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 arguments.

## 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 performs a basic addition operation. It takes two parameters, a and b, and computes their sum using the + operator. The resulting value is then returned by the function. The primary use case is for adding numerical types such as integers and floats.

# The function adds 'a' and 'b' and returns the result.  
return a + b

## 1.4 Usage Notes

* While designed for numbers, this function will also work with any data types that support the addition (+) operator, such as string concatenation.
* Providing incompatible types (e.g., an int and a str) will result in a TypeError.

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

15

## 1.5 Example

# Example usage with two integers  
result = num(5, 10)  
print(result)

**Output:**

15

## 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 creates and returns a list of a specified number of pseudo-random integers within a defined inclusive range.

## 2.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| count | int | The total number of random integers to generate in the list. |
| 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 generate a list of random integers. It operates in three main steps:

1. **Input Validation**: The function first checks if the count parameter is a non-negative number. If count is less than 0, it raises a ValueError to prevent invalid list creation.
2. **Range 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 start is always the lower bound and end is the upper bound, making the function more robust and user-friendly.
3. **Random Number Generation**: Finally, it uses a list comprehension combined with the random.randint(start, end) method to generate the list. It iterates count times, and in each iteration, it produces a single random integer that is inclusive of both the start and end values. These integers are collected into a list and returned.

# Internal logic for generating 3 numbers between 1 and 5  
# This demonstrates the core list comprehension  
[random.randint(1, 5) for \_ in range(3)]

## 2.4 Usage Notes

* The count parameter must be a non-negative integer. Providing a negative value will result in a ValueError.
* The function gracefully handles cases where the start value is greater than the end value by swapping them internally.
* The range defined by start and end is inclusive, meaning both start and end values can appear in the output list.
* This function requires the random module to be imported in the execution environment.

**Output Example**: The function returns a list of integers. The length of the list will be equal to the count argument. For example, generate\_random\_integers(5, 1, 10) might return [3, 1, 9, 10, 5].

## 2.5 Example

# Example usage of generate\_random\_integers  
# Generate 5 random integers between 10 and 20 (inclusive)  
random\_numbers = generate\_random\_integers(count=5, start=10, end=20)  
print(random\_numbers)  
  
# Example with default parameters (5 numbers between 0 and 100)  
default\_random\_numbers = generate\_random\_integers(count=5)  
print(default\_random\_numbers)  
  
# Example with swapped start and end values  
swapped\_range\_numbers = generate\_random\_integers(count=3, start=50, end=40)  
print(swapped\_range\_numbers)

**Output:**

[15, 11, 20, 18, 10]  
[87, 23, 5, 66, 91]  
[42, 48, 45]

*(Note: The actual output will vary with each execution due to the random nature of the function.)*

## 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 value.

## 3.3 Description

This function calculates the Fibonacci number at a specific 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 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 core logic resides in 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 steps through the Fibonacci sequence.

# Inside the loop for n > 0  
a, b = b, a + b

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 run, and the initial value of a (0) is correctly returned.

## 3.4 Usage Notes

* The input n must be a non-negative integer. Providing a negative value will result in a ValueError.
* The function is 0-indexed, meaning fibonacci(0) returns the first number in the sequence (0), fibonacci(1) returns the second (1), and so on.
* This iterative implementation is memory-efficient compared to a naive recursive approach, especially for large values of n, as it avoids a deep call stack.

**Output Example**: A possible return value for a valid input.

55

## 3.5 Example

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

**Output:**

55

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

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

## 4.1 Overview

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

## 4.2 parameters

* **items** (List[str]): A list of strings from which one item will be randomly selected. This list cannot be empty.

## 4.3 Description

This function provides a safe way to choose a random item from a list.

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 is empty, this condition evaluates to True, and the function raises a ValueError with the descriptive message “items must not be empty”. This prevents the underlying random.choice function from being called with an invalid argument, which would also result in an error.

If the items list is not empty, the function proceeds to call random.choice(items). This is a standard library function that takes a non-empty sequence and returns an element chosen uniformly at random. “Uniformly” means that every item in the list has an equal probability of being selected. The selected string is then returned as the output.

## 4.4 Usage Notes

* The input list items must not be empty. Providing an empty list will result in a ValueError.
* This function depends on Python’s built-in random module. Ensure it is imported in your script before calling this function.
* The selection is uniformly random, meaning each item in the list has an equal chance of being chosen on any given call.

**Output Example**: A single string from the input list. For an input of ['red', 'green', 'blue'], a possible return value is 'green'.

## 4.5 Example

import random  
from typing import List  
  
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 1: Choosing from a valid list  
colors = ["red", "green", "blue", "yellow"]  
random\_color = choose\_random\_item(colors)  
print(f"The chosen color is: {random\_color}")  
  
# Example 2: Handling the error case with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error when choosing from an empty list: {e}")

**Output:**

The chosen color is: blue  
Error when choosing from an empty list: items must not be empty

*(Note: The output for “The chosen color is:” will vary as it is selected randomly from the list.)*

## 4.6 FunctionDef shuffle\_copy(items)

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

## 5.1 Overview

The shuffle\_copy function returns a shuffled copy of a given list without modifying the original list.

## 5.2 parameters

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

## 5.3 Description

This function provides a safe, non-mutating way to shuffle the elements of a list. The process begins by creating a shallow copy of the input items list using the list() constructor. This step is crucial as it ensures that all subsequent operations are performed on a new list, leaving the original items list untouched.

Once the copy is created, the function utilizes the random.shuffle() method to rearrange the elements of the copy in a random, in-place order. Finally, this newly shuffled copy is returned to the caller.

# Internal logic  
copy = list(items)  
random.shuffle(copy)  
return copy

## 5.4 Usage Notes

* The primary feature of this function is that it is non-mutating. The original list provided as input will not be changed.
* The function relies on Python’s random module, which must be imported in the file where shuffle\_copy is defined.
* The function creates a shallow copy. For a list of simple data types like integers, this is perfectly safe. If the list contained mutable objects (e.g., other lists or dictionaries), the nested objects themselves would not be duplicated.

**Output Example**: A new list containing the same integers as the input items list, but arranged in a random sequence. For an input of [1, 2, 3], a possible output is [2, 3, 1].

## 5.5 Example

import random # This dependency is required for the shuffle\_copy function to work  
  
# Example usage  
original\_numbers = [1, 2, 3, 4, 5]  
shuffled\_numbers = shuffle\_copy(original\_numbers)  
  
print(f"Original List (unchanged): {original\_numbers}")  
print(f"Shuffled Copy: {shuffled\_numbers}")  
  
# Another run will produce a different random order  
shuffled\_again = shuffle\_copy(original\_numbers)  
print(f"Shuffled Again: {shuffled\_again}")

**Output:**

Original List (unchanged): [1, 2, 3, 4, 5]  
Shuffled Copy: [3, 5, 1, 4, 2]  
Shuffled Again: [5, 2, 4, 1, 3]  
# Note: The order of elements in the shuffled lists will be random on each execution.