Unit 1 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 as you solve the breakout problems for Unit 1. 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 1. In addition to the material below, you will be expected to know any required concepts from previous units.

Standard Concepts

> ‼ **This material is in scope for the Standard Unit 1 assessment.**

Built-In Functions

Print

`print(s)` Prints the message `s` to the console. Try it

- Accepts one parameter `s`: the message you would like to print

Example Usage:

```
# Example 1: Printing a string
print("Welcome to TIP102!") # Prints "Welcome to TIP102!" to the console

# Example 2: Printing an integer
print(100) # Prints 100 to the console

# Example 3: Printing a variable
s = "Welcome to CodePath!"
num = 7
print(s)   # Prints "Welcome to CodePath" to the console
print(num) # Prints 7 to the console

# Example 4: Printing a list
lst = ["TIP101", "TIP102", "TIP103"]
print(lst) # Prints ["TIP101", "TIP102", "TIP103"] to the console

# Example 5: Printing an expression
```

Length

`len(s)` Returns the length of a list or string. Try it

- Accepts one parameter `s`: the list or string we would like to get the length of
- Returns the number of items in a list or the number of characters in a string

Example Usage:

```
# Example 1: Getting the length of a list
lst = ['a', 'b', 'c']
lst_length = len(lst)
print(lst_length) # Output: 3

# Example 2: Getting the length of string
s = 'abcd'
s_length = len(s)
print(s_length) # Output: 4
```

Range

`range(start, stop, step)` Returns a sequence of numbers. Try it

- Accepts three parameters:
  - `start`: the first number in the sequence, this is an optional parameter. If we do not provide a `start`, the range will start from `0`.
  - `stop`: the last value in the sequence *exclusive*, meaning that the `stop` value is not actually included in the sequence. This is a *required* parameter
  - `step`: how much to increment each number in the sequence. If we do not provide a `step`, each successive number in the sequence will increment by 1.

Example Usage:

```
# Example 1: Just the stop value
range(10) # Evaluates to the sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

# Example 2: Start and stop value
range(1, 11) # Evaluates to the sequence: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

# Example 3: Start, stop, and step value
range(0, 30, 5) # Evaluates to the sequence: 0, 5, 10, 15, 20, 25
```

Im

Min

`min(x)` Returns the argument with the lowest value or the item with the lowest value in a list. Try it

```
# Example 1: Minimum item in a list
min([2, 3, 4, 1, 5]) # Evaluates to 1

# Example 2: Argument with minimum value
min(5, 2, 3) # Evaluates to 2

# Example 3: Smallest string
min(['a', 'b', 'c', 'aa']) # Evaluates to 'a'
```

Max

`max(x)` Returns the argument with the highest value or the item with the highest value in a list. Try it

```
# Example 1: Maximum item in a list
max([2, 3, 5, 1, 4]) # Evaluates to 5

# Example 2: Argument with maximum value
max(5, 2, 3) # Evaluates to 5

# Example 3: Maximum string
max(['a', 'b', 'c', 'aa']) # Evaluates to 'c'
```

List Methods & Syntax

Append Method

`lst.append(item)` Adds an item to the end of the list. Try it

- Accepts one parameter `item`: the item you would like to add to the list

Example Usage:

```
# Example 1: Add an integer to the list
lst = [1, 2, 3, 4]
lst.append(5)
print(lst) # Prints [1, 2, 3, 4, 5]
```

Sort Method

`lst.sort()` Sorts the list in ascending order. Try it

- Does not have any required parameters

```
# Example 1: List of integers
lst = [4, 2, 1, 3]
lst.sort()
print(lst) # Prints [1, 2, 3, 4]


# Example 2: List of strings
lst = ['b', 'a', 'd', 'c']
lst.sort()
print(lst) # Prints ['a', 'b', 'c', 'd']
```

String Methods & Syntax

Lower Method

`s.lower()` Returns `s` as a lowercase string.Try it

- Accepts no parameters
- Returns a lowercase string.

Example Usage:

```
# Example 1: Mixed case
s = 'Hello World!'
lowered = s.lower()
print(lowered) # Prints 'hello world!'

# Example 2: All uppercase
s = 'HELLO WORLD'
lowered = s.lower()
print(lowered) # Prints 'hello world'
```

Split Method

`s.split(seperator)` Splits the string into a list along whitespace or specified separator. Try it

- `s` is the string to split
- Accepts one parameter `separator`: The characters along which the string will be split. This is an *optional* parameter. By default, the string is split along any whitespace.
- Returns a list of the substrings of `s` split by the specified separator.

