Understanding Pure Functions with a Step-by-Step Example

Pure functions are a fundamental concept in functional programming. They enhance code readability, maintainability, and reliability by avoiding side effects. Here's a step-by-step guide based on the provided explanation, showcasing the transformation of an impure function into a pure one.

What Are Pure Functions?

A **pure function**:

1. Does not modify or interact with the global scope.
2. Returns consistent outputs for the same inputs (deterministic behavior).
3. Does not cause side effects like altering external data or states.

Example Walkthrough

Initial Impure Function

```
my_list = [1, 2, 3]  # Global list

def add_to_list(item):
    my_list.append(item)  # Modifies the global list
    return my_list

add_to_list(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

- **Problem**: The function modifies the global list `my_list`. Hence, it is **not a pure function**.

First Attempt: Returning a New Variable

```
my_list = [1, 2, 3]

def add_to_list(item):
    my_list.append(item)  # Still modifies the global list
    return my_list

new_list = add_to_list(4)
```

```
print(my_list)  # Output: [1, 2, 3, 4]
print(new_list)  # Output: [1, 2, 3, 4]
```

- **Issue**: Although the function returns `new_list`, the original `my_list` is still altered. The function remains impure.

Second Attempt: Accepting the List as an Argument

```
my_list = [1, 2, 3]

def add_to_list(lst, item):
    lst.append(item)  # Modifies the passed list
    return lst

new_list = add_to_list(my_list, 4)
print(my_list)  # Output: [1, 2, 3, 4]
print(new_list)  # Output: [1, 2, 3, 4]
```

- **Issue**: The function directly modifies the passed list (`lst`), which still results in an impure function.

Final Solution: Copying the List

```
my_list = [1, 2, 3]

def add_to_list(lst, item):
    new_list = lst.copy()  # Create a copy of the list
    new_list.append(item)  # Modify the copy
    return new_list  # Return the modified copy

new_list = add_to_list(my_list, 4)
print(my_list)  # Output: [1, 2, 3]  (Unchanged)
print(new_list)  # Output: [1, 2, 3, 4]
```

- **Solution**: The function now creates a **copy** of the original list (`lst.copy()`) and modifies it. The original list remains unchanged, making this function **pure**.

Benefits of Pure Functions

1. **Predictable Outputs**: Always produce the same output for the same input.
2. **Easier Debugging**: No side effects mean fewer unexpected changes in the code.
3. **Reusability**: Independent functions can be reused across different contexts.

4. **Concurrency-Friendly**: Pure functions avoid shared state, making them suitable for multi-threaded applications.
5. **Memoization**: Since outputs are predictable, caching results for repeated inputs becomes efficient.

Key Takeaways

- **Impure functions** can unintentionally modify external data, causing bugs and unpredictable behavior.
- By carefully copying data and avoiding external changes, you can refactor impure functions into **pure functions**.
- Pure functions are a cornerstone of functional programming, enabling cleaner, more maintainable, and extensible codebases.

By understanding and implementing pure functions, you'll write better code that's more aligned with best practices in software development.