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 input numbers.

## 1.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| a | int / float | The first number to be added. |
| b | int / float | The second number to be added. |

## 1.3 Description

The num function provides a straightforward implementation of addition. It accepts two arguments, a and b, which are expected to be numeric. The function uses the standard Python addition operator (+) to compute the sum of these two arguments. The resulting value is then immediately returned to the caller.

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

## 1.4 Usage Notes

* This function is primarily intended for use with numeric types such as integers (int) and floating-point numbers (float).
* If strings are passed as arguments, the function will perform string concatenation instead of mathematical addition. For example, num("hello", " world") would return "hello world".
* Providing incompatible types (e.g., an int and a str) will result in a TypeError.

**Output Example**: A numeric value representing the sum. For num(5, 10), the output is 15.

## 1.5 Example

# Example usage with two integers  
number1 = 10  
number2 = 25  
result = num(number1, number2)  
print(f"The result is: {result}")  
  
# Example usage with a float and an integer  
number3 = 15.5  
number4 = 5  
result\_float = num(number3, number4)  
print(f"The result is: {result\_float}")

**Output:**

The result is: 35  
The result is: 20.5

## 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 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 number of integers to generate. This value must be non-negative. |
| 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 robust way to generate a list of random integers. Its logic proceeds in three main steps:

1. **Input Validation**: The function first validates the count parameter. If count is a negative number, it is impossible to generate a list of that size, so the function raises a ValueError with the message “count must be non-negative”.
2. **Range Correction**: It then checks if the provided start value is greater than the end value. If this is the case, the function automatically swaps the two values. This ensures that random.randint receives a valid range (start <= end), making the function more resilient to user input errors.
3. **Random Number Generation**: The core of the function is a list comprehension: [random.randint(start, end) for \_ in range(count)].
   * It iterates count times.
   * In each iteration, it calls random.randint(start, end), which generates a single integer uniformly sampled from the inclusive range [start, end].
   * All generated integers are collected into a list, which is then returned.

This function depends on Python’s built-in random module.

# Internal logic for swapping start and end  
if start > end:  
 start, end = end, start

## 2.4 Usage Notes

* The function will raise a ValueError if the count argument is a negative integer.
* If start is greater than end, the function will automatically swap them to create a valid range; no error will be raised.
* The range defined by start and end is inclusive, meaning both start and end can appear in the output list.
* This function requires the random module to be imported in the script.

**Output Example**: A possible return value for generate\_random\_integers(5, 1, 100):

[42, 8, 99, 15, 73]

## 2.5 Example

# This example requires the 'random' module to be imported.  
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."""  
 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 7 random integers between 10 and 20.  
random\_numbers = generate\_random\_integers(7, 10, 20)  
print(random\_numbers)  
  
# Example with swapped start and end values (will work correctly)  
random\_numbers\_swapped = generate\_random\_integers(5, 50, 40)  
print(random\_numbers\_swapped)

**Output:**

# The actual output will vary with each execution due to randomness.  
[15, 11, 20, 18, 10, 13, 19]  
[42, 48, 45, 50, 41]

## 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 corresponding number.

## 3.3 Description

This function provides a memory-efficient, iterative method to calculate a number in the Fibonacci sequence. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.

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 function then enters a for loop that iterates n times. In each iteration, the values of a and b are updated using tuple unpacking: a takes the value of b, and b takes the value of the sum of the old a and b. This process effectively walks through the sequence: - Before loop: a=0, b=1 - After 1st iteration: a=1, b=1 - After 2nd iteration: a=1, b=2 - After 3rd iteration: a=2, b=3

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

# Logic for n = 3  
a, b = 0, 1  
# Iteration 1:  
a, b = 1, 0 + 1 # a=1, b=1  
# Iteration 2:  
a, b = 1, 1 + 1 # a=1, b=2  
# Iteration 3:  
a, b = 2, 1 + 2 # a=2, b=3  
# Loop ends, return a  
return 2

## 3.4 Usage Notes

* The index n is 0-based. For example, fibonacci(0) returns 0 and fibonacci(1) returns 1.
* The function will raise a ValueError if a negative integer is provided as an argument.
* This iterative implementation is preferred over simple recursion for larger values of n as it avoids deep recursion stacks and redundant calculations, resulting in better performance and memory usage.

**Output Example**:

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 chooses a single random item from a non-empty list of strings.

## 4.2 parameters

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

## 4.3 Description

This function provides a straightforward way to select one item at random from a given list. The function begins by checking if the input list items is empty. If it is, the function raises a ValueError with the message “items must not be empty” to prevent further execution with invalid input.

If the list contains one or more items, the function utilizes the random.choice() method. This method, part of Python’s standard random module, performs a uniform random selection from the sequence. This means every item in the items list has an equal probability of being chosen. The function then returns the single string that was selected.

# Internal logic for a non-empty list  
return random.choice(items)

## 4.4 Usage Notes

* The input list items must not be empty. Providing an empty list will raise a ValueError.
* This function depends on Python’s random module, which must be imported into the environment where the function is used.
* The selection is uniformly random, ensuring each item in the list has an equal chance of being returned.

**Output Example**: The function returns 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 # This import is necessary for the function to work  
  
# Example 1: Choosing from a list of options  
options = ["red", "green", "blue", "yellow"]  
selected\_color = choose\_random\_item(options)  
print(f"The selected color is: {selected\_color}")  
  
# Example 2: Demonstrating the error with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error when using an empty list: {e}")

**Output:**

# The output for the first example will vary with each execution.  
# A possible output is:  
The selected color is: blue  
  
# The output for the second example is always the same:  
Error when using an empty list: items must not be empty

## 4.6 FunctionDef shuffle\_copy

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

## 5.1 Overview

The shuffle\_copy function returns a shuffled copy of a given list without modifying the original list.

## 5.2 parameters

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

## 5.3 Description

This function provides a safe way to shuffle a list by ensuring the original data structure is not altered. The process is executed in three main steps:

1. A shallow copy of the input items list is created using the list() constructor. This new list is assigned to a local variable copy. This is the key step that prevents mutation of the original list.
2. The random.shuffle() function is then called on the copy. This function shuffles the elements of the copy 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 items list but in a new, randomized sequence.

# 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 remain unchanged after the function call.
* The function depends on Python’s standard random module. Ensure you have import random at the beginning of your script to use it.
* The randomness is based on the default random number generator in the random module.

**Output Example**: The function returns a new list. 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  
  
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]  
print(f"Original list before shuffling: {original\_list}")  
  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Original list after shuffling: {original\_list}")  
print(f"Newly created shuffled list: {shuffled\_list}")

**Output:**

Original list before shuffling: [10, 20, 30, 40, 50]  
Original list after shuffling: [10, 20, 30, 40, 50]  
Newly created shuffled list: [40, 10, 50, 20, 30]

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