1.Custom Data Type Conversion:

Write a function that takes a list of mixed data types (int, float, str, bool) and returns a dictionary with the count of each type.

def count\_data\_types(mixed\_list):

"""

Counts the occurrences of each data type (int, float, str, bool) in a list.

Args:

mixed\_list (list): List containing mixed data types.

Returns:

dict: Dictionary with data types as keys and their counts as values.

"""

type\_count = {"int": 0, "float": 0, "str": 0, "bool": 0}

for item in mixed\_list:

if isinstance(item, bool): # bool should be checked before int

type\_count["bool"] += 1

elif isinstance(item, int):

type\_count["int"] += 1

elif isinstance(item, float):

type\_count["float"] += 1

elif isinstance(item, str):

type\_count["str"] += 1

return type\_count

# Example usage

sample\_list = [1, 2.5, "hello", True, False, 3, 4.0, "world"]

print(count\_data\_types(sample\_list))

**Output:**

{'int': 2, 'float': 2, 'str': 2, 'bool': 2}

**Explanation:**

* We initialize a dictionary to store counts of each type.
* Use isinstance to check the type of each element.
* **Important:** bool must be checked before int because in Python bool is a subclass of int.

2.Dynamic Input Parser:

Implement a function that reads a line of input and parses it into appropriate Python data types (int, float, bool, str) based on its content.

type (int, float, bool, or str):

def parse\_input(user\_input):

"""

Parses a string input into its appropriate Python data type.

Rules:

- If input is "True" or "False" → bool

- If input is an integer → int

- If input is a float → float

- Otherwise → str

"""

# Check for boolean first

if user\_input.lower() == "true":

return True

elif user\_input.lower() == "false":

return False

# Check for integer

try:

int\_val = int(user\_input)

return int\_val

except ValueError:

pass

# Check for float

try:

float\_val = float(user\_input)

return float\_val

except ValueError:

pass

# Fallback to string

return user\_input

# Example usage

inputs = ["42", "3.14", "True", "hello"]

parsed = [parse\_input(i) for i in inputs]

print(parsed)

**Output:**

[42, 3.14, True, 'hello']

**Explanation:**

1. First, we handle bool because "True" and "False" could also be interpreted as strings.
2. Then we attempt int, and if it fails, we try float.
3. If both fail, we leave it as a str.

This function is robust for single-line inputs.

3.String Manipulation Suite:Given a string, write functions to:

Could you clarify what specific string operations you want in the suite? For example, common tasks include:

* Remove all vowels
* Count vowels/consonants/words/characters
* Reverse the string
* Check if it’s a palindrome
* Convert case (upper/lower/title)

If you want, I can create a **complete Python suite** with multiple functions handling all these operations in one place.

4. Remove all vowels

Count the frequency of each character

Return all substrings of length k

Custom Array Operations:

Implement a class that mimics Python’s list but only allows integers. Add methods for append, remove, pop, and slicing.

class StringUtils:

def \_\_init\_\_(self, text):

self.text = text

def remove\_vowels(self):

vowels = "aeiouAEIOU"

return ''.join(c for c in self.text if c not in vowels)

def char\_frequency(self):

freq = {}

for c in self.text:

freq[c] = freq.get(c, 0) + 1

return freq

def substrings\_of\_length(self, k):

return [self.text[i:i+k] for i in range(len(self.text) - k + 1)]

# Example usage

s = StringUtils("hello")

print("Without vowels:", s.remove\_vowels()) # hll

print("Char frequency:", s.char\_frequency()) # {'h':1, 'e':1, 'l':2, 'o':1}

print("Substrings of length 3:", s.substrings\_of\_length(3)) # ['hel','ell','llo']

4.2

class IntList:

def \_\_init\_\_(self):

self.data = []

def append(self, value):

if isinstance(value, int):

self.data.append(value)

else:

raise TypeError("Only integers are allowed")

def remove(self, value):

self.data.remove(value)

def pop(self, index=-1):

return self.data.pop(index)

def \_\_getitem\_\_(self, key):

return self.data[key]

def \_\_setitem\_\_(self, key, value):

if not isinstance(value, int):

raise TypeError("Only integers are allowed")

self.data[key] = value

def \_\_repr\_\_(self):

return str(self.data)

# Example usage

arr = IntList()

arr.append(10)

arr.append(20)

arr.append(30)

print(arr) # [10, 20, 30]

arr.pop()

print(arr) # [10, 20]

arr[1] = 50

print(arr) # [10, 50]

5.

Type Conversion Utility:

Write a function that takes a dictionary with string values and converts them to their most likely Python types (int, float, bool, str).

def convert\_dict\_types(input\_dict):

"""

Converts string values in a dictionary to appropriate Python types.

