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 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 parameters, a and b, which are expected to be numerical values (integers or floating-point numbers). The core logic of the function uses the + operator to compute the sum of a and b. The resulting value is then returned to the caller.

For example, if a is 10 and b is 5, the function will compute 10 + 5 and return 15.

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

## 1.4 Usage Notes

Important points to consider when using this function:

* The function relies on Python’s + operator. Ensure that both input arguments are of types that support addition (e.g., int, float). Passing incompatible types (like an int and a NoneType) will result in a TypeError.
* If strings are passed as arguments, the function will perform string concatenation instead of mathematical addition. For example, num("hello", "world") would return "helloworld".
* The data type of the returned value depends on the input types. For instance, adding an int and a float will result in a float.

**Output Example**: If the inputs are 15 and 20, the function returns:

35

## 1.5 Example

# Example usage with integers  
result = num(15, 20)  
print(result)  
  
# Example usage with floating-point numbers  
float\_result = num(10.5, 2.2)  
print(float\_result)

**Output:**

35  
12.7

### 1.5.1 FunctionDef num

# 2 Function: def num(a, b)

## 2.1 Overview

The num function is a placeholder that accepts two arguments but performs no operations and returns None.

## 2.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| a | Any | The first input parameter. It is not used within the function’s body. |
| b | Any | The second input parameter. It is not used within the function’s body. |

## 2.3 Description

The num function is defined with two parameters, a and b. The function body is currently empty, meaning it contains no logic or operations. Consequently, the function does not utilize the input parameters for any computation. In Python, a function that does not have an explicit return statement will implicitly return the None object.

## 2.4 Usage Notes

* This function currently acts as a stub or placeholder and lacks any functional implementation.
* Calling this function with any arguments will always result in a return value of None.
* It is likely intended for future development where functionality will be added.

## 2.5 Example

# Example usage  
# The function is called with two integer arguments.  
result = num(10, 20)  
print(result)  
  
# The function can be called with any data types.  
result\_str = num("hello", "world")  
print(result\_str)

**Output:**

None  
None

## 2.6 FunctionDef generate\_random\_integers

# 3 Function: generate\_random\_integers(count: int, start: int = 0, end: int = 100)

## 3.1 Overview

The generate\_random\_integers function generates a list of a specified number of pseudo-random integers within a given inclusive range.

## 3.2 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 of the random number range. Defaults to 0. |
| end | int | The inclusive upper bound of the random number range. Defaults to 100. |

## 3.3 Description

This function provides a convenient way to generate a list of pseudo-random integers. It operates in a few distinct steps:

1. **Input Validation**: The function first checks if the count parameter is a negative number. If it is, a ValueError is raised, as it’s impossible to generate a negative quantity of numbers.
2. **Range Correction**: It then verifies if the start value is greater than the end value. If this condition is true, the function automatically swaps the two values. This ensures that random.randint receives a valid range, making the function more robust and user-friendly.
3. **Integer Generation**: The function uses a list comprehension to efficiently generate the numbers. It iterates count times, and in each iteration, it calls random.randint(start, end). This standard library function returns a single random integer from the uniform distribution over the inclusive range [start, end].
4. **Return Value**: Finally, it returns the complete list containing count randomly generated integers.

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

## 3.4 Usage Notes

* The function requires the random module to be imported.
* A ValueError will be raised if a negative value is passed for the count parameter.
* The range defined by start and end is inclusive, meaning both start and end are potential values in the output list.
* If start is greater than end, the function will automatically swap them and proceed without error.

**Output Example**: A list of integers. The actual numbers will vary with each execution due to their random nature.

[42, 8, 99, 23, 76]

## 3.5 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(5, 10, 20)  
print(result)

**Output:**

[12, 19, 10, 15, 17]

### 3.5.1 FunctionDef generate\_random\_integers

# 4 Function: generate\_random\_integers(count: int, start: int = 0, end: int = 100) -> List[int]

## 4.1 Overview

The generate\_random\_integers function generates a list containing a specified number of random integers within a given inclusive range.

