Assignment 13 (MMN 13)
Course: 20606 – Programming and Data Analysis in Python
Units Covered: 5–7
Assignment Topics: Lists, Equality, Exceptions

Assignment Weight: 2 Points
Number of Questions: 4
Semester: 2025B
Submission Deadline: May 1, 2025

Important Notes:

- You **must use the exact class/function names** as written.
- Include **internal documentation only** (in English) within the code, both general and inline, explaining the program logic and functionality — as shown in Unit 1.9 and the examples.
- Follow **PEP 8** style guidelines: https://peps.python.org/pep-0008/
- You may only add functions **explicitly permitted** in the assignment instructions.
- **Do not** use advanced data structures (e.g., dictionaries) or advanced topics like OOP or recursion unless explicitly allowed.
- Use **constants** where applicable.
- Follow correct **indentation** and use **clear and conventional variable names** in English.
- Match the **output format exactly** as requested in the questions — spelling, capitalization, spacing, etc.
- You may use the **tester code from the course website** to test your code — copy your function into the tester, run it, then **delete the tester code before submitting**.
- Submit the assignment only via the **online submission system** on the course website.
- **Save your confirmation number** after submitting — it's proof of submission.

Page 2 — Question 1 (20 points)

Write a function named `complement` that receives a list `lst` of natural numbers. The function should return a new list containing all the natural numbers **not present** in `lst`, from 1 up to the **maximum** value in `lst`.

- If `lst` is empty, return an empty list.

Example:

```
lst = [1, 4, 5, 7, 8, 9]
# Returned list: [2, 3, 6]
lst = [1, 2, 3, 4]
# Returned list: []
lst = []
# Returned list: []
```

Notes:

- `lst` is a list (Python `list` object) of natural numbers (positive integers).
- You may assume all elements in `lst` are unique.
- **Do not** use the `in` operator for checking membership.
- **Do use** `in` only inside `for` loops (e.g., `for x in lst` is allowed).
- You **may not** sort `lst` or use the `max` function to find the maximum
- The list lst **must not be sorted at any stage**, but you **may use** the built-in function max, which receives a list and returns its maximum value

---

Question 2 (30 points)

This question deals with **rightward circular shifts** of lists.

A **right shift of size `k`** means that each element in the list is moved `k` positions to the right. Elements that move past the end of the list wrap around to the beginning.

Part A:

Write a function named `shift_k_right` that receives a list `lst` and a shift amount `k`. It returns a new list representing the result of shifting `lst` to the right by `k` positions.

- If `k` is negative or greater than the length of the list, raise a `ValueError`.

Example:

```
lst = [4,  1-, 9, 7, 11,2]
k = 2
# Returned list: [11,2, 4, 1-, 9, 7]
```

Part B:

Write a function named `shift_right_size` that receives two lists `a` and `b`.

- The function returns the **smallest** value of `k` for which a **right circular shift** of list `a` by `k` positions yields list `b`.
- If no such `k` exists (i.e., the lists are not circular shifts of each other), return `None`.

Examples:

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5]
# Returned value: 0 (no shift needed)

a = [1, 2, 3, 4, 5]
b = [4, 5, 1, 2, 3]
# Returned value: 3

a = [4, 1-, 9, 7, 11,2]
b = [4, 1-, 7, 9, 11,2]
# Returned value: None (not a valid circular shift)
```

Important:

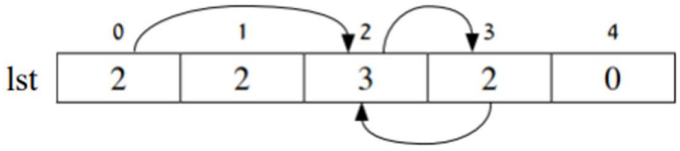
- You must **reuse the `shift_k_right` function from Part A** in this part.
- You may assume `a` and `b` are both Python `list` objects.
- If their lengths differ, immediately return `None`.

Question 3 (20 points)

This question involves analyzing lists using **index-based scanning**.

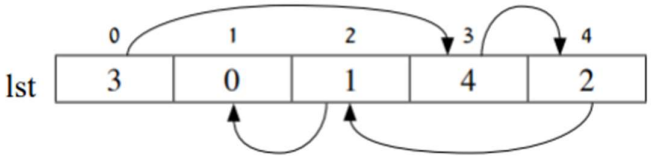
A **"value-guided scan"** is defined as follows:

- You begin at index 0.
- At each step, you move to the index specified by the value at the current index.
- This continues until one of two conditions is met:
    1. All the elements in the list have been visited **exactly once**.
    2. You reach an element with the value 0 (indicating the scan stops successfully).



If both conditions are met, the list is considered a **perfect list**.

Example for perfect list:



Note : A **perfect list** is one in which the "value-guided scan" satisfies both of the following conditions:

1. All cells in the list are visited exactly once.
2. The scan reaches a cell whose value is 0, at which point it terminates.

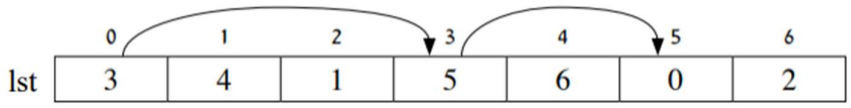
Example of a perfect list:

```
lst = [3, 0, 1, 4, 2]
```

This list is considered perfect because the scan passes through all cells and ends at the value 0.

Example of a non-perfect list:

1.



This list is **not** a perfect list because the scan terminates without visiting the cells at indices 1, 2, 4, and 6.

Write a function named **`is_perfect`** that receives a list **`lst`** of integers as a parameter.

- If the list is perfect, the function returns **`True`**. Otherwise, it returns **`False`.**
- If an out-of-range index is accessed during scanning, raise an **`IndexError`.**
- If an element in the list is not an integer, raise a **`TypeError`** (no need to handle the exception — just let it occur).

You may assume the parameter **`lst`** is a valid list (i.e. a variable of type **`list`**).

- If the list is empty, return `True`. There is no need to preserve the values of the list's elements.

  Important: Make sure to **avoid infinite loops**.

Special cases:

- If a move attempts to access an index out of bounds, raise an **`IndexError`**.
- If the list contains non-integer values, raise a **`TypeError`.**
- If the list is empty, return **`True`.**

More examples:

Let's say the list is:

```
lst = [2, 3, 2, 3, 0]
```

- Index path: 0 → 2 → 2 → 2 → ...
- Loop detected: Not all elements visited → Not perfect → return `False`

Now this list:

```
lst = [1, 2, 3, 4, 0]
```

- Index path: 0 → 1 → 2 → 3 → 4 → 0
- All elements visited exactly once, ends at 0 → Perfect → return `True`

Note:

- Avoid infinite loops — ensure each index is only visited once.
- You may assume `lst` is a Python `list`.

Question 4 (30 points)

This question focuses on **identity matrices** and submatrix operations.

An **identity matrix** of size `n` is a square matrix where all the elements on the main diagonal are `1`, and all other elements are `0`.

For example, a 5×5 identity matrix:

```
mat1 = [
    [1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1]
]
```

A **non-square matrix** (invalid for identity checks!):

```
mat2 = [
    [1, 0, 0],
    [0, 1, 0],
    [0, 0, 1, 0]  # Too many elements → not square
]
```

Part A:

Write a function named `identity_matrix` that receives a 2D list `mat` and returns:

- `True` if it is an identity matrix,
- `False` otherwise.

If any element is **not an integer**, raise a **`TypeError`.**
If the matrix is not square (number of rows ≠ number of columns), raise an **`IndexError`.**

Part B:

Write a function named `create_sub_matrix` that receives:

- A 2D square list `mat`
- An **odd** positive integer `size` that is **not larger than** the number of rows in `mat`

The function should return a square **submatrix** of dimension `size × size` **centered** in the original matrix.

Raise `IndexError` if not all rows are the same length.
You may assume `mat` is a 2D list and `size` is a valid odd number.

Example:

```
mat = [
    [1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [1, 0, 0, 0, 1]
]
```


```
]
```

Part C:

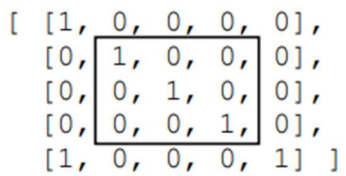
Write a function named `max_identity_matrix` that receives a 2D list `mat`.
It finds the **largest identity matrix** (centered in `mat`) and returns its size.

- Check progressively smaller odd-sized centered submatrices.
- If no identity matrix is found, return `0`.
- Raise `IndexError` if not all rows are of equal length.
- Raise `TypeError` if any value is not an integer.

Examples(Part C):

Input:

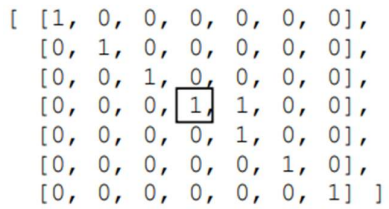
```
[
 [1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [1, 0, 0, 0, 1]
]
```



Output: 3

Input:

```
[
 [1, 0, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 1, 1, 0, 0],
 [0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 0, 1]
]
```



Output: 1

Input:

```
[
 [1, 0, 0],
 [0, 1, 0],
 [0, 0, 1.0]
]
```

Output: 0 , + Prints: "Not all values are int"