CS1 things to remember:

Sets:

- Use `set(S1)` to quickly convert lists, tuples, or other iterables to sets and eliminate duplicates.

```
# Convert a list with duplicate elements into a set
 S1 = [1, 2, 2, 3, 4, 4, 5]
set_S1 = set(S1) # Removes duplicates
print(set_S1) # Output: {1, 2, 3, 4, 5}
```

- Sets are very efficient for membership testing (checking if an item is in a set).

Check if an element is in the set

```
S1 = [1, 2, 2, 30, 4, 4, 5]
if 30 in set_S1:
    print("30 is in the set!")  # This will print.
```

For Loops:

- A **for loop** is used when you want to **repeat a block of code** for each element in a sequence (like a list, string, or range of numbers). The loop iterates over each element, one at a time, and executes the code block inside the loop for each element.
- Syntax:

```
for variable in sequence: # Code to execute for each item in the sequence
```

- `variable`: This is a temporary variable that takes the value of each element in the sequence one by one.
- `sequence`: This is the collection (like a list, tuple, string, or range) that you're iterating over.

Example:Finding the Maximum Value in a List

```
def find_max(L):
    if L == []:  #Edge case: empty list
        return None
    max_value = L[0]  #Assume the first number is the largest
    for i in L:
        if i > max_value:  # If a number is larger, update max_value
            max_value = i
    return max_value  # Return the largest value
```

List comprehensions:

- The general form of a simple list comprehension is:

```
[ expression(x) for x in sequence ]
```

Where sequence can be any of the types(lists, tuples, strings, ranges) or collections, such as sets. More generally:

```
[expression(x) for x in sequence1 for y in sequence2 ... if condition(x,y)]
```

```
>>>[(x,y) for x in range(3) for y in range(3) if x !=y]
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
#if x != y ensures only pairs where x and y are not the same are included.
>>>[(x,y) for x in range(3) for y in range(3) if x > y]
[(1,0), (2,0), (2,1)]
```

While Loops

- **A `while` loop repeatedly executes a block of code as long as a specified condition is `True`. It continues until the condition becomes `False`.**

Key Parts of a While Loop:

- **Condition: A logical expression checked before each iteration. If it is `True`, the loop runs; if `False`, the loop stops.**
- **Loop Body: The block of code executed repeatedly as long as the condition remains `True`.**
- **Update Step: Usually, something inside the loop updates variables so that the condition will eventually become `False` and stop the loop.**

A Simple Example: Counting from 1 to 5

```
i = 1
while i <= 5:
    print(i)
    i += 1  # Update step: increase i by 1
```

- **Condition: `i <= 5`**
- **Loop Body: `print(i)` and `i += 1`**

- **Update Step: `i += 1` ensures that eventually `i` will no longer be less than or equal to 5, stopping the loop.**

Key Points:

1. **Condition: The loop runs as long as the condition is `True`. If the condition is `False` initially, the loop won't run at all.**
2. **Infinite Loops: If the condition never becomes `False`, the loop will continue forever. This can happen if the update step is missing or incorrect.**

Example of While Loop:

```
def find_max_while(L):
    if L == []:  # Edge case: empty list
        return None

    max_value = L[0]  # Assume the first number is the largest
    index = 1  # Start from the second element

    while index < len(L):  # Loop through the list until the end
        if L[index] > max_value:  # If a number is larger, update max_value
            max_value = L[index]
        index += 1  # Move to the next index

    return max_value  # Return the largest value
```

- **Edge Case: The function first checks if the list is empty and returns `None` if it is.**
- **Initialization: It initializes `max_value` to the first element of the list and sets an `index` variable to `1` (the second element).**
- **While Loop: The loop continues as long as `index` is less than the length of the list.**
  - **Inside the loop, it checks if the current element (`L[index]`) is greater than `max_value`. If so, it updates `max_value`.**
  - **After checking, it increments `index` to move to the next element.**
- **Return Value: Finally, it returns the largest value found in the list.**

Example: Prompting for Input Until Correct Value

```
password = ""
correct_password = "abc123"

while password != correct_password:
    password = input("Enter the password: ")
```

```
print("Access granted!")
```

- **The loop continues to prompt the user for the password until they enter the correct value.**

Recursion:

A recursive function typically has:

1. **Base case(s)**: A condition that stops the recursion, preventing an infinite loop.
2. **Recursive case**: The part where the function calls itself with a smaller or simpler input.

A Simple Example: Factorial

The factorial of a number `n` (written as `n!`) is the product of all positive integers up to `n`. For example:

- `5! = 5 * 4 * 3 * 2 * 1 = 120`

The factorial of `n` can be defined recursively:

- Base case: `0! = 1`
- Recursive case: `n! = n * (n - 1)!`

Recursive Code for Factorial in Python

Here's how you could write a recursive function for factorial in Python:

```
def factorial(n):
    # Base case: when n is 0, return 1
    if n == 0:
        return 1
    else:
        # Recursive case: n * factorial of (n - 1)
        return n * factorial(n - 1)
```

Let's break down how this works:

- If you call `factorial(5)`, the function does this:
    - `factorial(5)` returns `5 * factorial(4)`
    - `factorial(4)` returns `4 * factorial(3)`

- `factorial(3)` returns `3 * factorial(2)`
- `factorial(2)` returns `2 * factorial(1)`
- `factorial(1)` returns `1 * factorial(0)`
- `factorial(0)` returns `1` (base case)

The recursive calls keep breaking the problem into smaller pieces until they hit the base case, at which point the results "unwind" and the final value is computed.

Key Points:

1. **Base Case**: Essential to prevent infinite recursion. Without it, the function would keep calling itself forever.
2. **Reduction Step**: Each recursive call should simplify the problem and move towards the base case.
3. **Call Stack**: Recursion uses the call stack, so each function call waits for the result of the next call.

Let's try another example: Fibonacci Sequence

The Fibonacci sequence is another famous example, where each number is the sum of the two preceding ones:

- `F(0) = 0`
- `F(1) = 1`
- `F(n) = F(n-1) + F(n-2)` for `n >= 2`

```
def fibonacci(n):
    # Base cases: F(0) = 0, F(1) = 1
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        # Recursive case: F(n) = F(n-1) + F(n-2)
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Try calling fibonacci(5):

- fibonacci(5) returns fibonacci(4) + fibonacci(3)
- fibonacci(4) returns fibonacci(3) + fibonacci(2)
- This continues until it reaches the base cases of fibonacci(0) and fibonacci(1).

String manipulation
Working with nested structures
Handling edge cases