Example Usage:

```
# Example 1: Split along whitespace
  = 'Never gonna give you up'
split = s.split()
```

```
s = 'Never-gonna-let-you-down'
split = s.split("-")
print(split) # Prints ['Never', 'gonna', 'let', 'you', 'down']
```

Join Method

`s.join(x)` Turns an iterable `x` into a string using `s` as a separator between elements in `x`. Try it

- `s` is the separator placed between items in the specified iterable
- Accepts one parameter `x`: the iterable whose items will be joined together into a string. This is a *required* parameter.
- Returns a string of the items in `x` separated by `s`

Example Usage:

```
# Example 1: Join items in a list separated by space
lst = ['Never', 'gonna', 'run', 'around', 'and', 'desert', 'you']
joined = ' '.join(lst)
print(joined) # Prints 'Never gonna run around and desert you'

# Example 2: Join items in a list separated by dash
lst = ['Never', 'gonna', 'make', 'you', 'cry']
joined = '-'.join(lst)
print(joined) # Prints 'Never-gonna-make-you-cry'
```

Strip Method

`s.strip(characters)` Removes whitespace or specified `characters` from either end of the string. Try it

- `s` is the string to remove whitespace or characters from.
- Accepts one parameter `characters`: the characters/substring to remove from either end of `s`. This is an *optional* parameter. If no value is specified, whitespace will be removed from the beginning and end of `s`.
- Returns the string `s` with whitespace or characters removed.

Example Usage:

```
# Example 1: Strip whitespace
s = '       Never gonna say goodbye       '
stripped = s.strip()
print(stripped) # Prints 'Never gonna say goodbye'

# Example 2: Strip question marks
s = '????Never gonna tell a lie and hurt you?????'
 tripped = s.strip('?')
print(stripped) # Prints 'Never gonna tell a lie and hurt you'
```

Python Syntax

Variables

In Python, variables do not need to be declared using a key word. We simply create variables by giving them a name and assigning a value to it.

Variable names use snake case and should have underscores between words.

Example Usage:

```
# Example 1: Integer variable
var1 = 10

# Example 2: String Variable
var2 = "Codepath"

# Example 3: Boolean Variable
my_boolean = True

print(var1) # Prints 10
print(var2) # Prints 'Codepath'
print(my_boolean) # Prints True
```

Python variables are dynamically typed, meaning we do not need to specify the type of a variable when declaring it, and the variable type can change over time.

Example Usage:

```
# Example: Changing x from an int to a string
x = 10
print(x) # Prints 10

x = "Hello"
print(x) # Prints 'Hello'
```

Conditionals

In Python, conditionals are defined using the `if`, `elif` and `else` keywords. The body of the conditional will execute if the expression placed after `if` or `elif` evaluates to `True`.

Example Usage:

```
# Example 1: Simple if statement
x = 3
if x > 1:
  print("This line will execute!")

if x > 5:
  print("This line will not execute!")

# Output: 'This line will execute!'

# Example 2: If-Elif statement
```

```
if x > y:
  print("x is greater than y")
elif y > x:
  print("y is greater than x")

# Output: 'y is greater than x'

# Example 3: If-Elif-Else statement
x = 20
y = 20

if x > y:
  print("x is greater than y")
elif y > x:
  print("y is greater than x")
else:
  print("x and y are equal")

# Output: 'x and y are equal'
```

Functions

In Python, functions are defined using the `def` keyword.

Example Usage:

```
# Example: Function that prints Hello world!
def function_example():
  print("Hello world!")
```

Functions can be called by writing the function name followed by parentheses.

Example Usage:

```
# Example: Calling a function
function_example() # Prints 'Hello world!'
```

We can add **parameters** to our function by placing them inside the parentheses of the function header separated by commas.

Similarly, when we call the function we can pass arguments for each parameter to our function by placing them in parentheses separated by commas.

Example Usage:

```
# Example: Function with 2 parameters
def function_w_parameters(parameter1, parameter2):
  print("Parameter 1: ", parameter1)
  print("Parameter 2: ", parameter2)

function_w_parameters("Interview", "Prep")
# Output:
```

We can **return** a value using the `return` keyword. By default, a function returns `None`.

