Debugging Tools in VSCode:

1. **Breakpoint**: A breakpoint is a marker you set in your code that pauses the execution when reached. This allows you to examine variables and program flow at that specific point.
2. **Step Over**: Executes the next line of code but doesn't step into any function calls. It's useful if you don't want to go inside a function but just want to see its outcome.
3. **Step Into**: Steps into the function on the current line if there is one. This is used when you want to dive deeper into a function to see what's happening inside.
4. **Step Out**: If you've stepped into a function, `Step Out` continues running the code until it exits the current function, returning you to the calling function.
5. **Continue**: Runs the program until the next breakpoint is encountered or the program finishes.
6. **Watch**: Allows you to monitor the value of specific variables or expressions as you step through the code.
7. **Call Stack**: Shows the function calls that are currently active. It's useful for understanding how you arrived at a particular point in the code.

Problem 1: Recursion and Stack Depth Issue

Problem:

```
python
Copy code
def factorial(n):
    # There's a bug here. Find it using the debugger.
    if n == 0:
        return 1
    return n * factorial(n - 1)

number = int(input("Enter a number to calculate its factorial: "))
result = factorial(number)
print(f"The factorial of {number} is {result}")
```

Expected Behavior:

For a positive number `n`, the factorial should be `n * (n-1) * (n-2) * ... * 1`. If `n` is negative, it should throw an error.

Observed Behavior:

If you input `-1`, the program enters an infinite loop due to recursion not handling negative numbers.

Debugging Steps & Observations:

1. **Breakpoint**: Set a breakpoint on the line `return n * factorial(n - 1)`.
2. **Step Into**: Observe the function calling itself. You'll notice that for `n = -1`, it keeps calling `factorial(n - 1)`indefinitely.

   - **Observation**: The recursive call does not have a condition to handle negative values, causing infinite recursion.
3. **Watch**: Add `n` to the `Watch` window to monitor how it changes with each recursion step.
   - **Observation**: You'll see `n` decreasing continuously into negative numbers without stopping.
4. **Fix**: Modify the code to handle negative input:

```
python
Copy code
def factorial(n):
    if n < 0:
        raise ValueError("Negative numbers are not allowed!")
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Solution:

Handle negative input by raising an exception.

Problem 2: Off-by-One Error in List Manipulation

Problem:

```
python
Copy code
def remove_duplicates(nums):
    if not nums:
        return nums
    write_index = 1
    for i in range(1, len(nums)):
        if nums[i] != nums[i - 1]:
            nums[write_index] = nums[i]
            write_index += 1
    return nums[:write_index]

numbers = [1, 2, 2, 3, 3, 4, 5, 5, 6]
result = remove_duplicates(numbers)
print(f"List after removing duplicates: {result}")
```

Expected Behavior:

The function should return `[1, 2, 3, 4, 5, 6]` without duplicates.

Observed Behavior:

For certain inputs, the function leaves one duplicate in the final list.

Debugging Steps & Observations:

1. **Breakpoint**: Place a breakpoint on the line inside the `for` loop: `if nums[i] != nums[i - 1]`.

2. **Step Over**: Use `Step Over` to iterate through the loop and watch how `write_index` and `nums` change.
    - **Observation**: In some cases, the comparison `nums[i] != nums[i - 1]` doesn't trigger as expected.
3. **Watch**: Monitor `write_index`, `nums`, and `i` to see if the indices are updated correctly.
    - **Observation**: You'll see that `write_index` updates correctly, but the `return nums[:write_index]` slice might be incorrect if `write_index` doesn't account for all unique values.
4. **Fix**: Adjust the function to handle edge cases:

    ```
    python
    Copy code
    def remove_duplicates(nums):
        if not nums:
            return nums
        write_index = 1
        for i in range(1, len(nums)):
            if nums[i] != nums[write_index - 1]:  # Corrected condition
                nums[write_index] = nums[i]
                write_index += 1
        return nums[:write_index]
    ```

Solution:

The condition was slightly off. Now the code correctly adjusts the comparison to avoid off-by-one errors.

Problem 3: Complex Data Handling (Dictionaries & Nested Structures)

Problem:

```
python
Copy code
def calculate_total_price(products):
    total_price = 0
    for product_id, product_details in products.items():
        if 'price' in product_details:
            total_price += product_details['price']
        if 'discount' in product_details:
            total_price -= product_details['discount']  # This might not be
correct.
    return total_price

inventory = {
    "p1": {"name": "Laptop", "price": 1000, "discount": 100},
    "p2": {"name": "Mouse", "price": 50, "discount": 10},
    "p3": {"name": "Keyboard", "price": 70},  # Missing discount
    "p4": {"name": "Monitor", "price": 300, "discount": 50},
}

total = calculate_total_price(inventory)
```

```
print(f"The total price of inventory is: {total}")
```

Expected Behavior:

The function should correctly calculate the total price considering discounts.

Observed Behavior:

The function subtracts the discount even if the product does not have one, leading to incorrect results.

Debugging Steps & Observations:

1. **Breakpoint**: Place a breakpoint inside the loop, especially at `if 'discount' in product_details`.
2. **Step Over**: Use `Step Over` to go through each item in the dictionary and see how the calculations change.
   - **Observation**: Notice how `total_price` gets updated incorrectly when a product doesn't have a discount.
3. **Step Into**: Step into the conditional blocks if needed to verify why a subtraction occurs when no discount exists.
4. **Watch**: Monitor `product_details` and `total_price` to see how calculations evolve.
   - **Observation**: The `discount` calculation is incorrect; missing products without a discount shouldn't impact the total.
5. **Fix**: Correct the logic to handle the absence of a discount:

   ```
   python
   Copy code
   def calculate_total_price(products):
       total_price = 0
       for product_id, product_details in products.items():
           if 'price' in product_details:
               total_price += product_details['price']
           if product_details.get('discount', 0):
               total_price -= product_details['discount']
       return total_price
   ```

Solution:

Use `.get()` to handle missing keys, ensuring calculations only subtract when a discount is present.

Let me know if you need more examples or detailed explanations for specific debugging scenarios!

The `.get('discount', 0)` method is used to safely access the value associated with the `'discount'` key in a dictionary. Here's what it does:

- **`'discount'`**: This is the key you are looking for in the dictionary.

- `0`: This is the default value that will be returned if the key `'discount'` is not found in the dictionary.

Explanation:

- If the `'discount'` key exists in the dictionary, `.get('discount', 0)` will return its value.
- If the `'discount'` key does not exist, it will return `0` instead of raising a `KeyError`.