Unit 2: Lists and Dictionaries

1. Lists

A list in Python is a collection of items that are **ordered**, **mutable** (can be changed), and allow **duplicate elements**. Lists are one of the most commonly used data types in Python and are define using square brackets `[]`.

1.1 Key Features of Lists

- **Ordered**: The items have a defined order that will not change unless you explicitly modify the li
- **Mutable**: You can add, remove, or change elements after the list is created.
- **Heterogeneous**: A list can contain elements of different data types (e.g., integers, strings, floa or even other lists).
- **Dynamic**: Lists in Python can grow or shrink dynamically as you add or remove elements.
- **Indexed**: Each element in a list can be accessed using its index, starting from `0` for the first element.

1.2 Working with Lists

1.2.1 Creating a List

You can create a list by placing a comma-separated sequence of elements inside square brackets.

```
# Examples of lists
empty_list = []                # An empty list
fruits = ["apple", "banana", "cherry"]  # A list of strings
numbers = [1, 2, 3, 4, 5]      # A list of integers
mixed = [1, "hello", 3.14, True]  # A mixed data type list
nested_list = [[1, 2], [3, 4]]  # A nested list
```

1.2.2 Accessing Elements

You can access list elements using their index. Negative indexing allows you to access elements fro the end of the list.

```
fruits = ["apple", "banana", "cherry"]

print(fruits[0])  # Output: apple (first element)
print(fruits[-1]) # Output: cherry (last element)
```

1.2.3 Adding Elements

- `append()` : Adds a single element to the end.
- `extend()` : Adds multiple elements to the end.
- `insert()` : Adds an element at a specific index.

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

1.2.4 Removing Elements

- `remove()` : Removes the first occurrence of a specific value.
- `pop()` : Removes an element at a specified index (or the last element if no index is provided).
- `clear()` : Removes all elements from the list.

```
numbers = [1, 2, 3, 4, 5]
numbers.remove(3)    # [1, 2, 4, 5]
numbers.pop()        # [1, 2, 4] (removes the last element)
numbers.pop(1)       # [1, 4] (removes element at index 1)
numbers.clear()      # [] (empty list)
```

1.3 List Built-in Methods

Method	Description	Example
`append(item)`	Adds an item to the end of the list.	`numbers = [1, 2, 3];` `numbers.append(4)` → `[1` `2, 3, 4]`
`extend(iterable)`	Adds all elements of an iterable (e.g., another list) to the end of the list.	`numbers = [1, 2];` `numbers.extend([3, 4])` → `[1, 2, 3, 4]`
`insert(index, item)`	Inserts an item at the specified index.	`numbers = [1, 3];` `numbers.insert(1, 2)` → `[1, 2, 3]`
`remove(item)`	Removes the first occurrence of the specified item.	`numbers = [1, 2, 3, 2];` `numbers.remove(2)` → `[1` `3, 2]`
`pop(index=-1)`	Removes and returns the item at the specified index (default is the last item).	`numbers = [1, 2, 3];` `numbers.pop()` → `[1, 2]`
`clear()`	Removes all items from the list.	`numbers = [1, 2, 3];` `numbers.clear()` → `[]`
`index(item, start, end)`	Returns the index of the first occurrence of the item between optional `start` and `end` indices.	`numbers = [1, 2, 3];` `numbers.index(2)` → `1`
`count(item)`	Returns the number of occurrences of the specified item.	`numbers = [1, 2, 2, 3];` `numbers.count(2)` → `2`
`sort(key=None, reverse=False)`	Sorts the list in ascending order (or descending if `reverse=True`).	`numbers = [3, 1, 2];` `numbers.sort()` → `[1, 2` `3]`
`reverse()`	Reverses the order of the list.	`numbers = [1, 2, 3];` `numbers.reverse()` → `[3` `2, 1]`
`copy()`	Returns a shallow copy of the list.	`numbers = [1, 2, 3];` `copy = numbers.copy()` – `[1, 2, 3]`

```
numbers = [5, 3, 8]
numbers.append(10)   # [5, 3, 8, 10]
numbers.sort()        # [3, 5, 8, 10]
print(numbers.pop())  # Output: 10
```

1.3.1 Special Features of Lists

- **Slicing**: Extract a portion of a sequence (like a list or string) using indices. It's done using the colon (:) operator.

```
nums = [10, 20, 30, 40, 50]
print(nums[1:4])  # Output: [20, 30, 40]
```

- **Negative Indexing**: Negative indexing allows you to access elements from the end of a sequence. The last element is -1, the second-to-last is -2, and so on.

```
print(nums[-1])  # Output: 50
```

2. Tuples

Tuples are a type of **sequence data structure** that are used to store collections of items. Unlike lis tuples are **immutable**, meaning their contents cannot be changed after creation.