Example Usage:

```
# Example: Function that returns sum of two numbers
def sum(a, b):
  return a + b

# Example: Function without a return value
def sum_without_returning(a, b):
  a + b

return_val1 = sum(4, 2)
return_val2 = sum_without_returning(4, 2)
print(return_val1) # Output: 6
print(return_val2) # Output: None
```

Formatted Strings

Formatted strings or **f-strings** allow us to insert variable expressions into Python strings.

To create an f-string, place f before the quotation marks and add curly brackets around any variables we add to the string.

Example Usage:

```
# Example 1: Adding a variable to a string
name = "Michael"
print(f"Welcome to Codepath, {name}!") # Prints 'Welcome to CodePath, Michael!'

# Example 2: Adding an expression to a string
a = 3
b = 5
print(f"The sum of {a} and {b} is {a + b}") # Prints 'The sum of 3 and 5 is 8'
```

Remainder Division

The `%` **operator** is also known as the modulo operator and performs remainder division. `x % y` returns the remainder after dividing `x` by `y`. Try it

Example Usage:

```
print(5 % 2) # Prints 1 because 5 / 2 = 2 remainder 1

print(10 % 2) # Prints 0 because 10 / 2 = 5 remainder 0
```

For Loops

`or num in nums` Iterates through every number in nums. Try it

executing a function multiple times.

Example Usage:

```
# Example 1: Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)  # Outputs each fruit on a new line

# Example 2: Using a for loop with a range
for i in range(5):
    print(i)  # Prints numbers 0 to 4
```

While Loops

`while condition` Iterates while the condition evaluates to `True`

While loops in Python allow you to repeat a block of code so long as a condition evaluates to `True`. This is particularly useful when the number of iterations we want to perform is not predetermined and depends on dynamic conditions.

Example Usage:

```
# Example 1: Indeterminate number of iterations
# Prompting user for input until they enter a valid response
user_input = ""
while user_input != "quit":
    user_input = input("Enter a command (type 'quit' to exit): ")
print("You exited the loop.")

# Example 2: Processing data until a condition is met
i = 1
while i < 6:
    print(i)
    i += 1
```

With a while loop, we must create any variables that are part of the conditions ourselves. For example, `i` in Example 2 above. No variables are automatically created as part of the loop declaration.

Additionally, we must be careful to ensure our loop body makes progress towards the termination of the loop. Otherwise, our loop will iterate infinitely, eventually crashing our program and computer.

```
# Example: Infinite Loop
i = 1
while i < 6:
    print(i)
```

Because `i` never increments in the loop above, `i` never becomes greater than `6` and `i < 6` will evaluate to `True` forever.

omparing Strings and Lists

lists.

Similarities

- **Ordered sequences** Both strings and lists are ordered sequences of data.
- **Indexed by Integers** Both strings and lists can be indexed using integers (e.g. `lst[0]` or `s[0]`).
- **Sliceable** Both strings and lists can be sliced to access a subsection of the string/list (e.g. `lst[1:3]` or `s[1:3]`).
- **Iterable** Both strings and lists are iterable meaning we can loop over them using a for loop.
- **Length** We can use the `len()` function to get the length of either a string or list

Differences

- **Content Type** Both strings and lists are ordered sequences of data, but strings are more limited in the type of data they can contain.
  - Strings are an ordered sequence of character elements.
  - Lists are an ordered sequence of elements that can be of any type, including integers, strings, and other lists.
- **Mutability**
  - Strings are immutable, meaning they are not changeable. To update a string, we must create a new string.

```
s = 'Try'
s[0] = 'C' # Results in TypeError: 'str' object does not support item assignment
```

  - Lists are mutable. We can update the content of a list without creating a new list.

```
lst = ['T', 'r', 'y']
lst[0] = 'C'
print(lst) # Prints ['C', 'r', 'y']
```

Advanced Concepts

‼ This material is in scope for the Advanced Unit 1 assessment. The Standard material above is also in scope for Advanced assessments.

Built-in Functions

Enumerate

`enumerate(x)` takes an iterable such as a list, dictionary, or string, and adds a counter to the function.
s often used to loop over the indices and values of an iterable simultaneously.

