TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: 126663

Unit 3 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem solving journey! Use this as a reference while you solve the breakout problems for Unit 3. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 3. In addition to the material below, you are expected to know any required concepts from previous units. You may also find advanced concepts and bonus concepts included at the bottom of this page helpful for solving problem set questions, but you are not required to know them for the unit assessment.

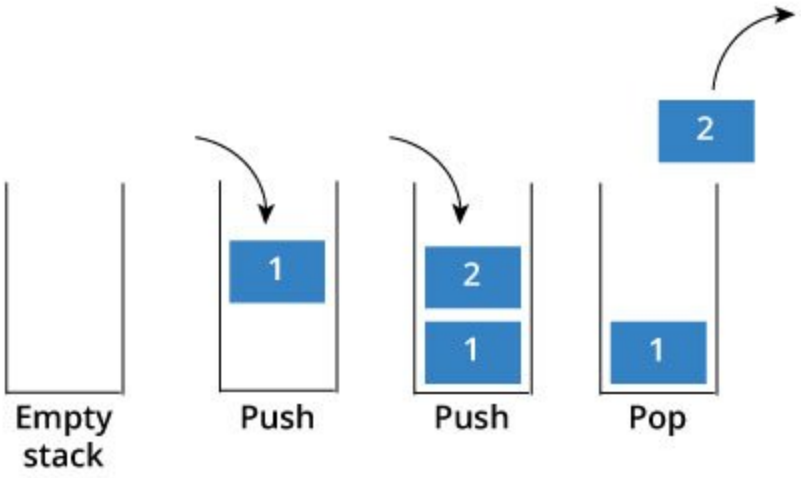
Standard Concepts

Stacks

Stacks are a special type of list where elements are always added and removed in a certain order. Stacks follow the Last In, First Out (LIFO) principle, which means that the last element added to the stack will be the first element to be removed.

Imagine the stack data structure as a stack of plates. When we add a new plate (element) to the stack, the plate gets added to the top of the existing stack. When we want to remove a plate from the stack, the plate that is easiest to remove is the plate on the top of the stack.

Adding a new element to the stack is also called *pushing* the element on to the stack. Removing an element from the stack is also called *popping* the element off of the stack.



Source: via Python Pool

Stack Implementation

Stacks can easily be implemented using the built-in list data type in Python and built-in `append()` and `pop()` methods.

```
stack = []

# Push new items onto the top of the stack
stack.append(1)
stack.append(2)

# Pop an item off the top of the stack
popped_item = stack.pop()

print(popped_item) # Prints: 2
print(stack) # Prints: [1]
```

`append()` and `pop()` both are both very fast functions (`O(1)` time complexity for those familiar with Big O), so using a normal Python list to implement a stack is an extremely efficient way to implement a stack.

When to Use a Stack

What are some common signs a problem could be solved using a stack?

- **Reversing Data**
  - Reversing the order of a sequence such as a list or string.
  - For example, Reverse String
- **Balancing Symbols**