2.1 Key Features of Tuples

- **Immutable**: Once a tuple is created, you cannot add, remove, or modify its elements.
- **Ordered**: Elements in a tuple maintain their order.
- **Allows duplicates**: Tuples can contain duplicate elements.
- **Can hold multiple data types**: A tuple can store a mix of integers, strings, lists, other tuples, etc.
- **Parentheses** `()`: Tuples are defined using parentheses, although commas (`,`) are the actua tuple separators.

2.2 Tuple Operators

2.2.1 Creating Tuples

You can create a tuple using parentheses `()` or the `tuple()` constructor.

```
# Example of tuple creation
t1 = (1, 2, 3)
t2 = tuple([4, 5, 6])  # Using the tuple() constructor
```

2.2.2 Accessing Elements

You can access elements using indexing and slicing.

- **Indexing:** Retrieves a single element.
- **Slicing:** Retrieves a sub-part of the tuple.

```
t = (10, 20, 30, 40, 50)

# Accessing elements by index
print(t[0])  # Output: 10
print(t[-1]) # Output: 50

# Accessing a range of elements (slicing)
print(t[1:4])  # Output: (20, 30, 40)
```

2.2.3 Concatenation (`+`)

Tuples can be combined using the `+` operator.

```
t1 = (1, 2)
t2 = (3, 4)
print(t1 + t2)  # Output: (1, 2, 3, 4)
```

2.2.4 Repetition (`*`)

You can repeat tuples using the `*` operator.

```
print(t1 * 2)   # Output: (1, 2, 1, 2)
```

2.2.5 Membership Testing

Check if an item exists in a tuple using the `in` or `not in` operators.

```
print(2 in t1)  # Output: True
print(5 not in t2) # Output: True
```

2.2.6 Iterating Through Tuples

You can iterate through a tuple using a `for` loop.

```
for item in t1:
    print(item)
```

2.3 Tuples Built-in Functions

Function	Description	Example	Outpu
`len()`	Returns the number of elements in the tuple.	`len((1, 2, 3))`	`3`
`max()`	Returns the maximum value in the tuple.	`max((10, 20, 30))`	`30`
`min()`	Returns the minimum value in the tuple.	`min((10, 20, 30))`	`10`
`sum()`	Returns the sum of all elements in the tuple (only for numeric elements).	`sum((1, 2, 3))`	`6`
`sorted()`	Returns a sorted list of the tuple elements (does not modify the original tuple).	`sorted((3, 1, 2))`	`[1, 2, 3]`
`tuple()`	Converts an iterable (like a list, string, or range) into a tuple.	`tuple([1, 2, 3])`	`(1, 2, 3)`
`all()`	Returns `True` if all elements in the tuple are `True` (or the tuple is empty).	`all((True, 1, "text"))`	`True`
`any()`	Returns `True` if at least one element in the tuple is `True`.	`any((False, 0, "text"))`	`True`
`enumerate()`	Returns an enumerate object that yields pairs of index and value for each element in the tuple.	`list(enumerate(("a", "b", "c")))`	`[(0, 'a'), (1, 'b'), (2, 'c')]`
`reversed()`	Returns a reverse iterator for the tuple.	`tuple(reversed((1, 2, 3)))`	`(3, 2, 1)`
`zip()`	Combines elements from multiple iterables into tuples.	`list(zip((1, 2), ('a', 'b')))`	`[(1, 'a'), (2, 'b')]`

Notes:

- These functions do not modify the tuple itself since tuples are immutable.

- Some functions (e.g., `max`, `min`, `sum`) are applicable only if all elements in the tuple are numeric or comparable.

2.4 Special Features of Tuples

Feature	Description
Immutability	Tuples cannot be modified after creation, ensuring data integrity.
Faster Than Lists	Tuples are faster due to immutability and fixed size.
Hashable	Can be used as keys in dictionaries or elements in sets if elements are hashable.
Heterogeneous Data	Can store elements of different data types.
Nested Tuples	Support for multi-level tuples.
Memory Efficient	Consume less memory compared to lists.
Packing & Unpacking	Group multiple values or assign tuple elements to variables.
Basic Operations	Support indexing, slicing, concatenation, repetition, and membership testing.
Readability	Often used to return multiple values from functions.
Immutable Constants	Ideal for storing fixed, unchanging data.
Lexicographical Comparison	Can be compared based on element order.

