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 numbers.

## 1.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| a | int / float | The first number to be used in the addition. |
| b | int / float | The second number to be used in the addition. |

## 1.3 Description

This function provides a straightforward way to perform addition. It accepts two arguments, a and b, and computes their sum using the + operator. The function then returns the resulting value. While designed for numeric types like integers and floats, the + operator’s behavior in Python means it will also concatenate other types that support it, such as strings.

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

# Internal logic  
return a + b

## 1.4 Usage Notes

* For mathematical addition, ensure that both a and b are numeric types (e.g., int or float).
* If string values are passed as arguments, the function will perform string concatenation instead of arithmetic addition (e.g., num("hello", "world") would return "helloworld").

**Output Example**: A numeric sum.

15

## 1.5 Example

# Example usage with integers  
result\_int = num(5, 10)  
print(result\_int)  
  
# Example usage with floating-point numbers  
result\_float = num(7.5, 2.5)  
print(result\_float)

**Output:**

15  
10.0

### 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 to accept two parameters, a and b. The body of the function is empty, meaning it contains no implementation or logic. Consequently, when called, the function executes nothing and does not perform any operations on the input arguments. In Python, a function that does not have an explicit return statement will implicitly return None upon completion.

# The function definition is empty  
def num(a, b):  
 pass # Implicitly returns None

## 2.4 Usage Notes

* This function can be called with any two arguments, but they will have no effect on the outcome.
* The function will always return None.
* It likely serves as a stub or placeholder for functionality that is intended to be implemented in the future.

## 2.5 Example

# Example usage  
# Calling the function with two integer arguments  
result = num(10, 20)  
print(result)  
  
# Calling with different 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) -> List[int]

## 3.1 Overview

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

## 3.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| count | int | The total number of integers to generate for 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. |

## 3.3 Description

This function provides a convenient way to generate a list of random integers. The logic proceeds as follows:

1. It first validates the count parameter. If count is a negative number, the function raises a ValueError because it’s not possible to generate a negative quantity of items.
2. Next, it ensures the range is valid by checking if the start value is greater than the end value. If it is, the function automatically swaps them. This makes the function robust, as the caller does not need to worry about the order of the range boundaries.
3. Finally, it uses a list comprehension to build the list. It iterates count times and, in each iteration, calls random.randint(start, end) to get a new random integer. The random.randint method generates an integer that is greater than or equal to start and less than or equal to end. The resulting integers are collected into a list which is then returned.

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

## 3.4 Usage Notes

* This function depends on Python’s built-in random module. Ensure it is imported in your script before calling this function.
* The count parameter must be a non-negative integer (>= 0).
* Both the start and end boundaries are inclusive, meaning they can appear in the output list.
* If start is provided with a value greater than end, the function will automatically swap them to form a valid range.

**Output Example**: A possible return value for generate\_random\_integers(5, 1, 10):

[3, 9, 1, 5, 7]

## 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 (inclusive).  
random\_numbers = generate\_random\_integers(5, 10, 20)  
print(random\_numbers)

**Output:**

(Note: The actual output will vary with each execution due to its random nature.)

[15, 11, 20, 18, 12]

### 3.5.1 FunctionDef generate\_random\_integers

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

## 4.1 Overview

The generate\_random\_integers function creates and returns a list containing a specified number of random integers within a defined inclusive range.

## 4.2 parameters

| Parameter | Type | Description |
| --- | --- | --- |
| count | int | The total number of random integers to generate and include in the output list. |
| start | int | The inclusive lower bound for the random number generation. This is an optional parameter with a default value of 0. |
| end | int | The inclusive upper bound for the random number generation. This is an optional parameter with a default value of 100. |

## 4.3 Description

This function provides a straightforward way to generate a collection of random integers. It operates by iterating count times. In each iteration, it produces a single random integer that falls between the start and end values, inclusive of both endpoints. These generated integers are collected into a list, which is the final return value of the function.

