# DATA C88C
# Summer 2025

# Python Lists, Mutability
Discussion 6: July 8, 2025

## List Mutation

The two most common mutation operations for lists are item assignment and the `append` method.

```
>>> s = [1, 3, 4]
>>> t = s  # A second name for the same list
>>> t[0] = 2  # this changes the first element of the list to 2, affecting both s and t
>>> s
[2, 3, 4]
>>> s.append(5)  # this adds 5 to the end of the list, affecting both s and t
>>> t
[2, 3, 4, 5]
```

There are many other list mutation methods:

- `append(elem)`: Add `elem` to the end of the list. Return `None`.
- `extend(s)`: Add all elements of iterable `s` to the end of the list. Return `None`.
- `insert(i, elem)`: Insert `elem` at index `i`. If `i` is greater than or equal to the length of the list, then `elem` is inserted at the end. This does not replace any existing elements, but only adds the new element `elem`. Return `None`.
- `remove(elem)`: Remove the first occurrence of `elem` in list. Return `None`. Errors if `elem` is not in the list.
- `pop(i)`: Remove and return the element at index `i`.
- `pop()`: Remove and return the last element.

**Q1: Nested Lists**

The mathematical constant $e$ is 2.718281828…

Draw an environment diagram to determine what is printed by the following code.

See the web version of this resource for the environment diagram.

If you have questions, ask them instead of just looking up the answer! First ask your group, and then the course staff.

### Q2: Apply in Place

Implement `apply_in_place`, which takes a one-argument function `fn` and a list `s`. It modifies `s` so that each element is the result of applying `fn` to that element. It returns `None`.

```
def apply_in_place(fn, s):
    """Replace each element x of s with fn(x).

    >>> original_list = [5, -1, 2, 0]
    >>> apply_in_place(lambda x: x * x, original_list)
    >>> original_list
    [25, 1, 4, 0]
    """
    for i in range(len(s)):
        s[i] = fn(s[i])
```

One approach is to use `for i in range(...)` to iterate over the indices (positions) of `s`.

# Immutable Lists

**Q3: Reverse (iteratively)**

Write a function `reverse_iter` that takes a list and returns a new list that is the reverse of the original. Use iteration! Do not use `lst[::-1]`, `lst.reverse()`, or `reversed(lst)`!

```
def reverse_iter(lst):
    """Returns the reverse of the given list.

    >>> reverse_iter([1, 2, 3, 4])
    [4, 3, 2, 1]
    >>> import inspect, re
    >>> cleaned = re.sub(r"#.*\\n", '', re.sub(r'"{3}[\s\S]*?"{3}', '', inspect.getsource
    (reverse_iter)))
    >>> print("Do not use lst[::-1], lst.reverse(), or reversed(lst)!") if any([r in
    cleaned for r in ["[::", ".reverse", "reversed"]]) else None
    """
    new, i = [], 0
    while i < len(lst):
        new = [lst[i]] + new
        i += 1
    return new
```

```
def reverse_iter(lst):
    """Returns the reverse of the given list.

    >>> reverse_iter([1, 2, 3, 4])
    [4, 3, 2, 1]
    >>> import inspect, re
    >>> cleaned = re.sub(r"#.*\\n", '', re.sub(r'"{3}[\s\S]*?"{3}', '', inspect.getsource
    (reverse_iter)))
    >>> print("Do not use lst[::-1], lst.reverse(), or reversed(lst)!") if any([r in
    cleaned for r in ["[::", ".reverse", "reversed"]]) else None
    """
    new, i = [], 0
    while i < len(lst):
        new = [lst[i]] + new
        i += 1
    return new
```

# Tree Recursion with Lists

To solve this problem, all you need are list literals (e.g., `[1, 2, 3]`), item selection (e.g., `s[0]`), list addition (e.g., `[1] + [2, 3]`), `len` (e.g., `len(s)`), and slicing (e.g., `s[1:]`). Use those!

The most important thing to remember about lists is that a non-empty list `s` can be split into its first element `s[0]` and the rest of the list `s[1:]`.

```
>>> s = [2, 3, 6, 4]
>>> s[0]
2
>>> s[1:]
[3, 6, 4]
```

**Q4: Max Product**

Implement `max_product`, which takes a list of numbers and returns the maximum product that can be formed by multiplying together non-consecutive elements of the list. Assume that all numbers in the input list are greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product of non-consecutive elements of s.

    >>> max_product([10, 3, 1, 9, 2])   # 10 * 9
    90
    >>> max_product([5, 10, 5, 10, 5])  # 5 * 5 * 5
    125
    >>> max_product([])                 # The product of no numbers is 1
    1
    """
    if s == []:
        return 1
    if len(s) == 1:
        return s[0]
    else:
        return max(s[0] * max_product(s[2:]), max_product(s[1:]))
        # OR
        return max(s[0] * max_product(s[2:]), s[1] * max_product(s[3:]))
```

First try multiplying the first element by the `max_product` of everything after the first two elements (skipping the second element because it is consecutive with the first), then try skipping the first element and finding the `max_product` of the rest. To find which of these options is better, use `max`.

A great way to get help is to talk to the course staff!

This solution begins with the idea that we either include `s[0]` in the product or not:

- If we include `s[0]`, we cannot include `s[1]`.
- If we don't include `s[0]`, we can include `s[1]`.

The recursive case is that we choose the larger of: - multiplying `s[0]` by the `max_product` of `s[2:]` (skipping `s[1]`) OR - just the `max_product` of `s[1:]` (skipping `s[0]`)

Here are some key ideas in translating this into code: - The built-in `max` function can find the larger of two numbers, which in this case come from two recursive calls. - In every case, `max_product` is called on a list of numbers and its return value is treated as a number.

An expression for this recursive case is:

```
max(s[0] * max_product(s[2:]), max_product(s[1:]))
```

Since this expression never refers to `s[1]`, and `s[2:]` evaluates to the empty list even for a one-element list `s`, the second base case (`len(s) == 1`) can be omitted if this recursive case is used.

The recursive solution above explores some options that we know in advance will not be the maximum, such as skipping both `s[0]` and `s[1]`. Alternatively, the recursive case could be that we choose the larger of: - multiplying `s[0]` by the `max_product` of `s[2:]` (skipping `s[1]`) OR - multiplying `s[1]` by the `max_product` of `s[3:]` (skipping `s[0]` and `s[2]`)

An expression for this recursive case is:

```
max(s[0] * max_product(s[2:]), s[1] * max_product(s[3:]))
```