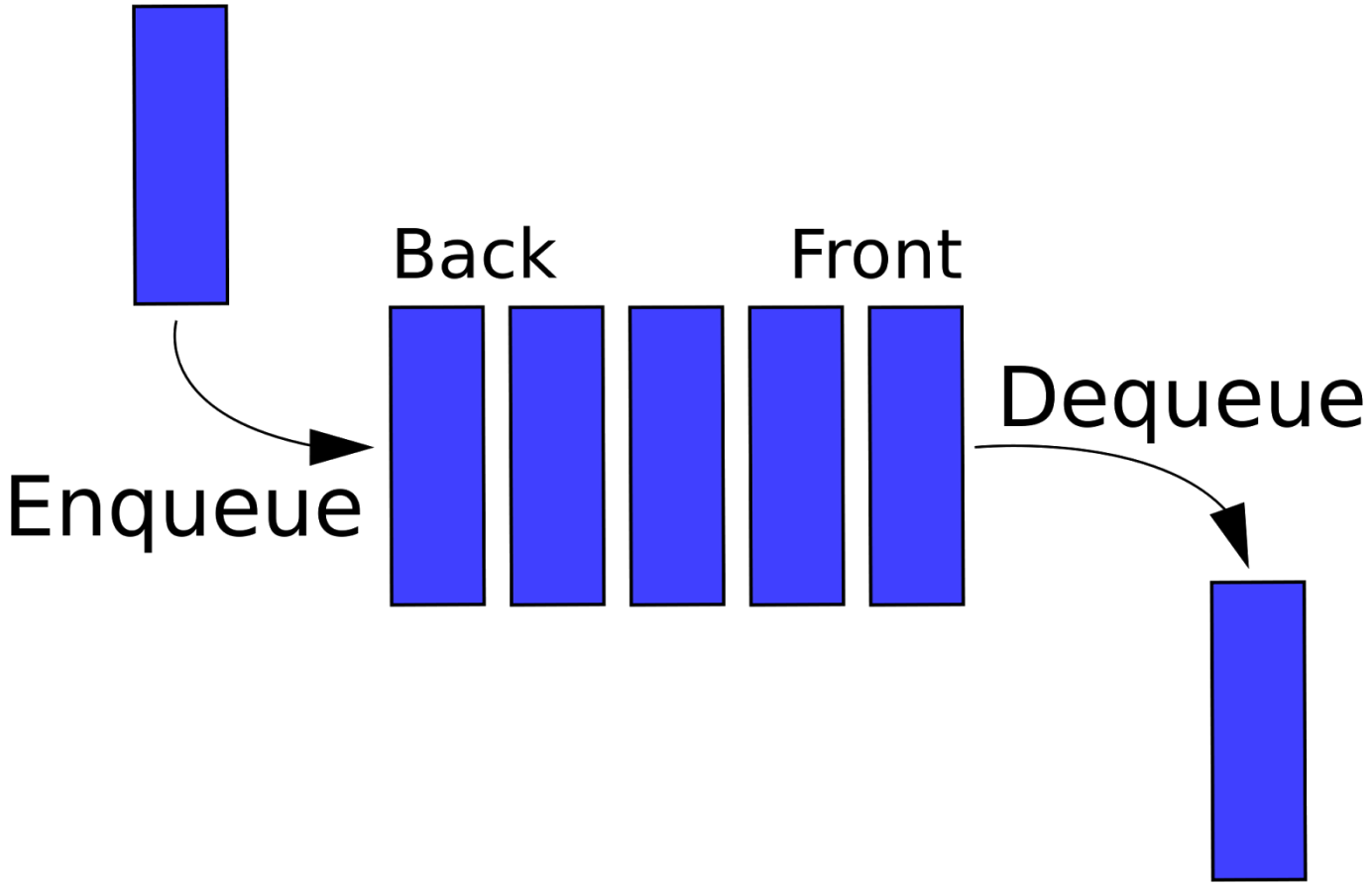
    - Stacks are commonly used to check for or manage pairs of symbols like parentheses, brackets, or braces that must open and close in a correct order.
    - For example, Valid Parentheses
- **Backtracking**
    - We won't cover backtracking algorithms in depth until Unit 8 and beyond, but this is one of the most common uses of a stack (often achieved via the recursive call stack).
    - The problem requires exploring multiple paths or states, and you may need to return to a previous state to try another path.
    - For example, depth first search algorithms (DFS) on trees and graphs
- **Expression Evaluation**
    - Stacks are commonly used to evaluate or convert expressions, particularly in postfix or infix form.
    - We won't cover these concepts too deeply within this course, but we encourage you to follow your curiosity!
    - For example, Reverse Polish Notation

Queues

Queues are a special type of list where elements are always added and removed in a certain order. Queues follow the First In, First Out (FIFO) principle, which means that the first element added to the queue will be the first element to be removed.

The term 'queue' comes from the British meaning of the word queue: a line of waiting people. Queue data structures follow the same logic as queues of real people! Imagine that each element in the queue is a person. New people add themselves to the end of the line and must wait until all of the people ahead of them have exited the line before they can exit the line. The person who has been waiting in line the longest is the first person to exit the line.

Practically speaking, this means that new elements are always appended to the end of a queue and removed from the beginning of the queue.



