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 operand for the addition operation. |
| b | int / float | The second operand for the addition operation. |

## 1.3 Description

This function provides a straightforward way to perform addition. It accepts two parameters, a and b, which are expected to be numerical values (integers or floating-point numbers). The core logic of the function is the expression a + b, which computes the sum of the two inputs. The resulting value is then returned by the function.

Due to Python’s dynamic typing, the + operator can also perform concatenation if strings are passed as arguments. However, the function’s name num implies its primary purpose is for numerical calculations.

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

## 1.4 Usage Notes

* The function will work with any data types that support the addition (+) operator, such as integers, floats, and strings.
* If you provide incompatible types (e.g., an integer and a string), the function will raise a TypeError.
* When used with strings, the function will perform concatenation, not mathematical addition.

**Output Example**: If the inputs are 5 and 10, the function returns an integer.

15

## 1.5 Example

# Example usage with integers  
result\_int = num(10, 5)  
print(result\_int)  
  
# Example usage with floats  
result\_float = num(7.5, 2.5)  
print(result\_float)

**Output:**

15  
10.0

## 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 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 convenient way to generate a list of random integers. It begins by validating its input parameters to ensure logical consistency.

First, it checks if the count parameter is a negative number. If count is less than zero, it is impossible to create a list of that size, so the function raises a ValueError.

Next, it ensures that the range [start, end] is correctly ordered. If the provided start value is greater than the end value, the function automatically swaps them. This makes the function more robust, as the user does not need to worry about the order of these two parameters.

Finally, the function uses a list comprehension in conjunction with the random.randint(start, end) method. It iterates count times, and in each iteration, it generates a single random integer that is uniformly sampled from the inclusive range [start, end]. These integers are collected into a list, which is the final return value.

# Internal logic for generating 5 numbers between 1 and 10  
import random  
start, end = 1, 10  
count = 5  
# The list comprehension is equivalent to this loop:  
result\_list = []  
for \_ in range(count):  
 random\_number = random.randint(start, end)  
 result\_list.append(random\_number)  
# result\_list is then returned

## 2.4 Usage Notes

* This function requires the random module to be imported.
* The count parameter must be a non-negative integer. Providing a negative value will result in a ValueError.
* If the start value is greater than the end value, the function will swap them internally; you do not need to handle this case yourself.
* The range defined by start and end is inclusive, meaning both start and end are possible values in the output list.

**Output Example**: A list of integers.

[42, 8, 99, 54, 23]

## 2.5 Example

import random  
from typing import List  
  
# Example usage: Generate 5 random integers between 10 and 50.  
random\_numbers = generate\_random\_integers(5, 10, 50)  
print(random\_numbers)

**Output:**

[43, 15, 22, 49, 37]

## 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 calculates the nth Fibonacci number, where the sequence starts with F₀ = 0 and F₁ = 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. The function then enters a for loop that iterates n times. In each iteration, the values of a and b are updated using tuple assignment: a, b = b, a + b. This operation effectively shifts the sequence forward: the new a takes the value of the old b, and the new b becomes the sum of the old a and b.

After the loop completes, the variable a holds the nth Fibonacci number. For an input of n=0, the loop does not execute, and the initial value of a (which is 0) is returned, correctly giving F₀.

# For n = 4:  
# Initial: a = 0, b = 1  
# Loop 1: a = 1, b = 1 (F₂)  
# Loop 2: a = 1, b = 2 (F₃)  
# Loop 3: a = 2, b = 3 (F₄)  
# Loop 4: a = 3, b = 5 (F₅)  
# Loop finishes, returns a = 3

The function returns the final value of a.

## 3.4 Usage Notes

* The index n is 0-based. For example, fibonacci(0) returns 0, and fibonacci(1) returns 1.
* The function only accepts non-negative integers for the parameter n. Providing a negative integer will result in a ValueError.
* This iterative implementation is efficient in terms of memory usage, avoiding the deep recursion stack that a naive recursive solution would create, making it suitable for calculating larger Fibonacci numbers.

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

34

## 3.5 Example

# Example usage  
# 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}")  
  
# 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 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, uniformly random item from a given list of strings.

## 4.2 Parameters

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

## 4.3 Description

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

The function first performs a validation check to ensure the input list items is not empty. If the list is empty (if not items:), it immediately raises a ValueError with the message “items must not be empty” to prevent errors in the subsequent random selection process.

If the list contains one or more elements, the function proceeds to use random.choice(items). This standard library function handles the core logic of picking an element from the sequence with a uniform probability distribution, meaning every item has an equal chance of being selected. The chosen string is then returned as the output.

# The function relies on the 'random' module  
import random  
  
# Example of internal logic  
items\_list = ["apple", "banana", "cherry"]  
if not items\_list:  
 raise ValueError("items must not be empty")  
# If the list is not empty, random.choice is called  
selected\_item = random.choice(items\_list)  
# selected\_item will be "apple", "banana", or "cherry"

## 4.4 Usage Notes

* The input list items must not be empty. Providing an empty list will result in a ValueError.
* This function requires the random module to be imported in the execution environment.
* The selection is uniformly random, ensuring each item in the list has an equal probability of being chosen.

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

"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.  
  
 Parameters:  
 items: A list of strings to choose from.  
  
 Returns:  
 A single string chosen uniformly at random.  
 """  
 if not items:  
 raise ValueError("items must not be empty")  
 return random.choice(items)  
  
# Example usage  
fruits = ["apple", "banana", "cherry", "date", "elderberry"]  
random\_fruit = choose\_random\_item(fruits)  
print(f"The randomly chosen fruit is: {random\_fruit}")  
  
# Example of error handling  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

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

## 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, leaving the original list unchanged.

## 5.2 parameters

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

## 5.3 Description

This function provides a safe way to shuffle a list without altering the original data structure. The process involves two 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 all subsequent operations are performed on a new list, preserving the integrity of the original list.
2. The random.shuffle() method is then called on the copy. This function shuffles the elements of the 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.

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

## 5.4 Usage Notes

* **Non-mutating:** This function is designed to be non-destructive. The original list passed as the items argument remains unchanged after the function call.
* **Dependency:** This function requires the random module to be imported in the script, as it utilizes random.shuffle() internally.
* **Randomness:** The order of the elements in the returned list is pseudo-random and will likely differ on each execution unless the random seed is explicitly set beforehand.

**Output Example**: A new list with the same elements as the input but in a different, random order. For example, [4, 1, 5, 3, 2] could be a possible output for an input of [1, 2, 3, 4, 5].

## 5.5 Example

import random  
from typing import List  
  
# The function definition is assumed to be present  
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 (unchanged): {original\_numbers}")  
print(f"Shuffled Copy: {shuffled\_numbers}")  
  
# Another run might produce a different order  
another\_shuffled\_copy = shuffle\_copy(original\_numbers)  
print(f"Another Shuffled Copy: {another\_shuffled\_copy}")

**Output:**

Original List (unchanged): [1, 2, 3, 4, 5]  
Shuffled Copy: [3, 5, 1, 2, 4]  
Another Shuffled Copy: [5, 2, 4, 1, 3]