Topic 3: Tuples, Sets and Dictionaries

Learning Outcomes:-

(a) Identify concept of tuples, sets and dictionaries

COMPUTER PROGRAMMING

LECTURE NOTES

CP125

Table of Contents - Tuples

LECTURE NOTES

TUPLES

Definition of Tuple:-
A built-in data type in Python.   ***A sequence of comma separated values that can contain elements of different types***

- must be created with commas between values, and conventionally the sequence is surrounded by parentheses ( ).

```
tuple_1 = (2, 'a', 4.5)
```

tuple name

tuple item/ element

COMPUTER PROGRAMMING

LECTURE NOTES

CP125

TUPLES

Why use Tuple?
a) **Faster than lists** (because they are immutable and take less memory). b) **Useful for fixed collections** (like coordinates, database records). c) **Can be used as dictionary keys** (unlike lists, since tuples are immutable).

Characteristics of Tuples

1 Tuples are Ordered

- Elements in a tuple **maintain their order**.
- This means indexing and slicing work **consistently**.

```
fruits = ("apple", "banana", "cherry")
print(fruits[0])   # Output: apple
print(fruits[1])   # Output: banana
print(fruits[2])   # Output: cherry
```

- Each element is accessed by index, starting with the first element at index 0.
- The order of elements **does not change** unless explicitly reassigned to another variable.

Characteristics of Tuples

2 Tuples are Immutable

- Once created, **tuples cannot be modified** (no addition, deletion, or updating of elements).
- This makes tuples **more secure** and **faster** than lists.

```
numbers = (1, 2, 3, 4)
numbers[1] = 10   # ❌ TypeError: 'tuple' object does not support item assignment
```

- **Fix:** To modify a tuple, convert it into a list, modify it, and convert it back.

Characteristics of Tuples

3 Tuples Allow Duplicate Element

- Tuples **can have duplicate values**.

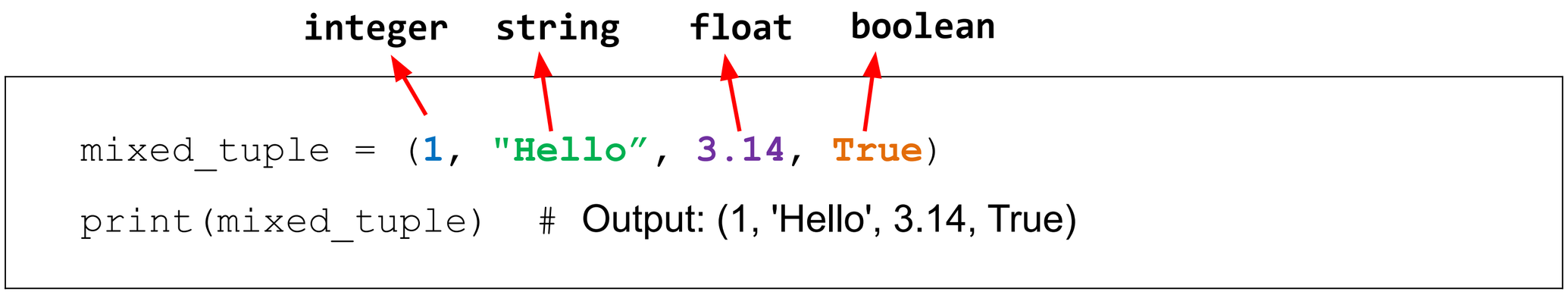
```
numbers = (10, 20, 10, 30, 40, 20)
print(numbers)   # Output: (10, 20, 10, 30, 40, 20)
```

- The values **10** and **20** appear multiple times without any issues.

Characteristics of Tuples

4 Tuples Support Multiple Data Types

- Tuples can store elements of **different data types**.



- The tuple contains an **integer (1)**, **string ("Hello")**, **float (3.14)**, and **boolean (True)**.

Characteristics of Tuples

5 Tuples are Indexable

- We can access tuple elements using **positive** and **negative indexing**.

