5
Built-in Collections

Python provides several built-in collection types designed to store and manipulate data efficiently. These collections can be categorized based on their properties and use cases, including sequences, sets, mappings, and binary data structures. Understanding these types is essential for writing efficient and readable Python code.

This chapter will cover the following topics:

- Sequence Types (Ordered and Indexable)
- Lists: Mutable Ordered Sequences
- Tuples: Immutable Ordered Sequences
- Mapping Types (Key-Value Pairs)
- Dictionaries: Key-Value Mappings
- Set Types (Unordered, Unique Elements)
- Frozensets: Immutable Unique Collections
- Binary Data Types (Handling Bytes)
- Negative Indexing and Slicing in Lists
- Workarounds for Modifying Tuples
- Mathematical Set Operations (Union, Intersection, Difference)
- Best Practices for Lists and Tuples
- Nested Collections: Combining Collections
- Choosing the Right Collection for a Task
- Performance Considerations for Common Operations
- Immutability vs. Mutability in Collections

Overview

Here is a comprehensive list of built-in collection types, categorized by their primary characteristics. This overview provides a high-level perspective on the available types. In the following sections, we will explore each type in detail.

Sequence Types (Ordered and Indexable)

Sequences are ordered collections that allow elements to be accessed via indexing and support slicing operations. They provide a flexible way to store and manipulate ordered data.

A **list** is a mutable sequence that can store elements of any type, including numbers, strings, and even other collections. Lists support dynamic resizing, allowing elements to be added, removed, or modified at any time. For example, a list can be defined as my_list = [1, 2, 3, "hello", True], where elements of different types coexist.

A **tuple** is an immutable sequence, meaning that once created, its elements cannot be changed. Tuples are commonly used when the data should remain constant throughout the program's execution. An example of a tuple is my_tuple = (1, 2, 3, "world"). Tuples are particularly useful for returning multiple values from a function.

Mapping Types (Key-Value Pairs)

Mappings are collections that store data as key-value pairs, providing fast lookup times. Unlike sequences, which use numeric indices, mappings allow access to values using keys.

A **dictionary** (dict) is the primary mapping type in Python. It associates unique, hashable keys with values, allowing for efficient data retrieval. A dictionary is defined as my_dict = {"name": "Alice", "age": 25}, where "name" and "age" are keys, and "Alice" and 25 are their corresponding values. Dictionaries are widely used for storing structured data, such as JSON-like configurations and database records.

Set Types (Unordered, Unique Elements)

Sets are unordered collections that store unique elements. Unlike sequences, sets do not allow duplicate values and support mathematical operations like union, intersection, and difference.

A **set** is a mutable collection that automatically removes duplicates when elements are added. For instance, my_set = {1, 2, 3, 4, 4, 5} results in {1, 2, 3, 4, 5} since duplicate values are discarded. Sets are useful for operations where uniqueness is required, such as removing duplicate entries from a dataset.

A **frozenset** is an immutable version of a set, meaning that its contents cannot be modified after creation. This makes it hashable and suitable for use as dictionary keys or elements in other sets. It is defined as my_frozenset = frozenset([1, 2, 3, 4]).

Binary Data Types (Handling Bytes)

Python includes specialized data types for working with binary data, which are essential for file handling, network communication, and low-level data processing.

A **bytes** object is an immutable sequence of byte values, commonly used for storing binary files such as images or encoded text. It is defined as my_bytes = b"hello", where each character is represented by its corresponding byte value.

A **bytearray** is a mutable counterpart to bytes, allowing modifications. It can be initialized with a list of integer values representing byte data, such as my_bytearray = bytearray([65, 66, 67]), which corresponds to ASCII characters 'A', 'B', and 'C'.

A **memoryview** provides a way to access and manipulate the underlying binary data of a bytes or bytearray object without creating a copy. This is useful for performance-critical applications that process large amounts of binary data. It can be created using my_memoryview = memoryview(my_bytes).

Each of these built-in collection types serves a distinct purpose, allowing Python developers to handle different types of data efficiently. Choosing the appropriate collection type depends on the specific requirements of the task, such as whether ordering, mutability, or uniqueness is a priority.

List

Lists in Python (CRUD Pattern)

The **CRUD** (Create, Read, Update, Delete) pattern is a common approach for managing data structures. In Python, lists support CRUD operations at both the **list level** (modifying the entire list) and the **element level** (modifying individual elements). This section explores how lists function within the CRUD framework.

Create (C) – Creating Lists and Adding Elements

Lists can be created in multiple ways and allow adding new elements dynamically.

Creating Lists

Lists are created using square brackets [] or the list() constructor.

```
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed_list = ["apple", 3.14, True]
nested_list = [[1, 2], [3, 4]]
converted_list = list("hello")  # ['h', 'e', 'l', 'l', 'o']
```

Adding Elements to a List (Updating the List)

Once a list is created, elements can be added using various methods.

- **append(value)**: Adds a single element to the end of the list.
- **extend(iterable)**: Adds multiple elements by merging another iterable.
- **insert(index, value)**: Inserts an element at a specific position.

```
my_list = [1, 2, 3]
my_list.append(4)            # [1, 2, 3, 4]
my_list.extend([5, 6, 7])    # [1, 2, 3, 4, 5, 6, 7]
my_list.insert(1, 99)        # [1, 99, 2, 3, 4, 5, 6, 7]
```

Read (R) – Accessing Elements and Slices

Reading elements from a list can be done using indexing, iteration, or list comprehension.

