Module 6 - Data Structures and Strings

Welcome to Module 6! Having mastered the flow control and modularity of your code, it's time to delve into how Python organizes and stores data. This module will introduce you to Python's powerful built-in **data structures** – Dictionaries, Lists, Sets, and Tuples – and delve deeper into **Strings**, which are fundamental for text manipulation.

Chapter 1: Dictionaries - Key-Value Pairs

Dictionaries are one of Python's most versatile and frequently used data structures. They allow you to store data in **key-value pairs**, which makes retrieving data very efficient and intuitive.

1.1 Introduction to Dictionaries

- **Concept:** A dictionary is an **unordered** collection of items. Each item consists of a **key** and an associated **value**. Think of a real-world dictionary where words (keys) have definitions (values).
- **Characteristics:**
    - **Mutable:** You can add, remove, and modify key-value pairs after the dictionary has been created.
    - **Unordered:** (In Python 3.7+ dictionaries retain insertion order, but conceptually, they are not indexed like lists). You access items by their key, not by their position.
    - **Keys must be Unique:** No two items can have the same key. If you try to add an item with an existing key, the old value associated with that key will be overwritten.
    - **Keys must be Immutable:** Keys must be of an immutable data type (e.g., strings, numbers, tuples). Lists, sets, and other dictionaries cannot be used as keys because they are mutable.
    - **Values can be Any Type:** Values can be of any data type (strings, numbers, lists, other dictionaries, functions, etc.).
- **Use Cases:**
    - Representing real-world objects with properties (e.g., a person with `name`, `age`, `city`).
    - Storing configurations where settings have names.
    - Mapping unique identifiers to associated data.
    - Counting frequencies (e.g., word counts in text).

1.2 Creating Dictionaries

Dictionaries are defined using curly braces `{}` with key-value pairs separated by colons `:`, and pairs separated by commas `,`.

1. Empty Dictionary:

Python
```
>>> empty_dict = {}
>>> print(empty_dict)
{}
>>> print(type(empty_dict))
<class 'dict'>

>>> another_empty_dict = dict() # Using the dict() constructor
>>> print(another_empty_dict)
{}
```

2. With Initial Values:

Python
```
>>> student = {
...     "name": "Alice",
...     "age": 20,
...     "major": "Computer Science",
...     "gpa": 3.8
... }
>>> print(student)
{'name': 'Alice', 'age': 20, 'major': 'Computer Science', 'gpa': 3.8}

>>> # Keys can be numbers too
>>> grades = {
...     1: "Excellent",
...     2: "Good",
...     3: "Average"
... }
>>> print(grades)
{1: 'Excellent', 2: 'Good', 3: 'Average'}
```

1.3 Accessing Values

You access values in a dictionary using their corresponding keys.

1. Using Square Brackets `[]`:

- This is the most common way to access values.
- If the key does not exist, it will raise a `KeyError`.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20}
>>> print(student["name"])
Alice
>>> print(student["age"])
20

>>> # Attempting to access a non-existent key causes an error
>>> print(student["city"])
KeyError: 'city'
```

2. Using the `get()` Method:

- This method is safer as it returns `None` (or a specified default value) if the key is not found, instead of raising a `KeyError`.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20}
>>> print(student.get("name"))
Alice
>>> print(student.get("city")) # Key 'city' does not exist, returns None
None

>>> # You can provide a default value if the key is not found
>>> print(student.get("city", "Not Available"))
Not Available
```

1.4 Modifying and Adding Items

Dictionaries are mutable, so you can change their content.

1. Adding New Key-Value Pairs:

- Simply assign a value to a new key.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20}
>>> student["major"] = "Computer Science" # Add new key-value pair
>>> print(student)
{'name': 'Alice', 'age': 20, 'major': 'Computer Science'}
```

2. Updating Existing Values:

- If you assign a value to an existing key, the old value associated with that key will be overwritten.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20, "major": "Computer Science"}
>>> student["age"] = 21 # Update the value for key 'age'
>>> print(student)
{'name': 'Alice', 'age': 21, 'major': 'Computer Science'}
```

1.5 Deleting Items

You can remove key-value pairs from a dictionary using several methods.

1. `del` Statement:

- Deletes the item with the specified key.
- Raises `KeyError` if the key does not exist.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 21, "major": "Computer Science"}
>>> del student["gpa"] # This would cause a KeyError if "gpa" wasn't there
>>> del student["age"]
>>> print(student)
{'name': 'Alice', 'major': 'Computer Science'}
```

2. `pop(key)` Method:

- Removes the item with the specified key and returns its value.
- Raises `KeyError` if the key is not found, unless a default value is provided.

<!-- end list -->

Python
```
>>> student = {"name": "Bob", "age": 22, "major": "Physics"}
>>> removed_major = student.pop("major")
>>> print(f"Removed major: {removed_major}")
Removed major: Physics
>>> print(student)
{'name': 'Bob', 'age': 22}

>>> # Pop with a default value (no error if key not found)
>>> removed_gpa = student.pop("gpa", "N/A")
>>> print(f"Removed GPA: {removed_gpa}")
Removed GPA: N/A
>>> print(student)
{'name': 'Bob', 'age': 22}
```

3. `popitem()` Method:

- Removes and returns an arbitrary (key, value) pair. In Python 3.7+, it removes and returns the last inserted item.
- Raises `KeyError` if the dictionary is empty.

<!-- end list -->

Python
```
>>> inventory = {"apple": 5, "banana": 3, "orange": 7}
>>> last_item = inventory.popitem()
>>> print(f"Removed: {last_item}")
Removed: ('orange', 7)
>>> print(inventory)
{'apple': 5, 'banana': 3}
```

4. `clear()` Method:

- Removes all items from the dictionary, leaving it empty.

<!-- end list -->

Python

```
>>> my_dict = {"a": 1, "b": 2}
>>> my_dict.clear()
>>> print(my_dict)
{}
```

1.6 Dictionary Functions and Methods

Python provides several built-in functions and dictionary methods to work with dictionaries.

1. `len()` Function:

- Returns the number of key-value pairs in the dictionary.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20, "major": "Computer Science"}
>>> print(len(student))
3
```

2. `keys()` Method:

- Returns a **view object** that displays a list of all the keys in the dictionary. This view is dynamic (reflects changes to the dictionary).

