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 be added. |
| b | int / float | The second number to be added. |

## 1.3 Description

This function provides a straightforward way to perform addition. It takes 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 to compute the sum of these two arguments using the + operator and return the resulting value.

For example, if a is 5 and b is 3, the function will compute 5 + 3 and return 8.

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

## 1.4 Usage Notes

* The function relies on Python’s built-in + operator. Ensure that the provided arguments are of types that support this operator for mathematical addition (e.g., int, float).
* If string values are passed, the function will perform string concatenation instead of mathematical addition. For example, num("hello", "world") would return "helloworld".
* The data type of the returned value will depend on the input types. For instance, adding an int and a float will result in a float.

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

8

## 1.5 Example

# Example usage  
result = num(5, 3)  
print(result)

**Output:**

8

## 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 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 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 robust way to generate a list of random integers. The process begins with input validation.

First, it checks if the count parameter is a negative number. If count is less than zero, the function raises a ValueError because it’s not possible to generate a negative number of items.

Next, it handles the range boundaries. If the provided start value is greater than the end value, the function automatically swaps them. This ensures that a valid range is always used for generation, preventing potential errors from random.randint.

Finally, the function uses a list comprehension to construct the output list. It iterates count times, and in each iteration, it calls random.randint(start, end) to produce a single pseudo-random integer. The random.randint method includes both start and end in the possible range of values. The resulting integers are collected into a list which is then returned.

# Internal logic for swapping bounds  
if start > end:  
 start, end = end, start  
# Core generation logic  
return [random.randint(start, end) for \_ in range(count)]

## 2.4 Usage Notes

* This function requires the random module to be imported to work correctly.
* The count parameter must be a non-negative integer (>= 0). Providing a negative value will result in a ValueError.
* The range defined by start and end is inclusive. For example, if start=1 and end=5, the possible numbers are 1, 2, 3, 4, and 5.
* The function gracefully handles cases where start is greater than end by swapping the values internally, so the order of these parameters does not strictly matter.

**Output Example**: A list of integers. The length of the list is determined by count.

[42, 17, 99, 5, 83]

## 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 1: Generate 5 random integers between 1 and 10.  
random\_numbers = generate\_random\_integers(5, 1, 10)  
print(f"Five random numbers between 1 and 10: {random\_numbers}")  
  
# Example 2: Use default parameters to generate 3 integers between 0 and 100.  
default\_numbers = generate\_random\_integers(3)  
print(f"Three random numbers with default range: {default\_numbers}")  
  
# Example 3: Demonstrate the swapping of start and end values.  
swapped\_range\_numbers = generate\_random\_integers(4, 50, 20)  
print(f"Four random numbers with swapped range (50, 20): {swapped\_range\_numbers}")

**Output:**

# Note: The actual output will vary with each execution due to randomness.  
Five random numbers between 1 and 10: [3, 8, 1, 10, 5]  
Three random numbers with default range: [76, 12, 98]  
Four random numbers with swapped range (50, 20): [33, 45, 21, 39]

## 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 of the desired Fibonacci number in the sequence.

## 3.3 Description

This function provides an 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 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 Fibonacci 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 simultaneously. The current value of b is assigned to a, and the sum of the old a and b is assigned to the new b. This operation, a, b = b, a + b, effectively advances one step in the Fibonacci sequence.
4. **Return Value**: After the loop completes, the variable a holds the value of the nth Fibonacci number. For n=0, the loop does not run, and the initial value of a (0) is returned. For n > 0, the loop updates a to the correct value before it is returned.

For example, to compute fibonacci(3): - Initialize 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. Providing a negative number will result in a ValueError.
* The function is 0-indexed, meaning fibonacci(0) returns the first number of the sequence (0), fibonacci(1) returns the second (1), and so on.
* This iterative implementation is memory-efficient compared to a simple recursive approach, as it avoids the overhead of a deep function call stack.