"""

def parse\_value(value):

# Check for boolean first

if isinstance(value, str):

if value.lower() == "true":

return True

elif value.lower() == "false":

return False

# Try int

try:

return int(value)

except ValueError:

pass

# Try float

try:

return float(value)

except ValueError:

pass

# Return as-is (already correct type or string)

return value

return {k: parse\_value(v) for k, v in input\_dict.items()}

# Example usage

data = {

"age": "25",

"height": "5.9",

"name": "Alice",

"is\_student": "True",

"city": "New York"

}

converted = convert\_dict\_types(data)

print(converted)

**Explanation:**

1. Checks for bool first (True/False).
2. Tries converting to int.
3. If that fails, tries float.
4. Otherwise, keeps it as a string.

6.

Prime Number Generator:

Write a generator function that yields all prime numbers up to n using nested loops and control statements.

def prime\_generator(n):

"""

Generator that yields prime numbers up to n.

"""

for num in range(2, n + 1):

is\_prime = True

for i in range(2, int(num \*\* 0.5) + 1): # check divisibility

if num % i == 0:

is\_prime = False

break # not prime, exit inner loop

if is\_prime:

yield num

# Example usage

for prime in prime\_generator(20):

print(prime, end=' ')

**Explanation:**

1. Loop through numbers from 2 to n.
2. For each number, check divisibility by numbers from 2 to sqrt(num) (efficient check).
3. If divisible, mark as not prime and break the loop.
4. If is\_prime remains True, yield the number.

7. Custom List Class with Advanced Features:

Implement a class that supports list operations (append, insert, remove, pop, reverse, sort) and also supports undo/redo of operations.

class AdvancedList:

def \_\_init\_\_(self):

self.data = []

self.history = [] # stores past states for undo

self.redo\_stack = [] # stores undone states for redo

# Internal method to save state before changes

def \_save\_state(self):

self.history.append(self.data.copy())

self.redo\_stack.clear() # clear redo after new operation

# List operations

def append(self, value):

self.\_save\_state()

self.data.append(value)

def insert(self, index, value):

self.\_save\_state()

self.data.insert(index, value)

def remove(self, value):

self.\_save\_state()

self.data.remove(value)

def pop(self, index=-1):

self.\_save\_state()

return self.data.pop(index)

def reverse(self):

self.\_save\_state()

self.data.reverse()

def sort(self):

self.\_save\_state()

self.data.sort()

# Undo the last operation

def undo(self):

if self.history:

self.redo\_stack.append(self.data.copy())

self.data = self.history.pop()

else:

print("Nothing to undo.")

# Redo the last undone operation

def redo(self):

if self.redo\_stack:

self.history.append(self.data.copy())

self.data = self.redo\_stack.pop()

else:

print("Nothing to redo.")

def \_\_getitem\_\_(self, key):

return self.data[key]

def \_\_setitem\_\_(self, key, value):

self.\_save\_state()

self.data[key] = value

def \_\_repr\_\_(self):

return str(self.data)

# Example usage

lst = AdvancedList()

lst.append(10)

lst.append(5)

lst.append(7)

print("Initial list:", lst)

lst.sort()

print("After sort:", lst)

lst.undo()

print("After undo:", lst)

lst.redo()

print("After redo:", lst)

lst.reverse()

print("After reverse:", lst)

lst.undo()

print("Undo reverse:", lst)

**Features included:**

* Standard list operations: append, insert, remove, pop, reverse, sort.
* Undo/redo of operations.
* Slicing and indexing via \_\_getitem\_\_ and \_\_setitem\_\_.
* Maintains history using copies of the list for safe undo/redo.

8. Set Operations:

Write a function that takes two lists and returns their union, intersection, and difference using set operations.

def set\_operations(list1, list2):

"""

Returns the union, intersection, and difference of two lists.

Args:

list1 (list): First list

list2 (list): Second list

Returns:

dict: Dictionary containing union, intersection, and difference

"""

set1 = set(list1)

set2 = set(list2)

result = {

"union": list(set1 | set2), # all unique elements

"intersection": list(set1 & set2), # common elements

"difference": list(set1 - set2) # elements in list1 not in list2

}

return result

# Example usage

list\_a = [1, 2, 3, 4]

list\_b = [3, 4, 5, 6]

result = set\_operations(list\_a, list\_b)

print("Union:", result["union"])

print("Intersection:", result["intersection"])

print("Difference:", result["difference"])

**Explanation:**

1. Convert both lists to set to remove duplicates and enable set operations.
2. | → union, & → intersection, - → difference.
3. Convert back to list to return results in list form.

9. Type Casting Engine:

Build a function that takes a list of values and a target type, and returns a new list with all values cast to the target type, handling exceptions gracefully.

def type\_cast\_list(values, target\_type):

"""

Casts each element in the list to the specified target type.