## 4.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| count | int | The total number of random integers to generate. This is a required argument. |
| start | int | The inclusive lower bound of the random number range. Defaults to 0. |
| end | int | The inclusive upper bound of the random number range. Defaults to 100. |

## 4.3 Description

This function produces a list of pseudo-random integers. The quantity of integers in the resulting list is determined by the count parameter.

Each integer in the list is generated within the numerical range defined by the start and end parameters. This range is inclusive, meaning that the values of start and end themselves are potential outcomes.

If the start and end parameters are not explicitly provided, the function uses a default range from 0 to 100. The core logic involves iterating count times, and in each iteration, generating a single random integer that falls within the specified bounds. These integers are then collected into a list which is returned as the final output.

A conceptual implementation would be:

import random  
  
def generate\_random\_integers(count: int, start: int = 0, end: int = 100) -> List[int]:  
 """  
 Generates a list of random integers.  
 """  
 # Using a list comprehension for a concise implementation  
 return [random.randint(start, end) for \_ in range(count)]

## 4.4 Usage Notes

* The value for start must be less than or equal to the value for end. If start is greater than end, the underlying random number generation will raise a ValueError.
* The count parameter should be a non-negative integer. If count is 0, an empty list will be returned.
* The numbers generated are pseudo-random and should not be used for cryptographic purposes. For security-sensitive applications, consider using the secrets module.

## 4.5 Example

# Example 1: Generate 5 random integers using the default range (0 to 100)  
default\_list = generate\_random\_integers(5)  
print(f"Generated list with default range: {default\_list}")  
  
# Example 2: Generate 10 random integers within a custom range of -50 to 50  
custom\_list = generate\_random\_integers(count=10, start=-50, end=50)  
print(f"Generated list with custom range: {custom\_list}")

**Output:**

# The output will vary due to the random nature of the function.  
# A possible output is shown below.  
  
Generated list with default range: [81, 12, 94, 3, 45]  
Generated list with custom range: [-22, 48, 1, -15, 0, 33, -49, 19, 25, -6]

## 4.6 FunctionDef fibonacci

# 5 Function: fibonacci(n: int)

## 5.1 Overview

The fibonacci function computes the nth number in the Fibonacci sequence using an iterative approach.

## 5.2 parameters

* n: int
  + The 0-indexed position in the Fibonacci sequence for which to find the value.

## 5.3 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 (F₀ and F₁).

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 old a and b is assigned to b. This process effectively shifts the sequence forward one step:

# Inside the loop  
a, b = b, a + b

After the loop completes, the variable a holds the nth Fibonacci number. For an input of n=0, the loop does not execute, and the initial value of a (which is 0) is returned, which is correct. For n > 0, the loop runs n times to calculate the correct value.

## 5.4 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. For example, fibonacci(0) returns the first number (0), and fibonacci(1) returns the second number (1).
* This iterative implementation is memory-efficient and avoids the recursion depth issues that can occur with naive recursive solutions for large values of n.

**Output Example**: A possible return value for a valid input.

34

## 5.5 Example

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

**Output:**

34

### 5.5.1 FunctionDef fibonacci

# 6 Function: fibonacci(n: int)

## 6.1 Overview

The fibonacci function calculates the n-th number in the Fibonacci sequence.

## 6.2 parameters

* n (int): A non-negative integer representing the position (index) in the Fibonacci sequence. The sequence is 0-indexed.

## 6.3 Description

This function computes a number in the Fibonacci sequence, which is a series of numbers where each number is the sum of the two preceding ones. The sequence implemented by this function starts with 0 and 1.

The function handles the base cases first: - If n is 0 or a negative number, it returns 0. - If n is 1, it returns 1.