  - `start`: the value to start the counter at. This is an *optional* parameter. If no default value is specified, the counter will start at 0.
- Returns a sequence of numbers coupled with values in given iterable

Example Usage:

```
# Example 1: Iterating over indices and characters in a string
my_string = 'code'
for index, char in enumerate(my_string):
  print(index, char)

# Prints:
# 0 c
# 1 o
# 2 d
# 3 e

# Example 2: Enumerate with start value specified
cereals = ['cheerios', 'fruity pebbles', 'cocoa puffs']
for count, cereal in enumerate(cereals, start=1):
  print(count, cereal)

# Prints:
# 1 cheerios
# 2 fruity pebbles
# 3 cocoa puffs
```

Zip

`zip(x, y)` takes two or more iterables and returns a sequence of tuples where each the first item from each iterable forms the first tuple in the sequence, the second item from each iterable forms a second tuple, and so on. It continues until it has placed all elements in the shortest iterable into a function. Try it

Zip is useful for iterating over multiple iterables in parallel or for combining data from separate iterables.

- Accepts two or more parameters:
  - `x`: an iterable object such as a list, dictionary, or string.
  - `y`: an iterable object such as a list, dictionary, or string.
  - Optionally accepts additional iterables to zip with `x` and `y`
- Returns a sequence of tuples pairing `x[i]` with `y[i]` where `0 <= i <= min(len(x), len(y))`

Example Usage:

```
# Example 1: Zipping Two Lists
```

```
print(list(zipped)) # Prints [('Alice', 25), ('Bob', 30), ('Charlie', 35)]

# Example 2: Zipping Lists of Different Lengths
names = ['Alice', 'Bob', 'Charlie', 'David']
ages = [25, 30, 35]
zipped = zip(names, ages)
print(list(zipped)) # Prints [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

Python Syntax

Nested Lists

Lists and other data structures used to store multiple items in a single variable can be **nested**. This means that lists can store other lists.

The container list is referred to as the **outer list**, while each list nested inside is referred to as an **inner list**.

Example Usage:

```
# Simple List
singers = ["Sabrina Carpenter", "FKA Twigs", "Elliot Smith"]

# Nested List
albums = [
            ["Sabrina Carpenter", "Short n' Sweet"],
            ["FKA Twigs", "Magdalene"],
            ["Elliot Smith", "Either/Or"]]

# Nested Lists Where Inner Lists Have Unequal Length
numbers = [
            [1],
            [1, 2],
            [1, 2, 3]]

# Triply Nested List
water_levels = ["Shallow", ["Deep", ["Deeper"]]]
```

You may also hear lists referred to by their dimensions. A simple list like `singers` in the above example may be referred to as a **1D** or 1-Dimensional list. A list of lists like `albums` or `numbers` may be referred to as a **2D list**, and so on.

Nested lists where each inner list has the same length are often referred to as **matrices**.

Example Usage:

```
matrix = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]
```

› can access and modify each inner list using normal list indexing syntax.

```
albums = [
            ["Sabrina Carpenter", "Short n' Sweet"],
            ["FKA Twigs", "Magdalene"],
            ["Elliot Smith", "Either/Or"]]

first_album = albums[0]
print(first_album) # Output: ["Sabrina Carpenter", "Short n' Sweet"]

