Tasks – List as Argument to a Function

Practice passing a **list as an argument** to a function: reading from it, returning results, and understanding that **mutating** the list inside the function (e.g. append, change by index) affects the **original** list the caller passed. Create each file, run it, and check the output.

Run scripts with: `python3 script_name.py`

Part 1 – Pass a list and read it

Task 1.1 – Function that takes a list and prints its length (`list_arg_len.py`)

- Create `list_arg_len.py`.
- Define a function `print_length(items)` that takes one parameter `items` (a list). Inside the function, print the length of the list: `print(len(items))`.
- In the script, create a list e.g. `nums = [10, 20, 30]` and call `print_length(nums)`. Then call `print_length(["a", "b"])`. The list is **passed** into the function; the function only **reads** it (len).

Expected output:

```
3
2
```

Task 1.2 – Function that returns the sum of a list (`list_arg_sum.py`)

- Create `list_arg_sum.py`.
- Define a function `sum_list(numbers)` that takes a list of numbers. Use a loop to add them up and **return** the sum. Do not change the list.

- Create `vals = [1, 2, 3, 4, 5]`, call `total = sum_list(vals)`, and print `total`. Then print `sum_list([10, 20])`. The function receives the list and returns a single value.

Expected output:

```
15
30
```

Task 1.3 – Function that prints each element of a list (`list_arg_print_each.py`)

- Create `list_arg_print_each.py`.
- Define a function `print_each(items)` that takes a list and, inside the function, loops over it and prints each element (one per line).
- Create `fruits = ["apple", "banana", "cherry"]` and call `print_each(fruits)`. Then call `print_each([1, 2, 3])`. Passing the list lets the function read and use its elements.

Expected output:

```
apple
banana
cherry
1
2
3
```

Part 2 – Return a new list (don't mutate)

Task 2.1 – Function that returns a doubled list (`list_arg_double.py`)

- Create `list_arg_double.py`.
- Define a function `double_list(numbers)` that takes a list of numbers. **Create a new list** (e.g. start with `result = []`), loop over the given list, append each number times 2 to the new list, then **return** that new list. Do **not** change the original list.

- Create `nums = [1, 2, 3]`, call `doubled = double_list(nums)`, and print `doubled`. Print `nums` as well — it should still be `[1, 2, 3]`. The function returned a **new** list.

Expected output:

```
[2, 4, 6]
[1, 2, 3]
```

Task 2.2 – Function that returns first and last (`list_arg_first_last.py`)

- Create `list_arg_first_last.py`.
- Define a function `first_and_last(items)` that takes a list. If the list has at least one element, return a **new** list containing the first element and the last element, e.g. `[items[0], items[-1]]`. If the list is empty, return an empty list.
- Call and print: `print(first_and_last([10, 20, 30, 40]))` and `print(first_and_last([]))`. The function **returns** a new list; it does not modify the argument.

Expected output:

```
[10, 40]
[]
```

Part 3 – Mutating the list inside the function

When you pass a list to a function, the parameter refers to the **same** list as the caller. If the function **mutates** it (append, change by index), the caller's list changes too.

Task 3.1 – Function that appends to the list (`list_arg_append.py`)

- Create `list_arg_append.py`.
- Define a function `add_item(items, value)` that takes a list and a value. Inside the function, **append** the value to the list: `items.append(value)`. Do not return the list (or you can return it; the important part is the mutation).

- In the script, create `my_list = [1, 2, 3]`, then call `add_item(my_list, 4)`. Print `my_list` **after** the call. You should see `[1, 2, 3, 4]` — the function changed the **original** list.

Expected output:

```
[1, 2, 3, 4]
```

Task 3.2 – Function that changes an element by index (`list_arg_change.py`)

- Create `list_arg_change.py`.
- Define a function `set_at_index(items, index, value)` that takes a list, an index, and a value. Inside the function, set `items[index] = value`.
- Create `data = [10, 20, 30, 40]`, call `set_at_index(data, 2, 99)`, then print `data`. The list should show 99 at index 2. The function mutated the list that was passed in.

Expected output:

```
[10, 20, 99, 40]
```

Task 3.3 – Function that clears the list (`list_arg_clear.py`)

- Create `list_arg_clear.py`.
- Define a function `clear_list(items)` that takes a list. Inside the function, use a loop to remove all elements (e.g. repeatedly `items.pop()` while the list is not empty), or use `items.clear()` if you know it. Do not reassign the parameter (e.g. `items = []` would not change the caller's list).
- Create `vals = [1, 2, 3]`, call `clear_list(vals)`, then print `vals`. It should be `[]`. Mutating the list (clearing it) affects the original.

Expected output:

```
[]
```

Part 4 – List argument with other parameters

Task 4.1 – Add a value to list if not already there (`list_arg_add_if_new.py`)

- Create `list_arg_add_if_new.py`.
- Define a function `add_if_missing(items, value)` that takes a list and a value. If `value` is **not** in the list (`value not in items`), append it. Otherwise do nothing. After the function, the caller's list may be modified.
- Create `seen = ["a", "b"]`, call `add_if_missing(seen, "c")`, print `seen`. Call `add_if_missing(seen, "b")` (already there), print `seen` again. You should see `["a", "b", "c"]` then still `["a", "b", "c"]`.

Expected output:

```
['a', 'b', 'c']
['a', 'b', 'c']
```

Task 4.2 – Function that takes list and number, returns count (`list_arg_count.py`)

- Create `list_arg_count.py`.
- Define a function `count_value(items, target)` that takes a list and a value. Loop over the list and count how many elements equal `target`. **Return** that count. Do not mutate the list.
- Create `nums = [1, 2, 2, 3, 2, 4]`, call and print `print(count_value(nums, 2))` (should be 3). Then print `count_value(nums, 5)` (should be 0). The list is read-only here.

Expected output:

```
3
0
```

Done

You've used: passing a **list as an argument**, **reading** from it (len, loop, sum), **returning** a new list or a single value, and **mutating** the list inside the function (append, change by index, clear) and seeing that the caller’s list is the same object, so it changes too. You also combined a list parameter with another parameter (e.g. value to add or count).