The function signature includes default values for start (0) and end (100), making it convenient for generating common ranges of random numbers without specifying these parameters every time. For example, calling generate\_random\_integers(5) would produce a list of 5 integers between 0 and 100.

A common implementation for this logic would use Python’s random module.

import random  
from typing import List  
  
def generate\_random\_integers(count: int, start: int = 0, end: int = 100) -> List[int]:  
 """Generates a list of random integers within a specified range."""  
 # A list comprehension is used for a concise implementation.  
 # It loops 'count' times, calling random.randint() in each iteration.  
 return [random.randint(start, end) for \_ in range(count)]

## 4.4 Usage Notes

* The value of start must be less than or equal to the value of end. If start is greater than end, the underlying random.randint() function will raise a ValueError.
* The function returns a List[int] with a length equal to the count argument.
* The output is non-deterministic; each call to the function will produce a different list of random integers.

## 4.5 Example

# To use this function, you would first need to import it.  
# from your\_module import generate\_random\_integers  
import random  
from typing import List  
  
# Example implementation for demonstration purposes  
def generate\_random\_integers(count: int, start: int = 0, end: int = 100) -> List[int]:  
 return [random.randint(start, end) for \_ in range(count)]  
  
# Generate a list of 5 random integers between 10 and 50.  
random\_numbers = generate\_random\_integers(count=5, start=10, end=50)  
print(random\_numbers)

**Output:**

[42, 15, 33, 48, 21]

*(Note: The actual output will vary as the numbers are generated randomly.)*

## 4.6 FunctionDef fibonacci

# 5 Function: fibonacci(n: int) -> int

## 5.1 Overview

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

## 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 specific number in the Fibonacci sequence, which is a series 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 because 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 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 the new b. This operation, a, b = b, a + b, effectively moves the sequence forward one step.

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

# Initialization for the sequence F(0)=0, F(1)=1  
a, b = 0, 1  
# Loop n times to find the nth number  
for \_ in range(n):  
 # Update a to the next number and b to the number after that  
 a, b = b, a + b  
# a now holds the nth Fibonacci number  
return a

## 5.4 Usage Notes

* The function expects a non-negative integer for the n parameter. A ValueError will be raised for negative inputs.
* The sequence is 0-indexed, meaning fibonacci(0) returns the first number (0), fibonacci(1) returns the second (1), and so on.
* This implementation is iterative and avoids the performance issues of deep recursion, making it suitable for calculating larger Fibonacci numbers efficiently.

**Output Example**: A single integer representing the Fibonacci number at the specified index.

## 5.5 Example

# Example usage: Find the 9th Fibonacci number (0-indexed)  
n\_index = 9  
result = fibonacci(n\_index)  
print(f"The Fibonacci number at index {n\_index} is: {result}")

**Output:**

The Fibonacci number at index 9 is: 34

### 5.5.1 FunctionDef fibonacci

# 6 Function: fibonacci(n: int) -> int

## 6.1 Overview

The fibonacci function is a placeholder intended to calculate the nth number in the Fibonacci sequence.

## 6.2 parameters

* n (int): The index (position) in the Fibonacci sequence to be calculated.

## 6.3 Description

This function is defined to compute a number from the Fibonacci sequence based on its position n. The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, typically starting with 0 and 1.

As currently implemented, the function is a stub with an empty body. It accepts an integer n as input but does not contain any logic to perform the calculation. Consequently, it does not explicitly return a value. In Python, a function that does not have a return statement implicitly returns None.

A complete implementation would involve logic, either iterative or recursive, to calculate the value at the n-th position of the sequence (e.g., 0, 1, 1, 2, 3, 5, 8, …).

## 6.4 Usage Notes

* This function is currently unimplemented. Calling it will not perform any calculation.
* The function will implicitly return None for any input value.
* The parameter n is intended to be a non-negative integer representing the position in the sequence.