Source: via Key to Programming

In Python, we use the `deque` class to create a new Queue.

Example Usage:

```
# 1. Import the deque module
from collections import deque

# 2. Initialize a new deque object (a new queue)
queue = deque()

#3. Add a new element, 1, to the end (right side) of the queue
queue.append(1)

#4. Remove an element from the beginning (left side) of the queue
removed_elt = queue.popleft()
print(removed_elt) # Prints 1
```

The above example can be broken down as follows:

1. The `deque` is a **class** or custom data type that is part of the `collections` library in Python. A library is a collection of pre-written code that we can import into our own programs. Importing gives us access to use the classes, functions, etc. defined inside the library.
2. Create a new instance of the `deque` class by assigning a variable to `deque` followed by parentheses. If you are unfamiliar with object oriented programming (OOP), this just means we create a new queue.

3. The `deque` class has a method `append()` that adds a new element to the end of the queue. `append()` has one parameter, the value we want to append to the queue. We use dot notation to append the integer `1` to our `deque` instance `queue`.
4. The `deque` class also has a method `popleft()` that removes and returns the element at the beginning of the queue. Similar to using list or string methods, we use dot notation to remove the element at the beginning of our `deque` instance `queue` (in this case `1` since there is only one element in the queue).

It is equally valid to add elements to the beginning of the queue and pop elements from the end. We can use the `appendleft()` and `pop()` elements to do this. The important thing is that we append and pop elements from *opposite ends* of the queue.

Example Usage:

```
# Import the deque module
from collections import deque

# Initialize a new deque object (a new queue)
queue = deque()

#3. Add a new element, 1, to the left side of the queue
queue.appendleft(1)

#4. Remove an element from the right side queue
removed_elt = queue.pop()
print(removed_elt) # Prints 1
```

It is also possible to model a queue using a normal list. However, we generally prefer to use `deque` as it optimizes the speed of removing nodes from the beginning of the queue and adding nodes to the end of the queue (under the hood, `deque` is implemented using a linked list which we will learn more about in future units!).

When to Use a Queue

What are some common signs a problem could be solved using a queue?

- **Processing Items**
  - Queues are commonly used to process items in a sequence in the order they arrive
  - For example, Number of Recent Calls (requires object oriented programming knowledge)
- **Breadth First Search Algorithm**
  - We won't cover BFS algorithms in depth until Unit 9 and beyond, but queues are almost always used to perform Breadth First Search (BFS) algorithms on trees and queuester
  - BFS is used to explore trees and graphs level by level (or in order of proximity). Queues help keep track of what part of the data structure should be explored next.

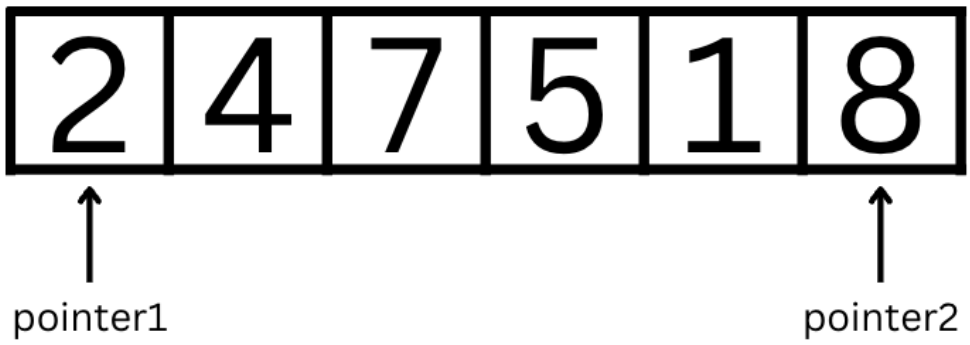
    - An example of BFS, is Level Order Traversal of a Binary Tree
- **Round-Robin Scheduling**
    - Queues are often used when tasks need to be processed in a cyclic order, particularly when time-sharing is involved.
    - For example, the Task Scheduler problem.