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20, "major": "Computer Science"}
>>> all_keys = student.keys()
>>> print(all_keys)
dict_keys(['name', 'age', 'major'])

>>> # You can convert it to a list if needed
>>> print(list(all_keys))
['name', 'age', 'major']
```

3. `values()` Method:

- Returns a **view object** that displays a list of all the values in the dictionary.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20, "major": "Computer Science"}
>>> all_values = student.values()
>>> print(all_values)
dict_values(['Alice', 20, 'Computer Science'])
>>> print(list(all_values))
['Alice', 20, 'Computer Science']
```

4. `items()` Method:

- Returns a **view object** that displays a list of a dictionary's key-value tuple pairs.

<!-- end list -->

Python
```
>>> student = {"name": "Alice", "age": 20, "major": "Computer Science"}
>>> all_items = student.items()
>>> print(all_items)
dict_items([('name', 'Alice'), ('age', 20), ('major', 'Computer Science')])
>>> print(list(all_items))
[('name', 'Alice'), ('age', 20), ('major', 'Computer Science')]
```

5. `update(other_dict)` Method:

- Adds key-value pairs from `other_dict` into the current dictionary.
- If a key from `other_dict` already exists in the current dictionary, its value is updated.

<!-- end list -->

Python
```
>>> person = {"name": "Bob", "age": 30}
>>> details = {"age": 31, "city": "New York"}
>>> person.update(details)
>>> print(person)
{'name': 'Bob', 'age': 31, 'city': 'New York'} # 'age' was updated, 'city'
was added
```

Looping Through Dictionaries:

You can iterate over dictionaries in various ways:

- **Looping through keys (default):**

  Python

  ```
  >>> for key in student: # or for key in student.keys():
  ...     print(key)
  name
  age
  major
  ```

- **Looping through values:**

  Python

  ```
  >>> for value in student.values():
  ...      print(value)
  Alice
  20 Computer
  Science
  ```

- **Looping through key-value pairs (most common):**

  Python

```
>>> for key, value in student.items():
...     print(f"{key}: {value}")
name: Alice
age: 20
major: Computer Science
```

Chapter 2: Lists - Ordered, Mutable Sequences

Lists are one of the most fundamental and versatile data structures in Python. They are used to store collections of items.

2.1 Introduction to Lists

- **Concept:** A list is an **ordered**, **mutable** collection of items. "Ordered" means the items have a defined sequence, and you can access them by their position (index). "Mutable" means you can change the list after it's created (add, remove, or modify items).
- **Characteristics:**
  - **Ordered:** Items are stored in a specific sequence, maintaining their order.
  - **Mutable:** Elements can be changed after the list is created.
  - **Heterogeneous:** A list can contain items of different data types (integers, strings, floats, even other lists or dictionaries).
  - **Indexed:** Each item has an index, starting from 0 for the first element.
  - **Dynamic Size:** Lists can grow or shrink in size as elements are added or removed.
- **Use Cases:**
  - Storing sequences of data (e.g., a list of names, numbers, or temperatures).
  - Implementing stacks and queues.
  - Collecting data from various sources.

2.2 Creating Lists

Lists are defined using square brackets `[]` with items separated by commas.

1. Empty List:

Python

```
>>> empty_list = []
>>> print(empty_list)
[]
>>> print(type(empty_list))
<class 'list'>

>>> another_empty_list = list() # Using the list() constructor
>>> print(another_empty_list)
[]
```

2. With Initial Values:

Python
```
>>> fruits = ["apple", "banana", "cherry"]
>>> print(fruits)
['apple', 'banana', 'cherry']

>>> numbers = [1, 2, 3, 4, 5]
>>> print(numbers)
[1, 2, 3, 4, 5]

>>> mixed_list = ["hello", 123, True, 3.14] # Lists can hold different data
types
>>> print(mixed_list)
['hello', 123, True, 3.14]

>>> nested_list = [[1, 2], [3, 4]] # Lists can contain other lists
>>> print(nested_list)
[[1, 2], [3, 4]]
```

3. From Other Iterables:

You can convert other iterable objects (like strings, tuples, sets, ranges) into lists using `list()`.

Python
```
>>> list_from_string = list("python")
>>> print(list_from_string)
['p', 'y', 't', 'h', 'o', 'n']

>>> list_from_tuple = list((10, 20, 30))
>>> print(list_from_tuple)
[10, 20, 30]

>>> list_from_range = list(range(5)) # Creates a list from 0 to 4
>>> print(list_from_range)
[0, 1, 2, 3, 4]
```

2.3 Accessing List Elements (Indexing)

List elements are accessed using their position (index).

1. Positive Indexing:

- Starts from `0` for the first element.

<!-- end list -->

Python
```
>>> fruits = ["apple", "banana", "cherry", "date"]
>>> print(fruits[0]) # First element
apple
>>> print(fruits[2]) # Third element
cherry
```

2. Negative Indexing:

- Starts from `-1` for the last element.
- `-1` refers to the last item, `-2` to the second-to-last, and so on.

<!-- end list -->

Python
```
>>> fruits = ["apple", "banana", "cherry", "date"]
>>> print(fruits[-1]) # Last element
date
>>> print(fruits[-3]) # Third from the end
banana
```

Common Error: `IndexError`

Attempting to access an index that is out of the list's bounds will raise an `IndexError`.

Python
```
>>> my_list = [10, 20]
>>> print(my_list[2]) # Index 2 does not exist (valid indices are 0, 1)
IndexError: list index out of range
```

2.4 List Operations

You can perform various operations directly on lists using operators or built-in functions.

1. Concatenation (+):

- Joins two or more lists to create a new list.

<!-- end list -->

Python
```
>>> list1 = [1, 2]
>>> list2 = [3, 4]
>>> combined_list = list1 + list2
>>> print(combined_list)
[1, 2, 3, 4]
```

2. Repetition (*):

- Repeats a list a specified number of times to create a new list.

