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 accepts two arguments, a and b, which are expected to be numeric types such as integers or floating-point numbers. The function computes their sum using the standard addition operator (+) and returns the resulting value.

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

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

## 1.4 Usage Notes

* The function is designed for numeric types. If strings are passed as arguments, it will perform string concatenation instead of mathematical addition (e.g., num("hello", "world") returns "helloworld").
* Passing incompatible types (e.g., an integer and a None type) 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)  
  
# Example usage with a float and an integer  
result\_float = num(7.5, 3)  
print(result\_float)

**Output:**

15  
10.5

## 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 given 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 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 convenient way to generate a list of pseudo-random integers. It operates in a few distinct steps:

1. **Input Validation**: The function first validates the count parameter. If count is a negative number, it raises a ValueError because it’s not possible to generate a negative number of items.
2. **Range Correction**: It checks if the start value is greater than the end value. If it is, the function automatically swaps them to ensure a valid range. This makes the function more robust and user-friendly, as the user does not need to worry about the order of the start and end parameters.
3. **Integer Generation**: Using a list comprehension, the function calls random.randint(start, end) for a total of count times. The random.randint() method generates an integer from a uniform distribution within the inclusive range [start, end], meaning both start and end are possible values. The generated integers are collected into a list which is then returned.

# Assumes the 'random' module is imported  
import random  
  
# Example of internal logic  
# If start=50 and end=10, they are swapped to start=10, end=50  
start, end = 50, 10  
if start > end:  
 start, end = end, start  
# The generation will now correctly use random.randint(10, 50)

## 2.4 Usage Notes

* This function depends on Python’s built-in random module, which must be imported before use.
* An error will be raised if a negative value is provided for the count parameter.
* The function gracefully handles cases where start > end by swapping the values internally.
* The range defined by start and end is inclusive, meaning both endpoints can be included in the output.

**Output Example**: A list of integers. The actual values will be random and vary with each execution. [42, 8, 99, 23, 76]

## 2.5 Example

import random  
  
# The function definition from the problem description  
def generate\_random\_integers(count: int, start: int = 0, end: int = 100):  
 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 in the default range [0, 100]  
result1 = generate\_random\_integers(5)  
print(f"Generated integers (default range): {result1}")  
  
# Example 2: Generate 7 random integers in a custom range [10, 20]  
result2 = generate\_random\_integers(count=7, start=10, end=20)  
print(f"Generated integers (custom range): {result2}")  
  
# Example 3: Demonstrate the swapping of start and end  
result3 = generate\_random\_integers(count=4, start=50, end=40)  
print(f"Generated integers (swapped range): {result3}")

**Output:**

# The output is random and will differ on each run.  
Generated integers (default range): [81, 9, 64, 3, 78]  
Generated integers (custom range): [15, 11, 20, 18, 10, 13, 19]  
Generated integers (swapped range): [42, 49, 45, 41]

## 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: An integer representing the 0-indexed position in the Fibonacci sequence for which to find the value.

## 3.3 Description

This function calculates a Fibonacci number based on its index n. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.

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, F(0) and F(1).

The function then enters a for loop that iterates n times. In each iteration, the values of a and b are updated. 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 moves one step forward in the sequence.

For example: - Initial: a = 0, b = 1 - After 1st iteration: a = 1, b = 1 (0 + 1) - After 2nd iteration: a = 1, b = 2 (1 + 1) - After 3rd iteration: a = 2, b = 3 (1 + 2)

After the loop completes, the variable a holds the nth Fibonacci number, which is then returned. If n is 0, the loop does not run, and the initial value of a (0) is returned correctly.

# Inside the function  
if n < 0:  
 raise ValueError("n must be non-negative")  
a, b = 0, 1  
for \_ in range(n):  
 a, b = b, a + b  
return a

## 3.4 Usage Notes

* The input n must be a non-negative integer. The function will raise a ValueError if a negative integer is provided.
* The sequence is 0-indexed, meaning fibonacci(0) returns the first element 0, fibonacci(1) returns the second element 1, and so on.
* This iterative implementation is efficient in terms of memory and performance for large values of n compared to a simple recursive solution, as it avoids redundant calculations and deep recursion stacks.

**Output Example**: The function returns a single integer value. 34

## 3.5 Example

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

**Output:**

The Fibonacci number at index 9 is: 34  
The Fibonacci number at index 0 is: 0

## 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 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 safe way to select a random element from a list. The logic proceeds as follows:

1. **Input Validation**: The function first checks if the provided items list is empty.
2. **Error Handling**: If the list is empty (if not items), it raises a ValueError with the message “items must not be empty”. This prevents runtime errors that would occur if an empty list were passed to the underlying selection mechanism.
3. **Random Selection**: If the list is not empty, the function utilizes the random.choice() method. This method takes the items list as an argument and returns one of its elements, chosen uniformly at random.
4. **Return Value**: The randomly selected string is returned as the result of the function.

# Internal logic example  
import random  
  
def choose\_random\_item(items: list[str]) -> str:  
 if not items:  
 raise ValueError("items must not be empty")  
 return random.choice(items)

## 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 depends on Python’s built-in random module. Ensure it is available in the execution environment.
* The selection is uniformly random, meaning each item in the list has an equal probability of being chosen.

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

"banana"

## 4.5 Example

# Assumes the 'random' module is imported within the file where choose\_random\_item is defined.  
# from random\_module\_one import choose\_random\_item  
  
# Example usage  
options = ["red", "green", "blue", "yellow"]  
try:  
 selected\_color = choose\_random\_item(options)  
 print(f"The chosen color is: {selected\_color}")  
except ValueError as e:  
 print(e)  
  
# Example with an empty list  
empty\_options = []  
try:  
 selected\_item = choose\_random\_item(empty\_options)  
 print(selected\_item)  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

The chosen color is: blue  
Error: items must not be empty

(Note: The first line of the output is random and will be one of the items from the options list in each run.)

## 4.6 FunctionDef shuffle\_copy(items)

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

## 5.1 Overview

The shuffle\_copy function returns a new list containing the same elements as the input list but in a randomized order, without altering 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 operating on a copy rather than the original. The process is executed in three main steps:

1. A shallow copy of the input items list is created using copy = list(items). This step is crucial as it ensures that any subsequent modifications do not affect the original list provided by the user.
2. The random.shuffle(copy) function is called. This function, from Python’s built-in random module, shuffles the elements of the copy list in-place. It rearranges the items within the list into a random sequence.
3. Finally, the function returns the copy list, which now contains the original elements in a new, randomized order.

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

## 5.4 Usage Notes

* This function is non-mutating, meaning it will not change the original items list passed to it. It is designed specifically to return a new, shuffled list.
* The function relies on Python’s random module. Ensure that import random is present at the beginning of the script where this function is used.
* Although the type hint specifies List[int], the function will work correctly with lists containing other data types (e.g., strings, floats, or mixed types).

**Output Example**: The function returns a list where the elements from the input are in a random order. For an input of [1, 2, 3, 4, 5], a possible output could be:

[4, 1, 5, 3, 2]

## 5.5 Example

import random  
from typing import List  
  
# Definition of the function  
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\_list = [10, 20, 30, 40, 50]  
print(f"Original list before shuffling: {original\_list}")  
  
shuffled\_list = shuffle\_copy(original\_list)  
print(f"Shuffled list (new): {shuffled\_list}")  
print(f"Original list after shuffling: {original\_list}")

**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]