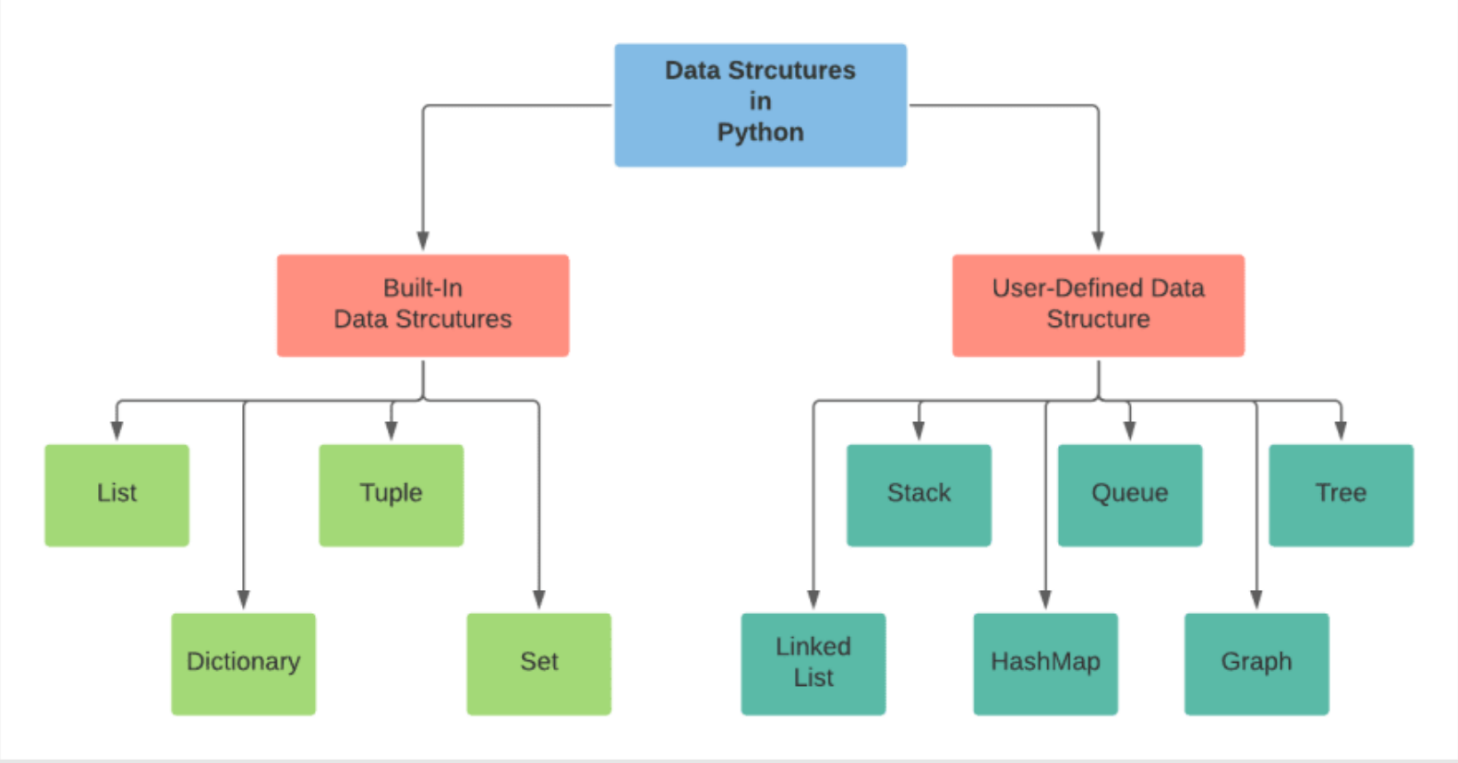
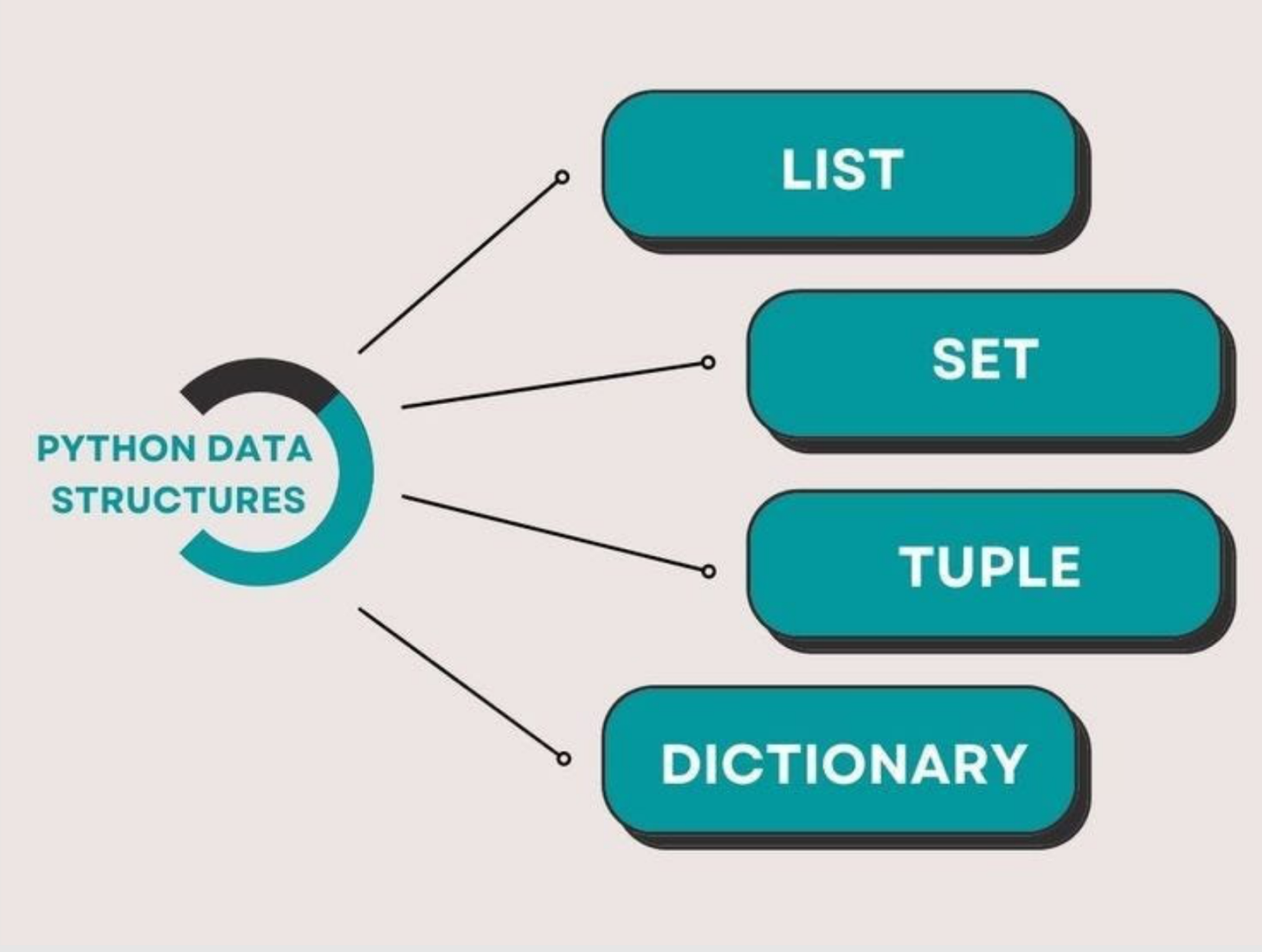
Built-in Data Structures :