<!-- end list -->

Python
```
>>> zeros = [0] * 5
>>> print(zeros)
[0, 0, 0, 0, 0]

>>> repeated_items = ["a", "b"] * 3
>>> print(repeated_items)
```

```
['a', 'b', 'a', 'b', 'a', 'b']
```

3. Membership Testing (`in`, `not in`):

- Checks if an item exists within a list. Returns `True` or `False`.

<!-- end list -->

Python
```
>>> fruits = ["apple", "banana", "cherry"]
>>> print("apple" in fruits)
True
>>> print("grape" in fruits)
False
>>> print("banana" not in fruits)
False
```

4. Length (`len()`):

- Returns the number of items in the list.

<!-- end list -->

Python
```
>>> fruits = ["apple", "banana", "cherry"]
>>> print(len(fruits))
3
```

5. Iteration (Looping):

- You can easily loop through all items in a list using a `for` loop.

<!-- end list -->

Python
```
>>> my_numbers = [10, 20, 30, 40]
>>> for number in my_numbers:
...     print(number)
10
20
30
40
```

2.5 Modifying Lists (Adding, Removing, Other Methods)

Lists are mutable, meaning you can change their content after creation.

A. Adding Elements:

1. **`append(item)`:** Adds an item to the end of the list.

   Python

```
>>> fruits = ["apple", "banana"]
>>> fruits.append("orange")
>>> print(fruits)
['apple', 'banana', 'orange']
```

2. **`insert(index, item)`:** Inserts an item at a specified index. Existing items are shifted to the right.

   Python

```
>>> fruits = ["apple", "orange"]
>>> fruits.insert(1, "banana") # Insert "banana" at index 1
>>> print(fruits)
['apple', 'banana', 'orange']
```

3. **`extend(iterable)`:** Appends all elements from another iterable (like another list, tuple, or string) to the end of the current list.

   Python

```
>>> list1 = [1, 2]
>>> list2 = [3, 4, 5]
>>> list1.extend(list2)
>>> print(list1)
[1, 2, 3, 4, 5]

>>> characters = ['a', 'b']
>>> characters.extend("cde") # Extends with individual characters
from the string
>>> print(characters)
['a', 'b', 'c', 'd', 'e']
```

B. Removing Elements:

1. **`remove(value)`:** Removes the **first occurrence** of a specified value from the list. Raises `ValueError` if the value is not found.

   Python

```
>>> shopping_list = ["milk", "bread", "milk", "eggs"]
>>> shopping_list.remove("milk") # Removes the first 'milk'
>>> print(shopping_list)
['bread', 'milk', 'eggs']

>>> # shopping_list.remove("butter") # Would raise ValueError
```

2. **`pop(index=-1)`:** Removes and returns the item at a given index. If no index is provided, it removes and returns the **last** item. Raises `IndexError` if the index is out of range.

   Python

```
>>> my_list = ["A", "B", "C", "D"]
>>> removed_item = my_list.pop(1) # Remove item at index 1 ("B")
>>> print(f"Removed: {removed_item}")
```

```
Removed: B
>>> print(my_list)
['A', 'C', 'D']

>>> last_item = my_list.pop() # Remove the last item ("D")
>>> print(f"Removed last: {last_item}")
Removed last: D
>>> print(my_list)
['A', 'C']
```

3. **`del` Statement:** Deletes items by index or slice.

   Python

```
>>> my_list = [10, 20, 30, 40, 50]
>>> del my_list[1] # Delete item at index 1 (20)
>>> print(my_list)
[10, 30, 40, 50]

>>> del my_list[1:3] # Delete items from index 1 (inclusive) to 3
(exclusive) (20, 30, 40)
>>> print(my_list)
[10, 50]

>>> del my_list # Deletes the entire list object
>>> # print(my_list) # This would now cause a NameError
```

4. **`clear()`:** Removes all items from the list, making it empty.

   Python

```
>>> my_list = [1, 2, 3]
>>> my_list.clear()
>>> print(my_list)
[]
```

C. Other Useful Methods:

1. **`sort(reverse=False)`:** Sorts the list in-place (modifies the original list). By default, it sorts in ascending order. Use `reverse=True` for descending.
   - **Note:** All elements in the list must be comparable (e.g., all numbers, or all strings).

   <!-- end list -->

   Python

```
>>> numbers = [5, 2, 8, 1, 9]
>>> numbers.sort() # Sorts in-place, ascending
>>> print(numbers)
[1, 2, 5, 8, 9]

>>> names = ["Charlie", "Alice", "Bob"]
>>> names.sort(reverse=True) # Sorts in-place, descending
>>> print(names)
['Charlie', 'Bob', 'Alice']
```

- **`sorted(iterable, reverse=False)` (Built-in function):** Returns a *new* sorted list without modifying the original.

<!-- end list -->

Python

```
>>> original_list = [5, 2, 8, 1, 9]
>>> new_sorted_list = sorted(original_list)
>>> print(new_sorted_list)
[1, 2, 5, 8, 9]
>>> print(original_list) # Original list remains unchanged
[5, 2, 8, 1, 9]
```

2. **`reverse()`:** Reverses the order of elements in the list in-place.

   Python

   ```
   >>> my_list = [1, 2, 3, 4]
   >>> my_list.reverse()
   >>> print(my_list)
   [4, 3, 2, 1]
   ```

   - **`reversed(seq)` (Built-in function):** Returns an iterator that yields elements in reverse order without modifying the original.

   <!-- end list -->

   Python

   ```
   >>> original_list = [1, 2, 3]
   >>> for item in reversed(original_list):
   ...     print(item)
   3
   2
   1
   >>> print(original_list) # Original list unchanged
   [1, 2, 3]
   ```

3. **`count(value)`:** Returns the number of times a specified value appears in the list.

   Python

   ```
   >>> items = [1, 2, 2, 3, 2, 4]
   >>> print(items.count(2))
   3
   >>> print(items.count(5))
   0
   ```

4. **`index(value, start=0, end=len(list))`:** Returns the index of the first occurrence of a specified value. Raises `ValueError` if the value is not found. You can specify a search range.

