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 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

This function provides a straightforward way to perform addition. It accepts two arguments, a and b, and executes the addition operation a + b. The resulting sum is then returned by the function. The function is designed to work with numeric types, including both integers and floating-point numbers.

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

## 1.4 Usage Notes

* Ensure that both parameters are numeric types (e.g., int, float) to perform a mathematical sum.
* If string types are provided for both a and b, the function will perform string concatenation instead of numerical addition.
* Providing incompatible types (e.g., an int and a str) will raise a TypeError.

**Output Example**: A single numeric value representing the sum. For instance, if a is 10 and b is 5, the output will be 15.

## 1.5 Example

# Example usage with two integers  
result = num(10, 5)  
print(result)  
  
# Example usage with two floats  
result\_float = num(7.5, 2.5)  
print(result\_float)

**Output:**

15  
10.0

## 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. This value must be non-negative. |
| start | int | The inclusive lower bound for the generated values. Defaults to 0. |
| end | int | The inclusive upper bound for the generated values. Defaults to 100. |

## 2.3 Description

This function provides a convenient way to create a list of pseudo-random integers. It operates in three main steps:

1. **Input Validation**: The function first checks if the count parameter is less than zero. If it is, a ValueError is raised, as it’s not possible to generate a negative number of integers.
2. **Range Correction**: It then compares the start and end parameters. If start is found to be greater than end, the function automatically swaps their values. This ensures that random.randint receives a valid range, making the function more robust and user-friendly.
3. **Integer Generation**: 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 call produces a single integer uniformly sampled from the inclusive range [start, end]. The resulting integers are collected into a list which is then returned.

# The core generation logic  
[random.randint(start, end) for \_ in range(count)]

## 2.4 Usage Notes

* The function requires the random module to be imported to work correctly.
* A ValueError will be raised if the count argument is a negative number.
* The range defined by start and end is inclusive, meaning both boundary values can be included in the results.
* If the provided start value is greater than the end value, the function will automatically swap them to form a valid range.

**Output Example**: The function returns a list of integers. For a call with count=5, start=1, and end=10, a possible output could be: [3, 10, 1, 7, 5]

## 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.  
random\_numbers = generate\_random\_integers(count=5, start=10, end=20)  
print(random\_numbers)  
  
# Example with swapped start and end values  
# The function will automatically handle this case.  
random\_numbers\_swapped = generate\_random\_integers(count=3, start=50, end=40)  
print(random\_numbers\_swapped)

**Output:**

[15, 11, 20, 18, 12]  
[43, 48, 41]

*(Note: The actual output will vary on each execution due to the random nature of the function.)*

## 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. It employs an iterative method for efficiency and to avoid recursion depth limits.

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, F(0) and F(1).

The core logic resides in a for loop that iterates n times. In each iteration, the values of a and b are updated simultaneously. The current value of b is assigned to a, and the sum of the old a and b is assigned to b. This process effectively steps through the sequence:

# Initial state: a=0, b=1  
# After 1st iteration: a=1, b=1 (0+1)  
# After 2nd iteration: a=1, b=2 (1+1)  
# After 3rd iteration: a=2, b=3 (1+2)  
# ...and so on.

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.

## 3.4 Usage Notes

* The index n is 0-indexed. For example, fibonacci(0) returns 0, and fibonacci(1) returns 1.
* The function will raise a ValueError if a negative integer is provided for n.
* This iterative implementation is memory-efficient and is preferred over simple recursive solutions for large values of n to prevent stack overflow errors.

**Output Example**: The function returns a single integer representing the Fibonacci number at the specified index.

## 3.5 Example

# Example usage  
# Calculate the 9th Fibonacci number (0-indexed)  
result = fibonacci(9)  
print(result)  
  
# Example of edge case  
try:  
 result\_zero = fibonacci(0)  
 print(f"fibonacci(0) = {result\_zero}")  
   
 # This will raise an error  
 fibonacci(-1)  
except ValueError as e:  
 print(e)

**Output:**

34  
fibonacci(0) = 0  
n must be non-negative

## 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 random string from a given list of strings.

## 4.2 parameters

* items: List[str]
  + A non-empty list of strings from which a single item will be chosen randomly.

## 4.3 Description

This function provides a simple way to get a random element from a sequence of strings. The core logic is built around Python’s random.choice() method.

Before attempting to select an item, the function first performs a validation check to ensure the input list items is not empty. If the list is empty (if not items), the function raises a ValueError with the message “items must not be empty” to prevent runtime errors from random.choice(), which cannot operate on an empty sequence.

If the list contains one or more items, the function proceeds to call random.choice(items). This method randomly selects a single element from the items list with a uniform probability distribution, meaning every item has an equal chance of being chosen. The selected string is then returned as the result.

# Internal logic for a non-empty list  
import random  
my\_list = ["apple", "banana", "cherry"]  
# The function effectively does this:  
selected = random.choice(my\_list)  
# selected could be "apple", "banana", or "cherry"

## 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 built-in random module. Ensure it is imported in the environment where the function is defined.
* The selection is uniformly random, meaning each item in the list has an equal probability of being chosen on any given call.

**Output Example**: The function returns a single string from the input list.

"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"]  
random\_fruit = choose\_random\_item(fruits)  
print(f"The randomly chosen fruit is: {random\_fruit}")  
  
# Example of error handling  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

The randomly chosen fruit is: cherry  
Error: items must not be empty

(Note: The chosen fruit in the first line of the output will vary with each execution.)

## 4.6 FunctionDef shuffle\_copy

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

## 5.1 Overview

The shuffle\_copy function creates and returns a randomly shuffled copy of a list, ensuring the original list remains unchanged.

## 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 without modifying the original data structure (a concept known as immutability). The process is straightforward:

1. The function first creates a shallow copy of the input items list by calling list(items). This new list is stored in a variable named copy.
2. It then uses the random.shuffle() method, which shuffles a sequence in-place. By applying this method to the copy, only the new list is reordered.
3. Finally, the function returns the shuffled copy, leaving the original items list in its initial state.

This approach is different from directly using random.shuffle(my\_list), which would modify my\_list directly and return None.

## 5.4 Usage Notes

* The primary benefit of this function is that it is non-mutating. The original list passed as the items parameter will not be altered.
* This function depends on Python’s built-in random module. Ensure that random is imported in your script before calling this function.
* The function creates a shallow copy. If the list contains mutable objects (like other lists or dictionaries), the references to those objects are shuffled, but the objects themselves are not duplicated.

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

[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.  
  
 Parameters:  
 items: A list of integers.  
  
 Returns:  
 A new list containing the same integers in random order.  
 """  
 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"Shuffled copy: {shuffled\_list}")  
print(f"Original list after shuffling: {original\_list}")

**Output:**

Original list before shuffling: [10, 20, 30, 40, 50]  
Shuffled copy: [40, 10, 50, 20, 30]  
Original list after shuffling: [10, 20, 30, 40, 50]