Accessing Elements

Lists support zero-based indexing, and elements can be retrieved using positive or negative indices.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])   # "apple"
print(fruits[-1])  # "cherry" (last element)
```

Reading Multiple Elements Using Slicing

Slicing allows retrieving a subset of the list using list[start:stop:step].

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8]
print(numbers[2:6])   # [2, 3, 4, 5] (elements from index 2 to 5)
print(numbers[:4])    # [0, 1, 2, 3] (first four elements)
print(numbers[::2])   # [0, 2, 4, 6, 8] (every second element)
```

Update (U) – Modifying Elements and Lists

Lists are **mutable**, meaning elements can be modified after creation.

Updating Elements in a List

Elements can be modified by assigning a new value to a specific index.

```
fruits[1] = "blueberry"
print(fruits)  # ["apple", "blueberry", "cherry"]
```

Updating Multiple Elements Using Slicing

A slice of a list can be updated with new values.

```
numbers[2:5] = [99, 88, 77]
print(numbers)  # [0, 1, 99, 88, 77, 5, 6, 7, 8]
```

Sorting and Reversing a List

Lists can be sorted or reversed in place.

```
numbers.sort()  # Sorts the list in ascending order
numbers.reverse()  # Reverses the list
```

Delete (D) – Removing Elements and Clearing Lists

Elements can be removed by value, index, or by clearing the entire list.

Removing Elements from a List

Python provides several ways to remove elements:

- **remove(value)**: Removes the first occurrence of a value.
- **pop(index)**: Removes and returns an element at a specific index (default is last).
- **del list[index]**: Deletes an element at a specific index.

```
fruits.remove("blueberry")  # Removes "blueberry"
```

```
last_fruit = fruits.pop()    # Removes and returns "cherry"

del numbers[3]               # Removes the element at index 3
```

Clearing a List

To remove all elements from a list:

```
fruits.clear()  # Becomes an empty list []
```

Deleting a List Entirely

To delete a list from memory:

```
del fruits  # The list is no longer accessible
```

Negative Indexing

Python lists support **negative indexing**, allowing elements to be accessed from the end of the list rather than the beginning. This feature provides a convenient way to work with elements at the tail of a list without needing to know its length. Negative indexing is also supported in slicing, enabling reverse traversal and efficient sublist extraction.

Accessing Elements with Negative Indexing

In Python, list indices start at 0 for the first element. Negative indices count backward from the end, with -1 referring to the last element, -2 to the second-to-last, and so on.

```
numbers = [10, 20, 30, 40, 50]

print(numbers[-1])  # 50 (last element)

print(numbers[-2])  # 40 (second-to-last element)

print(numbers[-5])  # 10 (first element, same as numbers[0])
```

Negative indexing is useful when working with lists of unknown or dynamic length, as it eliminates the need to calculate the position of the last few elements manually.

Negative Indexing in Slicing

Slicing allows extracting sublists using the syntax list[start:stop:step], and negative indices can be used for both start and stop positions.

```
letters = ['a', 'b', 'c', 'd', 'e', 'f']

print(letters[-4:])

print(letters[-5:-2])

print(letters[:-3])
```

Using negative indices in slicing provides an intuitive way to extract elements from the end of the list without needing to compute their exact positions.

Reversing a List Using Negative Step

The **step** parameter in slicing controls how elements are selected. Using a negative step reverses the list.

```
numbers = [1, 2, 3, 4, 5, 6]


print(numbers[::-1])
print(numbers[-1:-4:-1])
print(numbers[-3::-1])
```

This method is particularly useful for reversing lists without modifying the original data.

Combining Negative Indexing with Regular Indexing

Negative and positive indices can be mixed in slicing operations, creating flexible and readable expressions.

```
data = [100, 200, 300, 400, 500, 600, 700]
print(data[2:-2])
print(data[-5:4])
print(data[1:-1:2])
```

By combining negative and positive indices, lists can be efficiently sliced regardless of their length.

Negative indexing provides an easy way to access elements from the end of a list, avoiding the need for manual length calculations. It seamlessly integrates with slicing operations, enabling flexible sublist extraction and reverse traversal. Whether accessing a single element, slicing a portion of a list, or reversing its order, negative indexing enhances the readability and efficiency of list operations in Python.

Python lists provide a flexible and efficient way to store ordered collections of data. By applying the CRUD pattern, we can systematically manage lists and their elements through creation, retrieval, modification, and deletion. Lists are widely used due to their dynamic nature, rich set of built-in methods, and support for iteration, making them a fundamental data structure in Python.

Tuple

A **tuple** is an immutable, ordered collection in Python that can store elements of different data types. Unlike lists, tuples cannot be modified after creation, making them useful for storing fixed data structures where immutability is a requirement. This section explores tuples through the **CRUD** pattern: **Create, Read, Update, and Delete**.

Create (C) – Creating Tuples

Tuples are created using parentheses () or the tuple() constructor. They can store numbers, strings, other tuples, and even lists.

Creating Tuples

A tuple can be defined with values separated by commas inside parentheses:

```
empty_tuple = ()

single_element_tuple = (42,)  # A comma is required for a single-element tuple

numbers_tuple = (1, 2, 3, 4, 5)

mixed_tuple = ("hello", 3.14, True)

nested_tuple = ((1, 2), (3, 4))
```

Tuples can also be created using the tuple() constructor:

```
converted_tuple = tuple([10, 20, 30])

string_tuple = tuple("abc")
```

Since tuples are immutable, they are typically used when the data should remain constant throughout the program.

Read (R) – Accessing Elements and Slicing Tuples

Since tuples maintain a fixed order, elements can be accessed using **indexing** and **slicing**, just like lists.