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

print(colors[1])    # Output: blue (Positive Indexing)
print(colors[-1])   # Output: yellow (Negative Indexing)
```

- **Positive indexing** starts from 0. e.g. [0] – red, [1] – blue, [2] – green, [3] - yellow
- **Negative indexing** starts from -1 (last element). e.g. [-1] – yellow, [-2] green, [-3] – blue, [-4]- red

Characteristics of Tuples

6 Tuples are Memory Efficient and Faster than Lists

- Because tuples are **immutable**, Python **optimizes** their storage and access speed.
- This makes them **faster than lists** when performing lookups

```
import sys


list_data = [1,2,3,4,5,6,7,8,9,10]

tuple_data = (1,2,3,4,5,6,7,8,9,10)
```

Characteristics of Tuples

6 **Tuples are Memory Efficient and Faster than Lists**

```
print(sys.getsizeof(list_data))   # Output: Larger memory usage
print(sys.getsizeof(tuple_data)) # Output: Smaller memory usage

Output
136 #bytes
120 #bytes
```

- **Tuples consume less memory** than lists, making them **more efficient**.

Summary of Tuples

Feature	Description
Ordered	Elements maintain their position
Immutable	Cannot be modified after creation
Allows Duplicates	Can contain duplicate elements
Supports Multiple Data Types	Can store integers, strings, floats, etc.
Indexable	Can access elements using positive/negative indexing
Memory Efficient	Uses less memory than lists

Review Activity : Characteristics of Tuples

Try this !  **Exercise 1**

Which of the following statements is <u>true</u> about tuples in Python?

a) Tuples are mutable

b) Tuples use square brackets []

c) Tuples can store different data types

d) Tuples do not allow duplicate values

Review Activity : Characteristics of Tuples

Try this !  **Exercise 2**

What happens if you try to modify an element in a tuple?

a) The element is successfully changed

b) A TypeError occurs because tuples are immutable

c) The element is removed from the tuple

d) The element is replaced with None

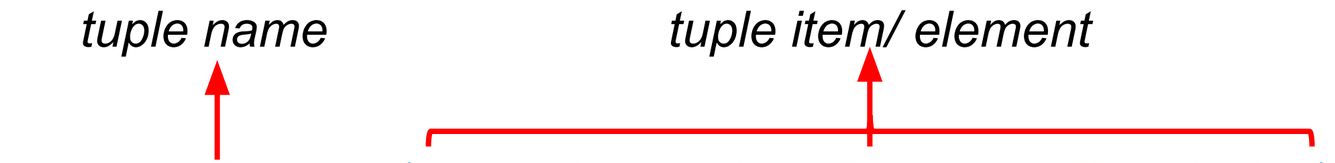
CRUD Operations

1 CREATE (C) – Creating a Tuple

Syntax

```
# Creating a tuple using parentheses □ ( )
tuple_name = (tuple_item1/element1, tuple_item2/element2,
tuple_item3/element3, ...)
```

Example




```
my_tuple = (10, 20, 30, "Python", 3.14)

print(my_tuple)                    # Output: (10, 20, 30, 'Python', 3.14)
```

CRUD Operations

1 CREATE (C) – Creating a Tuple

Syntax

```
#  Creating a tuple using the tuple() function
tuple_name = tuple([tuple item1/element1, tuple item2/element2,
tuple item3/element3, ...])
```

Example

tuple name *tuple function* *tuple item/ element*

```
new_tuple = tuple([100, 200, 300])
print(new_tuple)    #  Output: (100, 200, 300)
```

**function name is followed by parentheses () which may contain parameters*

CRUD Operations

1 CREATE (C) – Creating a Tuple

Tuple Type	Syntax Example
Empty Tuple	`empty_tuple = ()` or `empty_tuple = tuple()`
Single Element Tuple	`single_element_tuple = (5,)` ** you have to add a comma after the item, otherwise Python will not recognize it as a tuple.*
Mixed Data Type Tuple	`mixed_tuple = ("Hello", 100, 3.14, True)`

Review Activity : Creating a Tuple

Try this !  **Exercises**

1. Write a Python program to create a tuple named **`vehicles`** that contains the following values: `"car", "motorcycle", "bicycle"`, and `"bus"`. Then, print the tuple.
2. Write a Python program to create an **empty tuple**, a **tuple with a single element** (number 5), and a **tuple with multiple data types** (`"Hello", 10, 3.14, True`). Print all the tuples.

CRUD Operations

2 READ (R) – Accessing Tuple Elements

Indexing

- Tuples are indexed staring from 0.
- can access elements by specifying their index inside square brackets.

Syntax