For any n greater than 1, the function iteratively calculates the value. It initializes two variables, a and b, to the first two numbers of the sequence (0 and 1). It then loops n-1 times, updating a and b in each step to represent the next pair of consecutive numbers in the sequence. The new b is calculated as the sum of the old a and b.

# Assumed implementation logic  
if n <= 0:  
 return 0  
elif n == 1:  
 return 1  
else:  
 a, b = 0, 1  
 # Loop n-1 times to get to the n-th element  
 for \_ in range(n - 1):  
 a, b = b, a + b  
 return b

After the loop completes, the variable b holds the n-th Fibonacci number, which is then returned.

## 6.4 Usage Notes

* The input n must be an integer. The function is designed for non-negative integers.
* If a negative integer is provided for n, the function will return 0.
* Be aware that Fibonacci numbers grow exponentially. While Python’s integers can handle arbitrary size, calculations for extremely large values of n can be computationally intensive and result in very large numbers.

## 6.5 Example

# Example usage  
# Calculate the 9th Fibonacci number (0-indexed)  
# The sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...  
result = fibonacci(9)  
print(result)

**Output:**

34

## 6.6 FunctionDef choose\_random\_item

# 7 Function: choose\_random\_item(items: List[str])

## 7.1 Overview

The choose\_random\_item function selects and returns a single, uniformly random item from a given list of strings.

## 7.2 parameters

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

## 7.3 Description

This function provides a simple way to get a random element from a list. The logic proceeds in two main steps:

1. **Validation**: The function first checks if the provided items list is empty using the condition if not items:. This is a crucial safeguard to ensure the function operates correctly. If the list is empty, it raises a ValueError with the message “items must not be empty”, preventing the program from proceeding with invalid input.
2. **Random Selection**: If the list is not empty, the function utilizes Python’s built-in random.choice(items) method. This method takes a sequence as input and returns a single item chosen uniformly at random from that sequence. The chosen string is then returned as the output of the choose\_random\_item function.

# Internal logic example  
import random  
  
def choose\_random\_item(items: list[str]) -> str:  
 # 1. Validate that the list is not empty  
 if not items:  
 raise ValueError("items must not be empty")  
 # 2. Return a random choice from the list  
 return random.choice(items)

## 7.4 Usage Notes

Important points to consider when using this function:

* 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 it is imported in the scope where this function is defined or called.
* The selection is uniformly random, meaning every item in the list has an equal probability of being chosen.

**Output Example**: The function returns a single string. If the input is ['apple', 'banana', 'cherry'], a possible return value is 'banana'.

## 7.5 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  
fruits = ["apple", "banana", "cherry", "date"]  
random\_fruit = choose\_random\_item(fruits)  
print(f"The randomly chosen fruit is: {random\_fruit}")  
  
# Example of what happens with an empty list  
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.)

### 7.5.1 FunctionDef choose\_random\_item

# 8 Function: choose\_random\_item(items: List[str]) -> str

## 8.1 Overview

The choose\_random\_item function selects and returns a single, randomly chosen element from a given list of strings.

## 8.2 parameters

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

## 8.3 Description

This function is designed to perform a random selection from a collection of items. It accepts a single argument, items, which is a list of strings.

The intended logic is to use a random selection mechanism to pick one of the strings from this list. The standard Python approach for this task is to use the random.choice() function from the built-in random module. This method takes a non-empty sequence (like a list) and returns a randomly selected element.

A complete implementation would look like the following:

import random  
  
def choose\_random\_item(items: List[str]) -> str:  
 """  
 Selects and returns a single random item from a list of strings.  
 """  
 return random.choice(items)

The function takes the items list, passes it to random.choice(), and returns the resulting string.

## 8.4 Usage Notes

* The input list items must not be empty. Providing an empty list will cause a random.choice() call to raise an IndexError.
* The selection is pseudo-random, meaning the sequence of chosen items is determined by the initial seed of the random number generator.
* While the type hint specifies List[str], the underlying random.choice() function works with any non-empty sequence (e.g., list of integers, tuples, etc.).