## 6.5 Example

# Example usage  
result = fibonacci(10)  
print(result)

**Output:**

None

## 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 element from a given list of strings.

## 7.2 parameters

* **items** (List[str]): A list of strings from which a random item will be chosen. This list cannot be empty.

## 7.3 Description

This function provides a safe way to pick a random item from a list. The logic proceeds as follows:

1. It first performs a validation check to see if the input list items is empty using the condition if not items:.
2. If the list is empty, the function raises a ValueError with the message “items must not be empty”. This is a crucial guard clause to prevent the underlying random.choice function from failing, as it cannot operate on an empty sequence.
3. If the list is not empty, the function proceeds to call random.choice(items). This standard library function selects a single item from the items list. The selection is uniformly random, meaning every item in the list has an equal probability of being chosen.
4. Finally, the randomly selected string is returned as the result.

# Internal logic for selecting an item  
return random.choice(items)

## 7.4 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 in the scope where this function is defined or used.
* The function is type-hinted to accept a list of strings (List[str]) and return a single string (str).

**Output Example**: The function returns a single string from the input list. For an input of ["red", "green", "blue"], a possible return value is:

"green"

## 7.5 Example

# You must import the 'random' module for this function to work  
import random  
from typing import List  
  
# Definition of the function  
def choose\_random\_item(items: List[str]) -> str:  
 if not items:  
 raise ValueError("items must not be empty")  
 return random.choice(items)  
  
# Example usage  
colors = ["red", "green", "blue", "yellow", "purple"]  
random\_color = choose\_random\_item(colors)  
print(f"The chosen color is: {random\_color}")  
  
# Example of what happens with an empty list  
try:  
 choose\_random\_item([])  
except ValueError as e:  
 print(f"Error: {e}")

**Output:**

The chosen color is: blue  
Error: items must not be empty

(Note: The chosen color in the first line of the output is random and will likely be different each time the code is run.)

### 7.5.1 FunctionDef choose\_random\_item

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

## 8.1 Overview

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

## 8.2 parameters

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

## 8.3 Description

This function provides a straightforward way to get a random element from a list. It takes a single argument, items, which is expected to be a list of strings. Internally, the function utilizes Python’s random module, specifically the random.choice() method. This method is designed to pick a random item from a non-empty sequence. The function then returns the selected string.

If the provided items list is empty, calling this function will result in an IndexError because random.choice() cannot select an item from an empty sequence.

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

## 8.4 Usage Notes

* The input list items must not be empty. Passing an empty list will raise an IndexError.
* The function is dependent on Python’s random module.
* The selection is pseudo-random and its properties depend on the underlying random number generator used by the random module.

## 8.5 Example

import random  
from typing import List  
  
# To make the random choice predictable for this example, we set a seed.  
random.seed(42)  
  
def choose\_random\_item(items: List[str]) -> str:  
 """  
 Randomly selects and returns one item from a list of strings.  
 """  
 if not items:  
 raise IndexError("Cannot choose from an empty list")  
 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}")

**Output:**

The randomly selected color is: yellow

## 8.6 FunctionDef shuffle\_copy

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

## 9.1 Overview

The shuffle\_copy function returns a new, randomly shuffled copy of a given list, ensuring the original input list is not altered.

## 9.2 parameters

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

## 9.3 Description

This function provides a safe way to shuffle a list without side effects. The process is straightforward:

1. A shallow copy of the input items list is created using copy = list(items). This is a crucial step that isolates the operation from the original data.
2. The random.shuffle() function is then called on this copy. random.shuffle() shuffles the elements of a list in-place.
3. Finally, the function returns the modified copy, which now contains the same elements as the original items list but in a random order.