Accessing Elements by Index

Tuples support zero-based indexing:

```
colors = ("red", "green", "blue")

print(colors[0])

print(colors[-1])

print(colors[1])
```

Slicing Tuples

Tuples can be sliced to extract sub-tuples using tuple[start:stop:step]:

```
numbers = (0, 1, 2, 3, 4, 5, 6)

print(numbers[2:5])  # (2, 3, 4)

print(numbers[:4])   # (0, 1, 2, 3)

print(numbers[-3:])  # (4, 5, 6)

print(numbers[::-1]) # (6, 5, 4, 3, 2, 1, 0) (reversed tuple)
```

Update (U) – Modifying Tuples (Workarounds)

Tuples are **immutable**, meaning they cannot be changed after creation. However, there are ways to work around this limitation.

Reassigning a Tuple

While individual elements cannot be modified, a tuple variable can be reassigned:

```
coordinates = (10, 20)

coordinates = (30, 40)  # Reassignment creates a new tuple
```

Concatenating and Extending Tuples

Tuples can be combined to create new tuples:

```
tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

new_tuple = tuple1 + tuple2  # (1, 2, 3, 4, 5, 6)
```

Converting a Tuple to a List for Modification

If modification is necessary, a tuple can be converted into a list, updated, and then converted back into a tuple:

```
fruits = ("apple", "banana", "cherry")

fruits_list = list(fruits)

fruits_list[1] = "blueberry"

fruits = tuple(fruits_list)

print(fruits)
```

Although this workaround exists, if modification is needed frequently, a list might be a better choice.

Delete (D) – Removing Elements and Tuples

Since tuples are immutable, **individual elements cannot be deleted**, but entire tuples can be removed.

Deleting a Tuple

To remove a tuple from memory:

```
colors = ("red", "green", "blue")

del colors  # The tuple no longer exists
```

Attempting to access colors after deletion will raise an error.

Removing Elements (Workaround)

To "remove" an element, a new tuple must be created without the unwanted elements:

```
numbers = (1, 2, 3, 4, 5)
```

Tuples provide an efficient and memory-friendly way to store immutable sequences of data. By applying the **CRUD pattern**, we see that tuples support **creation** and **reading** operations easily but do not allow direct updates or deletions. However, reassignment, concatenation, and conversion to lists offer workarounds for modification. Tuples are particularly useful for fixed collections, return values from functions, and situations where immutability ensures data consistency and security.

Dict

A **dictionary** (dict) is a built-in data structure in Python that stores key-value pairs. Unlike lists and tuples, which use numerical indexing, dictionaries allow fast lookups using **unique, hashable keys**. They are **mutable**, meaning elements can be added, modified, and removed dynamically. This section explores dictionaries through the **CRUD** pattern: **Create, Read, Update, and Delete**.

Create (C) – Creating Dictionaries

Dictionaries can be created using curly braces {} with key-value pairs separated by colons, or with the dict() constructor.

Creating Dictionaries

empty_dict = {}

person = {"name": "Alice", "age": 30, "city": "New York"}

numeric_keys = {1: "one", 2: "two", 3: "three"}

mixed_dict = {"id": 101, "price": 19.99, "tags": ["sale", "new"]}

Dictionaries can also be created using dict():

person_dict = dict(name="Bob", age=25, city="London")

Adding New Key-Value Pairs

New entries can be added by assigning a value to a **new key**:

person["email"] = "alice@example.com"

print(person) # {'name': 'Alice', 'age': 30, 'city': 'New York', 'email': 'alice@example.com'}

Read (R) – Accessing Dictionary Elements

Dictionary values can be accessed using **keys**, either directly or with the get() method.

Accessing Values by Key

print(person["name"]) # "Alice"

print(person["age"])   # 30

Attempting to access a non-existent key **raises a KeyError**:

print(person["phone"]) # KeyError: 'phone'

Using get() to Avoid Errors

The get() method returns None (or a default value) instead of raising an error if the key is missing:

print(person.get("phone"))   # None

print(person.get("phone", "N/A")) # "N/A" (default value)

Checking for Key Existence

```
if "email" in person:
    print("Email exists:", person["email"])
```

Iterating Over a Dictionary

Dictionaries can be looped through using keys(), values(), or items():

```
for key in person.keys():
    print(key)  # Prints all keys


for value in person.values():
    print(value)  # Prints all values


for key, value in person.items():
    print(f"{key}: {value}")  # Prints key-value pairs
```

Update (U) – Modifying Dictionaries

Dictionaries are mutable, allowing existing keys to be updated and new key-value pairs to be added.

Modifying Existing Values

```
person["age"] = 31  # Updates the age
print(person)  # {'name': 'Alice', 'age': 31, 'city': 'New York', 'email': 'alice@example.com'}
```

Updating Multiple Keys with update()

```
person.update({"age": 32, "city": "San Francisco"})
print(person)  # {'name': 'Alice', 'age': 32, 'city': 'San Francisco', 'email': 'alice@example.com'}
```

Delete (D) – Removing Elements and Clearing Dictionaries

Dictionary elements can be removed using del, pop(), or popitem(). The entire dictionary can also be cleared.

Removing Elements by Key

```
del person["email"]
print(person)  # {'name': 'Alice', 'age': 32, 'city': 'San Francisco'}
```

Removing and Returning a Value with pop()

```
age = person.pop("age")
print(age)  # 32
print(person)  # {'name': 'Alice', 'city': 'San Francisco'}
```

Removing the Last Inserted Item with popitem()

The popitem() method removes the **last inserted key-value pair**:

last_item = person.popitem()

print(last_item) # ('city', 'San Francisco')

print(person) # {'name': 'Alice'}

Clearing All Elements

person.clear()

print(person) # {}

Deleting the Dictionary

del person # Dictionary is completely removed

print(person) # NameError: name 'person' is not defined

Dictionaries provide an efficient way to store and retrieve key-value pairs, offering fast lookups and dynamic updates. Using the **CRUD pattern**, we see that dictionaries support easy **creation** and **reading**, flexible **updates**, and multiple ways to **delete** elements or the entire dictionary. Their ability to store structured data in a readable format makes them a fundamental part of Python programming.

