Table of Contents

## 0.1 FunctionDef num(a, b)

# 1 Function: def num(a, b)

## 1.1 Overview

The num function calculates the sum of two arguments or concatenates them.

## 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 of a type compatible with a. |

## 1.3 Description

The num function provides a straightforward implementation of the addition operator (+). It takes two parameters, a and b, and returns the result of a + b.

The behavior of the function is dependent on the data types of the input arguments: - If a and b are numeric types (such as int or float), the function performs arithmetic addition and returns their sum. - If a and b are sequence types (such as str or list), the function performs concatenation, joining the two sequences together.

The core logic is simply to return the output of the expression:

return a + b

## 1.4 Usage Notes

* Ensure that the data types of a and b are compatible for the + operator. For instance, attempting to add an int to a str will result in a TypeError.
* The function can be used with any Python objects that have defined the \_\_add\_\_ method.

**Output Example**: A numeric sum or a concatenated sequence. For num(5, 10), the output would be: 15

## 1.5 Example

# Example 1: Adding two integers  
sum\_result = num(5, 10)  
print(f"The sum is: {sum\_result}")  
  
# Example 2: Concatenating two strings  
string\_result = num("Hello, ", "World!")  
print(f"The concatenated string is: '{string\_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 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 convenient way to generate multiple 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 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 robust against incorrect range ordering.
3. **Generation**: Using a list comprehension, the function iterates count times. In each iteration, it calls random.randint(start, end) to produce a single pseudo-random integer that is uniformly distributed within the inclusive range [start, end]. These integers are collected into a list.
4. **Return Value**: The function returns the newly created list containing count random integers.

# The 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 in your script.
* 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, meaning both start and end can appear in the output list.
* If start is provided with a value greater than end, the function will automatically swap them to form a valid range.

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

[45, 8, 99, 23, 67]

## 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.  
result = generate\_random\_integers(5, 10, 20)  
print(result)

**Output:**

(Note: The output is random and will vary with each execution)

[15, 11, 20, 18, 12]

## 2.6 FunctionDef fibonacci(n)

# 3 Function: fibonacci(n: int) -> int

## 3.1 Overview

The fibonacci function computes the nth number in the Fibonacci sequence using an efficient iterative approach.

## 3.2 parameters

* n (int): 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 logic proceeds as follows:

1. **Input Validation**: The function first checks if the input n is a negative number. Since the Fibonacci sequence is not defined for negative indices, it raises a ValueError if this 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. If n is 0, the loop is skipped entirely.
4. **Calculation**: Inside the loop, the core logic a, b = b, a + b is executed. This is a tuple assignment that simultaneously updates the values:
   * a is updated to the current value of b.
   * b is updated to the sum of the old values of a and b. This process effectively walks through the sequence, with a and b always holding two consecutive Fibonacci numbers.
5. **Return Value**: After the loop completes, a holds the nth Fibonacci number. For n=0, the loop doesn’t run and the initial value of a (0) is returned. For n > 0, the loop runs n times, and the final value of a is the correct result.

# For n = 5:  
# Initial: a = 0, b = 1  
# Loop 1: a = 1, b = 1 (0 + 1)  
# Loop 2: a = 1, b = 2 (1 + 1)  
# Loop 3: a = 2, b = 3 (1 + 2)  
# Loop 4: a = 3, b = 5 (2 + 3)  
# Loop 5: a = 5, b = 8 (3 + 5)  
# Loop finishes, returns a  
return 5

## 3.4 Usage Notes

* The function only accepts non-negative integers. Providing a negative number will result in a ValueError.
* The sequence is 0-indexed, meaning fibonacci(0) returns the first number of the sequence, which is 0.
* This iterative implementation is highly efficient and is preferred over simple recursion for larger values of n as it avoids excessive function calls and potential stack overflow errors.

**Output Example**: The function returns a single integer value. 55

## 3.5 Example

# Example usage  
# Find the 10th Fibonacci number (0-indexed)  
index = 10  
result = fibonacci(index)  
print(f"The Fibonacci number at index {index} is: {result}")  
  
# Example with an edge case  
index\_zero = 0  
result\_zero = fibonacci(index\_zero)  
print(f"The Fibonacci number at index {index\_zero} is: {result\_zero}")  
  
# Example that would raise an error  
try:  
 fibonacci(-1)  
except ValueError as e:  
 print(f"Error caught as expected: {e}")

**Output:**

The Fibonacci number at index 10 is: 55  
The Fibonacci number at index 0 is: 0  
Error caught as expected: n must be non-negative

## 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 safe way to select a random element from a list. The core logic is implemented in two main steps:

1. **Validation**: The function first checks if the provided items list is empty using the if not items: condition. If the list is empty, it raises a ValueError with the message “items must not be empty”. This prevents runtime errors that would occur if random.choice were called on an empty sequence.
2. **Random Selection**: 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, selects a single item from the items list. Each item in the list has an equal probability of being chosen. The selected string is then returned as the result.

# Internal logic for selecting an item  
return random.choice(items)

## 4.4 Usage Notes

* The input list items must not be empty. Providing an empty list will result in a ValueError.
* This function relies on Python’s random module. Ensure it is imported in the environment where the function is called.
* The selection is uniformly random, meaning every item in the list has an equal chance of being returned.

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

"banana"

## 4.5 Example

import random  
from typing import List  
  
# Definition of the function  
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  
options = ["red", "green", "blue", "yellow"]  
chosen\_color = choose\_random\_item(options)  
print(f"The chosen color is: {chosen\_color}")  
  
# Example of what happens with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

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

*(Note: The actual color output will vary as it is chosen randomly from the list.)*

## 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 to be shuffled.

## 5.3 Description

This function provides a safe way to shuffle a list by operating on a copy rather than the original data. The process is as follows:

1. A shallow copy of the input items list is created using copy = list(items). This is a critical step to ensure that the original list passed to the function is not mutated.
2. The random.shuffle() function is then called on this newly created copy. The random.shuffle() method shuffles the sequence (the list copy) in place.
3. Finally, the function returns the copy, which now holds the same elements as the original items list but arranged in a random order.

# Internal logic of the function  
import random  
  
# Assume 'items' is [1, 2, 3, 4, 5]  
copy = list(items) # copy is now a new list: [1, 2, 3, 4, 5]  
random.shuffle(copy) # copy is shuffled in-place, e.g., it might become [3, 1, 5, 2, 4]  
return copy # The shuffled list [3, 1, 5, 2, 4] is returned

## 5.4 Usage Notes

* This function is non-mutating. It will always return a new list and leave the original input list unchanged.
* The function relies on Python’s random module. Ensure this module is imported (import random) in your script before calling this function.
* Although the type hint specifies List[int], the function will work correctly with a list containing elements of any type (e.g., strings, floats, or mixed types).

**Output Example**: A new list with the same elements as the input, but in a random order.

[4, 1, 5, 2, 3]

## 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\_list = [1, 2, 3, 4, 5]  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Original List: {original\_list}")  
print(f"Shuffled Copy: {shuffled\_list}")

**Output:**

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

*(Note: The order of elements in the “Shuffled Copy” will vary with each execution due to its random nature.)*