If casting fails for a value, it is skipped.

Args:

values (list): List of values to cast

target\_type (type): Target Python type (int, float, str, bool)

Returns:

list: List of successfully casted values

"""

casted\_values = []

for val in values:

try:

casted\_values.append(target\_type(val))

except (ValueError, TypeError):

# Skip values that cannot be cast

print(f"Warning: Could not cast {val} to {target\_type.\_\_name\_\_}")

return casted\_values

# Example usage

values = ["10", "20.5", "hello", True, "False"]

print("To int:", type\_cast\_list(values, int)) # skips "20.5", "hello", "False"

print("To float:", type\_cast\_list(values, float)) # skips "hello", "False"

print("To bool:", type\_cast\_list(values, bool)) # True for most non-empty/True values

print("To str:", type\_cast\_list(values, str)) # all converted to strings

**Explanation:**

1. Iterates through the list and attempts to cast each value to the target\_type.
2. Uses try-except to gracefully skip values that cannot be cast.
3. Returns a new list with successfully casted values.
4. Prints a warning for transparency when a cast fails.

9. List Comprehension Challenge:

Given a list of numbers, use list comprehensions to:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 1️⃣ Filter even numbers

even\_numbers = [x for x in numbers if x % 2 == 0]

# 2️⃣ Filter odd numbers

odd\_numbers = [x for x in numbers if x % 2 != 0]

# 3️⃣ Square of each number

squares = [x\*\*2 for x in numbers]

# 4️⃣ Cube of each number

cubes = [x\*\*3 for x in numbers]

# 5️⃣ Numbers greater than 5

greater\_than\_5 = [x for x in numbers if x > 5]

# 6️⃣ Convert numbers to strings

string\_numbers = [str(x) for x in numbers]

# 7️⃣ Create tuples of (number, square)

number\_square\_tuples = [(x, x\*\*2) for x in numbers]

# 8️⃣ Numbers divisible by 2 or 3

div\_by\_2\_or\_3 = [x for x in numbers if x % 2 == 0 or x % 3 == 0]

# 9️⃣ Flatten a nested list

nested\_list = [[1,2],[3,4],[5,6]]

flattened = [item for sublist in nested\_list for item in sublist]

# 1️⃣0️⃣ Conditional expression (square if even, cube if odd)

conditional\_calc = [x\*\*2 if x % 2 == 0 else x\*\*3 for x in numbers]

# Display results

print("Even:", even\_numbers)

print("Odd:", odd\_numbers)

print("Squares:", squares)

print("Cubes:", cubes)

print("Greater than 5:", greater\_than\_5)

print("As strings:", string\_numbers)

print("Number-Square tuples:", number\_square\_tuples)

print("Divisible by 2 or 3:", div\_by\_2\_or\_3)

print("Flattened list:", flattened)

print("Conditional calc:", conditional\_calc)

10. Find all even numbers

Square all odd numbers

Create a list of tuples (number, square) for numbers divisible by 3

Set Membership and Manipulation:

Implement a function that takes a string and returns a set of unique characters, then removes all vowels from the set.

**1️⃣ List Comprehension Challenge**

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Find all even numbers

even\_numbers = [x for x in numbers if x % 2 == 0]

# Square all odd numbers

squared\_odds = [x\*\*2 for x in numbers if x % 2 != 0]

# Create list of tuples (number, square) for numbers divisible by 3

div3\_tuples = [(x, x\*\*2) for x in numbers if x % 3 == 0]

print("Even numbers:", even\_numbers)

print("Squared odds:", squared\_odds)

print("Tuples divisible by 3:", div3\_tuples)

**Output:**

Even numbers: [2, 4, 6, 8, 10]

Squared odds: [1, 9, 25, 49, 81]

Tuples divisible by 3: [(3, 9), (6, 36), (9, 81)]

**2️⃣ Set Membership and Manipulation**

def unique\_chars\_without\_vowels(input\_string):

"""

Returns a set of unique characters from the string,

removing all vowels.

"""

vowels = set("aeiouAEIOU")

char\_set = set(input\_string) # get unique characters

return char\_set - vowels # remove vowels

# Example usage

s = "Hello World"

result = unique\_chars\_without\_vowels(s)

print("Unique characters without vowels:", result)

**Output:**

Unique characters without vowels: {'r', 'H', 'd', ' ', 'l', 'W'}

✅ **Explanation:**

1. **List Comprehensions:**
   * Filter, map, and create tuples in a single line.
2. **Set Operations:**
   * Convert string to a set to get unique characters.
   * Subtract the vowels set to remove them.

11. Function Decorator for Logging:

Write a decorator that logs the arguments and return value of any function it wraps.

Here’s a Python **decorator** that logs both the arguments and the return value of any function it wraps:

def log\_decorator(func):

"""

A decorator that logs function arguments and return value.