Sets in Python (CRUD Pattern)

A **set** is an **unordered**, **mutable**, and **unique** collection in Python. Unlike lists and tuples, sets do not allow duplicate values and do not maintain order. They are optimized for **fast membership testing and mathematical operations** like union, intersection, and difference. This section explores sets using the **CRUD** pattern: **Create, Read, Update, and Delete**.

Create (C) – Creating Sets

Sets can be created using curly braces {} or the set() constructor. They are particularly useful for ensuring uniqueness and performing mathematical set operations.

Creating Sets

empty_set = set() # Using set(), since {} creates an empty dictionary

numbers = {1, 2, 3, 4, 5}

mixed_set = {3.14, "apple", True, 42}

Creating a Set from a List (Removing Duplicates)

When a list contains duplicate elements, converting it to a set removes duplicates:

unique_numbers = set([1, 2, 2, 3, 4, 4, 5])

print(unique_numbers) # {1, 2, 3, 4, 5}

Since sets are **unordered**, their elements may appear in a different order when printed.

Read (R) – Accessing Set Elements

Since sets are unordered, elements cannot be accessed using **indexing** or **slicing** like lists or tuples.

Checking for Membership (in keyword)

The best way to retrieve data from a set is by checking if an element exists:

```
fruits = {"apple", "banana", "cherry"}


print("apple" in fruits)  # True
print("orange" in fruits)  # False
```

Iterating Over a Set

Since sets are iterable, they can be looped through using a for loop:

```
for fruit in fruits:
    print(fruit)  # Order may vary
```

Finding the Length of a Set (len())

The len() function returns the number of elements in a set:

```
print(len(fruits))  # 3
```

Update (U) – Modifying Sets

Unlike lists, sets do not support **index-based modification** since they are unordered. However, elements can be added or removed dynamically.

Adding Elements to a Set (add())

New elements can be added using the add() method:

```
fruits.add("orange")
print(fruits)  # {"apple", "banana", "cherry", "orange"}
```

Adding Multiple Elements (update())

The update() method can add multiple elements from an iterable (e.g., list, tuple, or another set):

```
fruits.update(["grape", "kiwi"])
print(fruits)  # Order may vary, but all elements will be unique
```

Delete (D) – Removing Elements and Clearing Sets

Elements can be removed using different methods, each with specific behavior.

Removing an Element (remove())

The remove() method deletes a specific element but raises a KeyError if the element is not found:

```
fruits.remove("banana")

print(fruits)

# If "banana" is not in the set, it raises an error
```

Removing an Element Safely (discard())

The discard() method removes an element **without raising an error** if the element does not exist:

```
fruits.discard("banana")  # No error, even if "banana" was already removed
```

Removing and Returning a Random Element (pop())

Since sets are unordered, the pop() method removes and returns an **arbitrary** element:

```
random_fruit = fruits.pop()

print(random_fruit)  # Random element from the set
```

Clearing All Elements (clear())

To empty a set, use the clear() method:

```
fruits.clear()

print(fruits)  # set()
```

Deleting the Set (del)

The del keyword completely removes the set from memory:

```
del fruits

# print(fruits)  # Raises a NameError since the set no longer exists
```

Mathematical Set Operations

Sets support mathematical operations like **union**, **intersection**, **difference**, and **symmetric difference**.

Union (| or union())

Combines elements from both sets, removing duplicates:

```
set1 = {1, 2, 3}

set2 = {3, 4, 5}


union_set = set1 | set2  # {1, 2, 3, 4, 5}

# Alternative: set1.union(set2)
```

Intersection (& or intersection())

Finds common elements between sets:

```
intersection_set = set1 & set2  # {3}
```

Alternative: set1.intersection(set2)

Difference (- or difference())

Finds elements in one set but not in another:

difference_set = set1 - set2 # {1, 2}

Alternative: set1.difference(set2)

Symmetric Difference (^ or symmetric_difference())

Finds elements that are in one set or the other, but not both:

symmetric_diff_set = set1 ^ set2 # {1, 2, 4, 5}

Alternative: set1.symmetric_difference(set2)

Sets provide an **efficient way** to store unique elements and perform fast membership checks and mathematical operations. Using the **CRUD pattern**, we see that sets support **creation** and **reading** via membership checks and iteration, while **updating** and **deleting** elements work differently than lists due to their **unordered nature**. Their ability to remove duplicates automatically makes them useful for data processing and mathematical computations.

Best Practices for Using Lists and Tuples

When working with lists in Python, a key best practice is to use them for items of the same type that share the same meaning or purpose. This ensures clarity, consistency, and ease of iteration, making the code more maintainable and predictable. However, in the case of tuples, this rule can be relaxed, as tuples often serve a different purpose in structuring data.

Using Lists for Homogeneous Data

A list is best suited for storing multiple elements of the same type that represent similar entities. This approach improves readability, facilitates bulk operations, and ensures that iterating over the list produces meaningful and expected results.

Example: List with Homogeneous Data

ages = [25, 30, 28, 35, 40] # All elements are integers representing ages

names = ["Alice", "Bob", "Charlie", "David"] # All elements are strings (names)

prices = [10.99, 5.49, 3.75] # All elements are floats representing prices

Here, each list contains elements of the same type and meaning, making operations like sorting, filtering, or aggregation straightforward.

