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

## 1.3 Description

This function takes two parameters, a and b, and returns the result of the expression a + b. The behavior of the function is dependent on the data types of the provided arguments.

If the arguments a and b are numeric (such as int or float), the function will perform arithmetic addition and return their sum.

# When a=5 and b=10, the function returns 15  
num(5, 10)

If the arguments are of a sequence type like a string (str), the function will perform concatenation.

# When a="hello" and b=" world", the function returns "hello world"  
num("hello", " world")

## 1.4 Usage Notes

* This function will raise a TypeError if the operands a and b are of incompatible types for the + operator (e.g., attempting to add an int to a str).
* The caller is responsible for ensuring that the arguments are of compatible types, as the function does not perform any internal type checking or conversion.

**Output Example**: A numeric return value from adding two integers.

15

## 1.5 Example

# Example 1: Adding two integers  
result\_sum = num(5, 10)  
print(f"The sum is: {result\_sum}")  
  
# Example 2: Concatenating two strings  
result\_concat = num("doc", "umentation")  
print(f"The concatenated string is: {result\_concat}")

**Output:**

The sum is: 15  
The concatenated string is: 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 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 random integers to generate. This value must be non-negative. |
| start | int | The inclusive lower bound for the random integer values. Defaults to 0. |
| end | int | The inclusive upper bound for the random integer values. Defaults to 100. |

## 2.3 Description

This function provides a robust way to generate a list of random integers. The process begins with input validation.

First, it checks if the count parameter is a negative number. If count is less than zero, the function raises a ValueError to prevent invalid list creation.

Next, it ensures the range is valid by comparing start and end. If start is found to be greater than end, the function automatically swaps their values. This convenience feature allows the user to provide the bounds in any order without causing an error.

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) to produce a single integer. The random.randint method samples uniformly from the inclusive range [start, end]. The resulting integers are collected into a list and returned.

# Internal logic for swapping bounds  
if start > end:  
 start, end = end, start  
# Internal logic for list 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.
* A ValueError will be raised if the count argument is a negative integer.
* The function gracefully handles cases where start > end by swapping the values, so the order of range bounds does not matter.
* The generated integers are sampled uniformly from the range, inclusive of both start and end.

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

[42, 18, 99, 5, 73]

## 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 7 random integers between 50 and 60.  
random\_numbers = generate\_random\_integers(count=7, start=50, end=60)  
print(random\_numbers)

**Output:**

[58, 51, 55, 52, 60, 55, 59]

## 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.

## 3.3 Description

This function calculates a Fibonacci number based on its index n. The logic begins by validating the input. If n is a negative number, a ValueError is raised, as the Fibonacci sequence is not defined for negative indices.

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

It then enters 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 previous a and b is assigned to the new b. This process effectively walks through the sequence:

# Initial state  
a, b = 0, 1  
  
# After 1st iteration (for n=1)  
a, b = 1, 1 # (0 + 1)  
  
# After 2nd iteration (for n=2)  
a, b = 1, 2 # (1 + 1)  
  
# After 3rd iteration (for n=3)  
a, b = 2, 3 # (1 + 2)

After the loop completes, the variable a holds the nth Fibonacci number, which is then returned.

## 3.4 Usage Notes

* The input n must be a non-negative integer. Providing a negative value will result in a ValueError.
* The function uses a 0-indexed sequence, meaning fibonacci(0) returns the first number of the sequence, which is 0.
* This iterative implementation is efficient in terms of memory, as it avoids the deep recursion stacks that can occur with naive recursive solutions, making it suitable for calculating larger Fibonacci numbers.

**Output Example**: A single integer representing the calculated Fibonacci number.

13

## 3.5 Example

# Example usage  
# Calculate the 9th Fibonacci number (0-indexed)  
result = fibonacci(9)  
print(result)

**Output:**

34

## 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 non-empty list of strings from which a random item will be chosen.

## 4.3 Description

This function provides a simple way to randomly select one element from a sequence. The core logic is implemented in two steps:

1. **Validation**: The function first checks if the provided items list is empty using if not items:. 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". This prevents runtime errors from the underlying random.choice method, which cannot handle empty sequences.
2. **Selection**: If the list is not empty, the function utilizes random.choice(items). This method, from Python’s built-in random module, takes a sequence as input and returns a single item chosen from it with uniform probability. Every item in the list has an equal chance of being selected.

The selected string is then returned as the result.

# Internal logic example  
import random  
  
def choose\_random\_item(items: list[str]) -> str:  
 if not items:  
 raise ValueError("items must not be empty")  
 # random.choice handles the selection of a single random element  
 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 depends on Python’s random module. Ensure that import random is present at the top of the script where this function is defined or used.
* The selection is uniformly random, meaning each item in the list has an equal probability of being chosen.

**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  
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"]  
selected\_color = choose\_random\_item(options)  
print(f"The randomly 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 randomly selected color is: blue  
Error: items must not be empty

*(Note: The selected color in the first line of the output 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 shuffled copy of a given list without modifying the original list.

## 5.2 parameters

* items: List[int] - A list of integers that you want to shuffle.

## 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 step is crucial as it ensures that the original list passed to the function remains unchanged.
2. The random.shuffle() function is then called on this copy. random.shuffle() 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 list but in a new, random sequence.

# Internal logic breakdown  
import random  
  
def shuffle\_copy(items: List[int]) -> List[int]:  
 # 1. Create a shallow copy of the input list  
 copy = list(items)  
   
 # 2. Shuffle the copy in-place  
 random.shuffle(copy)  
   
 # 3. Return the shuffled copy  
 return copy

## 5.4 Usage Notes

* **Non-mutating**: The primary benefit of this function is that it does not mutate (change) the input list. The original list remains in its initial order after the function call.
* **Shallow Copy**: The function creates a shallow copy. For a list of integers, this is sufficient. If the list contained mutable objects (like other lists or dictionaries), changes to those nested objects would be reflected in both the original and the copied list.
* **Randomness**: The shuffling is pseudo-random and depends on the current state of Python’s random module. The output order will be different each time the function is called, unless the random seed is explicitly set beforehand using random.seed().

**Output Example**: A possible return value for an input of [1, 2, 3, 4, 5].

[4, 1, 5, 2, 3]

## 5.5 Example

import random  
  
# Assume random.shuffle is available from the random module  
# For reproducibility, we can set a seed  
random.seed(42)  
  
original\_list = [10, 20, 30, 40, 50]  
print(f"Original list before shuffling: {original\_list}")  
  
# Get a shuffled copy of the list  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Shuffled list (new): {shuffled\_list}")  
print(f"Original list after shuffling: {original\_list}")

**Output:**

Original list before shuffling: [10, 20, 30, 40, 50]  
Shuffled list (new): [10, 50, 20, 40, 30]  
Original list after shuffling: [10, 20, 30, 40, 50]