3. Dictionaries

A dictionary is a collection data type that stores data in key-value pairs. It is mutable, unordered (pr to Python 3.7), and allows for fast access, modification, and storage of data using unique keys.

Dictionaries are often used to represent real-world data mappings, such as a telephone directory or student database.

3.1 Key Features of Dictionaries

- **Key-Value Pairs:** Each element in a dictionary consists of a unique **key** and its correspondinc **value**.
    - Example: `{"name": "Sandy", "age": 20}`

- **Mutable:** You can modify the dictionary by adding, updating, or deleting elements.
- **Keys Must Be Immutable:** Keys can be of types like strings, numbers, or tuples (if they conta only immutable elements), but not lists or other dictionaries.
- **Unordered (Up to Python 3.6):** Dictionaries did not maintain the order of insertion before Python 3.7. From Python 3.7 onwards, dictionaries maintain insertion order.
- **Fast Lookups:** Values can be quickly retrieved by referencing their keys.

3.2 Working with Dictionaries

3.2.1 Creating a Dictionary

You can create a dictionary using curly braces `{}` or the `dict()` constructor.

```
# Using curly braces
student = {"name": "Sandy", "age": 20, "course": "BCA"}

# Using dict() constructor
info = dict(name="Sandy", age=20, course="BCA")
```

3.2.2 Accessing Elements

Values in a dictionary can be accessed using their keys.

```
# Accessing values
print(student["name"])  # Output: Sandy

# Using the get() method (avoids KeyError if key doesn't exist)
print(student.get("age"))  # Output: 20
print(student.get("grade", "Not Assigned"))  # Output: Not Assigned
```

3.2.3 Modifying a Dictionary

You can add, update, or delete key-value pairs.

```
# Adding a new key-value pair
student["grade"] = "A"
print(student)  # Output: {'name': 'Sandy', 'age': 20, 'course': 'BCA', 'grade':
'A'}

# Updating an existing key-value pair
student["age"] = 22
print(student)  # Output: {'name': 'Sandy', 'age': 22, 'course': 'BCA', 'grade':
'A'}

# Removing a key-value pair
del student["course"]
print(student)  # Output: {'name': 'Sandy', 'age': 22, 'grade': 'A'}

# Using pop() method
grade = student.pop("grade")
print(grade)  # Output: A
print(student)  # Output: {'name': 'Sandy', 'age': 22}
```

3.3 Built-in Functions for Dictionaries

These functions can be applied to dictionaries.

Function	Description	Example	Output
`len(dict)`	Returns the number of key-value pairs in the dictionary.	`len({"a": 1, "b": 2})`	`2`
`max(dict)`	Returns the largest key (based on comparison).	`max({"a": 1, "b": 2})`	`'b'`
`min(dict)`	Returns the smallest key (based on comparison).	`min({"a": 1, "b": 2})`	`'a'`
`sorted(dict)`	Returns a sorted list of the keys.	`sorted({"b": 1, "a": 2})`	`['a', 'b']`
`dict()`	Creates a new dictionary.	`dict([("a", 1), ("b", 2)])`	`{'a': 1, 'b': 2}`

3.4 Built-in Methods for Dictionaries

These methods are specifically designed for dictionary objects.

Method	Description	Example	Output
`dict.keys()`	Returns a view object containing all keys in the dictionary.	`{"a": 1, "b": 2}.keys()`	`dict_keys(['a', 'b'])`
`dict.values()`	Returns a view object containing all values in the dictionary.	`{"a": 1, "b": 2}.values()`	`dict_values([1, 2])`
`dict.items()`	Returns a view object of key-value pairs as tuples.	`{"a": 1, "b": 2}.items()`	`dict_items([('a' 1), ('b', 2)])`
`dict.get(key[, default])`	Returns the value associated with the key; if the key doesn't exist, returns the specified default value.	`{"a": 1}.get("b", 0)`	`0`
`dict.update(other)`	Updates the dictionary with key-value pairs from another dictionary or an iterable of pairs.	`{"a": 1}.update({"b": 2})`	`{'a': 1, 'b': 2}`