Why Avoid Heterogeneous Lists?

Using lists to store mixed data types can reduce clarity and make operations unpredictable.

person = ["Alice", 30, True, "New York"]

This list is harder to work with because the meaning of each element is not immediately clear. A dictionary or a tuple would be a better choice for structured, heterogeneous data.

Relaxing the Rule for Tuples

Unlike lists, tuples are often used for grouping related but different types of data into a single unit. This is because tuples are typically employed for structuring data rather than performing repetitive operations on similar items.

Example: Tuples for Heterogeneous Data

person = ("Alice", 30, "New York")

In this case, the tuple represents a single entity with multiple attributes (name, age, city). The use of different types is appropriate because each position in the tuple has a specific meaning.

When to Use Lists vs. Tuples

Use Case	List	Tuple
Storing multiple values of the same type	✅ Best practice	❌ Not recommended
Grouping related attributes of an entity	❌ Use a dictionary or tuple	✅ Best practice
Mutability required	✅ Can be modified	❌ Immutable
Iterating and performing batch operations	✅ Ideal for iteration and modification	❌ Less flexible for iteration

---

Conclusion

When using lists, it is best practice to store elements of the same type and meaning to ensure consistency and predictability. However, in the case of tuples, this restriction is relaxed since tuples often group related but different types of data into a structured unit. By following this distinction, Python code becomes more readable, maintainable, and efficient.

Nested Collections: Combining Collections

Python allows collections to be nested within each other, enabling complex data structures that are highly flexible and efficient for various use cases. Nested collections are particularly useful when working with hierarchical data, structured records, and multidimensional storage. This section explores how different collections, such as **lists, dictionaries, sets, and tuples**, can be combined to form more advanced data structures.

Lists of Dictionaries

A common use case for nested collections is storing multiple dictionaries inside a list. This structure is useful for managing structured data, such as records in a database or JSON-like objects.

```
students = [
    {"name": "Alice", "age": 25, "grades": [85, 90, 92]},
    {"name": "Bob", "age": 22, "grades": [78, 80, 89]},
    {"name": "Charlie", "age": 23, "grades": [92, 88, 95]}
]
```

Here, each student is represented as a dictionary, and all student records are stored in a list. This allows easy iteration and retrieval.

Accessing and Modifying Data

Elements within this structure can be accessed using list indexing followed by dictionary key access:

```
print(students[0]["name"])  # "Alice"
print(students[1]["grades"])  # [78, 80, 89]


# Modifying a student's grade
students[2]["grades"].append(97)
print(students[2]["grades"])  # [92, 88, 95, 97]
```

This approach is commonly used when dealing with APIs, where JSON responses are often structured as lists of dictionaries.

Dictionaries of Lists

A dictionary where values are lists is useful when multiple values are associated with a single key. This is commonly used for **grouping** data.

```
subjects = {
    "Math": ["Alice", "Bob", "Charlie"],
    "Science": ["Alice", "Charlie"],
    "History": ["Bob", "Charlie"]
}
```

Accessing and Modifying Data

```
print(subjects["Math"])  # ['Alice', 'Bob', 'Charlie']


# Adding a new student to Science
```

```
subjects["Science"].append("Bob")


# Creating a new subject
subjects["English"] = ["Alice", "Charlie"]
```

This structure allows efficient categorization of elements.

Dictionaries of Sets

Using **sets** as dictionary values is useful when uniqueness is required. For instance, when tracking which students are enrolled in which courses without duplication:

```
enrollments = {
    "Math": {"Alice", "Bob", "Charlie"},
    "Science": {"Alice", "Charlie"},
    "History": {"Bob", "Charlie"}
}
```

Operations on Dictionary of Sets

```
# Adding a student
enrollments["Math"].add("David")


# Removing a student
enrollments["Science"].discard("Charlie")


# Checking membership
if "Alice" in enrollments["History"]:
    print("Alice is in History class")
```

Sets ensure that each student is only listed once in each course.

Lists of Sets

A **list of sets** can be useful for operations that require **grouping unique values** while maintaining order.

```
unique_values = [
    {1, 2, 3},
    {3, 4, 5},
    {6, 7, 8}
```

]

Set Operations on Lists

```
# Finding common elements between two sets
common = unique_values[0] & unique_values[1]  # {3}


# Merging all sets
merged_set = set().union(*unique_values)  # {1, 2, 3, 4, 5, 6, 7, 8}
```

This structure is useful when handling grouped but non-duplicated data.

Dictionaries of Dictionaries

A **dictionary of dictionaries** is useful when each key represents a larger dataset with multiple properties.

```
company = {
    "Alice": {"role": "Engineer", "salary": 70000},
    "Bob": {"role": "Manager", "salary": 85000},
    "Charlie": {"role": "Analyst", "salary": 65000}
}
```

Accessing and Modifying Nested Dictionaries

```
print(company["Alice"]["role"])  # "Engineer"


# Updating salary
company["Bob"]["salary"] += 5000


# Adding a new employee
company["David"] = {"role": "Intern", "salary": 40000}
```

This is commonly used in **databases, APIs, and structured configurations**.

Conclusion

Nested collections provide a powerful way to organize and manipulate complex data. By combining lists, dictionaries, and sets, Python enables efficient storage, retrieval, and modification of hierarchical and structured information. Understanding these structures is essential for working with real-world datasets, APIs, and object modeling.

(Un)packing

Unpacking in Python

Unpacking is a powerful feature in Python that allows you to **assign values from an iterable (like a list or tuple) to multiple variables in a single line**. It makes code more readable and concise, and it is commonly used in function returns, loops, and working with data structures like dictionaries.

