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 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 provides a straightforward way to perform addition. It accepts two parameters, a and b, which are expected to be numeric types. The function then uses the standard addition operator (+) to compute their sum. The resulting value from the expression a + b is immediately returned to the caller.

# The function's logic is a single addition operation  
return a+b

The primary purpose of this function is to encapsulate the addition of two numbers.

## 1.4 Usage Notes

* The function is designed to work with numeric types such as integers (int) and floating-point numbers (float).
* If string types are passed as arguments, the function will perform string concatenation instead of arithmetic addition. For example, num("hello", "world") would return "helloworld".

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

## 1.5 Example

# Example usage with integers  
result\_int = num(10, 5)  
print(f"The sum of integers is: {result\_int}")  
  
# Example usage with floats  
result\_float = num(7.5, 2.2)  
print(f"The sum of floats is: {result\_float}")

**Output:**

The sum of integers is: 15  
The sum of floats is: 9.7

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

# 2 Function: generate\_random\_integers(count: int, start: int = 0, end: int = 100)

## 2.1 Overview

The generate\_random\_integers function 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 number of random integers to generate. This value must be non-negative. |
| start | int | The inclusive lower bound for the generated values. Defaults to 0. |
| end | int | The inclusive upper bound for the generated values. Defaults to 100. |

## 2.3 Description

This function provides a convenient way to generate a list of random integers. It operates in a few distinct steps:

1. **Input Validation**: It first checks if the count parameter is less than zero. If it is, a ValueError is raised, as it’s impossible to generate a negative number of integers.
2. **Range Correction**: The function then compares the start and end parameters. If start is found to be greater than end, it automatically swaps their values. This ensures that the range is always valid for the random number generation logic, making the function more robust against user error.
3. **Generation**: Finally, it uses a list comprehension to generate the list of integers. It calls random.randint(start, end) for a total of count times, collecting each result into a list which is then returned. The use of random.randint means the range [start, end] is inclusive, so both the start and end values can be part of the output.

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

## 2.4 Usage Notes

* This function requires the random module to be imported.
* A ValueError will be raised if count is a negative number.
* The function gracefully handles cases where start > end by swapping the values internally.
* The generated integers are sampled uniformly from the inclusive range [start, end].

**Output Example**: A possible return value for generate\_random\_integers(5, 1, 10). The actual numbers will vary with each execution.

[8, 2, 10, 5, 3]

## 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 7 random integers between 50 and 60.  
random\_list = generate\_random\_integers(7, 50, 60)  
print(random\_list)  
  
# Example with swapped start and end values  
# The function will internally treat this as start=1, end=10  
swapped\_list = generate\_random\_integers(5, 10, 1)  
print(swapped\_list)

**Output:**

[58, 51, 55, 60, 53, 50, 52]  
[5, 9, 2, 10, 3]

*(Note: The actual output will vary as the numbers are generated randomly.)*

## 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. This must be a non-negative integer.

## 3.3 Description

This function calculates a Fibonacci number based on its index n. The logic proceeds as follows:

1. **Input Validation**: The function first checks if the input n is negative. 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 to advance the sequence.
4. **Calculation**: The core of the logic is the tuple assignment a, b = b, a + b. In each step:
   * The current value of b is assigned to a.
   * The sum of the previous a and b is assigned to b. This effectively shifts the sequence forward one position, with a holding the value of F(i) and b holding F(i+1) at the end of each iteration i.
5. **Return Value**: After the loop completes n iterations, the variable a holds the nth Fibonacci number. 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.

## 3.4 Usage Notes

* The input n must be a non-negative integer. Providing a negative number will result in a ValueError.
* The function is 0-indexed, meaning fibonacci(0) returns the first number in the sequence, which is 0.
* This iterative implementation is efficient and avoids the performance issues and potential stack overflow errors associated with naive recursive solutions, especially for large values of n.

**Output Example**:

55

## 3.5 Example

# Example usage to find the 10th Fibonacci number  
n\_index = 10  
result = fibonacci(n\_index)  
print(f"The Fibonacci number at index {n\_index} is: {result}")

**Output:**

The Fibonacci number at index 10 is: 55

## 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 string from a provided list of strings.

## 4.2 parameters

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

## 4.3 Description

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

The function first performs a validation check to ensure the input list items is not empty. If the list is empty, it raises a ValueError with the message “items must not be empty” to prevent runtime errors from the underlying random selection logic.

If the list contains one or more items, the function then utilizes random.choice(items) to perform the selection. The random.choice() method selects one item uniformly at random from the sequence, meaning each item has an equal probability of being chosen. The selected string is then returned as the result.

## 4.4 Usage Notes

* The input list items must contain at least one element. Passing an empty list will result in a ValueError.
* This function depends on Python’s built-in random module. Ensure it is imported in the scope where the function is used.
* The selection is uniformly random, giving every item in the list an equal chance of being picked.

**Output Example**: A single string from the input list.

"example\_string"

## 4.5 Example

import random # This import is necessary for the function to work  
  
# 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 = ["red", "green", "blue", "yellow"]  
selected\_color = choose\_random\_item(options)  
print(f"The randomly selected color is: {selected\_color}")  
  
# Example of what happens with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

The randomly selected color is: blue  
Error: items must not be empty

*(Note: The selected color will vary with each execution as it is chosen randomly.)*

## 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 of integers, leaving the original list unmodified.

## 5.2 parameters

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

## 5.3 Description

This function provides a safe method for shuffling a list by ensuring the original data structure remains intact. The logic proceeds in the following steps:

1. A shallow copy of the input items list is created using the expression copy = list(items). This is a critical step that allocates a new list in memory, preventing any changes to the original list that was passed as an argument.
2. The random.shuffle() method is then called on the newly created copy. This function, from Python’s standard random module, shuffles the elements of the copy list *in-place*, rearranging them into a random permutation.
3. Finally, the function returns the copy, which now contains the same elements as the original items list but in a new, randomized order.

## 5.4 Usage Notes

* The primary feature of this function is that it is non-mutating. The original list passed as the items parameter will not be changed.
* This function requires Python’s built-in random module. Ensure import random is included in your script before using this function.
* Although the type hint specifies List[int], the function’s logic will work correctly with lists containing elements of any data type (e.g., strings, floats, or objects).

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

[4, 1, 5, 2, 3]

## 5.5 Example

import random  
from typing import List  
  
# The function relies on the 'random' module.  
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]  
shuffled\_numbers = shuffle\_copy(original\_numbers)  
  
print(f"Original List (unchanged): {original\_numbers}")  
print(f"Shuffled Copy: {shuffled\_numbers}")  
  
# Another example with different data types  
original\_items = ['apple', 'banana', 'cherry']  
shuffled\_items = shuffle\_copy(original\_items)  
print(f"\nOriginal Items (unchanged): {original\_items}")  
print(f"Shuffled Items: {shuffled\_items}")

**Output:**

Original List (unchanged): [10, 20, 30, 40, 50]  
Shuffled Copy: [40, 10, 50, 20, 30]  
  
Original Items (unchanged): ['apple', 'banana', 'cherry']  
Shuffled Items: ['banana', 'cherry', 'apple']

*(Note: The actual order of the elements in the shuffled lists will vary with each execution.)*