# Example 2: Updating a Row in a Matrix
matrix = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]

matrix[2] = [100, 200, 300]
print(matrix) # Output: [[1, 2, 3], [4, 5, 6], [100, 200, 300]]
```

We can use **multiple indices** to access and modify elements within the inner lists.

Example Usage:

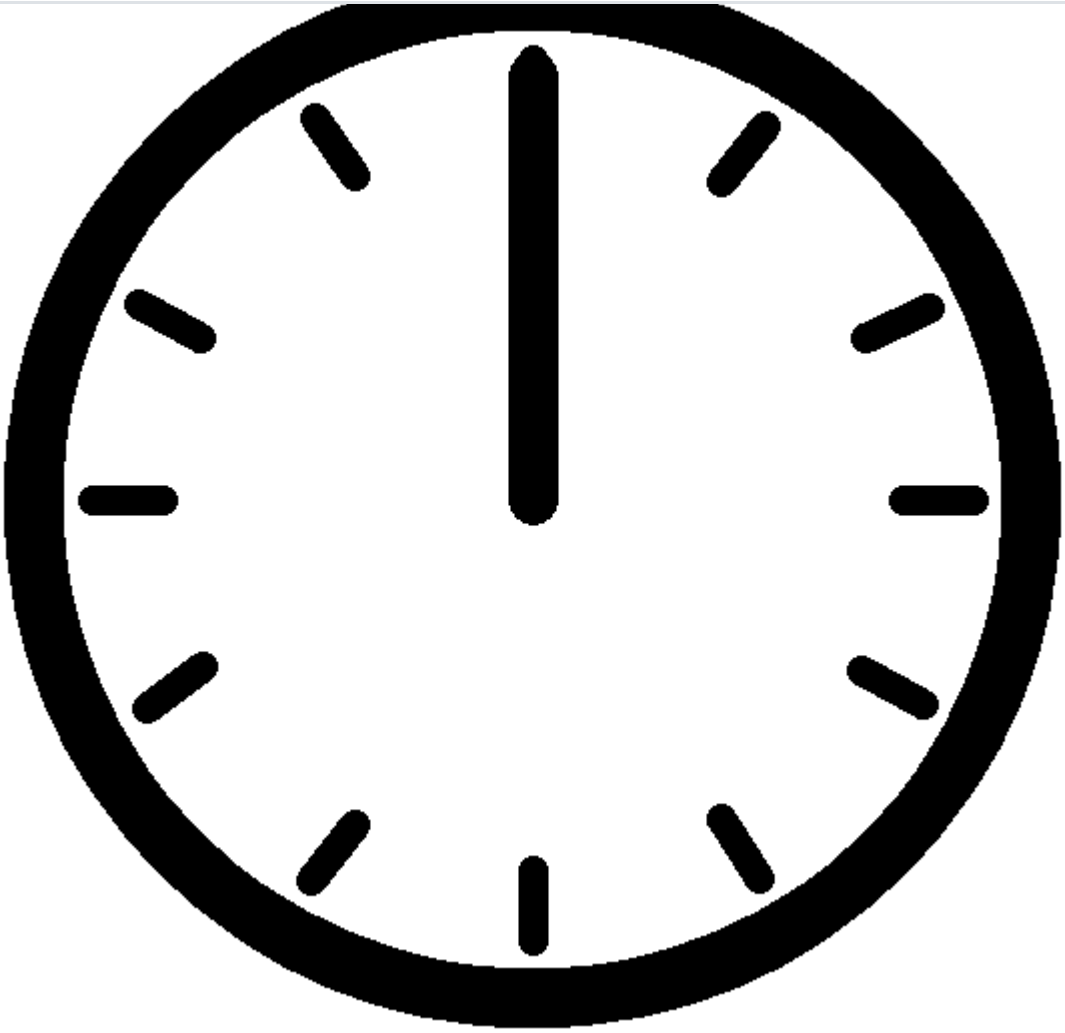
```
# Example 1: Retrieving a Singer
albums = [
            ["Sabrina Carpenter", "Short n' Sweet"],
            ["FKA Twigs", "Magdalene"],
            ["Elliot Smith", "Either/Or"]]

fka_twigs = albums[1][0]
print(fka_twigs) # Output: FKA Twigs

# Example 2: Updating a cell in a matrix
matrix = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]]
matrix[1][1] = "Surprise!"
print(matrix) # Output: [[1, 2, 3], [4, 'Suprise!', 6], [7, 8, 9]]
```

Nested Loops

Just as a nested list means a list within a list, a **nested loop** means a loop within a loop! They allow us to perform repeated actions within repeated actions.

When we nest loops, the inner loop will run completely every time the outer loop runs once.

Imagine the outer loop as the short hour hand of an analog clock, and the inner loop as the longer minute hand. The minute hand has to complete a full rotation around the clock before the hour hand increments once.



Example Usage:

```
for i in range(1, 4):
    print("Outer loop incremented")
    for j in range(1, 4):
        print(f"i = {i}, j = {j}")

# Output:
# i = 1, j = 1
# i = 1, j = 2
# i = 1, j = 3
# i = 2, j = 1
# i = 2, j = 2
# i = 2, j = 3
# i = 3, j = 1
# i = 3, j = 2
# i = 3, j = 3
```

Nested loops are often used to iterate over nested lists and matrices.

Example Usage:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
```

```
for row in matrix:
    for item in row:
        print(item, " ")
    print()  # Print a new line after each row

# Output:
# 1 2 3
# 4 5 6
# 7 8 9
```

A nested for loop is the most common example of a nested loop, but we can also nest other types of loops. We can nest while loops together or nest while loops within for loops and vice versa.