Basic Unpacking with Lists and Tuples

In Python, you can unpack values from a list or tuple directly into variables:

```
# Unpacking a tuple
coordinates = (10, 20, 30)
x, y, z = coordinates


print(x)  # Output: 10
print(y)  # Output: 20
print(z)  # Output: 30
```

The same works with lists:

```
values = [100, 200, 300]
a, b, c = values


print(a, b, c)  # Output: 100 200 300
```

Rules of Unpacking:

- The number of variables **must match** the number of elements in the iterable.
- Otherwise, Python raises a ValueError.

x, y = (10, 20, 30) # ValueError: too many values to unpack

Using the * Operator for Extended Unpacking

Python 3 introduced **extended unpacking** using *, which allows you to capture multiple elements into a list.

```
numbers = [1, 2, 3, 4, 5]
first, *middle, last = numbers


print(first)   # Output: 1
print(middle)  # Output: [2, 3, 4]
print(last)    # Output: 5
```

Use cases of * unpacking:

- **Capturing the rest of the list** while extracting specific values.
- **Ignoring unnecessary elements** in a function return.
- **Reversing lists dynamically** using *reversed(iterable).

Example: Unpacking a function return:

```
def get_values():
    return 10, 20, 30, 40


a, *b, c = get_values()


print(a)  # Output: 10
print(b)  # Output: [20, 30]
print(c)  # Output: 40
```

Unpacking in Loops

When working with **tuples in lists**, unpacking helps extract values easily.

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three')]


for number, word in pairs:
    print(f"Number: {number}, Word: {word}")
```

Output:

```
Number: 1, Word: one
Number: 2, Word: two
Number: 3, Word: three
```

Unpacking Dictionaries

Dictionaries store key-value pairs, and unpacking can be useful when iterating over them.

Example: Unpacking dictionary items

```
student = {"name": "Alice", "age": 25, "grade": "A"}


for key, value in student.items():
    print(f"{key}: {value}")
```

Output:

name: Alice

age: 25

grade: A

✅ **Other Dictionary Unpacking Tricks**
You can unpack dictionary keys or values using **:

```
data = {"x": 10, "y": 20}

new_data = {**data, "z": 30}


print(new_data)

# Output: {'x': 10, 'y': 20, 'z': 30}
```

Swapping Variables Using Unpacking

Python allows **swapping variable values** without using a temporary variable.

```
a, b = 5, 10

a, b = b, a  # Swap values


print(a, b)  # Output: 10 5
```

Why is this useful?

- It is **faster and cleaner** than using a temporary variable.
- It works with **any data type** (strings, lists, etc.).

Summary of Unpacking Techniques

Technique	Example	Use Case
Basic Unpacking	x, y = (10, 20)	Assign values from tuples/lists
Extended Unpacking (*)	first, *middle, last = [1, 2, 3, 4]	Capture multiple values dynamically
Loop Unpacking	for key, value in dict.items()	Iterate over tuples and dictionaries easily
Swapping Variables	a, b = b, a	Swap values without a temp variable
Dictionary Unpacking (**)	{**data, "z": 30}	Merge dictionaries dynamically

Unpacking in Python makes working with sequences **more readable and efficient**. Whether you're dealing with **lists, tuples, dictionaries, or function returns**, unpacking simplifies code structure while improving performance.

List Comprehensions

List Comprehensions in Python

List comprehensions provide a **concise and efficient way** to create lists in Python. Instead of using loops and append() calls, list comprehensions allow you to **generate lists in a single line of code**, making your code more readable and faster.

Basic Syntax of List Comprehensions

The general syntax of a list comprehension is:

```
[expression for item in iterable]
```

This is equivalent to:

```
result = []
for item in iterable:
    result.append(expression)
```

Example: Creating a list of squares

Using a traditional loop:

```
squares = []
for x in range(5):
    squares.append(x ** 2)


print(squares)  # Output: [0, 1, 4, 9, 16]
```

Using list comprehension:

```
squares = [x ** 2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

✅ **Advantages of List Comprehensions:**

- **More concise** than traditional loops.
- **Faster execution** (optimized for performance).
- **Improves readability** in many cases.

Adding Conditional Logic to List Comprehensions

You can add **conditions** to filter elements while generating a list.

List of even numbers from 0 to 10

```
evens = [x for x in range(11) if x % 2 == 0]
```

```
print(evens)  # Output: [0, 2, 4, 6, 8, 10]
```

Equivalent loop approach:

```
evens = []
for x in range(11):
    if x % 2 == 0:
        evens.append(x)
```

List comprehensions with conditions help filter data easily.

Using if-else in List Comprehensions

You can include an if-else condition inside a list comprehension:

```
numbers = [1, 2, 3, 4, 5]
labels = ["even" if x % 2 == 0 else "odd" for x in numbers]


print(labels)  # Output: ['odd', 'even', 'odd', 'even', 'odd']
```

Equivalent loop approach:

```
labels = []
for x in numbers:
    if x % 2 == 0:
        labels.append("even")
    else:
        labels.append("odd")
```

Use if-else for conditional transformations inside comprehensions.

Nested List Comprehensions

List comprehensions can be **nested** to generate multi-dimensional lists.

Example: Generating a 3x3 matrix using nested list comprehensions

```
matrix = [[x for x in range(3)] for _ in range(3)]


print(matrix)
# Output: [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

Example: Flattening a 2D list into a 1D list

```
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in nested_list for num in row]
```

```
print(flattened)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Nested comprehensions are useful for working with matrices and multi-dimensional data.