Two Pointer

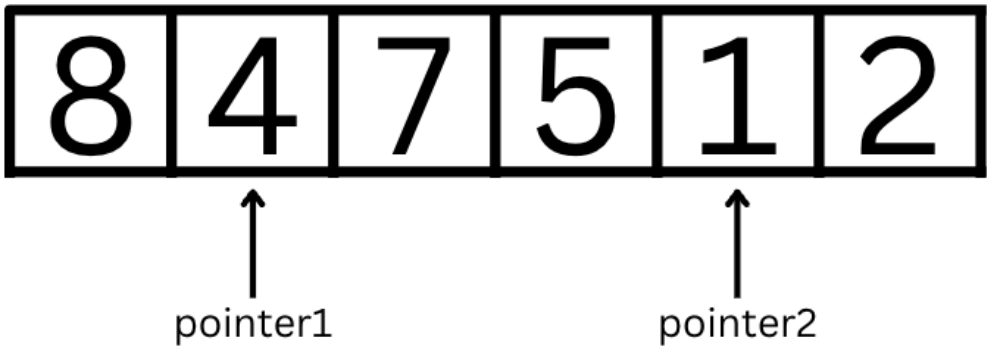
The **two-pointer approach** is a common technique in which we initialize two pointer variables to track different indices or places in a list or string and move them to new indices based on certain conditions.

Opposite Direction Pointers

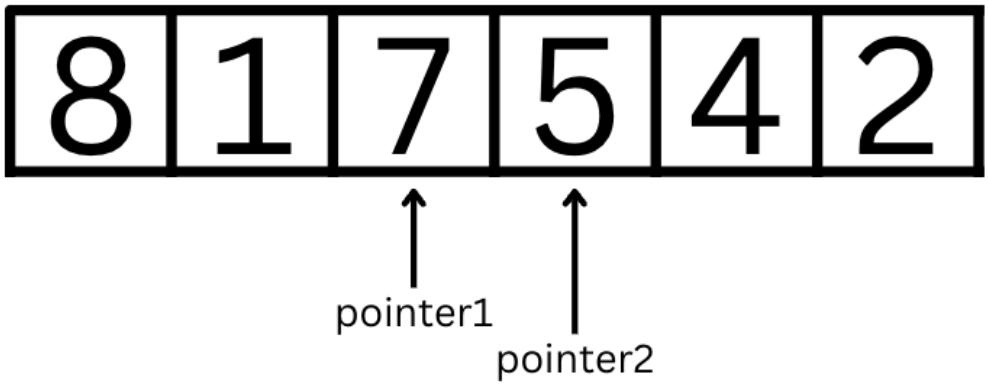
In the most common variation of the two-pointer approach, we initialize one variable to point at the beginning of a list and a second variable/pointer to point at the end of list. We then shift the pointers to move inwards through the list towards each other, until our problem is solved or the pointers reach the opposite ends of the list.



In the above case, `pointer1 = 0` and `pointer2 = 5`. Let's say we wanted to **reverse the integer list**. The two pointers would allow us to swap the values at each pointer and then move inwards to continue swapping. The next iteration would have `pointer1 = 1` and `pointer2 = 4` to swap those values.

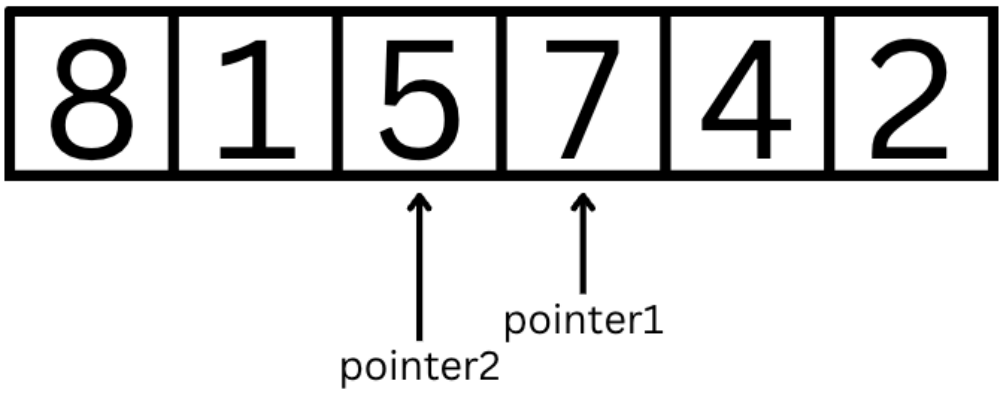


After those values are swapped, the next iteration would have `pointer1 = 2` and `pointer2 = 3` to swap those values.



