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, list | The first operand for the addition or concatenation operation. |
| b | int, float, str, list | The second operand, which must be of a type compatible with a. |

## 1.3 Description

This function takes two arguments, a and b, and computes their sum using the + operator. The behavior of the function is dependent on the data types of the input parameters.

If a and b are numeric types (such as int or float), the function performs arithmetic addition and returns their numerical sum.

# Numeric addition  
result = num(10, 20)  
# result is 30

If a and b are sequence types (such as str or list), the function performs concatenation and returns a new, combined sequence.

# String concatenation  
result = num("hello", " world")  
# result is "hello world"

The function does not include any internal type checking. Therefore, providing incompatible types (e.g., an int and a str) will result in a TypeError.

## 1.4 Usage Notes

* Ensure that both a and b are of compatible types that support the + operator.
* The function will raise a TypeError if the operands are of unsupported types for addition, such as attempting to add a number to a string.
* The order of parameters matters for non-commutative operations like string concatenation. For example, num("a", "b") returns "ab", while num("b", "a") returns "ba".

**Output Example**: For numeric inputs 5 and 3, the output would be:

8

## 1.5 Example

# Example usage with integers  
result\_int = num(5, 10)  
print(result\_int)  
  
# Example usage with strings  
result\_str = num("doc", "umentation")  
print(result\_str)

**Output:**

15  
documentation

## 1.6 FunctionDef generate\_random\_integers

# 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. |
| start | int | The inclusive lower bound of the random number range. Defaults to 0. |
| end | int | The inclusive upper bound of the random number range. Defaults to 100. |

## 2.3 Description

This function provides a straightforward way to create a list of pseudo-random integers. It operates in a few distinct steps:

First, it validates the count parameter. If count is a negative number, the function cannot produce a list of that length and will raise a ValueError.

Next, it ensures the integrity of the numerical range. It checks if the start value is greater than the end value. If it is, the function automatically swaps them to ensure that start is always the lower bound and end is the upper bound. This makes the function more robust and user-friendly.

Finally, the function uses a list comprehension to generate the random numbers. It iterates count times, and in each iteration, it calls random.randint(start, end). This standard library function returns a single random integer that is greater than or equal to start and less than or equal to end. The resulting integers are collected into a list, which is then returned.

# Internal logic for swapping bounds  
if start > end:  
 start, end = end, start  
# 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. Providing a negative value will result in a ValueError.
* The function is inclusive of both the start and end values.
* If the provided start value is greater than the end value, the function will automatically swap them to form a valid range.

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

[42, 8, 99, 54, 23]

## 2.5 Example

import random  
from typing import List  
  
# Definition of the function from the documentation  
def generate\_random\_integers(count: int, start: int = 0, end: int = 100) -> List[int]:  
 """Return a list of pseudo-random integers."""  
 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 1 and 50.  
random\_numbers = generate\_random\_integers(5, 1, 50)  
print(random\_numbers)

**Output:**

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

[34, 12, 48, 2, 25]

## 2.6 FunctionDef fibonacci

# 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 value. Must be a non-negative integer.

## 3.3 Description

This function calculates a Fibonacci number based on its index n. The logic begins by handling invalid input; if n is a negative number, a ValueError is raised.

The function initializes two variables, a and b, to 0 and 1 respectively. These represent the first two numbers in the Fibonacci sequence (F₀ and F₁).

It 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, which is then returned. For n=0, the loop does not run, and the initial value of a (0) is returned correctly.

# For n = 3:  
# Initial state: a = 0, b = 1  
# Iteration 1: a becomes 1, b becomes 0 + 1 = 1  
# Iteration 2: a becomes 1, b becomes 1 + 1 = 2  
# Iteration 3: a becomes 2, b becomes 1 + 2 = 3  
# Loop ends. The function returns a, which is 2.

## 3.4 Usage Notes

* The input n must be a non-negative integer. The function will raise a ValueError for negative inputs.
* The function is 0-indexed, meaning fibonacci(0) returns the first element of the sequence, 0.
* This iterative implementation is memory-efficient compared to a naive recursive approach, as it avoids a deep call stack.

**Output Example**: A single integer representing the Fibonacci number at the specified index.

55

## 3.5 Example

# Example usage  
# Calculate the 10th Fibonacci number (0-indexed)  
result = fibonacci(10)  
print(result)

**Output:**

55

## 3.6 FunctionDef choose\_random\_item

# 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 list of strings from which to choose. This list must not be empty.

## 4.3 Description

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

1. It first performs a validation check on the input list items. The condition if not items: evaluates to True if the list is empty.
2. If the list is found to be empty, the function immediately stops execution and raises a ValueError with the message “items must not be empty”. This prevents potential errors from the underlying random.choice method, which requires a non-empty sequence.
3. If the list is not empty, the function calls 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.
4. The randomly chosen string is then returned as the result of the function.

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

## 4.4 Usage Notes

* The input list items must contain at least one element. Providing an empty list will raise a ValueError.
* This function depends on Python’s built-in random module. Ensure it is imported in your script (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 from the input list. "banana"

## 4.5 Example

import random  
from typing import List  
  
# The function being documented  
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"]  
selected\_color = choose\_random\_item(options)  
print(f"The selected color is: {selected\_color}")  
  
# Example of what happens with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

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

*(Note: The selected color will vary with each execution as it is chosen randomly.)*

## 4.6 FunctionDef shuffle\_copy

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

## 5.1 Overview

The shuffle\_copy function returns a new list containing a shuffled version of the elements from the input list, 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. 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 remains unchanged, adhering to the non-mutating principle of the function.
2. The random.shuffle() method is then called on this copy. This method shuffles the elements of the list in-place, rearranging them into a random order.
3. Finally, the function returns the modified copy, which now contains the same elements as the original items list but in a new, randomized sequence.

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

## 5.4 Usage Notes

* This function is non-mutating. It will always return a new list and will not alter the original list provided as the items parameter.
* The function relies on the random module. Ensure that import random is present in the file where this function is used.
* Although the type hint specifies List[int], the underlying logic will work correctly with lists containing elements of any type (e.g., strings, floats, or mixed types).

**Output Example**: The function returns a list where the elements are the same as the input but in a random order.

## 5.5 Example

import random # This import is required for the function to work  
  
# Define an original list of integers  
original\_numbers = [10, 20, 30, 40, 50]  
  
# Get a shuffled copy of the list  
shuffled\_numbers = shuffle\_copy(original\_numbers)  
  
print(f"Original List: {original\_numbers}")  
print(f"Shuffled Copy: {shuffled\_numbers}")

**Output:**

Original List: [10, 20, 30, 40, 50]  
Shuffled Copy: [30, 50, 10, 20, 40]  
# Note: The actual order of elements in the output will be random with each execution.