Table of Contents

## 0.1 FunctionDef num(a, b)

# 1 Function: def num(a, b)

## 1.1 Overview

The num function returns the result of applying the addition operator (+) to two input arguments.

## 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, which must be type-compatible with a for the + operator. |

## 1.3 Description

This function provides a straightforward implementation of Python’s addition operator (+). It takes two parameters, a and b, and directly returns the value of a + b.

The behavior of the function is dependent on the data types of the input arguments: - If both a and b are numeric (e.g., int or float), the function performs arithmetic addition and returns their sum. - If both a and b are strings, the function performs string concatenation, joining b to the end of a.

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

Attempting to use this function with incompatible types (e.g., adding a string to an integer) will result in a TypeError.

## 1.4 Usage Notes

* Ensure that both parameters are of compatible types for the + operation. For example, you can add an int to a float, but you cannot add an int to a str.
* The function can be used for both mathematical summation and string joining, depending on the input types.

**Output Example**: A numeric return value from an addition operation.

15

## 1.5 Example

# Example 1: Adding two integers  
sum\_result = num(7, 8)  
print(f"The sum is: {sum\_result}")  
  
# Example 2: Concatenating two strings  
concat\_result = num("Hello, ", "world!")  
print(f"The concatenated string is: {concat\_result}")

**Output:**

The sum is: 15  
The concatenated string is: Hello, world!

## 1.6 FunctionDef generate\_random\_integers(count)

# 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 defined 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. The logic proceeds as follows:

1. **Input Validation**: The function first validates the count parameter. If count is a negative number, it raises a ValueError because it’s impossible to generate a negative number of items.
2. **Range Correction**: It then checks if the start value is greater than the end value. If this condition is true, it swaps the two values. This ensures that the range is always valid (start <= end), making the function more robust and user-friendly.
3. **Random Number Generation**: Finally, the function uses a list comprehension to build the list of integers. It iterates count times, and in each iteration, it calls random.randint(start, end). This random.randint method generates a single integer uniformly sampled from the inclusive range [start, end]. The resulting integers are collected into a list which is then returned.

# Internal logic for generation  
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.
* The range defined by start and end is inclusive, meaning both start and end can appear in the output list.
* If start is provided as a larger number than end, the function will automatically swap them to form a valid range.
* If start and end are not provided, the function defaults to generating integers between 0 and 100, inclusive.

**Output Example**: A possible return value for generate\_random\_integers(5) might look like this:

[42, 8, 99, 23, 75]

## 2.5 Example

import random  
from typing import List  
  
# Definition of the function  
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.  
random\_list = generate\_random\_integers(5, 10, 20)  
print(random\_list)

**Output:**

[15, 11, 20, 18, 12]

## 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 corresponding number.

## 3.3 Description

This function calculates a Fibonacci number based on its index n. The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones (0, 1, 1, 2, 3, 5, …).

The function begins by validating the input n. If n is a negative number, it raises a ValueError, as 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, F₀ and F₁.

The core logic resides in 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 simultaneously sets a to the current value of b and b to the sum of the previous a and b. This effectively advances one step in the Fibonacci sequence.

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

# Inside the loop for n=5:  
# Initial: a = 0, b = 1  
# 1. a = 1, b = 1 (0 + 1)  
# 2. a = 1, b = 2 (1 + 1)  
# 3. a = 2, b = 3 (1 + 2)  
# 4. a = 3, b = 5 (2 + 3)  
# 5. a = 5, b = 8 (3 + 5)  
# Loop ends. Returns a, which is 5.

## 3.4 Usage Notes

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

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

## 3.5 Example

# Example usage  
# Find the 9th Fibonacci number (0-indexed)  
result = fibonacci(9)  
print(result)  
  
# Example with a base case  
result\_zero = fibonacci(0)  
print(result\_zero)

**Output:**

34  
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 random item from a given list of strings.

## 4.2 parameters

* items: List[str] - A list of strings from which a single item will be chosen. This list must not be empty.

## 4.3 Description

This function provides a safe way to select a random element from a list. It begins by performing a crucial validation check on the input items list.

Using the condition if not items:, the function determines if the provided list is empty. If it is, the function raises a ValueError with the message “items must not be empty”. This preventative measure ensures that the program does not attempt to perform an impossible operation and clearly informs the developer of the incorrect input.

If the list is not empty, the function proceeds to use the random.choice(items) method. This method, part of Python’s standard random library, takes the sequence items and returns one element chosen uniformly at random. “Uniformly” means that every item in the list has an equal probability of being selected. The chosen string is then returned as the function’s output.

## 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 into the script.
* The selection is uniformly random, meaning each item in the input list has an equal chance of being chosen on any given call.

**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.  
  
 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"]  
try:  
 random\_fruit = choose\_random\_item(fruits)  
 print(f"A random fruit from the list is: {random\_fruit}")  
  
 # Example with an empty list  
 empty\_list = []  
 print("\nAttempting to choose from an empty list...")  
 choose\_random\_item(empty\_list)  
except ValueError as e:  
 print(f"Caught an error: {e}")

**Output:**

A random fruit from the list is: cherry  
Attempting to choose from an empty list...  
Caught an error: items must not be empty

*(Note: The specific fruit chosen in the first part 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 randomly shuffled copy of a list of integers, leaving the original list unchanged.

## 5.2 parameters

* items: (List[int]) The 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 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 new list from the original, preventing any side effects or mutation of the input data.
2. The random.shuffle() function is then called on this copy. random.shuffle() modifies a sequence in-place by rearranging its items into a random order.
3. Finally, the function returns the shuffled copy. The original items list remains in its initial order.

# The function creates a copy  
copy = list(items)  
# It then shuffles the copy in-place  
random.shuffle(copy)  
# It returns the modified copy, not the original list  
return copy

## 5.4 Usage Notes

* This function is non-destructive. The original list passed as the items parameter will not be modified.
* The function depends on Python’s built-in random module. Ensure that import random is present in your script before calling this function.
* While the type hint specifies List[int], the underlying logic will work correctly with lists containing elements of any type.

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

## 5.5 Example

import random  
from typing import List  
  
# The function must be defined or imported to be used  
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]  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Original List (unchanged): {original\_list}")  
print(f"Shuffled Copy: {shuffled\_list}")

**Output:**

Original List (unchanged): [10, 20, 30, 40, 50]  
Shuffled Copy: [40, 10, 50, 20, 30]

*(Note: The order of elements in “Shuffled Copy” is random and will likely differ with each execution.)*