Example Usage:

```
numbers = [3, 5, 2]

# Outer for loop iterates over each number in the list
for num in numbers:
    print(f"Counting down from {num}:")

    # Inner while loop counts down from the current number to 0
    while num >= 0:
        print(num)
        num -= 1  # Decrement the number by 1 each time

    print("---")  # Separator to indicate moving to the next number

# Output:
# Counting down from 3:
# 3
# 2
# 1
# 0
# ---
# Counting down from 5:
# 5
# 4
# 3
# 2
# 1
# 0
# ---
# Counting down from 2:
# 2
# 1
# 0
# ---
```

⚠️ While nested loops are powerful, they do have some drawbacks:

- **Performance:** Nested loops can significantly increase the time it takes your code to run, especially when used on large inputs or when deeply nested.

loops deeply.

List Comprehensions

`result_list = [expression for element in lst]` evaluates an expression on each element in a list and stores the result of each evaluation in `result_list`. Try It

List comprehensions are a shorthand syntax for creating a new list using values of an existing list.

For example, say you want to double each value in a list. With a traditional for loop, you would to:

1. Initialize a new list to hold the list
2. Initialize a for loop
3. Double each element and append it

```
nums = [1, 2, 3, 4, 5]
doubled = []

for num in nums:
    doubled.append(num * 2)

print(doubled) # Output: [2, 4, 6, 8, 10]
```

A list comprehension reduces the above three steps to a single line of code.

```
nums = [1, 2, 3, 4, 5]
doubled = [value * 2 for value in nums]
print(doubled) # Output: [2, 4, 6, 8, 10]
```

We can also add a condition to our list comprehension.

`result_list = [expression for element in lst if condition]` evaluates an expression on each element in a list if the condition is true and stores the result of each evaluation in `result_list`

For example, say we wanted to find strings with length greater than `5`. Without a list comprehension we would have the following code:

```
words = ["I", "Love", "Codepath!"]
result = []

for word in words:
    if len(word) > 5:
        result.append(word)

print(result) # Output: ['Codepath!']
```

With a list comprehension, we can condense our code to:

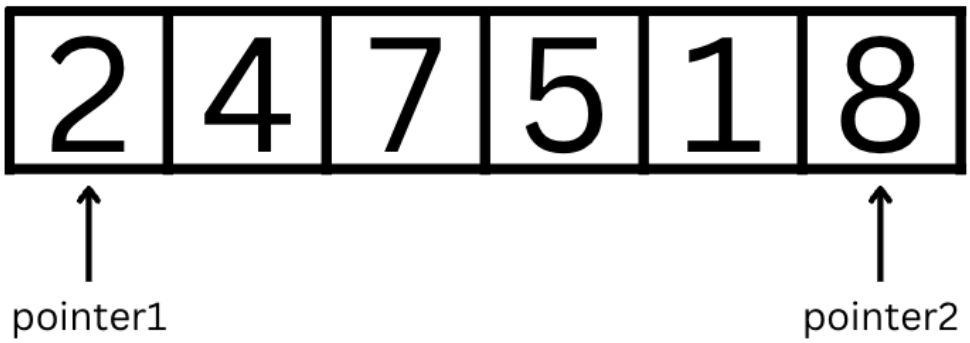
```
words = ["I", "Love", "Codepath!"]
 esult = [word for word in words if len(word) > 5]
,rint(result) # Output: ['Codepath!']
```