   Python

```
>>> letters = ['a', 'b', 'c', 'b', 'd']
>>> print(letters.index('b'))
1
>>> print(letters.index('b', 2)) # Find 'b' starting from index 2
3
>>> # print(letters.index('z')) # Would raise ValueError
```

5. **`copy()`:** Returns a shallow copy of the list. This is important to avoid unintended modifications when assigning lists.

   Python

```
>>> original = [1, 2, 3]
>>> # Bad way to copy (both variables point to the same list in
memory)
>>> alias_list = original
>>> alias_list.append(4)
>>> print(original) # original is also modified!
[1, 2, 3, 4]

>>> # Good way to copy (creates a new independent list)
>>> original = [1, 2, 3]
>>> copied_list = original.copy()
>>> copied_list.append(4)
>>> print(original) # original is unaffected
[1, 2, 3]
>>> print(copied_list)
[1, 2, 3, 4]
```

2.6 List Slicing

Slicing allows you to extract a portion (a "slice") of a list to create a new list.

Syntax: `list[start:end:step]`

- `start`: (Optional) The starting index (inclusive). Default is 0.
- `end`: (Optional) The ending index (exclusive). Default is end of list.
- `step`: (Optional) The step (or stride) for jumping through elements. Default is 1.

Examples:

Python

```
>>> my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

>>> # Slice from start to end (creates a copy of the list)
>>> print(my_list[:])
['a', 'b', 'c', 'd', 'e', 'f', 'g']

>>> # Slice from index 2 up to (but not including) index 5
>>> print(my_list[2:5])
['c', 'd', 'e']

>>> # Slice from beginning up to (but not including) index 4
>>> print(my_list[:4])
['a', 'b', 'c', 'd']
```

```
>>> # Slice from index 3 to the end
>>> print(my_list[3:])
['d', 'e', 'f', 'g']

>>> # Slice with a step (every second element from start to end)
>>> print(my_list[::2])
['a', 'c', 'e', 'g']

>>> # Reverse a list using slicing (common idiom)
>>> print(my_list[::-1])
['g', 'f', 'e', 'd', 'c', 'b', 'a']

>>> # Modify a slice (this changes the original list)
>>> numbers = [10, 20, 30, 40, 50]
>>> numbers[1:4] = [25, 35, 45] # Replace elements at index 1, 2, 3
>>> print(numbers)
[10, 25, 35, 45, 50]
```

2.7 List Comprehension

List comprehension provides a concise and readable way to create new lists based on existing iterables. It's often much shorter and more efficient than using a `for` loop with `append()`.

Syntax: **`[expression for item in iterable if condition]`**

- `expression`: The item you want to put into the new list.
- `item`: The variable representing each element in the iterable.
- `iterable`: The sequence you're iterating over (e.g., another list, range, string).
- `condition`: (Optional) A filter that determines if `item` should be processed.

Examples:

1. **Simple transformation:** Create a list of squares from another list of numbers.

   Python

   ```
   >>> numbers = [1, 2, 3, 4, 5]
   >>> squares = [n**2 for n in numbers]
   >>> print(squares)
   [1, 4, 9, 16, 25]
   ```

2. **Filtering:** Create a list of even numbers from a range.

   Python

   ```
   >>> even_numbers = [num for num in range(10) if num % 2 == 0]
   >>> print(even_numbers)
   [0, 2, 4, 6, 8]
   ```

3. **Transformation and Filtering:** Convert a list of words to uppercase, but only for words longer than 3 characters.

   Python

```
>>> words = ["apple", "bat", "cat", "dog", "elephant"]
>>> long_words_upper = [word.upper() for word in words if len(word) >
3]
>>> print(long_words_upper)
['APPLE', 'ELEPHANT']
```

4. **Nested List Comprehension (for 2D lists):**

   Python

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> flattened = [num for row in matrix for num in row]
>>> print(flattened)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions are a powerful and widely used feature in Python for their conciseness and efficiency.

Chapter 3: Sets - Unordered Collections of Unique Elements

Sets are another built-in data structure used to store a collection of unique items. They are based on the mathematical concept of a set.

3.1 Introduction to Sets

- **Concept:** A set is an **unordered** collection of **unique** elements. This means a set cannot have duplicate values.
- **Characteristics:**
  - **Unordered:** Items in a set do not have a defined order or index. You cannot access elements by index like lists or tuples.
  - **Mutable (but elements must be immutable):** You can add or remove elements from a set. However, the elements *within* a set must be immutable (e.g., numbers, strings, tuples). Lists, dictionaries, or other sets cannot be elements of a set.
  - **Unique Elements:** Duplicates are automatically removed. If you add an existing element to a set, it simply won't change the set.
- **Use Cases:**
  - **Removing Duplicates:** Easily get a collection of unique items from a list.
  - **Membership Testing:** Very fast checking if an item is present in a large collection.
  - **Mathematical Set Operations:** Performing union, intersection, difference, etc.

3.2 Creating Sets

Sets are created using curly braces `{}` or the `set()` constructor.

1. Empty Set:

- **Important:** You must use `set()` for an empty set. `{}` creates an empty dictionary.

<!-- end list -->

Python
```
>>> empty_set = set()
>>> print(empty_set)
set()
>>> print(type(empty_set))
<class 'set'>
```

2. With Initial Values:

- Using curly braces for non-empty sets. Duplicates are automatically removed.

<!-- end list -->

Python
```
>>> fruits = {"apple", "banana", "cherry"}
>>> print(fruits) # Order might not be as entered due to 'unordered' nature
{'cherry', 'banana', 'apple'}

>>> numbers = {1, 2, 3, 2, 1, 4} # Duplicates 1 and 2 are removed
>>> print(numbers)
{1, 2, 3, 4}
```

3. From Iterables:

You can convert other iterable objects into sets using `set()`.