Python provides several built-in data structures that are

efficient, versatile, and designed to handle a variety of tasks. These data structures include **lists**, **tuples**, **dictionaries**, **sets**, and **strings**. Below is a detailed explanation of each type, their characteristics, and the methods associated with them.

1. Lists

Definition

A **list** is an ordered, mutable collection that allows duplicate elements and can store items of mixed data types. Lists are highly versatile and widely used for storing and managing sequences of data.

Characteristics

- **Ordered**: Lists maintain the order of elements. Each item has a specific index starting from 0.
- **Mutable**: You can modify lists by adding, removing, or changing elements.
- **Dynamic**: Lists can grow or shrink in size as needed.
- **Heterogeneous**: A single list can contain items of different data types, such as integers, strings, or

even other lists.

Common Use Cases

- Storing dynamic collections of data.
- Iterating over sequences of elements.
- Performing various operations like sorting, filtering, and aggregating.

Methods

- **Adding Elements**:
- append(item): Adds an item to the end of the list.
- extend(iterable): Adds all elements of an iterable (e.g., another list) to the end of the list.
- insert(index, item): Inserts an item at a specified index.
- **Removing Elements**:
- remove(item): Removes the first occurrence of the specified item.

- pop(index): Removes and returns the item at the specified index. If no index is provided, it removes the last item.
- clear(): Removes all elements from the list.
- **Finding Elements**:
- index(item): Returns the index of the first occurrence of the specified item.
- count(item): Returns the number of occurrences of the specified item.
- **Sorting and Reversing**:
- sort(): Sorts the list in ascending order (can use the reverse=True parameter for descending order).
- reverse(): Reverses the order of the list.
- **Copying**:
- copy(): Creates a shallow copy of the list.
- **Slicing**:

- Lists support slicing to extract subsets of elements using the list[start:stop:step] syntax.

2. Tuples

Definition

A **tuple** is an ordered, immutable collection of items. Once created, the elements in a tuple cannot be changed, making it ideal for representing fixed sets of data.

Characteristics

- **Ordered**: Elements maintain their position.
- **Immutable**: You cannot modify, add, or remove elements after a tuple is created.
- **Heterogeneous**: Like lists, tuples can store elements of different data types.