Two Pointer Technique

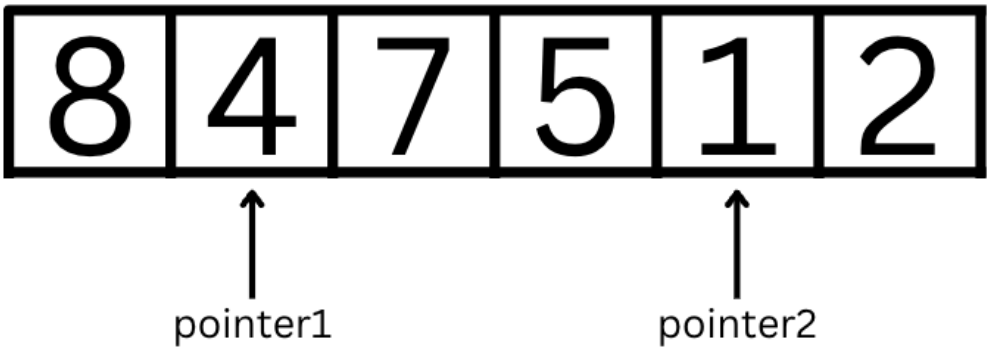
The **two-pointer approach** is a common technique in which we initialize two pointer variables to track different indices or places in a list or string and move them to new indices based on certain conditions.

Opposite Direction Pointers

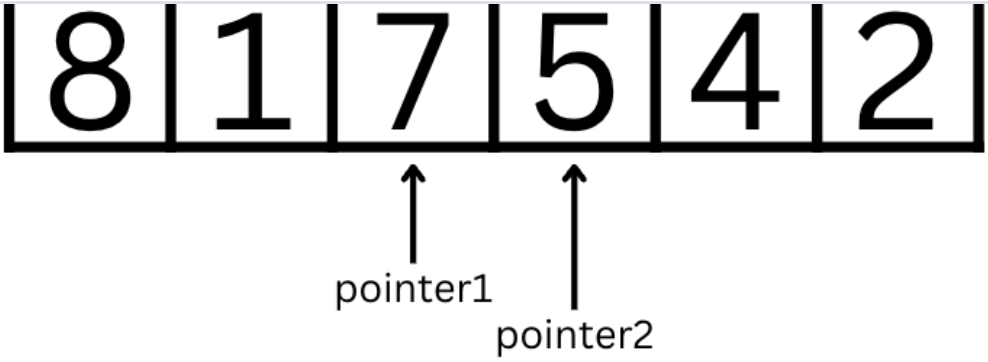
In the most common variation of the two-pointer approach, we initialize one variable to point at the beginning of a list and a second variable/pointer to point at the end of list. We then shift the pointers to move inwards through the list towards each other, until our problem is solved or the pointers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

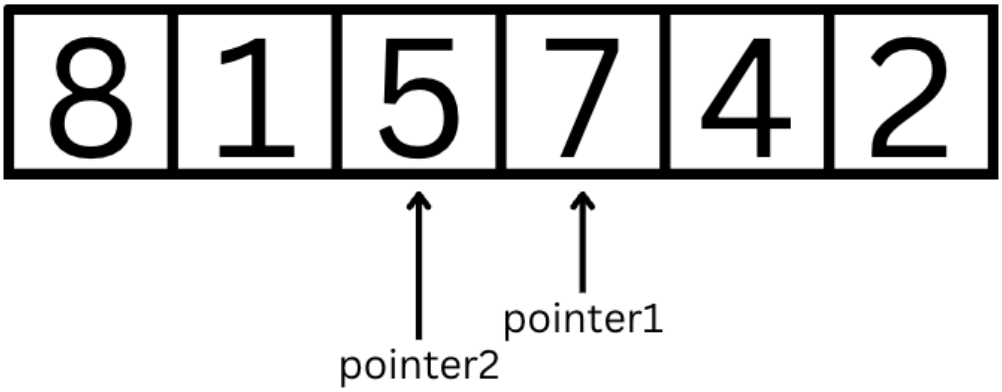


After those values are swapped, the next iteration would have `pointer1 = 2` and `pointer2 = 3` to swap those values.



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:



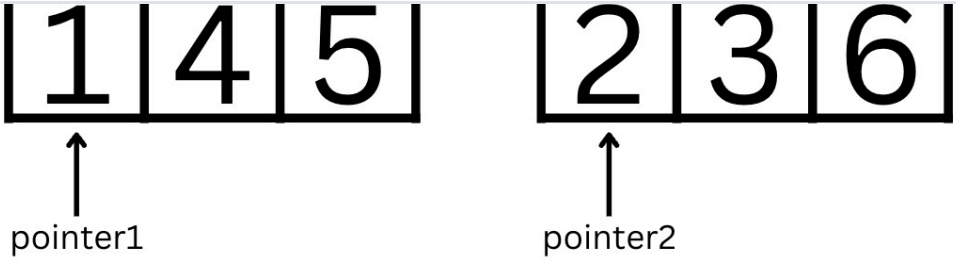
Example Usage:

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

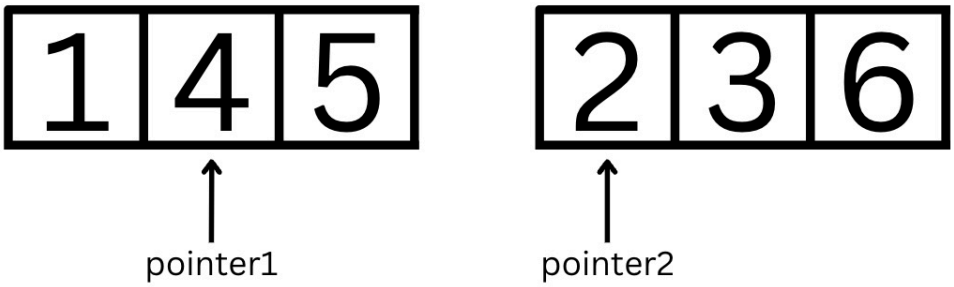
Same Direction Pointers

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.

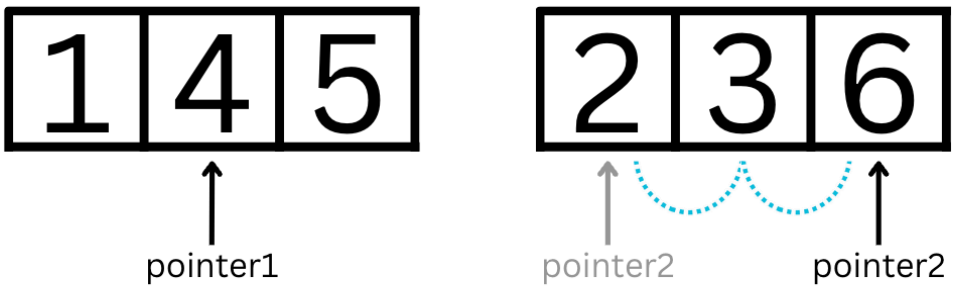
For example, let's say we want to merge two *sorted* lists. Since each list is sorted, we can start off by comparing the values at the beginning of the list with both `pointer1 = 0` and `pointer2 = 0`:



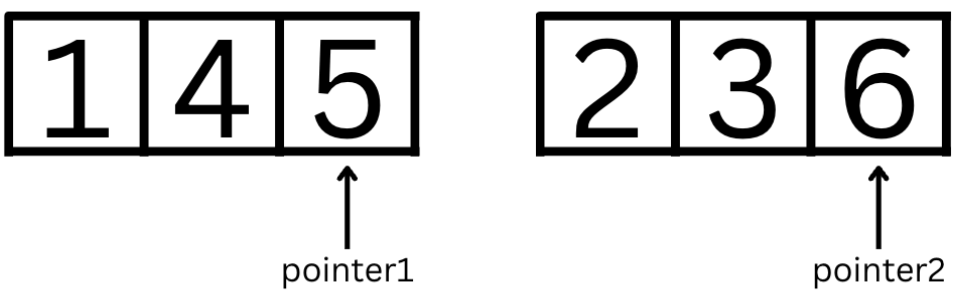
Then, we would increment the pointer of the value that was less and put into the merged list. Currently, the merged list would have `[1]` and `pointer1 = 1` and `pointer2 = 0`.



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.



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**

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 1 assessments**, and you do not need to memorize them! Click on each concept to read more about how to use it.

- `lst.insert(x, item)` Inserts `item` into list at index `x`
- `lst.remove(item)` Removes `item` from list
- `lst.pop(x)` Removes element at index `x` from list
- `lst.copy()` Creates a copy of a list
- `abs(x)` Returns the absolute value of a number `x`.
- `s.isalpha()` Returns True if all characters in given string are alphabetic letters (a-z).
- `s.isalnum()` Returns True if all characters in given string are alphanumeric (a-z or 0-9).
- `s.find(x)` Returns start index of the first occurrence of substring `x` in a given string. Returns `-1` if `x` is not in the string.
- `s.count(x)` Returns the frequency of the substring `x` in the given string.
- Sets A useful data type that allow you to group multiple pieces of data together. Data in sets is unordered, unchangeable, and cannot contain duplicates.