Python
```
>>> set_from_list = set(["a", "b", "c", "a"])
>>> print(set_from_list)
{'b', 'c', 'a'}

>>> set_from_string = set("programming") # Unique characters from the
string
>>> print(set_from_string)
{'r', 'm', 'g', 'i', 'p', 'o', 'a', 'n'}
```

3.3 Basic Set Operations

1. Adding Elements (`add(item)`):

- Adds a single item to the set. If the item already exists, the set remains unchanged.

<!-- end list -->

Python
```
>>> my_set = {1, 2, 3}
>>> my_set.add(4)
>>> print(my_set)
```

```
{1, 2, 3, 4}

>>> my_set.add(2) # 2 already exists, set remains unchanged
>>> print(my_set)
{1, 2, 3, 4}
```

2. Removing Elements (`remove(item)` vs. `discard(item)`):

- **`remove(item)`:** Removes a specified item. Raises `KeyError` if the item is not found.

  Python

  ```
  >>> my_set = {1, 2, 3}
  >>> my_set.remove(2)
  >>> print(my_set)
  {1, 3}

  >>> # my_set.remove(5) # Would raise KeyError
  ```

- **`discard(item)`:** Removes a specified item. **Does not raise an error** if the item is not found. Safer for removal.

  Python

  ```
  >>> my_set = {1, 2, 3}
  >>> my_set.discard(2)
  >>> print(my_set)
  {1, 3}

  >>> my_set.discard(5) # 5 not in set, no error
  >>> print(my_set)
  {1, 3}
  ```

3. `pop()`:

- Removes and returns an arbitrary (random) element from the set. Raises `KeyError` if the set is empty.

  Python

  ```
  >>> my_set = {10, 20, 30}
  >>> popped_element = my_set.pop()
  >>> print(f"Popped: {popped_element}")
  Popped: 10 # (Could be 20 or 30 too, depends on internal hashing)
  >>> print(my_set)
  {20, 30}
  ```

4. `clear()`:

- Removes all elements from the set, making it empty.

  Python

  ```
  >>> my_set = {1, 2, 3}
  >>> my_set.clear()
  ```

```
>>> print(my_set)
set()
```

5. Membership Testing (`in`, `not in`):

- Very efficient for checking if an item is present.

<!-- end list -->

Python
```
>>> fruits = {"apple", "banana", "cherry"}
>>> print("apple" in fruits)
True
>>> print("grape" not in fruits)
True
```

3.4 Mathematical Set Operations

Sets support standard mathematical set operations.

1. Union (All unique elements from both sets):

- Operator: `|`
- Method: `union()`

<!-- end list -->

Python
```
>>> set1 = {1, 2, 3}
>>> set2 = {3, 4, 5}
>>> print(set1 | set2)
{1, 2, 3, 4, 5}
>>> print(set1.union(set2))
{1, 2, 3, 4, 5}
```

2. Intersection (Common elements in both sets):

- Operator: `&`
- Method: `intersection()`

<!-- end list -->

Python
```
>>> set1 = {1, 2, 3}
>>> set2 = {3, 4, 5}
>>> print(set1 & set2)
{3}
>>> print(set1.intersection(set2))
{3}
```

3. Difference (Elements in the first set but NOT in the second):

- Operator: -

- Method: `difference()`

<!-- end list -->

Python
```
>>> set1 = {1, 2, 3, 6}
>>> set2 = {3, 4, 5}
>>> print(set1 - set2) # Elements in set1 but not in set2
{1, 2, 6}
>>> print(set2 - set1) # Elements in set2 but not in set1
{4, 5}
>>> print(set1.difference(set2))
{1, 2, 6}
```

4. Symmetric Difference (Elements in either set, but NOT in both):

- Operator: `^`
- Method: `symmetric_difference()`

<!-- end list -->

Python
```
>>> set1 = {1, 2, 3}
>>> set2 = {3, 4, 5}
>>> print(set1 ^ set2)
{1, 2, 4, 5}
>>> print(set1.symmetric_difference(set2))
{1, 2, 4, 5}
```

5. Subset, Superset, Disjoint:

- **`issubset()` (<=):** Returns `True` if all elements of one set are in another.
- **`issuperset()` (>=):** Returns `True` if one set contains all elements of another.
- **`isdisjoint()`:** Returns `True` if two sets have no elements in common (their intersection is empty).

<!-- end list -->

Python
```
>>> A = {1, 2, 3}
>>> B = {1, 2, 3, 4, 5}
>>> C = {6, 7}

>>> print(A.issubset(B)) # Is A a subset of B?
True
>>> print(A <= B)
True

>>> print(B.issuperset(A)) # Is B a superset of A?
True
>>> print(B >= A)
True

>>> print(A.isdisjoint(C)) # Do A and C have no common elements?
True
```

```
>>> print(A.isdisjoint(B)) # Do A and B have no common elements?
False
```

Chapter 4: Strings - Immutable Sequences of Characters

You've used strings extensively already, but let's explore their capabilities and common operations in more detail.

4.1 Introduction to Strings

- **Concept:** A string is an **ordered**, **immutable** sequence of characters. It's used to represent text.
- **Characteristics:**
    - **Ordered:** Characters have a defined sequence and can be accessed by index (like lists).
    - **Immutable:** Once a string is created, you cannot change individual characters within it. Any operation that appears to "modify" a string actually creates a *new* string.
    - **Iterable:** You can loop through a string character by character.
- **Defining Strings:** Enclosed in single quotes (`'...'`), double quotes (`"..."`), or triple quotes (`'''...'''` or `"""..."""`) for multi-line strings.

<!-- end list -->

Python
```
>>> single_quote = 'Hello'
>>> double_quote = "World"
>>> multi_line = """This is a
... multi-line string."""
>>> print(multi_line)
This is a
multi-line string.
```

4.2 String Functions and Methods

Strings come with a rich set of built-in methods for manipulation and inspection.

1. `len()` (Built-in function):

- Returns the number of characters in the string.

  Python

  ```
  >>> text = "Python"
  >>> print(len(text))
  6
  ```

2. Case Conversion:

- **`upper()`:** Returns a new string with all characters converted to uppercase.

