Defining My Own List Class

In [ ]:

```
class my_list:
    def __init__(self, initial_data=None):
        # Initialize the list with initial data if provided, else an empty list
        self._data = list(initial_data) if initial_data else []

    def append(self, item):
        # Append an item to the list
        self._data.append(item)

    def remove(self, item):
        # Remove the first occurrence of an item from the list
        self._data.remove(item)

    def pop(self, index=-1):
        # Remove and return item at index (default last)
        return self._data.pop(index)

    def insert(self, index, item):
        # Insert an item at a given position
        self._data.insert(index, item)

    def __getitem__(self, index):
        # Get item by index
        return self._data[index]

    def __setitem__(self, index, value):
        # Set item at a specific index
        self._data[index] = value

    def __delitem__(self, index):
        # Delete item at a specific index
        del self._data[index]

    def __len__(self):
        # Return the length of the list
        return len(self._data)

    def __iter__(self):
        # Return an iterator for the list
        return iter(self._data)

    def __repr__(self):
        # Return the string representation of the list
        return repr(self._data)

    def __contains__(self, item):
        # Check if an item is in the list
        return item in self._data

    def __add__(self, other):
        # Concatenate two lists
        if isinstance(other, my_list):
            return my_list(self._data + other._data)
        elif isinstance(other, list):
            return my_list(self._data + other)
        else:
            raise TypeError("Can only concatenate my_list or list to my_list")

    def __iadd__(self, other):
        # In-place add for list concatenation
        if isinstance(other, my_list):
            self._data += other._data
        elif isinstance(other, list):
            self._data += other
        else:
            raise TypeError("Can only concatenate my_list or list to my_list")
        return self

    def __eq__(self, other):
        # Check equality
        if isinstance(other, my_list):
            return self._data == other._data
        elif isinstance(other, list):
            return self._data == other
        else:
            return False
```

Performing Basic Functions

In [ ]:

```
# creating a list
my_list1 = my_list([1, 2, 3, 4, 5])

# adding an element to the list
my_list1.append(6)

# removing an element from the list
my_list1.remove(3)

# modifying an element in the list
my_list1[0] = 10

print("Updated List:", my_list1)
```

```
Updated List: [10, 2, 4, 5, 6]
```

Defining My Own Dict Class

In [ ]:

```
class my_dict:
    def __init__(self, initial_data=None):
        # Initialize the dictionary with initial data if provided, else an empty
        self._data = dict(initial_data) if initial_data else {}

    def __getitem__(self, key):
        # Get item by key
        return self._data[key]

    def __setitem__(self, key, value):
        # Set item at a specific key
        self._data[key] = value

    def __delitem__(self, key):
        # Delete item by key
        del self._data[key]

    def __contains__(self, key):
        # Check if key is in the dictionary
        return key in self._data

    def get(self, key, default=None):
        # Get item by key with a default value
        return self._data.get(key, default)

    def keys(self):
        # Return the keys of the dictionary
        return self._data.keys()

    def values(self):
        # Return the values of the dictionary
        return self._data.values()

    def items(self):
        # Return the items of the dictionary
        return self._data.items()

    def update(self, other):
        # Update the dictionary with another dictionary
        if isinstance(other, my_dict):
            self._data.update(other._data)
        elif isinstance(other, dict):
            self._data.update(other)
        else:
            raise TypeError("Can only update with another my_dict or dict")

    def pop(self, key, default=None):
        # Remove the specified key and return the corresponding value
        return self._data.pop(key, default)

    def __repr__(self):
        # Return the string representation of the dictionary
        return repr(self._data)

    def __len__(self):
        # Return the number of items in the dictionary
        return len(self._data)

    def __iter__(self):
        # Return an iterator over the keys of the dictionary
        return iter(self._data)

    def clear(self):
        # Remove all items from the dictionary
        self._data.clear()

    def copy(self):
        # Return a shallow copy of the dictionary
        return my_dict(self._data.copy())

    def setdefault(self, key, default=None):
        # Insert key with a value of default if key is not in the dictionary
        return self._data.setdefault(key, default)

    def __eq__(self, other):
        # Check equality
        if isinstance(other, my_dict):
            return self._data == other._data
        elif isinstance(other, dict):
            return self._data == other
        else:
            return False
```

Performing Basic Function

In [ ]:

```
# create dictionary
my_dict1 = my_dict({'name': 'John', 'age': 25, 'city': 'Delhi'})

# adding
my_dict1['gender'] = 'Male'

# removing
del my_dict1['age']

# modifying
my_dict1['city'] = 'Mumbai'

print("Updated Dictionary:", my_dict1)
```

```
Updated Dictionary: {'name': 'John', 'city': 'Mumbai', 'gender': 'Male'}
```

Defining My Own Set Class

In [ ]:

```
class my_set:
    def __init__(self, initial_data=None):
        # Initialize the set with initial data if provided, else an empty set
        self._data = set(initial_data) if initial_data else set()

    def add(self, element):
        # Add an element to the set
        self._data.add(element)

    def remove(self, element):
        # Remove an element from the set, raises KeyError if not found
        self._data.remove(element)

    def discard(self, element):
        # Remove an element from the set if present
        self._data.discard(element)

    def pop(self):
        # Remove and return an arbitrary set element, raises KeyError if empty
        return self._data.pop()

    def clear(self):
        # Remove all elements from the set
        self._data.clear()

    def __contains__(self, element):
        # Check if an element is in the set
        return element in self._data
    def __iter__(self):
        # Return an iterator for the set
        return iter(self._data)

    def __len__(self):
        # Return the number of elements in the set
        return len(self._data)

    def __repr__(self):
        # Return the string representation of the set
        return repr(self._data)

    def __eq__(self, other):
        # Check equality
        if isinstance(other, my_set):
            return self._data == other._data
        elif isinstance(other, set):
            return self._data == other
        else:
            return False

    def union(self, *others):
        # Return the union of sets as a new my_set
        new_set = self._data.union(*(other._data if isinstance(other, my_set) el
        return my_set(new_set)

    def intersection(self, *others):
        # Return the intersection of sets as a new my_set
        new_set = self._data.intersection(*(other._data if isinstance(other, my_
        return my_set(new_set)

    def difference(self, *others):
        # Return the difference of sets as a new my_set
        new_set = self._data.difference(*(other._data if isinstance(other, my_se
        return my_set(new_set)

    def symmetric_difference(self, other):
        # Return the symmetric difference of sets as a new my_set
        if isinstance(other, my_set):
            new_set = self._data.symmetric_difference(other._data)
        else:
            new_set = self._data.symmetric_difference(other)
        return my_set(new_set)

    def issubset(self, other):
        # Report whether another set contains this set
        if isinstance(other, my_set):
            return self._data.issubset(other._data)
        else:
            return self._data.issubset(other)

    def issuperset(self, other):
        # Report whether this set contains another set
        if isinstance(other, my_set):
            return self._data.issuperset(other._data)
        else:
            return self._data.issuperset(other)

    def isdisjoint(self, other):
        # Return True if two sets have a null intersection
        if isinstance(other, my_set):
            return self._data.isdisjoint(other._data)
        else:
            return self._data.isdisjoint(other)
```

Performing Basic Functions

In [ ]:

```
# creating set
my_set1 = my_set({1, 2, 3, 4, 5})

# adding
my_set1.add(6)

# removing
my_set1.remove(3)

# modifying
my_set1.discard(1)
my_set1.add(10)

print("Updated Set:", my_set1)
```

```
Updated Set: {2, 4, 5, 6, 10}
```