Table of Contents

## 0.1 FunctionDef num(a, b)

# 1 Function: def num(a, b)

## 1.1 Overview

The num function returns the sum of two arguments, a and b.

## 1.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| a | int / float / str | The first operand. |
| b | int / float / str | The second operand. |

## 1.3 Description

This function takes two parameters, a and b, and applies the addition operator (+) to them. The core logic is simply to compute and return the result of a + b. The behavior of the function depends on the data types of the input arguments. If both are numbers (integers or floats), it performs mathematical addition. If both are strings, it performs string concatenation.

# The function returns the result of the addition operation  
return a + b

## 1.4 Usage Notes

* When a and b are both numbers (e.g., int or float), the function returns their mathematical sum.
* If a and b are both strings, the function concatenates them and returns the resulting string.
* Passing arguments of incompatible types (e.g., an int and a str) will result in a TypeError.

**Output Example**: A numeric value representing the sum.

## 1.5 Example

# Example usage with integers  
result = num(10, 5)  
print(result)  
  
# Example usage with strings  
string\_result = num("hello", " world")  
print(string\_result)

**Output:**

15  
hello world

## 1.6 FunctionDef generate\_random\_integers

# 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 random 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. It operates in three main steps:

1. **Input Validation**: It first validates the count parameter. If count is a negative number, the function raises a ValueError because it’s impossible to generate a negative quantity of numbers.
2. **Range Correction**: The function checks if the start value is greater than the end value. If this condition is true, it automatically swaps the two values. This ensures that random.randint receives a valid range (start <= end) and makes the function more robust by handling inverted range inputs gracefully.
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, which is the final return value.

# Internal logic for generating the list  
return [random.randint(start, end) for \_ in range(count)]

## 2.4 Usage Notes

* This function depends on Python’s built-in random module. Ensure it is imported before use.
* A ValueError will be raised if the count argument is less than 0.
* 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 and proceed without error.

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

[45, 12, 89, 67, 33]

## 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(count=5, start=10, end=20)  
print(result)

**Output:**

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

[15, 11, 20, 18, 12]

## 2.6 FunctionDef fibonacci

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

## 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, typically starting with 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 (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 to the next pair in the sequence using tuple assignment: a, b = b, a + b. This simultaneously sets the new a to the old b’s value, and the new b to 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 an input of n=0, the loop does not run, and the initial value of a (0) is correctly returned.

# Internal logic for n=3  
# Initial state  
a, b = 0, 1  
  
# Loop 1: \_ = 0  
a, b = 1, 0 + 1 # a=1, b=1  
  
# Loop 2: \_ = 1  
a, b = 1, 1 + 1 # a=1, b=2  
  
# Loop 3: \_ = 2  
a, b = 2, 1 + 2 # a=2, b=3  
  
# Loop finishes, return a  
return 2 # F₃ = 2

## 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 function is 0-indexed. For example, fibonacci(0) returns the first number in the sequence, 0.
* This iterative implementation is efficient in terms of memory usage, avoiding the deep recursion stack that can occur with a naive recursive solution, making it suitable for calculating larger Fibonacci numbers.

**Output Example**: An integer representing the calculated Fibonacci number.

## 3.5 Example

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

**Output:**

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

## 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 element from a given list of strings.

## 4.2 parameters

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

## 4.3 Description

This function provides a simple way to get a random item from a list. The core logic is built around Python’s built-in random module.

First, the function performs a crucial validation check using if not items:. It verifies if the provided items list is empty. If the list is empty, it is impossible to choose an item, so the function raises a ValueError with the message “items must not be empty” to prevent further errors.

If the list is not empty, the function proceeds to call random.choice(items). The random.choice() method is a standard library function that takes a non-empty sequence (like a list) as input and returns a single item chosen uniformly at random from that sequence. The item returned by random.choice() is then returned as the final output of the choose\_random\_item function.

## 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 script where it is used.
* 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 ['red', 'green', 'blue'], a possible return value is 'green'.

## 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  
options = ["apple", "banana", "cherry", "date"]  
random\_fruit = choose\_random\_item(options)  
print(f"Randomly selected fruit: {random\_fruit}")  
  
# Example of error handling  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

Randomly selected fruit: banana  
Error: items must not be empty

(Note: The selected fruit 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, randomly shuffled copy of a given list, ensuring the original input list remains unchanged.

## 5.2 parameters

* **items** List[int]: A list of integers that will be copied and then shuffled.

## 5.3 Description

This function provides a safe way to shuffle a list without altering the original data structure, a concept known as non-mutation. The process is straightforward:

1. A shallow copy of the input items list is created using list(items). This new list is stored in a local variable named copy.
2. The random.shuffle() method is then called on the 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 holds the same elements as the original list but in a new, randomized sequence.

By operating on a copy, the function guarantees that the list provided by the caller is not mutated.

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

## 5.4 Usage Notes

* This function is non-mutating, meaning the original list passed as the items parameter will not be modified.
* The function depends on Python’s built-in random module. You must ensure import random is present in your script for random.shuffle() to be accessible.
* The returned list will always contain the exact same elements and have the same length as the input list; only their order is changed.

**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], a possible output could be [2, 3, 1].

## 5.5 Example

import random  
from typing import List  
  
# Definition of the function  
def shuffle\_copy(items: List[int]) -> List[int]:  
 copy = list(items)  
 random.shuffle(copy)  
 return copy  
  
# Example usage  
original\_numbers = [10, 20, 30, 40, 50]  
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 shuffled copy will vary with each execution.