```
# Accessing elements using indexing
tuple_name = (tuple_item1/element1, tuple_item2/element2,
 tuple_item3/element3, ...)
print(tuple_name[index])
```

CRUD Operations

2 READ (R) – Accessing Tuple Elements

Indexing

Example

```
my_tuple = (10, 20, 30, "Python", 3.14)

# Indexing
print(my_tuple[0])   # Output: 10  index start with 0
print(my_tuple[-1])  # Output: 3.14
```

Understanding -1 Index:

- -1 refers to the **last** element in the sequence.
- -2 refers to the **second last** element, and so on.

CRUD Operations

2 READ (R) – Accessing Tuple Elements

Slicing

- Can access a range of elements by specifying a **start** and **end index** .
- The **start index** is inclusive, while the **end index** is exclusive

Syntax

```
# Accessing elements using slicing
tuple_name = (tuple_item1/element1, tuple_item2/element2,
 tuple_item3/element3, ...)
print(tuple_name[start:end:step])
```

CRUD Operations

2 READ (R) – Accessing Tuple Elements

Slicing

Example

```
my_tuple = (10, 20, 30, "Python", 3.14)

print(my_tuple[1:4])    # Output: (20, 30, 'Python')
```

- **my_tuple[1:4]** extracts elements from index 1 (inclusive) to 4 (exclusive).

 start:stop, # This will return the items from **position 1 to 3.** item in **position 4 is NOT included**

CRUD Operations

② READ (R) – Accessing Tuple Elements

Slicing

Example

```
my_tuple = (10, 20, 30, "Python", 3.14)

print(my_tuple[:3])      # Output: (10, 20, 30) → First 3 elements
```

- **Omitting start** → Defaults to 0 (beginning of tuple)

  #returns the items **from** the **beginning** to INDEX 2 only, **NOT included**, **“Python"**:

CRUD Operations

② READ (R) – Accessing Tuple Elements

Slicing

Example

```
my_tuple = (10, 20, 30, "Python", 3.14)

print(my_tuple[3:])     # Output: (Python, 3.14) → From index 3 to end
```

- **Omitting end** → Defaults to the length of the tuple (until the last element).

CRUD Operations

2 READ (R) – Accessing Tuple Elements

Slicing

Example

```
my_tuple = (10, 20, 30, "Python", 3.14)

print(my_tuple[-3:-1])     # Output: (30, Python) → From index -3 to index -1
```

- Negative indexing means **starting** from the **end** of the tuple.
- This example returns the items from index **-3 (included) to index -1 (excluded)**
- Remember that the **last item** has the index **-1**

CRUD Operations

2 READ (R) – Accessing Tuple Elements

Looping – `for` loop

Syntax

```
for item in tuple_name:
    # Process each item
```

CRUD Operations

2 READ (R) – Accessing Tuple Elements

Looping – `for` loop

Example

```
my_tuple = (10, 20, 30, 40)


for item in my_tuple:
    print(item)              # Output : 10
                                      20
                                      30
                                      40
```

Review Activity : Accessing Tuple Elements

Try this ! 

1.
```
animals = ("lion", "tiger", "elephant", "giraffe")
```

What will be the output of the following statement?

```
print(animals[-2])
```

2.
```
numbers = (5, 15, 25, 35, 45, 55)
```

What will be the output of the following statement?

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

Review Activity : Accessing Tuple Elements

Try this !

3.
```
colors = ("red", "blue", "green", "yellow", "purple", "orange")
```

What will be the output of the following slicing operation?