"""

def wrapper(\*args, \*\*kwargs):

print(f"Calling function '{func.\_\_name\_\_}' with args: {args}, kwargs: {kwargs}")

result = func(\*args, \*\*kwargs)

print(f"Function '{func.\_\_name\_\_}' returned: {result}")

return result

return wrapper

# Example usage

@log\_decorator

def add(a, b):

return a + b

@log\_decorator

def greet(name, greeting="Hello"):

return f"{greeting}, {name}!"

# Test calls

add(5, 7)

greet("Alice")

greet("Bob", greeting="Hi")

**Sample Output:**

Calling function 'add' with args: (5, 7), kwargs: {}

Function 'add' returned: 12

Calling function 'greet' with args: ('Alice',), kwargs: {}

Function 'greet' returned: Hello, Alice!

Calling function 'greet' with args: ('Bob',), kwargs: {'greeting': 'Hi'}

Function 'greet' returned: Hi, Bob!

**Explanation:**

1. \*args and \*\*kwargs allow the decorator to accept any number of positional and keyword arguments.
2. Logs the function name, its arguments, and the returned value.
3. Returns the original function result so it behaves normally.

11. Deep Copy vs Shallow Copy Demonstration:

Write code to demonstrate the difference between deep copy and shallow copy for nested lists.

Here’s a clear Python demonstration showing the difference between **shallow copy** and **deep copy** for nested lists:

import copy

# Original nested list

original = [[1, 2, 3], [4, 5, 6]]

# Shallow copy

shallow = copy.copy(original)

# Deep copy

deep = copy.deepcopy(original)

# Modify the nested list in the original

original[0][0] = 100

print("Original:", original)

print("Shallow Copy:", shallow)

print("Deep Copy:", deep)

**Sample Output:**

Original: [[100, 2, 3], [4, 5, 6]]

Shallow Copy: [[100, 2, 3], [4, 5, 6]] # Nested list references are shared

Deep Copy: [[1, 2, 3], [4, 5, 6]] # Nested list is independent

**Explanation:**

1. **Shallow Copy (copy.copy)**
   * Creates a new outer list, but **inner lists are still references** to the original inner lists.
   * Modifying a nested element affects both original and shallow copy.
2. **Deep Copy (copy.deepcopy)**
   * Creates a completely **independent copy**, including all nested objects.
   * Modifying the original does **not affect** the deep copy.

12. Custom Implementation of Built-in Functions:

Implement your own versions of max, min, and sum functions for lists without using the built-in ones.

Here’s a Python implementation of **custom max, min, and sum functions** for lists without using the built-in functions:

def custom\_max(lst):

if not lst:

raise ValueError("custom\_max() arg is an empty list")

max\_val = lst[0]

for item in lst[1:]:

if item > max\_val:

max\_val = item

return max\_val

def custom\_min(lst):

if not lst:

raise ValueError("custom\_min() arg is an empty list")

min\_val = lst[0]

for item in lst[1:]:

if item < min\_val:

min\_val = item

return min\_val

def custom\_sum(lst):

total = 0

for item in lst:

total += item

return total

# Example usage

numbers = [5, 3, 8, 2, 10]

print("Custom max:", custom\_max(numbers)) # 10

print("Custom min:", custom\_min(numbers)) # 2

print("Custom sum:", custom\_sum(numbers)) # 28

**Explanation:**

1. **custom\_max:** Start with the first element as max, iterate, and update if a bigger element is found.
2. **custom\_min:** Start with the first element as min, iterate, and update if a smaller element is found.
3. **custom\_sum:** Initialize total = 0 and add each element in the list.

13.

Advanced Control Flow:

Write a function that takes a list of numbers and returns a new list with only the numbers that are not skipped by a custom rule (e.g., skip multiples of 3, break if a negative number is found, continue otherwise).

Here’s a Python function demonstrating **advanced control flow** using for, continue, and break statements with a custom rule:

def filter\_numbers(numbers):

"""

Returns a new list with numbers filtered based on these rules:

- Skip multiples of 3 (continue)

- Stop processing if a negative number is found (break)

- Include all other numbers

"""

result = []

for num in numbers:

if num < 0:

break # stop processing on negative number

if num % 3 == 0:

continue # skip multiples of 3

result.append(num)

return result

# Example usage

nums = [1, 2, 3, 4, 5, 6, -1, 7, 8]

filtered = filter\_numbers(nums)

print("Filtered list:", filtered)

**Output:**

Filtered list: [1, 2, 4, 5]

**Explanation:**

1. **continue**: Skips multiples of 3, so they are not added to the result.
2. **break**: Stops the loop entirely if a negative number is found.
3. **Other numbers**: Added to the new list.