- **`lower()`:** Returns a new string with all characters converted to lowercase.
- **`capitalize()`:** Returns a new string with the first character capitalized and the rest lowercase.
- **`title()`:** Returns a new string where the first letter of each word is capitalized.

Python

```
>>> s = "Hello Python"
>>> print(s.upper())
HELLO PYTHON
>>> print(s.lower())
hello python
>>> print(s.capitalize())
Hello python
>>> print(s.title())
Hello Python
```

3. Stripping Whitespace:

- **`strip()`:** Returns a new string with leading and trailing whitespace removed.
- **`lstrip()`:** Removes leading whitespace.
- **`rstrip()`:** Removes trailing whitespace.

Python

```
>>> s_padded = "   Hello World   "
>>> print(s_padded.strip())
Hello World
>>> print(s_padded.lstrip())
Hello World
>>> print(s_padded.rstrip())
   Hello World
```

4. Checking Content:

- **`startswith(prefix)`:** Returns `True` if the string starts with the specified prefix.
- **`endswith(suffix)`:** Returns `True` if the string ends with the specified suffix.
- **`isdigit()`:** Returns `True` if all characters are digits and there is at least one character.
- **`isalpha()`:** Returns `True` if all characters are alphabetic and there is at least one character.
- **`isalnum()`:** Returns `True` if all characters are alphanumeric (letters or numbers) and there is at least one character.
- **`isspace()`:** Returns `True` if all characters are whitespace and there is at least one character.

Python

```
>>> name = "Alice"
>>> print(name.startswith("Al"))
True
>>> print(name.endswith("ice"))
True

>>> print("123".isdigit())
```

```
True
>>> print("abc".isalpha())
True
>>> print("A123".isalnum())
True
>>> print("   ".isspace())
True
```

5. Finding and Replacing:

- **`find(substring)`:** Returns the lowest index in the string where the substring is found. Returns `-1` if not found.
- **`index(substring)`:** Same as `find()`, but raises a `ValueError` if the substring is not found.
- **`replace(old, new, count=-1)`:** Returns a new string with all (or up to `count`) occurrences of `old` replaced by `new`.

  Python

```
>>> text = "hello world hello"
>>> print(text.find("world"))
6
>>> print(text.find("python"))
-1

>>> print(text.replace("hello", "hi"))
hi world hi
>>> print(text.replace("hello", "hi", 1)) # Replace only first
occurrence
hi world hello
```

6. Splitting and Joining:

- **`split(separator=None)`:** Splits the string into a list of substrings based on the `separator`. If `separator` is `None` (default), it splits by any whitespace and removes empty strings.

  Python

```
>>> sentence = "This is a sample sentence"
>>> words = sentence.split() # Splits by whitespace
>>> print(words)
['This', 'is', 'a', 'sample', 'sentence']

>>> csv_data = "apple,banana,cherry"
>>> fruits_list = csv_data.split(',')
>>> print(fruits_list)
['apple', 'banana', 'cherry']
```

- **`join(iterable)`:** Joins elements of an iterable (e.g., a list of strings) into a single string, using the string on which `join()` is called as the separator.

  Python

```
>>> my_words = ["Hello", "World", "Python"]
```

```
>>> joined_by_space = " ".join(my_words)
>>> print(joined_by_space)
Hello World Python

>>> joined_by_dash = "-".join(my_words)
>>> print(joined_by_dash)
Hello-World-Python
```

4.3 String Formatting

String formatting allows you to create dynamic strings by inserting values into predefined placeholders.

1. `%` Operator (Old Style):

- Similar to C's `sprintf`. Less common in modern Python.

  Python

```
>>> name = "Alice"
>>> age = 30
>>> print("My name is %s and I am %d years old." % (name, age))
My name is Alice and I am 30 years old.
```

2. `str.format()` Method:

- Uses curly braces `{}` as placeholders. More readable and flexible than `%`.

  Python

```
>>> name = "Bob"
>>> age = 25
>>> print("My name is {} and I am {} years old.".format(name, age))
My name is Bob and I am 25 years old.

>>> # Can specify argument by position or name
>>> print("My name is {0} and I am {1} years old. {0} loves
Python.".format(name, age))
My name is Bob and I am 25 years old. Bob loves Python.

>>> print("The price is {price:.2f} for {item}.".format(item="apple",
price=1.2345))
The price is 1.23 for apple.
```

3. F-strings (Formatted String Literals - Modern and Recommended):

- Introduced in Python 3.6. They are prefixed with `f` or `F`.
- Allow embedding expressions directly inside string literals by placing them inside curly braces `{}`.
- Highly readable, concise, and often faster.

  Python

```
>>> name = "Charlie"
```

```
>>> age = 35
>>> print(f"My name is {name} and I am {age} years old.")
My name is Charlie and I am 35 years old.

>>> # Can include expressions
>>> print(f"Next year, {name} will be {age + 1} years old.")
Next year, Charlie will be 36 years old.

>>> price = 19.99
>>> quantity = 3
>>> total = price * quantity
>>> print(f"You bought {quantity} items at ${price:.2f} each. Total:
${total:.2f}")
You bought 3 items at $19.99 each. Total: $59.97

>>> # Can call functions
>>> print(f"Lowercased name: {name.lower()}")
Lowercased name: charlie
```

Recommendation: For most modern Python code, **use f-strings** for string formatting.

Chapter 5: Tuples - Ordered, Immutable Sequences

Tuples are similar to lists in many ways, but with one critical difference: they are immutable.

5.1 Introduction to Tuples

- **Concept:** A tuple is an **ordered**, **immutable** collection of items.
- **Characteristics:**
  - **Ordered:** Items have a defined sequence and can be accessed by index (like lists).
  - **Immutable:** Once a tuple is created, you cannot change its elements (add, remove, or modify). This is their main distinction from lists.
  - **Heterogeneous:** Can contain items of different data types.
  - **Indexed:** Each item has an index, starting from 0.
- **Use Cases:**
  - **Returning Multiple Values from Functions:** Functions often return multiple values as a tuple.
  - **Fixed Collections of Data:** When you have a collection of items that should not change throughout the program's execution (e.g., coordinates, RGB colors).
  - **Dictionary Keys:** Since tuples are immutable, they can be used as keys in dictionaries (unlike lists).