Common Use Cases

- Representing constant data (e.g., geographic coordinates, configuration settings).

- Returning multiple values from a function.
- Using as keys in dictionaries (since tuples are hashable).

Methods

- **Finding Elements**:
- count(item): Returns the number of occurrences of the specified item.
- index(item): Returns the index of the first occurrence of the specified item.

Additional Notes

- Tuples are more memory-efficient than lists.
- They support slicing and unpacking, similar to lists.

3. Dictionaries

Definition

A **dictionary** is an unordered collection of key-value

pairs, where each key is unique, and values can be of any data type.

Characteristics

- **Unordered**: Items are stored without a specific order (insertion order is preserved from Python 3.7 onwards).
- **Mutable**: You can add, update, or remove key-value pairs.
- **Keys must be immutable**: Strings, numbers, or tuples can be used as keys, but lists cannot.

Common Use Cases

- Storing mappings (e.g., user IDs and names, product codes and prices).
- Quick lookups and updates based on unique keys.
- Storing JSON-like data structures.

Methods

- **Adding/Updating Elements**:

- dict[key] = value: Adds or updates the value associated with the specified key.

- update(other_dict): Updates the dictionary with key-value pairs from another dictionary.

- **Removing Elements:**

the key-value pair with the specified key and returns the value.

- popitem(): Removes and returns the last inserted key-value pair (useful for Python 3.7+).

- del dict[key]: Deletes the key-value pair with the specified key.

- clear(): Removes all key-value pairs from the dictionary.

- **Accessing Elements**:

- dict[key]: Retrieves the value associated with the specified key (throws a KeyError if the key doesn't exist).

- get(key, default=None): Retrieves the value

associated with the key, returning the default value if the key doesn't exist.

- **Other Useful Methods**:

- keys(): Returns a view object of all the keys in the dictionary.

- values(): Returns a view object of all the values in the dictionary.

- items(): Returns a view object of all key-value pairs in the dictionary as tuples.

- copy(): Creates a shallow copy of the dictionary.

4. Sets

Definition

A **set** is an unordered, mutable collection of unique elements. Sets do not allow duplicate elements and are commonly used for membership tests and eliminating duplicates.

Characteristics

- **Unordered**: Sets do not maintain a specific order of elements.

- **Mutable**: You can add or remove elements, but the elements themselves must be immutable (e.g., strings, numbers, tuples).

- **No duplicates**: Duplicate elements are automatically removed when added to a set.

Common Use Cases

- Removing duplicates from a list.

- Performing mathematical set operations (e.g., union, intersection, difference).

- Checking membership in a collection.

Methods

- **Adding/Removing Elements**:

- add(item): Adds an item to the set.

- remove(item): Removes the specified item from the set (throws a KeyError if the item is not present).

- discard(item): Removes the specified item without throwing an error if it doesn’t exist.
- pop(): Removes and returns an arbitrary item from the set.
- clear(): Removes all elements from the set.
- **Mathematical Operations**:
- union(other_set): Returns a set containing all unique elements from both sets.
- intersection(other_set): Returns a set containing only elements common to both sets.
- difference(other_set): Returns a set containing elements in the first set but not in the second.
- symmetric_difference(other_set): Returns a set containing elements not common to both sets.
- **Other Methods**:
- issubset(other_set): Returns True if the set is a subset of another set.

- issuperset(other_set): Returns True if the set is a superset of another set.

- copy(): Creates a shallow copy of the set.

5. Strings

Definition

A **string** is an immutable sequence of characters. Strings are widely used in Python for representing and manipulating textual data.

Characteristics

- **Immutable**: Once created, the contents of a string cannot be modified.

- **Ordered**: Strings maintain the order of characters, and you can access individual characters using indexing.

- **Homogeneous**: Strings only store text (characters).

Common Use Cases

- Storing and manipulating textual data.
- Parsing, formatting, and processing strings.

Methods

- **String Manipulation**:
- upper(): Converts all characters to uppercase.
- lower(): Converts all characters to lowercase.
- strip(): Removes leading and trailing whitespace.
- replace(old, new): Replaces all occurrences of old with new.
- split(delimiter): Splits the string into a list based on the specified delimiter.
- join(iterable): Joins elements of an iterable into a string using the string as a separator.
- **Searching and Checking**:
- find(substring): Returns the index of the first occurrence of the substring or -1 if not found.

- startswith(prefix): Returns True if the string starts with the specified prefix.
- endswith(suffix): Returns True if the string ends with the specified suffix.
- isalpha(): Returns True if the string contains only alphabetic characters.
- isdigit(): Returns True if the string contains only digits.
- **Formatting**:
- format(): Formats a string using placeholders.
- f-strings: A more modern and concise way to format strings (e.g., f"Hello, {name}").

Summary of Built-in Data Structures

This summary provides a foundation for understanding Python's built-in data structures and their applications.