Dictionary and Set Comprehensions

List comprehensions aren't limited to lists. You can also use them for **dictionaries and sets**.

Dictionary Comprehension

```
# Squaring numbers and storing them in a dictionary
squares_dict = {x: x ** 2 for x in range(5)}

print(squares_dict)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Set Comprehension

Getting unique numbers from a list

```
unique_numbers = {x for x in [1, 2, 2, 3, 4, 4, 5]}

print(unique_numbers)  # Output: {1, 2, 3, 4, 5}
```

Dictionary comprehensions are great for key-value transformations, while set comprehensions help remove duplicates.

List Comprehensions vs. Map and Filter

Python's map() and filter() functions achieve similar results as list comprehensions.

Task	List Comprehension	Using map() and filter()
Squaring numbers	[x ** 2 for x in range(5)]	list(map(lambda x: x ** 2, range(5)))
Filtering even numbers	[x for x in range(10) if x % 2 == 0]	list(filter(lambda x: x % 2 == 0, range(10)))

List comprehensions are generally more readable than map() and filter().

Summary of List Comprehensions

Feature	Example	Use Case
Basic Comprehension	[x ** 2 for x in range(5)]	Creating lists dynamically
Filtering with if	[x for x in range(10) if x % 2 == 0]	Selecting elements conditionally

Feature	Example	Use Case
Using if-else	["even" if x % 2 == 0 else "odd" for x in numbers]	Conditional transformation
Nested Comprehensions	[[x for x in range(3)] for _ in range(3)]	Working with multi-dimensional data
Dictionary Comprehension	{x: x ** 2 for x in range(5)}	Creating key-value pairs dynamically
Set Comprehension	{x for x in [1,2,2,3]}	Extracting unique values

List comprehensions provide an elegant and efficient way to create and manipulate lists. By replacing traditional loops with **one-liner comprehensions**, you can make your code more readable, expressive, and performant.

Comparison of Collections

Python provides a variety of collection types, each with its strengths and trade-offs. Choosing the right collection depends on the specific requirements of a task, such as whether elements need to be **ordered**, **unique**, or **mutable**. This section explores when to use each collection type, their performance considerations, and the distinction between **mutable** and **immutable** collections.

When to Use Each Collection Type

Each built-in collection type in Python is optimized for specific use cases.

Lists (list) – Ordered and Mutable

Use a **list** when:

- The order of elements matters.
- Frequent modifications (appending, inserting, removing) are needed.
- Elements may contain duplicates.
- Random access by index is required.

Example:

```
tasks = ["write report", "email client", "prepare slides"]

tasks.append("meet with team")  # Adding an item

print(tasks[0])  # Accessing an item
```

Tuples (tuple) – Ordered and Immutable

Use a **tuple** when:

- The collection should be **read-only**.
- Performance optimization is needed (tuples are slightly faster than lists).

- The data should be **hashable** (e.g., using it as a dictionary key).

Example:

coordinates = (40.7128, -74.0060) # Immutable geographic coordinates

Dictionaries (dict) – Key-Value Mapping

Use a **dictionary** when:

- Data should be accessed by a unique **key** instead of an index.
- Fast lookups are needed (average O(1) time complexity).
- It is important to store structured data efficiently.

Example:

employee = {"name": "Alice", "position": "Engineer", "salary": 70000}

print(employee["position"]) # Fast lookup

Sets (set) – Unordered and Unique Elements

Use a **set** when:

- Duplicates should be automatically removed.
- Fast membership testing is needed (O(1) time complexity).
- Mathematical set operations (union, intersection, difference) are required.

Example:

unique_numbers = {1, 2, 3, 3, 4} # Duplicates are removed

print(2 in unique_numbers) # Fast membership test

Frozensets (frozenset) – Immutable Sets

Use a **frozenset** when:

- A **set** needs to be **immutable** and hashable.
- It should be used as a dictionary key or stored inside another set.

Example:

immutable_set = frozenset([1, 2, 3])

Bytes, Bytearray, and Memoryview – Binary Data Handling

Use **bytes, bytearray, or memoryview** when:

- Working with **binary data** (e.g., file I/O, networking).
- Performance optimizations for low-level operations are needed.

Example:

byte_data = b"hello" # Immutable bytes