Method	Description	Example	Output
`dict.pop(key[, default])`	Removes the key and returns its value; if the key doesn't exist, returns the default value.	`{"a": 1}.pop("a")`	`1`
`dict.popitem()`	Removes and returns the last inserted key-value pair (from Python 3.7+).	`{"a": 1, "b": 2}.popitem()`	`('b', 2)`
`dict.clear()`	Removes all key-value pairs from the dictionary.	`{"a": 1, "b": 2}.clear()`	`{}`
`dict.copy()`	Returns a shallow copy of the dictionary.	`{"a": 1, "b": 2}.copy()`	`{'a': 1, 'b': 2}`
`dict.fromkeys(iterable[, value])`	Creates a new dictionary with keys from the iterable and values set to the specified value.	`dict.fromkeys(["a", "b"], 0)`	`{'a': 0, 'b': 0}`

Method	Description	Example	Output
`dict.setdefault(key[, default])`	Returns the value of the key if it exists; if not, inserts the key with the default value and returns it.	`{"a": 1}.setdefault("b", 2)`	`2` (and dictionary becomes `{'a': 1, 'b': 2}`)

3.5 Dictionary Keys

Dictionary keys are the identifiers used to access values in a dictionary. They are unique, immutabl and serve as the primary way to organize and retrieve data in a dictionary.

3.5.1 Key Properties

Property	Description
Uniqueness	Duplicate keys are not allowed; the latest value replaces the old one.
Immutable Types	Keys must be of immutable types like strings, numbers, or tuples.
Fast Lookup	Keys enable fast value retrieval using hash-based lookups.
Case Sensitivity	Keys are case-sensitive (e.g., `"key"` and `"KEY"` are different keys).## 4. Sets

3.5.2 Creating Keys

Keys can be of various immutable data types:

```
# String keys
string_key_dict = {"name": "Sandy"}

# Numeric keys
numeric_key_dict = {1: "one", 2: "two"}

# Tuple keys
tuple_key_dict = {(1, 2): "coordinates"}
```

4. Sets in Python

A **set** is an unordered, mutable collection of unique elements. It is defined using curly braces `{}` or the built-in `set()` function.

They are commonly used to store distinct values and perform mathematical set operations like unio intersection, and difference.

4.1 Characteristics of Sets

- **Unordered**: Elements do not maintain a specific order.
- **Unique**: Duplicate elements are not allowed.
- **Mutable**: You can add or remove items from a set, but the elements themselves must be immutable (e.g., numbers, strings, tuples).

4.2 Working with Sets

4.2.1 Creating Sets

- Using curly braces:

```python
my_set = {1, 2, 3, 4}
print(my_set)  # Output: {1, 2, 3, 4}
```

- Using the `set()` constructor:

```python
my_set = set([1, 2, 3, 4])
print(my_set)  # Output: {1, 2, 3, 4}
```

- Creating an empty set:

```python
empty_set = set()  # Correct way
```

4.2.2 Adding Elements

Use `add()` to add a single element.

```python
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}
```

4.2.3 Removing Elements

- `remove()` raises an error if the item does not exist.
- `discard()` does not raise an error if the item is absent.

```python
my_set = {1, 2, 3}
my_set.remove(2)   # Removes 2
my_set.discard(5)   # Does nothing
```

- `pop()` removes and returns an arbitrary element.

```python
my_set.pop()
print(my_set)   # Output: {2, 3} (arbitrary removal)
```

- `clear()` removes all elements.

```python
my_set.clear()
print(my_set)   # Output: set()
```

4.3 Common Set Methods

Method	Description	Example
`add(element)`	Adds a single element to the set.	`s = {1, 2}; s.add(3)` → `{1, 2, 3}`
`update(iterable)`	Adds multiple elements (from an iterable) to the set.	`s = {1, 2}; s.update([3, 4])` → `{1, 2, 3, 4}`
`remove(element)`	Removes the specified element; raises `KeyError` if the element is not found.	`s = {1, 2, 3}; s.remove(2)` → `{1, 3}`
`discard(element)`	Removes the specified element without error if it's not found.	`s = {1, 2}; s.discard(3)` → `{1, 2}`
`pop()`	Removes and returns an arbitrary element; raises `KeyError` if the set is empty.	`s = {1, 2, 3}; s.pop()` → Remaining elements: `{2, 3}`
`clear()`	Removes all elements from the set.	`s = {1, 2, 3}; s.clear()` → `set()`
`union(set)`	Returns a new set containing all elements from both sets.	`s1 = {1, 2}; s2 = {3, 4}; s1.union(s2)` → `{1, 2, 3, 4}`
`intersection(set)`	Returns a new set with elements common to both sets.	`s1 = {1, 2, 3}; s2 = {2, 3 4}; s1.intersection(s2)` → `{2, 3}`
`difference(set)`	Returns a new set with elements in the first set but not in the second.	`s1 = {1, 2, 3}; s2 = {2, 3 4}; s1.difference(s2)` → `{1}`
`symmetric_difference(set)`	Returns a new set with elements in either set, but not both.	`s1 = {1, 2, 3}; s2 = {3, 4}; s1.symmetric_difference(s2` → `{1, 2, 4}`
`issubset(set)`	Checks if the set is a subset of another.	`s1 = {1, 2}; s2 = {1, 2, 3}; s1.issubset(s2)` → `Tru`