```
print(colors[-5:-2])
```

CRUD Operations

3 UPDATE (U) – - Modifying a Tuple (Not Directly Possible)

- Since tuples are **immutable**, they cannot be modified directly.
- However, we can create a **new tuple** with updated values.

1. **Reassigning a Tuple (Creating a New Tuple)**
2. **Converting Tuple to List (Mutable Workaround)**
3. **Concatenation (Adding Elements)**
4. **Removing an Element (Using List Conversion)**

CRUD Operations

3 UPDATE (U) – - Modifying a Tuple (Not Directly Possible)

Reassigning a Tuple (Creating a New Tuple)

- Since tuples cannot be modified directly, you can create a **new tuple** with updated values:

Example

```
my_tuple = (10, 20, 30)
my_tuple = (new_value)          # Creating a new tuple
print(my_tuple)                 # Output: (10, 25, 30) - display the latest tuple
items
```

CRUD Operations

3 UPDATE (U) – - Modifying a Tuple (Not Directly Possible)

Converting Tuple to List (Mutable Workaround)

- To modify values, **convert the tuple to a list**, update it, and convert it back to a tuple:

Syntax

```
my_tuple = (tuple_item1/element1, tuple_item2/element2,
 tuple_item3/element3, ...)
temp_list = list(my_tuple)          # Convert to list
temp_list[index] = new_value        # Modify value
my_tuple = tuple(temp_list)         # Convert back to tuple
print(my_tuple)                     # Output
```

CRUD Operations

3 UPDATE (U) – - Modifying a Tuple (Not Directly Possible)

Converting Tuple to List (Mutable Workaround)

Example

```
my_tuple = (10, 14, 17, 22)
temp_list = list(my_tuple)          # Convert to list
temp_list[2] = 99                   # Modify value
my_tuple = tuple(temp_list)         # Convert back to tuple
print(my_tuple)                     # Output : 10, 14, 99, 22
```

CRUD Operations

3 UPDATE (U) – - Modifying a Tuple (Not Directly Possible)

Concatenation (Adding Elements)

- Since tuples cannot be changed, we can **concatenate** tuples to simulate updating:

Syntax

```
new_tuple = old_tuple + (new_value,)
```

CRUD Operations

3 UPDATE (U) – Modifying a Tuple (Not Directly Possible)

Concatenation (Adding Elements)

Example

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

my_tuple = my_tuple + (5,)   # Adding element 5 □ The extra comma (,)
                                                    ensures that Python treats
 it as a tuple.
print(my_tuple)                    # Output: (1, 2, 3, 5)
```

Review Activity : Modifying a Tuple (Not Directly Possible)

Try this !  **Task 1**

Given the following tuple:

```
my_tuple = (5, 10, 15, 20)
```

1. Why does the following code result in an error?

```
my_tuple[1] = 50
```

2. How can we correctly update the second element `(10)` to `50`?
3. Correct Python code to update the tuple:

Review Activity : Modifying a Tuple (Not Directly Possible)

Try this !  **Task 2**

Given the following tuple:

```
my_tuple = (5, 10, 15, 20)
```

How to add 25 at the end of the tuple?

CRUD Operations

4 DELETE (D) – Removing Elements from a Tuple

Deleting the Entire Tuple

- Since tuples are immutable, elements cannot be deleted, but the **entire tuple can be deleted**.

Syntax

```
# Deleting a tuple
del tuple_name
```

CRUD Operations

4 DELETE (D) – Removing Elements from a Tuple

Deleting the Entire Tuple

Example

```
# Deleting a tuple
my_tuple = (10, 20, 30)
del my_tuple   # Deletes the entire tuple


# print(my_tuple)   # This will raise a NameError since the tuple is deleted
```

CRUD Operations

4 DELETE (D) – Removing Elements from a Tuple

Removing an Element (Using List Conversion)

- To remove an element, convert it to a list first.

Syntax

```
temp_list = list(my_tuple)
temp_list.remove(value)   # OR del temp_list[index]
my_tuple = tuple(temp_list)
```

CRUD Operations

4 DELETE (D) – Removing Elements from a Tuple

Removing an Element (Using List Conversion)

Example

```
temp_list = list((1, 2, 3, 4))      # Convert tuple to list
temp_list.remove(2)                 # Remove element
new_tuple = tuple(temp_list)        # Convert back to tuple
print(new_tuple)                    # Output: (1, 3, 4)
```

Review Activity : Removing Elements from a Tuple

Try this !  **Task 1**

Given the following tuple:

```
animals = ("cat", "dog", "rabbit", "parrot", "fish")
```

1. Try to remove `"rabbit"` using the remove() method. What happens?
2. Explain why tuples do not support direct deletion of elements.
3. Provide a correct approach to remove "`rabbit`" and print the updated tuple.