```
mutable_byte_data = bytearray(b"hello")  # Mutable bytes
```

Performance Considerations for Common Operations

Different collections have varying efficiency for common operations. The table below summarizes their average time complexities:

Operation	List (list)	Tuple (tuple)	Dict (dict)	Set (set)	Frozenset (frozenset)
Indexing (O(1))	✅ Fast	✅ Fast	❌ Not applicable	❌ Not applicable	❌ Not applicable
Appending (O(1))	✅ Fast	❌ Not possible	❌ Not applicable	✅ Fast	❌ Not possible
Insertion (O(n))	❌ Slow	❌ Not possible	✅ Fast	❌ Not applicable	❌ Not applicable
Lookup (O(n))	❌ Slower	❌ Slower	✅ Fast (O(1))	✅ Fast (O(1))	✅ Fast (O(1))
Deletion (O(n))	❌ Slower	❌ Not possible	✅ Fast (O(1))	✅ Fast (O(1))	❌ Not possible
Iteration (O(n))	✅ Fast	✅ Faster	✅ Fast	✅ Fast	✅ Fast

Key observations:

- **Tuples are faster** than lists for iteration but **cannot be modified**.
- **Dictionaries and sets provide the fastest lookups** due to hash table implementation.
- **Lists are best for maintaining order**, while **sets are best for uniqueness**.

Immutability vs. Mutability

Python collections fall into two categories: **mutable** and **immutable**.

Mutable Collections (Can Be Modified)

- list
- dict
- set
- bytearray

These collections allow adding, removing, and modifying elements.

Example:

data = [1, 2, 3]

data.append(4) # ✅ Allowed

data[0] = 99 # ✅ Allowed

Immutable Collections (Cannot Be Modified After Creation)

- tuple
- frozenset
- bytes

These collections do **not allow modification** after they are created, making them **thread-safe** and **hashable** (usable as dictionary keys).

Example:

coords = (10, 20)

coords[0] = 99 # ❌ TypeError: 'tuple' object does not support item assignment

When to Choose Mutable vs. Immutable Collections

- Use **mutable collections** when **modifications** are required (e.g., dynamic lists, dictionaries).
- Use **immutable collections** for **data integrity and performance** (e.g., fixed records, function return values).

Choosing the right collection type depends on whether order, mutability, uniqueness, or performance is the priority. Lists are useful for sequential data, dictionaries for key-value mappings, and sets for unique elements with fast lookups. Tuples and frozensets provide immutable alternatives when modification is not needed. Understanding these trade-offs helps in writing efficient and scalable Python programs.

Summary

Python provides a variety of built-in collections designed to store and manipulate data efficiently. These collections can be categorized into sequences, sets, mappings, and binary data structures, each offering different features based on ordering, mutability, and uniqueness. Understanding these types is essential for writing efficient and readable Python code.

Sequences are ordered collections that support indexing and slicing. Lists are mutable sequences that allow dynamic resizing, making them useful for handling variable-sized data. In contrast, tuples are immutable, ensuring data remains constant throughout the program. Lists should ideally contain elements of the same type to ensure clarity and consistency, whereas tuples can be used to group heterogeneous data logically.

Mappings provide key-value pairs for fast lookups. The dictionary is Python's primary mapping type, allowing efficient data retrieval using unique, hashable keys. Dictionaries support dynamic updates, making them useful for structured data storage such as configurations and records.

Sets, on the other hand, store unique, unordered elements and support mathematical set operations like union and intersection. They automatically remove duplicates and provide fast membership

testing. A regular set is mutable, whereas a frozenset is its immutable counterpart, suitable for use as dictionary keys or elements within other sets.

Python also includes specialized types for handling binary data. Bytes are immutable sequences of byte values, often used for storing binary files. Bytearray provides a mutable alternative, allowing modifications to byte sequences, while memoryview enables efficient access to binary data without creating copies.

Lists follow the CRUD pattern, supporting creation through direct initialization or the list constructor, reading via indexing and slicing, updating by modifying elements, and deletion using various removal methods. Negative indexing allows convenient access to elements from the end of a list, seamlessly integrating with slicing operations. Tuples, while immutable, can be reassigned or concatenated to create modified versions. Dictionaries offer efficient key-based data retrieval, easy updates, and multiple methods for removing elements. Sets allow fast additions and deletions, and their mathematical operations make them useful for data comparisons.

When choosing a collection type, it is important to consider its properties. Lists are best for maintaining order and handling sequential data, whereas tuples provide a lightweight, immutable alternative. Dictionaries are ideal for key-based lookups, while sets excel at uniqueness enforcement. Performance considerations also play a role, as operations like indexing, lookup, and iteration vary in efficiency across different collection types.

Nested collections allow the combination of lists, dictionaries, and sets to create complex data structures. Lists of dictionaries are commonly used for structured records, while dictionaries of lists enable grouping elements under a shared key. Sets within dictionaries help track unique associations, and dictionaries of dictionaries provide hierarchical data storage.

Choosing the right collection depends on whether order, mutability, uniqueness, or efficiency is the priority. Lists provide flexibility, dictionaries offer fast lookups, and sets enforce uniqueness, while tuples and frozensets ensure immutability when needed. By understanding these collections and their trade-offs, Python developers can write more efficient and maintainable code.

Exercises

1. Write a Python program that creates a list of five city names and prints the second and last city using both positive and negative indexing.

2. Create a list containing the numbers 1 to 5. Add the number 6 at the end, insert 0 at the beginning, and replace the third element with 99. Print the modified list.

3. Define a tuple with three elements: a string, an integer, and a float. Attempt to modify the second element and observe the result. Then, use an appropriate workaround to create a modified version of the tuple.

4. Create a dictionary representing a student with keys "name", "age", and "grades", where "grades" is a list of numbers. Print the student's name, add a new grade to the list, and update the age by one year.

5. Write a program that defines two sets: set_A = {1, 2, 3, 4} and set_B = {3, 4, 5, 6}. Perform and print the results of the following operations:

- Union of both sets
- Intersection of both sets
- Difference between set_A and set_B
- Check whether the number 2 is in set_B

6. Create a list of dictionaries where each dictionary represents a book with keys "title", "author", and "year". Write a function that prints the titles of all books published after the year 2000.
7. Write a Python program that initializes a dictionary with country names as keys and their capitals as values. Allow the user to input a country name and return the capital if it exists, handling the case where the country is not in the dictionary.
8. Create a dictionary where the keys are programming languages ("Python", "Java", "C++") and the values are sets of people who know that language. Add a new person to the "Python" set and remove someone from "C++". Ensure no duplicate names exist.
9. Create a nested dictionary representing employees in a company, where each key is an employee ID and the value is another dictionary with "name", "position", and "salary". Implement functions to add a new employee, update a salary, and delete an employee by ID.
10. Write a Python script that measures the time taken to perform **lookups** in a list, a dictionary, and a set containing the same elements. Use the timeit module to compare lookup speeds and explain the differences in execution time.