Method	Description	Example
`issuperset(set)`	Checks if the set is a superset of another.	`s1 = {1, 2, 3}; s2 = {1, 2}; s1.issuperset(s2)` → `True`
`isdisjoint(set)`	Checks if two sets have no elements in common.	`s1 = {1, 2}; s2 = {3, 4}; s1.isdisjoint(s2)` → `True`
`copy()`	Returns a shallow copy of the set.	`s1 = {1, 2, 3}; s2 = s1.copy()` → `s2` = `{1, 2, 3}`

4.4 Comparing Sets

Sets can be compared using relational operators (`==` , `!=` , `<` , `>` , `<=` , `>=`).

4.4.1Equality (`==`)

Checks if two sets have the same elements, regardless of order.

```
set1 = {1, 2, 3}
set2 = {3, 2, 1}
print(set1 == set2)  # True
```

4.4.2 Inequality (`!=`)

Checks if two sets are not equal.

```
print(set1 != set2)  # False
```

4.4.3 Subset (`<`)

A set is a subset of another if all elements of the first set are in the second.

```
set1 = {1, 2}
set2 = {1, 2, 3}
print(set1 < set2)  # True
```

4.4.4 Superset (`>`)

A set is a superset of another if it contains all elements of the second set.

```
print(set2 > set1)  # True
```

4.4.5 Disjoint (`isdisjoint()`)

Two sets are disjoint if they have no elements in common.

```
set1 = {1, 2, 3}
set2 = {4, 5}
print(set1.isdisjoint(set2))  # True
```

4.5 Mathematical Set Operations

Python sets support operations like union, intersection, difference, and symmetric difference, simila mathematical sets.

- **Union (`|` or `union()`)**: Combines elements from both sets.

  ```
  set1 = {1, 2, 3}
  set2 = {3, 4, 5}
  print(set1 | set2)  # {1, 2, 3, 4, 5}
  ```

- **Intersection (`&` or `intersection()`)**: Finds common elements.

  ```
  print(set1 & set2)  # {3}
  ```

- **Difference (`-` or `difference()`)**: Elements in the first set but not in the second.

  ```
  print(set1 - set2)  # {1, 2}
  ```

- **Symmetric Difference (`^` or `symmetric_difference()`)**: Elements in either set but not bot

  ```
  print(set1 ^ set2)  # {1, 2, 4, 5}
  ```

4.6 Set Comprehensions

Set Comprehensions in Python provide a concise way to create sets based on existing iterables (lik lists, tuples, or other sets).

A set comprehension has a similar syntax to list comprehensions but produces a `set` as the outpu

Syntax

```
{expression for item in iterable if condition}
```

4.6.1 Key Points:

- **Set Properties**:
  - A set does not allow duplicate values.
  - The resulting set is unordered.
- **Purpose**:
  - To simplify the process of creating sets programmatically.
  - To make the code more readable and concise.

Example 1: Square of Numbers

```
squared_set = {x**2 for x in range(5)}
print(squared_set)  # Output: {0, 1, 4, 9, 16}
```

Example 2: Filtering with Conditions

```
even_set = {x for x in range(10) if x % 2 == 0}
print(even_set)  # Output: {0, 2, 4, 6, 8}
```

4.7 Use Cases of Sets

- Removing duplicates from a list:

  ```
  my_list = [1, 2, 2, 3, 4, 4]
  unique_set = set(my_list)
  print(unique_set)  # {1, 2, 3, 4}
  ```

- Checking for common elements:

  ```
  print(bool(set1 & set2))  # True if they share any elements
  ```