5.2 Creating Tuples

Tuples are defined using parentheses `()` with items separated by commas.

1. Empty Tuple:

Python
```
>>> empty_tuple = ()
>>> print(empty_tuple)
()
>>> print(type(empty_tuple))
<class 'tuple'>
```

2. With Initial Values:

Python
```
>>> coordinates = (10, 20)
>>> print(coordinates)
(10, 20)

>>> rgb_color = (255, 0, 128)
>>> print(rgb_color)
(255, 0, 128)

>>> mixed_tuple = ("apple", 123, True)
>>> print(mixed_tuple)
('apple', 123, True)
```

3. Single-Item Tuple (Crucial Comma!):

- To create a tuple with a single item, you **must include a comma** after the item. Without the comma, Python treats it as a regular expression in parentheses.

<!-- end list -->

Python
```
>>> single_item_tuple = (5,) # The comma is essential!
>>> print(single_item_tuple)
(5,)
>>> print(type(single_item_tuple))
<class 'tuple'>

>>> not_a_tuple = (5) # This is just an integer 5
>>> print(not_a_tuple)
5
>>> print(type(not_a_tuple))
<class 'int'>
```

4. Tuple Packing and Unpacking:

- **Packing:** Creating a tuple without parentheses (though often used for clarity).

  Python

  ```
  >>> packed_data = 1, "hello", True
  >>> print(packed_data)
  (1, 'hello', True)
  >>> print(type(packed_data))
  <class 'tuple'>
  ```

- **Unpacking:** Assigning elements of a tuple (or any iterable) to multiple variables. This is very common and powerful.

Python

```
>>> point = (10, 25)
>>> x, y = point # Unpacking the tuple into x and y
>>> print(f"X: {x}, Y: {y}")
X: 10, Y: 25

>>> # Swapping variables easily
>>> a = 5
>>> b = 10
>>> a, b = b, a # Pythonic swap!
>>> print(f"a: {a}, b: {b}")
a: 10, b: 5
```

5.3 Accessing Tuple Elements

Accessing tuple elements is identical to accessing list elements, using positive or negative indexing and slicing.

1. Indexing:

Python

```
>>> my_tuple = ("a", "b", "c", "d")
>>> print(my_tuple[0])
a
>>> print(my_tuple[-1])
d
```

2. Slicing:

Python

```
>>> my_tuple = (10, 20, 30, 40, 50)
>>> print(my_tuple[1:4])
(20, 30, 40)
>>> print(my_tuple[::-1]) # Reverse a tuple
(50, 40, 30, 20, 10)
```

Important: Slicing a tuple returns a *new* tuple.

5.4 Tuple Operations

Tuples support many of the same operations as lists that do not involve modification.

1. Concatenation (+):

Python

```
>>> tuple1 = (1, 2)
>>> tuple2 = (3, 4)
>>> combined = tuple1 + tuple2
>>> print(combined)
(1, 2, 3, 4)
```

2. Repetition (*):

Python
```
>>> repeated_tuple = ("x",) * 3
>>> print(repeated_tuple)
('x', 'x', 'x')
```

3. Membership Testing (`in`, `not in`):

Python
```
>>> my_tuple = ("red", "green", "blue")
>>> print("green" in my_tuple)
True
```

4. Length (`len()`):

Python
```
>>> my_tuple = (1, 2, 3)
>>> print(len(my_tuple))
3
```

5. Iteration:

Python
```
>>> for item in (1, 2, 3):
...     print(item)
1
2
3
```

6. Built-in Tuple Functions:

- **`count(value)`:** Returns the number of times a specified value appears.
- **`index(value)`:** Returns the index of the first occurrence of a value.

<!-- end list -->

Python
```
>>> data = (1, 2, 3, 2, 4, 2)
>>> print(data.count(2))
3
>>> print(data.index(3))
2
```

5.5 When to Use Tuples vs. Lists

The choice between a list and a tuple depends on whether you need mutability and the semantic meaning of your data.

Feature	List	Tuple
Mutability	**Mutable** (can change after creation)	**Immutable** (cannot change after creation)
Syntax	`[]` (square brackets)	`()` (parentheses)

Performance	Generally slightly slower for iteration than tuples (due to mutability overhead)	Generally slightly faster for iteration than lists (due to immutability)

Rule of Thumb:

- Use a **List** when you have a collection of items that you expect to **change**, add to, remove from, or reorder.
- Use a **Tuple** when you have a collection of items that should **not change** after they are created, or when you are representing a fixed record (like a database row or coordinates).

Chapter 6: Coding Challenges for Data Structures and Strings

Time to practice with Python's core data structures and string manipulation techniques!

Challenge 1: Student Grade Manager (Dictionaries)

Goal: Create a program that allows you to store and manage student names and their grades using a dictionary.

Concepts Covered: Dictionaries (`{}`), adding/updating items, `get()`, looping (`items()`), `input()`.

Requirements:

1. Initialize an empty dictionary called `student_grades`.
2. Implement a loop that presents a menu to the user:
    - 1. Add/Update Student Grade
    - 2. View Student Grade
    - 3. View All Grades
    - 4. Exit
3. **Add/Update Student Grade:**
    - Prompt for student name and their grade (assume it's a number).
    - Add or update the entry in `student_grades`. Print confirmation.
4. **View Student Grade:**
    - Prompt for student name.

    - Use `get()` to retrieve the grade. If the student is not found, print "Student not found." Otherwise, print the student's name and grade.
5. **View All Grades:**
    - Iterate through `student_grades.items()` and print each student's name and grade. If the dictionary is empty, print "No grades entered yet."
6. **Exit:** Break the loop.
7. Handle invalid menu choices.