Finally, we know to stop once `pointer1 == pointer2` (*they are at the same index, mostly when the length of the list or string is odd*) or when `pointer1 > pointer2` (*the pointers have intersected, and pointer1 is past pointer2*).

The end result has the integer list completely reversed:



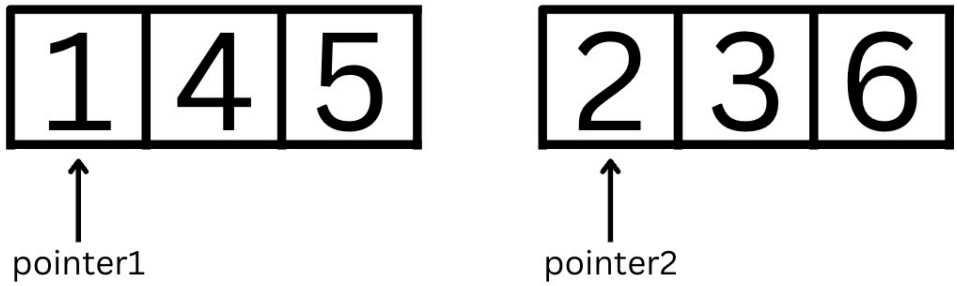
Example Usage:

```
left_pointer = 0
right_pointer = len(word) - 1
while left_pointer < right_pointer:
    pass
    left_pointer += 1
    right_pointer -= 1
```

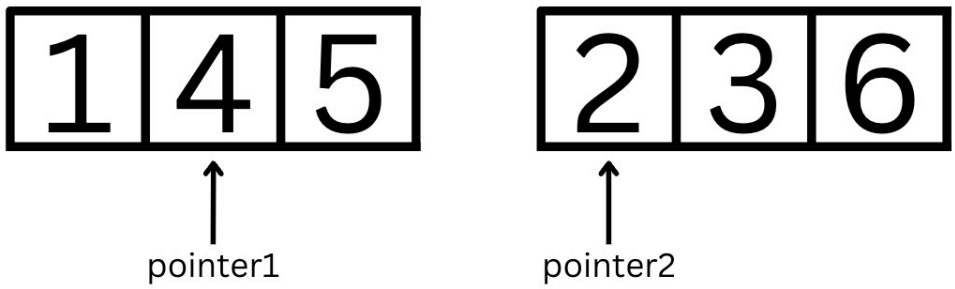
Same Direction Pointers

Another common variation of the two-pointer approach is to point one pointer at the beginning of one string or list and a second pointer at the beginning of a second string or list, then increment each pointer conditionally to solve a problem. This allows us to keep track of values in two lists at the same time.

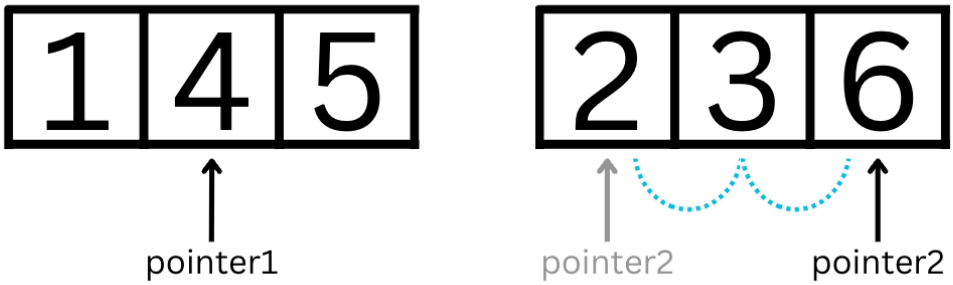
For example, let's say we want to merge two *sorted* lists. Since each list is sorted, we can start off by comparing the values at the beginning of the list with both `pointer1 = 0` and `pointer2 = 0`:



