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 provided arguments.

## 1.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| a | int, float | The first operand for the addition operation. |
| b | int, float | The second operand for the addition operation. |

## 1.3 Description

This function provides a straightforward way to perform addition. It accepts two parameters, a and b, which are expected to be numerical values (integers or floating-point numbers). The core logic of the function is centered around the + operator. It computes the sum of a and b and immediately returns the resulting value.

The function’s behavior is defined by Python’s + operator. For example, if a is 5 and b is 10, the function will execute 5 + 10 and return 15.

# The core operation of the function  
return a + b

## 1.4 Usage Notes

* This function is primarily intended for use with numeric types such as int and float.
* If strings are passed as arguments, the function will perform string concatenation instead of arithmetic addition (e.g., num("hello", "world") returns "helloworld").
* Passing incompatible types (e.g., an int and a str) will result in a TypeError.

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

15

## 1.5 Example

# Example usage with two integers  
result = num(5, 10)  
print(result)

**Output:**

15

## 1.6 FunctionDef generate\_random\_integers(count, start, end)

# 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 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 straightforward way to generate a list of random integers. It operates in three main steps:

1. **Input Validation**: The function first checks if the count parameter is a non-negative number. If count is less than zero, it raises a ValueError because it’s impossible to generate a negative number of items.
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 the range is always valid for the random number generation logic that follows, preventing potential errors.
3. **Generation**: Using a list comprehension, the function iterates count times. In each iteration, it calls random.randint(start, end) to produce a single integer that is uniformly sampled from the inclusive range [start, end]. These integers are collected into a list which is then returned.

# Internal logic for generating 5 numbers between 1 and 10  
# This assumes the random module is imported  
import random  
start, end = 1, 10  
count = 5  
result = [random.randint(start, end) for \_ in range(count)]  
# result might be [3, 10, 1, 7, 5]

## 2.4 Usage Notes

* This function requires the random module to be imported in the script where it is used.
* A ValueError will be raised if the count argument is a negative integer.
* The range defined by start and end is inclusive, meaning both start and end can appear in the output list.
* The function gracefully handles cases where start > end by swapping the values internally, so the order of these arguments does not strictly matter.

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

[42, 1, 99, 50, 23]

## 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)

**Output:**

[15, 11, 20, 18, 11]

## 2.6 FunctionDef fibonacci(n)

# 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 provides an efficient, iterative method to calculate a specific number in the Fibonacci sequence. The sequence starts with 0 and 1.

The function first validates 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 the first two numbers of the sequence, 0 and 1, respectively.

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 walks through the sequence:

* Iteration 1: a becomes 1, b becomes 1 (0+1)
* Iteration 2: a becomes 1, b becomes 2 (1+1)
* Iteration 3: a becomes 2, b becomes 3 (1+2)
* and so on…

After the loop completes n iterations, the variable a will hold the nth Fibonacci number, which is then returned.

# Initialization for n=3  
a, b = 0, 1  
  
# Loop 1: range(3) -> 0  
a, b = b, a + b # a becomes 1, b becomes 1  
  
# Loop 2: range(3) -> 1  
a, b = b, a + b # a becomes 1, b becomes 2  
  
# Loop 3: range(3) -> 2  
a, b = b, a + b # a becomes 2, b becomes 3  
  
# Loop finishes, return a  
return a # returns 2

## 3.4 Usage Notes

* The function uses a 0-indexed sequence, meaning fibonacci(0) returns the first element, which is 0.
* The input n must be a non-negative integer. The function will raise a ValueError for any negative input.
* This iterative implementation is memory-efficient and avoids the recursion depth limits that can be an issue with naive recursive solutions for large values of n.

**Output Example**: The function returns a single integer value.

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(items)

# 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 provided 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 safe way to select a random item from a list. The logic proceeds as follows:

1. **Input Validation**: The function first checks if the provided items list is empty using the if not items: condition.
2. **Error Handling**: If the list is found to be empty, it immediately raises a ValueError with the message “items must not be empty”. This prevents the program from crashing due to an invalid operation on an empty sequence.
3. **Random Selection**: If the list is not empty, the function utilizes random.choice(items). This standard library method efficiently and uniformly selects a single random element from the input sequence.
4. **Return Value**: The randomly chosen string is then returned as the result of the function call.

# The function assumes the 'random' module is imported.  
import random  
  
# Example of internal logic  
items\_list = ["apple", "banana", "cherry"]  
if not items\_list:  
 raise ValueError("items must not be empty")  
selected\_item = random.choice(items\_list)  
# selected\_item will be one of "apple", "banana", or "cherry"

## 4.4 Usage Notes

* The input list items must not be empty. Passing an empty list will raise a ValueError.
* This function requires the random module to be imported in the execution environment.
* Each item in the list has an equal probability of being selected.

**Output Example**: A single string chosen from the input list. "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 with a valid list  
fruits = ["apple", "banana", "cherry", "date", "elderberry"]  
random\_fruit = choose\_random\_item(fruits)  
print(f"A random fruit was chosen: {random\_fruit}")  
  
# Example of error handling with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error caught: {e}")

**Output:**

A random fruit was chosen: cherry  
Error caught: items must not be empty

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

## 4.6 FunctionDef shuffle\_copy(items)

# 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 list remains unchanged.

## 5.2 parameters

* items: List[int]
  + A list of integers that you want to create a shuffled copy of.

## 5.3 Description

This function provides a safe way to shuffle a list without modifying 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 copy = list(items). This step is crucial as it isolates the original list from any subsequent modifications.
2. The random.shuffle() function is then called on the copy. This function 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 creating a copy first, the function guarantees that the list provided by the caller remains in its original state.

# Demonstration of non-mutation  
import random # shuffle\_copy depends on the random module  
  
original\_list = [10, 20, 30]  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Original list after function call: {original\_list}")  
# The original\_list will still be [10, 20, 30]

## 5.4 Usage Notes

* This function does not modify (mutate) the input list items. It always returns a new list.
* The function relies on Python’s built-in random module. Ensure this module is imported in your script before calling shuffle\_copy.
* The order of elements in the returned list is non-deterministic and will be different on each execution.

**Output Example**: A new list containing the same elements as the input items list, but in a random order.

## 5.5 Example

import random # This module is required for random.shuffle()  
  
# Definition of the function (assuming it's in the same file or imported)  
def shuffle\_copy(items: list) -> list:  
 copy = list(items)  
 random.shuffle(copy)  
 return copy  
  
# Example usage  
my\_numbers = [1, 2, 3, 4, 5, 6, 7]  
shuffled\_numbers = shuffle\_copy(my\_numbers)  
  
print(f"Original List: {my\_numbers}")  
print(f"Shuffled Copy: {shuffled\_numbers}")

**Output:**

Original List: [1, 2, 3, 4, 5, 6, 7]  
Shuffled Copy: [4, 1, 7, 3, 5, 2, 6]

*(Note: The actual order of elements in “Shuffled Copy” will vary with each run.)*