COMPUTER PROGRAMMING

LECTURE NOTES

CP125

TUPLES

List vs Tuple

Feature	List	Tuple
Mutable	✅ Yes (Can be modified)	❌ No (Immutable)
Syntax	[ ] (Square brackets)	( ) (Parentheses)
Performance	Slower (due to mutability)	Faster (due to immutability)
Memory Usage	Uses more memory	Uses less memory
Methods Available	Many built-in methods (e.g., append(), remove(), sort())	Limited methods (count(), index())
Use Case	When you need to modify data frequently	When you need fixed data that won't change

Summary

- Definition
- Purpose of Using Tuple
- Characteristics
- CRUD Operation – Creating
- CRUD Operation – Reading
- CRUD Operation – Updating
- CRUD Operation – Deleting

Topic 3: Tuples, Sets and Dictionaries

Learning Outcomes:-

(a) Identify concept of tuples, sets and dictionaries

COMPUTER PROGRAMMING

LECTURE NOTES

CP125

Table of Contents - Sets

Definition of Set:-
A collection of elements or objects of unique values enclosed in a pair of curly brackets.

- To create a set with initial elements, you can specify the elements enclosed in braces, just like in mathematics:

```
my_set = {1, 3, 4.5}
```

set name

set item/ element

curly bracket

LECTURE NOTES

SETS

- Like sets in mathematics, members in a set are unordered and unindexed.
- Sets do not allow duplicate values, making them useful for storing distinct items.
- A set is created using a set literal or the set function.
- Alternatively, you can use the set function to convert any sequence into a set:

```
numb = [1, 3, 4.5]  # A list
my_set = set(numb)  # converts the numb list into a set using the set() function.
```

Why use Set?

a) To store **unique elements** efficiently.

b) To perform **mathematical set operations** like union, intersection, and difference.

c) To enable **fast membership testing** (in keyword).

d) To remove duplicates from a list easily.

Characteristics of Sets

1 Unordered (No Fixed Order)

- Sets do **not** maintain the order of elements.
- Unlike lists and tuples, items in a set may appear in a different order each time you access or print them.

```
my_set = {10, 20, 30, 40}
print(my_set)   # Output: {40, 10, 20, 30} (Order may vary)
```

- ✅ **Implication:** You **cannot** use indexing or slicing with sets.

Characteristics of Sets

2 **Mutable** (Changeable)

- You can **add** or **remove** elements after creating a set.
- However, **set elements themselves must be immutable** (e.g., numbers, strings, tuples).

```
my_set = {1, 2, 3}
my_set.add(4)       # Adding an element
my_set.remove(2)    # Removing an element

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

- 🚫 **Note:** You **cannot** add lists or dictionaries to a set because they are mutable.

Characteristics of Sets

3 Unique Elements (No Duplicates)

- A set automatically removes duplicate values when created.
- This ensures all elements inside a set are distinct.

```
my_set = {1, 2, 2, 3, 4, 4, 5}
print(my_set)   # Output: {1, 2, 3, 4, 5}  (Duplicates removed)
```

✅ **Implication:** Sets are useful for eliminating duplicate data from lists or other collections.

Characteristics of Sets

4 No Indexing or Slicing

- Unlike lists or tuples, sets do **not** support indexing.
- Since sets are unordered, individual elements **cannot** be accessed by an index.