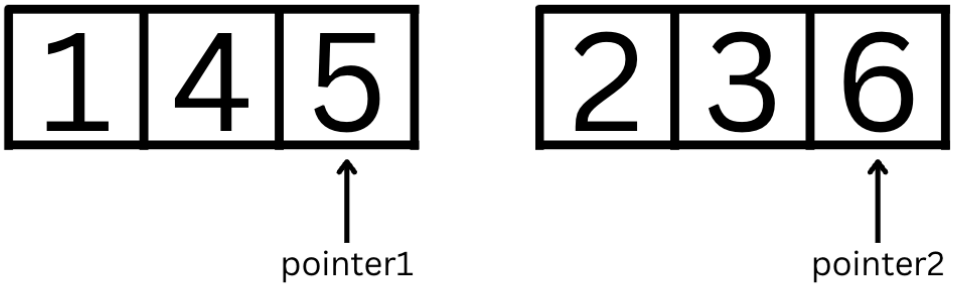
Then, we would increment the pointer of the value that was less and put into the merged list. Currently, the merged list would have `[1]` and `pointer1 = 1` and `pointer2 = 0`.



Comparing the pointer values, `pointer2` is incremented twice in a row and the merged list would have `[1,2,3]`



Now, with `pointer1 = 1` and `pointer2 = 2` we compare values again and `pointer1` is incremented.



After the last comparison, we have our sorted merged list as `[1,2,3,4,5,6]`. Unlike the previous variation with the 2 pointers pointing to the same list, we know to stop because both pointers have reached the end of their respective lists.

Example Usage:

```
    nums1_pointer = 0
    nums2_pointer = 0

    while nums1_pointer < len(nums1) and nums2_pointer < len(nums2):
        # <if conditional>
            # <operation>
            nums1_pointer += 1
        else:
            # <operation>
            nums2_pointer += 1
```

When to use the Two-Pointer Technique

Often, knowing when to apply a particular data structure or algorithm comes from practice and experience. However, there are some indicators we can look out for that may hint a problem can be solved with a two-pointer approach.

- **Data Structure**

    - This technique is most commonly applied to strings, arrays, and linked lists (covered in future units!)
- **Reducing Nested Loops**
    - This technique is often applied to improved solutions that can be 'brute forced' using multiple for loops.
    - If you have an approach that uses multiple for loops, you may want to consider wether it's possible to eliminate repetititve calculations through strategic movements of mulitple pointers.
- **Searching for Pairs or Triplets**
    - The two-pointer technique is commonly used to find pairs or triplets within a sorted array that satisfy certain conditions.
    - A classic example of this is the Two Sum Problem.
- **In Place Operations**
    - The two-pointer technique is often used when performing operations on a sequence in place, without creating an extra data structure to hold the result.
    - A popular example of this is Removing Duplicates from a Sorted Array.
- **Comparing Opposite Ends of a Sequence**
    - If the problem involves comparing or processing elements from both ends of a sequence, that's often a indicator the Opposite Direction Pointers technique can be used.
    - A popular example of this is Valid Palindrome
- **Partitioning Problems**
    - If the problem requires dividing or partitioning the data into multiple parts
    - For example, Partitioning Array According to a Given Pivot

Helper Functions

If you find your functions getting too long or performing lots of different tasks, it might be a good indicator that you should add a **helper function**. Helper functions are functions we write to implement a subtask of our primary task.

We can then call our function inside of our main function, to keep our main function shorter and easy to read.

Example Usage:

```
# Example: Calculating a Bill Total

# Helper Function: Calculate the price of one item
def calculate_price(item_price, quantity):
    return item_price * quantity

# Primary Function: Calculate total cost of bill
def calculate_total(bill):
    total = o
    for item_price, quantity in bill.items():
        # Call helper function
        total += calculate_price(item_price, quantity)
    return total
```

Creating helper functions helps us follow the Single Responsibility Principle (SRP) which says that a function should perform only a single task or action. SRP helps us create clean, readable, and maintainable code!

Python Syntax

Unpacking

Unpacking is a method of assigning multiple variables at once, commonly used with the 2-pointer approach.