By operating on a copy, the function guarantees that the list passed as an argument remains in its original state after the function call.

import random  
  
# Original list  
original\_list = [10, 20, 30, 40, 50]  
  
# The shuffle\_copy function does not exist in the standard random library,  
# so we define it here for the example.  
def shuffle\_copy(items: list) -> list:  
 copy = list(items)  
 random.shuffle(copy)  
 return copy  
  
# Get a shuffled copy  
shuffled\_list = shuffle\_copy(original\_list)  
  
print(f"Original list (unchanged): {original\_list}")  
print(f"New shuffled list: {shuffled\_list}")

## 9.4 Usage Notes

* This function requires the random module to be imported to use random.shuffle().
* The primary benefit of this function is its non-mutating behavior. It is a “safe” alternative to calling random.shuffle() directly on a list that you need to preserve.
* Although the type hint specifies List[int], the function will work correctly with lists containing elements of any type (e.g., strings, floats, or mixed types).

**Output Example**: The output is a new list with the same elements as the input, but in a random order. The exact order will vary with each execution.

[4, 1, 5, 2, 3]

## 9.5 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  
  
# Define an original list of numbers  
my\_numbers = [1, 2, 3, 4, 5, 6, 7]  
  
print(f"Original list before shuffling: {my\_numbers}")  
  
# Get a shuffled copy of the list  
shuffled\_numbers = shuffle\_copy(my\_numbers)  
  
print(f"New shuffled list: {shuffled\_numbers}")  
print(f"Original list after shuffling: {my\_numbers}")

**Output:**

Original list before shuffling: [1, 2, 3, 4, 5, 6, 7]  
New shuffled list: [4, 7, 2, 1, 6, 3, 5]  
Original list after shuffling: [1, 2, 3, 4, 5, 6, 7]

### 9.5.1 FunctionDef shuffle\_copy

# 10 Function: shuffle\_copy(items: List[int]) -> 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: A list of integers (List[int]) that will be shuffled. The original list provided to this parameter remains unmodified.

## 10.3 Description

This function provides a non-destructive way to shuffle a list. It operates in two main steps:

1. **Create a Copy:** It first creates a shallow copy of the input items list. This step is crucial as it ensures that the original list passed by the user is not altered.
2. **Shuffle the Copy:** It then shuffles the elements of this newly created copy in-place, randomizing their order. A common implementation would use an algorithm like the Fisher-Yates shuffle.

Finally, the function returns the new, shuffled list.

A standard implementation would look like this:

import random  
from typing import List  
  
def shuffle\_copy(items: List[int]) -> List[int]:  
 # Create a shallow copy of the input list  
 copied\_list = items[:]  
 # Shuffle the copied list in-place  
 random.shuffle(copied\_list)  
 # Return the new, shuffled list  
 return copied\_list

## 10.4 Usage Notes

* The primary feature of this function is that it is **non-destructive**. The original list you pass as an argument will not be changed.
* The function returns a **new list**. You must assign the result to a variable to access the shuffled sequence.
* Due to the nature of randomization, calling the function multiple times with the identical input list will likely produce different output lists each time.
* Although the type hint specifies List[int], the underlying logic works for lists containing elements of any type (e.g., str, float, or custom objects).

## 10.5 Example

# Example usage  
original\_numbers = [1, 2, 3, 4, 5, 6, 7, 8]  
  
print(f"Original list before shuffling: {original\_numbers}")  
  
shuffled\_numbers = shuffle\_copy(original\_numbers)  
  
print(f"Original list after shuffling: {original\_numbers}")  
print(f"New shuffled list: {shuffled\_numbers}")  
  
# Another shuffle will produce a different order  
another\_shuffled\_list = shuffle\_copy(original\_numbers)  
print(f"Another shuffled list: {another\_shuffled\_list}")

**Output:**

Original list before shuffling: [1, 2, 3, 4, 5, 6, 7, 8]  
Original list after shuffling: [1, 2, 3, 4, 5, 6, 7, 8]  
New shuffled list: [5, 2, 8, 1, 7, 4, 6, 3]  
Another shuffled list: [3, 7, 1, 8, 5, 2, 4, 6]