Example Interaction:

```
--- Student Grade Manager ---
1. Add/Update Student Grade
2. View Student Grade
3. View All Grades
4. Exit
Enter your choice: 1
Enter student name: Alice
Enter grade: 95
Grade for Alice added/updated.

Enter your choice: 1
Enter student name: Bob
Enter grade: 88
Grade for Bob added/updated.

Enter your choice: 3
--- All Grades ---
Alice: 95
Bob: 88

Enter your choice: 2
Enter student name: Alice
Alice's grade: 95

Enter your choice: 2
Enter student name: Charlie
Student not found.

Enter your choice: 4
Goodbye!
```

Challenge 2: List Manipulator

Goal: Perform various operations on a list based on user input, including adding, removing, and sorting elements, and using slicing.

Concepts Covered: Lists (`[]`), `append()`, `insert()`, `remove()`, `pop()`, `sort()`, `len()`, list slicing, `input()`.

Requirements:

1. Initialize an empty list called `my_items`.
2. Implement a loop that presents a menu:
    -
        1. Add Item (to end)

- 2. Insert Item (at specific position)
- 3. Remove Item by Value
- 4. Remove Last Item
- 5. View All Items
- 6. Sort Items (Alphabetical)
- 7. View Sub-List (Slicing)
- 8. Exit

3. Implement the logic for each menu choice using appropriate list methods.
    - For "Insert Item," ask for both the item and the index.
    - For "Remove Item by Value," handle `ValueError` if the item isn't found.
    - For "View Sub-List," ask for start and end indices. Handle `IndexError` or bad input if necessary (simple error message is fine).
4. After each operation (except exit), display the current state of `my_items`.

Example Interaction:

```
--- List Manipulator ---
Current List: []
1. Add Item
2. Insert Item
3. Remove Item by Value
4. Remove Last Item
5. View All Items
6. Sort Items
7. View Sub-List
8. Exit
Enter your choice: 1
Enter item to add: Apple
Current List: ['Apple']

Enter your choice: 1
Enter item to add: Banana
Current List: ['Apple', 'Banana']

Enter your choice: 2
Enter item to insert: Cherry
Enter index: 1
Current List: ['Apple', 'Cherry', 'Banana']

Enter your choice: 6
```

```
Current List: ['Apple', 'Banana', 'Cherry'] # Sorted!

Enter your choice: 3
Enter item to remove: Cherry
Current List: ['Apple', 'Banana']

Enter your choice: 7
Enter start index: 0
Enter end index: 1
Sub-list: ['Apple']
Current List: ['Apple', 'Banana']

Enter your choice: 8
Goodbye!
```

Challenge 3: Unique Word Counter (Sets and Strings)

Goal: Read a sentence from the user, count the total words, and list all unique words.

Concepts Covered: Sets (`set()`), `add()`, string methods (`lower()`, `split()`), `len()`.

Requirements:

1. Prompt the user to "Enter a sentence:".
2. Convert the input sentence to lowercase to handle case-insensitivity (e.g., "Hello" and "hello" are the same word).
3. Split the sentence into individual words.
4. Count the total number of words.
5. Create a set of unique words from the split words.
6. Print:
    - The total number of words.
    - The number of unique words.
    - The list of unique words (you can print the set directly or convert it to a list for ordered display, e.g., `sorted(unique_words_set)`).

Example Interaction:

```
Enter a sentence: Python is powerful and Python is fun
Total words: 7
Unique words: 5
Unique words list (sorted): ['and', 'fun', 'is', 'powerful', 'python']
```

Challenge 4: User Profile Formatter (String Formatting)

Goal: Ask a user for their details and then display a formatted profile using f-strings.

Concepts Covered: String formatting (f-strings), `input()`, basic data types.

Requirements:

1. Ask the user for:
    - Their Name
    - Their Age (as an integer)

    - Their City
    - Their Favorite Programming Language
2. Store these inputs in appropriate variables.
3. Using f-strings, print a formatted user profile like this:

```
--- User Profile ---
Name: [User's Name]
Age: [User's Age] years old
City: [User's City]
Favorite Language: [User's Favorite Language]
--------------------
```

   Ensure the age is displayed as an integer.

Example Interaction:

```
Enter your name: Jane Doe
Enter your age: 28
Enter your city: London
Enter your favorite programming language: Python
--- User Profile ---
Name: Jane Doe
Age: 28 years old
City: London
Favorite Language: Python
--------------------
```

Challenge 5: Basic Inventory (Tuple and Dictionary Mix)

Goal: Represent a simple inventory system where each item is a tuple, and the inventory itself is a dictionary mapping item IDs to item details.

Concepts Covered: Tuples, Dictionaries, accessing nested data, iteration.

Requirements:

1. Create an initial dictionary called `inventory` where:
    - Keys are `item_id` (integers).
    - Values are tuples containing `(item_name, quantity, price_per_unit)`. Example: `inventory = {101: ("Laptop", 5, 1200.00), 102: ("Mouse", 20, 25.50)}`
2. Print the initial inventory clearly.
3. Add a new item to the inventory (get `item_id`, `name`, `qty`, `price` from user and create a new tuple).
4. Update the quantity of an existing item (get `item_id` and new `quantity` from user). Remember, you can't change the tuple, so you'll need to create a *new tuple* with the updated quantity and assign it back to the existing `item_id` key.
5. Print the updated inventory.

Example Interaction:

```
--- Inventory System ---
Initial Inventory:
ID 101: ('Laptop', 5, 1200.0)
```

```
ID 102: ('Mouse', 20, 25.5)

Add New Item:
Enter new Item ID: 103
Enter Item Name: Keyboard
Enter Quantity: 15
Enter Price: 75.0
Item 103 added.

Updated Inventory:
ID 101: ('Laptop', 5, 1200.0)
ID 102: ('Mouse', 20, 25.5)
ID 103: ('Keyboard', 15, 75.0)

Update Item Quantity:
Enter Item ID to update: 101
Enter new Quantity: 3
Quantity for ID 101 updated.

Final Inventory:
ID 101: ('Laptop', 3, 1200.0)
ID 102: ('Mouse', 20, 25.5)
ID 103: ('Keyboard', 15, 75.0)
```