To assign multiple variables at once, we can use the following syntax to assign the value `1` to `a` and the value `2` to `b`:

```
a, b = 1, 2
```

If there is an incorrect number of variables for the values given, a `ValueError` will be thrown. This can be applied to when swapping values (*as seen in the **Beginning and End** 2-Pointer Approach*):

```
pointer_one, pointer_two = pointer_two, pointer_one
```

Note that this swaps the **index** each pointer points to, and not the **value**. To swap values in a list, we would use:

```
nums[pointer_one], nums[pointer_two] = nums[pointer_two], nums[pointer_one]
```

Unpacking is not limited to one data type. It also works with strings, and we can even assign multiple values from a list of strings:

```
inventory = [["apples", 3], ["carrots", 5]]
[[item, quantity], [item2, quantity2]] = inventory
# [item, quantity] << ["apples", 3]
# [item2, quantity2] << ["carrots", 5]
```

Inner Functions

Inner/Nested Functions

An Inner Function, also called a nested function, is a function defined inside of another function.

Example Usage:

```
# Outer function
def print_sum(x, y):

    # Inner function
    def get_sum():
        return x + y

    total = get_sum()
    print(f"{x} + {y} is {total}")

print_sum(1, 2) # Prints: 1 + 2 is 3
```

The inner function has access to any variables and parameters defined by the outer function. In the example above, the inner function `get_sum()` has access to `print_sum()`'s parameters even though they are not passed in to `get_sum()`.

Inner functions are only available within the scope of the outer function. They are commonly used to define helper functions that will only be called by the main function, and won't be used anywhere else in the program.

Example Usage:

```
def total_sum_of_digits(numbers):
    # Inner function to calculate the sum of digits of a single number
    def sum_of_digits(n):
        total = 0
        while n > 0:
            total += n % 10
            n //= 10
        return total

    # Main function logic
    total_sum = 0
    for number in numbers:
        total_sum += sum_of_digits(number)

    return total_sum
```

The function `total_sum_of_digits()` above accepts a list of integers, and returns the sum of digits of *all* numbers in the list. The logic of the helper function `sum_of_digits()` which calculates the sum of digits for a single number is specific to the main function and unlikely to be used in the larger program. Encapsulating `sum_of_digits()` within `total_sum_of_digits()` as an inner function can help keep code organized and modular.

Advanced Concepts

There are no extra advanced syntax or concepts for this unit! 😎 The advanced component comes from the difficulty of the problems!

Bonus Syntax & Concepts

The following concepts are nice to know and may improve your code readability or help you solve certain problems more easily and efficiently. However, they are not *required* to solve any of the problems in this unit. These concepts are **not in scope for either the Standard or Advanced Unit 3 assessments**, and you do not need to memorize them! Click on each concept to read more about how to use it.

Diving Deeper: Stacks vs Queues

Stacks and Queues are both special types of lists that maintain certain properties about insertion and removal order.

Problems solved with a stack are often implement a *recursive* solution, because under the hood, computers use a stack data structure to track the sequence of recursive function calls. When we initiate a series of recursive calls, we are essentially making a stack. Every time a recursive call is made, a new element - the function call and any arguments it is passed - is added on to the stack. The stack the computer uses to track function calls is referred to as the call stack. Recursion and the call stack will be covered more in depth in future units.

In contrast, problems solved using queues generally use an *iterative* solution involving a while loop.

Although stack problems are often solved recursively, they may always be solved iteratively.

Example Usage:

```
# Reverse a string using a iterative stack solution
def reverse_string_iterative(s):
    stack = []

    # Push all characters of the string onto the stack
    for char in s:
        stack.append(char)

    reversed_string = ""

    # Pop all characters from the stack and append to the result
    while stack:
        reversed_string += stack.pop()

    return reversed_string

# Reverse a string using a recursive stack solution
def reverse_string_recursive(s):
    # Base case: if the string is empty or has only one character
    if len(s) <= 1:
        return s

    # Recursive case: reverse the rest of the string and append the first character to the end
    # reverse_string_recursive(s[1:]) is added to call stack
    return reverse_string_recursive(s[1:]) + s[0]
```