## 8.5 Example

The following example demonstrates the usage of the choose\_random\_item function. For this example to be executable, a sample implementation is provided.

import random  
  
# Sample implementation for the example  
def choose\_random\_item(items: List[str]) -> str:  
 if not items:  
 raise IndexError("Cannot choose from an empty list")  
 return random.choice(items)  
  
# A list of possible choices  
options = ["Option A", "Option B", "Option C", "Option D"]  
  
# Call the function to get a random choice  
selected\_option = choose\_random\_item(options)  
  
print(f"Selected Item: {selected\_option}")

**Output:**

Selected Item: Option C

*(Note: The actual output will vary with each execution because the item is chosen randomly.)*

## 8.6 FunctionDef shuffle\_copy

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

## 9.1 Overview

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

## 9.2 parameters

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

## 9.3 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() method 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 list but in a new, randomized sequence.

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

## 9.4 Usage Notes

* This function is non-mutating, meaning the original list you pass as an argument will not be altered.
* The function depends on Python’s built-in random module. Ensure this module is available in the environment.
* The order of the elements in the returned list is non-deterministic and will be different on each execution.

**Output Example**: A new list containing the same integers as the input list, but in a random order. For an input of [1, 2, 3, 4, 5], a possible output could be [3, 1, 5, 2, 4].

## 9.5 Example

import random  
from typing import List  
  
# Definition of the function  
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]  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Original List: {original\_list}")  
print(f"Shuffled Copy: {shuffled\_list}")

**Output:**

Original List: [10, 20, 30, 40, 50]  
Shuffled Copy: [30, 50, 10, 20, 40]

### 9.5.1 FunctionDef shuffle\_copy

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

## 10.1 Overview

The shuffle\_copy function creates and returns a new list containing the same elements as the input list but arranged in a random order.

## 10.2 parameters

* items: List[int] - The list of integers to be shuffled. This list will not be modified.

## 10.3 Description

This function provides a safe way to shuffle a list by operating on a copy rather than modifying the original list in place. The process is as follows:

1. The function accepts a list of integers, items, as its input.
2. It first creates a shallow copy of the items list. This is a crucial step to ensure that the original list passed by the caller remains unchanged.
3. Next, it applies an in-place shuffling algorithm (such as the Fisher-Yates shuffle) to the newly created copy, randomly reordering its elements.
4. Finally, the function returns the shuffled copy.

This non-destructive approach is useful when you need to preserve the original order of a list for other parts of your program while also working with a randomized version of it.

## 10.4 Usage Notes

* The original list passed as the items argument is not altered. This function is non-destructive.
* The returned list will have the exact same elements and length as the input list, only in a different, random order.
* Because the shuffling is random, successive calls to shuffle\_copy with the same input list will likely produce different output lists.

## 10.5 Example

import random  
from typing import List  
  
# A possible implementation for demonstration purposes, as the original code is a signature.  
def shuffle\_copy(items: List[int]) -> List[int]:  
 """Creates a shuffled copy of a list."""  
 items\_copy = items.copy()  
 random.shuffle(items\_copy)  
 return items\_copy  
  
# Example usage  
my\_numbers = [10, 20, 30, 40, 50]  
shuffled\_numbers = shuffle\_copy(my\_numbers)  
  
print(f"Original list: {my\_numbers}")  
print(f"Shuffled copy: {shuffled\_numbers}")  
  
# Another call to show randomness  
another\_shuffled\_copy = shuffle\_copy(my\_numbers)  
print(f"Another shuffled copy: {another\_shuffled\_copy}")

**Output:**

Original list: [10, 20, 30, 40, 50]  
Shuffled copy: [30, 50, 10, 20, 40]  
Another shuffled copy: [50, 20, 40, 10, 30]

*(Note: The actual order of elements in the shuffled lists will be random and may differ with each execution.)*