable operation from the caller’s perspective.

The function operates in three main steps: 1. It first creates a shallow copy of the input items list by calling list(items). This new list, assigned to the copy variable, is an independent object in memory. 2. It then uses the random.shuffle() method to rearrange the elements of the copy list in-place. The random.shuffle() function shuffles the sequence randomly. 3. Finally, it returns the shuffled copy, leaving the original items list in its initial, unmodified state.

# Internal logic breakdown  
def shuffle\_copy(items: List[int]) -> List[int]:  
 # 1. Create a new list, independent of the original 'items'  
 copy = list(items)  
   
 # 2. Shuffle the new 'copy' list in-place  
 random.shuffle(copy)  
   
 # 3. Return the shuffled copy  
 return copy

## 5.4 Usage Notes

* **Non-Mutation**: The primary feature of this function is that it does not modify the original list provided as the items parameter. This prevents unintended side effects in your code.
* **Random Module Dependency**: This function depends on Python’s standard random module. You must ensure import random is present at the top of your script for this function to work.
* **Shallow Copy**: The function creates a shallow copy. For a list of simple data types like integers, this is sufficient. If the list contained mutable objects (e.g., other lists or dictionaries), the nested objects themselves would not be copied.

**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 is not changed.

import random  
from typing import List  
  
def shuffle\_copy(items: List[int]) -> List[int]:  
 """Return a shuffled copy of the given list without mutating the input."""  
 copy = list(items)  
 random.shuffle(copy)  
 return copy  
  
# Example usage  
original\_numbers = [10, 20, 30, 40, 50]  
print(f"Original list before shuffling: {original\_numbers}")  
  
shuffled\_numbers = shuffle\_copy(original\_numbers)  
print(f"Shuffled list (new): {shuffled\_numbers}")  
print(f"Original list after shuffling: {original\_numbers}")

**Output:**

Original list before shuffling: [10, 20, 30, 40, 50]  
Shuffled list (new): [30, 50, 10, 20, 40]  
Original list after shuffling: [10, 20, 30, 40, 50]

*(Note: The order of elements in the “Shuffled list” will vary with each execution due to its random nature.)*