Table of Contents

## 0.1 FunctionDef num(a, b)

# 1 Function: def num(a, b)

## 1.1 Overview

The num function returns the sum of two numbers or the concatenation of two strings.

## 1.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| a | int, float, str | The first operand for the addition or concatenation operation. |
| b | int, float, str | The second operand for the addition or concatenation operation. |

## 1.3 Description

The num function takes two arguments, a and b, and applies the addition operator (+) to them. The behavior of the operator is determined by the data types of the input arguments.

* If both a and b are numeric types (such as int or float), the function will perform arithmetic addition and return their sum.
* If both a and b are strings, the function will perform string concatenation, joining string b to the end of string a.

The function directly returns the result of the a + b operation.

# Numeric addition  
result\_sum = num(10, 5) # result\_sum will be 15  
  
# String concatenation  
result\_concat = num("hello", " world") # result\_concat will be "hello world"

## 1.4 Usage Notes

* This function can be used for both mathematical addition and string joining. Ensure that both parameters are of a compatible type for the + operator.
* Passing arguments of incompatible types (e.g., an integer and a string) will raise a TypeError.

**Output Example**: A numeric return value from the function.

15

## 1.5 Example

# Example 1: Adding two integers  
sum\_result = num(25, 75)  
print(f"The sum is: {sum\_result}")  
  
# Example 2: Concatenating two strings  
concat\_result = num("docu", "mentation")  
print(f"The concatenated string is: {concat\_result}")

**Output:**

The sum is: 100  
The concatenated string is: documentation

## 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 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. It operates in the following sequence:

1. **Input Validation**: It 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**: The function checks if the start value is greater than the end value. If it is, the function automatically swaps them. This ensures that start is always the lower bound and end is the upper bound, making the function more robust and forgiving of incorrect parameter order.
3. **Generation**: Using a list comprehension, the function calls random.randint(start, end) for a total of count times. The random.randint() method generates a single integer uniformly sampled from the inclusive range [start, end].
4. **Return Value**: The function collects these generated integers into a list and returns it.

# Internal logic for generating the list  
return [random.randint(start, end) for \_ in range(count)]

## 2.4 Usage Notes

* This function depends on Python’s built-in random module. Ensure it is imported (import random) before use.
* The count parameter must be a non-negative integer (>= 0). Providing a negative value will result in a ValueError.
* Both the start and end values are *inclusive*, meaning they can appear in the generated list.
* If the start value is provided as greater than the end value, the function will automatically swap them to form a valid range.

**Output Example**: A list containing integers. The contents will be random and vary with each execution.

[15, 8, 92, 43, 77]

## 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 (inclusive).  
random\_numbers = generate\_random\_integers(5, 10, 20)  
print(random\_numbers)

**Output:**

[12, 18, 10, 15, 20]

*(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 calculates the nth number in the Fibonacci sequence using an iterative method.

## 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 an efficient, iterative way to compute a Fibonacci number. The logic proceeds as follows:

1. **Input Validation**: The function first checks if the input n is less than 0. Since the Fibonacci sequence is not defined for negative indices, it raises a ValueError if the 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 0-indexed Fibonacci sequence (F₀ = 0, F₁ = 1).
3. **Iteration**: The function then enters a for loop that iterates n times. In each iteration, the core Fibonacci logic is executed:

* a, b = b, a + b
* This simultaneous assignment updates a to the value of b, and b to the sum of the old a and b. This effectively shifts the sequence forward one step.

1. **Return Value**: After the loop completes, the variable a holds the nth Fibonacci number. For n=0, the loop does not run, and the initial value of a (0) is correctly returned. For n > 0, the loop updates a to the desired value.

## 3.4 Usage Notes

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

**Output Example**:

55

## 3.5 Example

# Example usage to find the 10th Fibonacci number  
n\_value = 10  
result = fibonacci(n\_value)  
print(f"The Fibonacci number at index {n\_value} is: {result}")  
  
# Example with edge case n=0  
result\_zero = fibonacci(0)  
print(f"The Fibonacci number at index 0 is: {result\_zero}")

**Output:**

The Fibonacci number at index 10 is: 55  
The Fibonacci number at index 0 is: 0

## 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, uniformly random item from a given list of strings.

## 4.2 parameters

* **items** (List[str]): A non-empty list of strings from which to choose a random item.

## 4.3 Description

This function provides a simple way to get a random element from a list. The core logic is handled by Python’s built-in random.choice() method, which ensures that every item in the list has an equal probability of being selected.

Before attempting to select an item, the function first performs a validation check: if not items:. If the provided items list is empty, it raises a ValueError to prevent runtime errors from random.choice() and to inform the user that the input is invalid. If the list is not empty, it proceeds to return a single string chosen at random.

# Internal logic for a non-empty list  
return random.choice(items)

## 4.4 Usage Notes

* This function requires the random module to be imported in the script where it is used.
* The input list items must not be empty. Providing an empty list will result in a ValueError.
* Although the type hint specifies List[str], the underlying random.choice function can work with any non-empty sequence (like tuples or other lists). However, this implementation is explicitly typed for a list of strings.

**Output Example**: A single string from the input list. For an input of ["apple", "banana", "cherry"], a possible output is:

"banana"

## 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 usage:  
fruit\_options = ["apple", "banana", "cherry", "date", "elderberry"]  
selected\_fruit = choose\_random\_item(fruit\_options)  
print(f"The randomly selected fruit is: {selected\_fruit}")  
  
# Example of what happens with an empty list:  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

The randomly selected fruit is: cherry  
Error: items must not be empty

*(Note: The selected fruit in the first line of the output 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 creates and returns a new list containing the elements of the input list in a randomized order, without modifying the original list.

## 5.2 parameters

* items (List[int]): A list of integers that you want to shuffle.

## 5.3 Description

This function provides a safe way to shuffle a list by ensuring the original data structure is not altered. The process is as follows:

1. A shallow copy of the input items list is created using copy = list(items). This step is crucial as it isolates the operation from the original list.
2. The random.shuffle() function is then called on this copy. random.shuffle() shuffles the elements of a list in-place, randomizing their order.
3. Finally, the function returns the shuffled copy, leaving the original items list untouched.

This non-mutating behavior is useful when you need to preserve the original order of a list for other operations while also needing a randomized version of it.

## 5.4 Usage Notes

* This function does not modify (mutate) the original list provided as the items parameter. It always returns a new list.
* The function relies on Python’s built-in random module. Ensure this module is imported (import random) in the scope where shuffle\_copy is defined or used.
* While the type hint specifies List[int], the function will work correctly with lists containing any type of element, as random.shuffle is type-agnostic.

**Output Example**: The function returns a new list with the same elements as the input but in a random order. The exact order will vary with each execution.

[4, 1, 5, 3, 2]

## 5.5 Example

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.  
  
 Parameters:  
 items: A list of integers.  
  
 Returns:  
 A new list containing the same integers in random order.  
 """  
 copy = list(items)  
 random.shuffle(copy)  
 return copy  
  
# Example usage  
original\_list = [1, 2, 3, 4, 5]  
shuffled\_list = shuffle\_copy(original\_list)  
  
print("Original List:", original\_list)  
print("Shuffled Copy:", shuffled\_list)

**Output:**

Original List: [1, 2, 3, 4, 5]  
Shuffled Copy: [3, 5, 1, 2, 4]