**Output Example**: A single integer representing the calculated Fibonacci number.

## 3.5 Example

# Example usage of the fibonacci function  
# Calculate the 9th Fibonacci number (0-indexed)  
n\_value = 9  
result = fibonacci(n\_value)  
print(f"The Fibonacci number at index {n\_value} is: {result}")

**Output:**

The Fibonacci number at index 9 is: 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, uniformly random element from a given list of strings.

## 4.2 parameters

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

## 4.3 Description

This function provides a safe way to select a random item from a list. The core logic is broken down into two main steps:

1. **Validation**: The function first checks if the provided items list is empty using the condition if not items:. This is a crucial guard clause to prevent runtime errors from the underlying random.choice method, which cannot operate on an empty sequence.
2. **Selection**: If the list is not empty, the function proceeds to call random.choice(items). This standard library function takes the list as input and returns one of its elements, with each element having an equal probability of being selected.

If the validation step fails (i.e., the list is empty), a ValueError is raised with the descriptive message “items must not be empty” to immediately inform the developer of the incorrect usage.

# Internal logic for a non-empty list  
return random.choice(items)  
  
# Internal logic for an empty list  
if not items:  
 raise ValueError("items must not be empty")

## 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 relies on Python’s built-in random module. Ensure it is imported in the file where this function is defined or called (e.g., import random).
* The selection is uniformly random, meaning every item in the list has an equal chance of being chosen on any given call.

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

"banana"

## 4.5 Example

import random  
from typing import List  
  
# The function definition from the documentation  
def choose\_random\_item(items: List[str]) -> str:  
 if not items:  
 raise ValueError("items must not be empty")  
 return random.choice(items)  
  
# Example 1: Choosing from a list of fruits  
fruit\_options = ["apple", "banana", "cherry", "date"]  
random\_fruit = choose\_random\_item(fruit\_options)  
print(f"A randomly chosen fruit is: {random\_fruit}")  
  
# Example 2: Demonstrating the error with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Caught expected error: {e}")

**Output:**

A randomly chosen fruit is: cherry  
Caught expected error: items must not be empty

*(Note: The output for the first print statement will vary with each execution due to its random nature.)*

## 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 random order, without modifying the original list.

## 5.2 parameters

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

## 5.3 Description

This function provides a safe way to shuffle a list by ensuring the original data structure remains unchanged. The logic proceeds in three 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 modification to the original items list passed to the function.
2. The random.shuffle(copy) method is then called on this new copy. This function, part of Python’s standard random module, rearranges the elements of the copy list into a random order. The shuffling happens in-place on the copy.
3. Finally, the function returns the copy, which now contains the same elements as the original list but in a new, randomized sequence.

# The function relies on the 'random' module  
import random  
  
# Step 1: A copy is made  
original\_list = [10, 20, 30]  
copy\_of\_list = list(original\_list) # copy\_of\_list is now [10, 20, 30]  
  
# Step 2: The copy is shuffled in-place  
random.shuffle(copy\_of\_list) # copy\_of\_list might now be [30, 10, 20]  
  
# Step 3: The shuffled copy is returned  
# The function returns copy\_of\_list

## 5.4 Usage Notes

* **Non-mutating:** The primary feature of this function is that it does not alter the input items list. It operates exclusively on a copy.
* **Dependency:** This function requires the random module to be imported (e.g., import random) to use the random.shuffle() method.
* **Randomness:** The output list’s order is non-deterministic. Each call to the function will likely produce a different order of elements.

**Output Example**: A new list with the same elements as the input but in a randomized order. For an input of [1, 2, 3], a possible output is [3, 1, 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."""  
 copy = list(items)  
 random.shuffle(copy)  
 return copy  
  
# Example usage  
original\_numbers = [1, 2, 3, 4, 5]  
shuffled\_numbers = shuffle\_copy(original\_numbers)  
  
print(f"Original List: {original\_numbers}")  
print(f"Shuffled Copy: {shuffled\_numbers}")

**Output:**

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

*(Note: The order of elements in “Shuffled Copy” will vary with each execution.)*