1. First-class function vs. high-order function

### **First-Class Functions**

A programming language is said to have **first-class functions** if functions are treated as first-class citizens. This means functions can:

1. Be assigned to variables
2. Be passed as arguments to other functions
3. Be returned from other functions
4. Be stored in data structures (e.g., lists, dictionaries)

#### **Example of First-Class Functions**

def greet(name):

return f"Hello, {name}!"

# Assigning function to a variable

greeting = greet

print(greeting("Alice")) # Output: Hello, Alice!

# Passing function as an argument

def call\_function(func, value):

return func(value)

print(call\_function(greet, "Bob")) # Output: Hello, Bob!

### **Higher-Order Functions (HOFs)**

A **higher-order function** is a function that does at least one of the following:

1. Takes one or more functions as arguments
2. Returns a function as a result

#### **Example of Higher-Order Functions**

# Function that takes another function as an argument

def apply\_operation(func, x, y):

return func(x, y)

def add(a, b):

return a + b

def multiply(a, b):

return a \* b

print(apply\_operation(add, 5, 3)) # Output: 8

print(apply\_operation(multiply, 5, 3)) # Output: 15

# Function that returns another function

def power(exponent):

def exponentiate(base):

return base \*\* exponent

return exponentiate

square = power(2)

cube = power(3)

print(square(4)) # Output: 16

print(cube(3)) # Output: 27

### **Common Built-in Higher-Order Functions in Python**

**map()** – Applies a function to all elements in an iterable  
  
numbers = [1, 2, 3, 4]

squared = map(lambda x: x \*\* 2, numbers)

print(list(squared)) # Output: [1, 4, 9, 16]

**filter()** – Filters elements based on a condition  
  
numbers = [1, 2, 3, 4, 5]

evens = filter(lambda x: x % 2 == 0, numbers)

print(list(evens)) # Output: [2, 4]

**reduce()** (from functools) – Performs cumulative operations  
  
from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x \* y, numbers)

print(product) # Output: 24

2. Contains Duplicate  
 **Checking for Duplicates in Python**

When working with lists or other collections in Python, you should check whether a list contains duplicate values. Here are different ways to check for duplicates:

### **Method 1: Using set() (Fastest for Checking Existence)**

Since sets do not allow duplicates, converting a list to a set and comparing lengths can determine if duplicates exist.

def has\_duplicates(lst):

return len(lst) != len(set(lst))

numbers = [1, 2, 3, 4, 5, 2]

print(has\_duplicates(numbers)) # Output: True

**Time Complexity:** O(n)O(n)O(n) (on average) since set operations are generally O(1)O(1)O(1).

### **Method 2: Using collections.Counter (To Count Duplicates)**

If you need to check which elements are duplicated, Counter from the collections module is useful.

from collections import Counter

def find\_duplicates(lst):

counter = Counter(lst)

return [item for item, count in counter.items() if count > 1]

numbers = [1, 2, 3, 4, 5, 2, 3, 3]

print(find\_duplicates(numbers)) # Output: [2, 3]

**Time Complexity:** O(n)O(n)O(n)

### **Method 3: Using a Set While Iterating (Efficient for Large Lists)**

This method stops as soon as a duplicate is found.

def has\_duplicates(lst):

seen = set()

for num in lst:

if num in seen:

return True

seen.add(num)

return False

numbers = [1, 2, 3, 4, 5, 2]

print(has\_duplicates(numbers)) # Output: True

**Time Complexity:** O(n)O(n)O(n) (on average)

### **Method 4: Using Nested Loops (Inefficient for Large Lists)**

This brute-force method checks every pair of elements but is slow for large lists.

def has\_duplicates(lst):

for i in range(len(lst)):

for j in range(i + 1, len(lst)):

if lst[i] == lst[j]:

return True

return False

numbers = [1, 2, 3, 4, 5, 2]

print(has\_duplicates(numbers)) # Output: True

**Time Complexity:** O(n2)O(n^2)O(n2) (Not recommended for large lists)

### **Method 5: Using List Comprehension to Find Duplicates**

def find\_duplicates(lst):

return list(set([x for x in lst if lst.count(x) > 1]))

numbers = [1, 2, 3, 4, 5, 2, 3, 3]

print(find\_duplicates(numbers)) # Output: [2, 3]

**Time Complexity:** O(n2)O(n^2)O(n2) (because count(x) runs in O(n)O(n)O(n) for each element)

### **Best Approach?**

* **If only checking for duplicates:** set() method (O(n))
* **If finding duplicate elements:** Counter or set method (O(n))
* **For extensive lists:** Use set() while iterating to stop early

3. Valid Anagram

**Valid Anagram in Python**

An **anagram** is a word or phrase that is formed by rearranging the letters of another word. To check if two strings are anagrams, they must contain the same characters with the same frequency.

### **Method 1: Using sorted() (Simple & Readable)**

We can sort both strings and compare them. If they are anagrams, their sorted versions will be identical.

def is\_anagram(s: str, t: str) -> bool:

return sorted(s) == sorted(t)

print(is\_anagram("listen", "silent")) # Output: True

print(is\_anagram("hello", "world")) # Output: False

**Time Complexity:** O(nlog⁡n)O(n \log n)O(nlogn) (due to sorting)  
**Space Complexity:** O(n)O(n)O(n) (to store sorted versions)

### **Method 2: Using collections.Counter (Counting Frequencies)**

We can count the frequency of characters in both strings and compare them.

from collections import Counter

def is\_anagram(s: str, t: str) -> bool:

return Counter(s) == Counter(t)

print(is\_anagram("listen", "silent")) # Output: True

print(is\_anagram("hello", "world")) # Output: False

**Time Complexity:** O(n)O(n)O(n)  
**Space Complexity:** O(1)O(1)O(1) (only storing 26 letters for lowercase alphabets)

### **Method 3: Using a Fixed-Size Frequency Array (Efficient for Lowercase Letters)**

If we assume only lowercase English letters (a-z), we can use an array of size 26 to track character counts.

def is\_anagram(s: str, t: str) -> bool:

if len(s) != len(t):

return False

count = [0] \* 26 # 26 letters in English alphabet

for char in s:

count[ord(char) - ord('a')] += 1

for char in t:

count[ord(char) - ord('a')] -= 1

return all(c == 0 for c in count)

print(is\_anagram("listen", "silent")) # Output: True

print(is\_anagram("hello", "world")) # Output: False

**Time Complexity:** O(n)O(n)O(n)  
**Space Complexity:** O(1)O(1)O(1) (fixed-size array)

### **Edge Cases to Consider**

1. **Different lengths** → Return False
2. **Empty strings** → Return True (since both are trivially anagrams)
3. **Case sensitivity** → Convert to lowercase before checking (s.lower(), t.lower())
4. **Non-alphabetic characters** → If needed, preprocess the strings