```
my_set = {10, 20, 30}
print(my_set[0])   # ❌ TypeError: 'set' object is not subscriptable
```

✅ **Implication:** Use loops (for item in `my_set`) or membership tests (in) to work with set elements

Characteristics of Sets

5 Supports Mathematical Set Operations

- Sets allow operations like **union, intersection, difference**, and **symmetric difference**.
- These operations make sets powerful when working with group-based data.
-

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

print(A | B)   # Union: {1, 2, 3, 4, 5, 6}
print(A & B)   # Intersection: {3, 4}
print(A - B)   # Difference: {1, 2}
print(A ^ B)   # Symmetric Difference: {1, 2, 5, 6}
```

Characteristics of Sets

6 Can Store Different Data Types

- Sets can store **multiple types of immutable data** (e.g., integers, floats, strings, tuples).

```
my_set = {1, 3.5, "Hello", (2, 4)}
print(my_set)   # Output: {1, 3.5, 'Hello', (2, 4)}
```

Note: Lists and dictionaries cannot be stored inside sets.

✅ **Implication:** Sets allow diverse data types but require immutability.

Review Activity : Characteristics of Sets

Try this !  **Exercise 1**

Which of the following is true about Python sets?

A) Sets allow duplicate values

B) Sets are ordered collections

C) Sets are mutable

D) Set elements can be accessed by index

Review Activity : Characteristics of Sets

Try this !  **Exercise 2**

Which statement accurately describes the ordering of elements in a Python set?

A) Elements in a set are stored in the order they were added.

B) Elements in a set are sorted in ascending order.

C) Elements in a set are unordered; their arrangement is arbitrary.

D) Elements in a set can be accessed using an index.

Review Activity : Characteristics of Sets

Try this !  **Exercise 3**

Considering Python sets, which of the following operations is valid?

A) Accessing an element by its index.

B) Adding a duplicate element to the set.

C) Removing an element from the set.

D) Slicing a subset of elements from the set.

CRUD Operations

1 CREATE (C) – Creating a Set

- A set in Python is created using curly braces { } or the set( ) constructor.

Syntax

```
# Creating a set using curly braces/ define set
set_name = {element1, element2, element3, ...}

# Creating an empty set using set() (not {})
set_name = set()
```

CRUD Operations

1 CREATE (C) – Creating a Set

- A set in Python is created using curly braces { } or the set( ) constructor.

Example

```
# Creating a set
my_set = {1, 2, 3, "Python", 3.14}
print(my_set)   # Output: {1, 2, 3, 'Python', 3.14}


# Creating an empty set
empty_set = set()
print(type(empty_set))   # Output: <class 'set'>
```

Review Activity : Creating Set

Try this !  **Exercise 1**

Which of the following is the correct way to create an empty set in Python?

A) my_set = {}

B) my_set = set()

C) my_set = []

D) my_set = set([])

Review Activity : Creating Set

Try this ! **Exercise 2**

What will be the output of the following code?

```
my_list = [1, 2, 3, 4, 4, 5]
my_set = set(my_list)
print(my_set)
```

CRUD Operations

② READ (R) – Accessing Set Elements

- Sets do **not support indexing** because they are unordered. We must use iteration.

Syntax

```
# Iterating through a set
for element in set_name:
    print(element)

# Checking membership
if value in set_name:
    print("Exists")
```

CRUD Operations

2 **READ (R) –** Accessing Set Elements

Example

 # Prints elements in random order

```
my_set = {1,2,3, "Python", 3.14}

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

```
1
2
3.14
3
Python
```

CRUD Operations

2 **READ (R) –** Accessing Set Elements

Example

 # Check existence using `in`

```
my_set = {1,2,3, "Python", 3.14}

print ("Python" in my_set)
```

```
True
```

```
my_set = {1,2,3, "Python", 3.14}

print (4.15 in my_set)
```

```
False
```

Review Activity : Assessing Set Elements

Try this ! **Exercise 1**

Given the set below, write Python code to print all elements:

```
colors = {'red', 'green', 'blue'}
```

Review Activity : Assessing Set Elements

Try this ! **Exercise 2**

What will be the output of the following code?

```
fruits = {'apple', 'banana', 'cherry'}
print('banana' in fruits)
```

CRUD Operations

3 UPDATE (U) – Modifying Set

- can **add** elements, **remove** elements, or **update** a set with another collection.

Syntax

```
# Adding a single element
set_name.add(value)

# Adding multiple elements
set_name.update([value1, value2, ...])
```

CRUD Operations

3 UPDATE (U) – Modifying Set

Example

```
my_set = {1, 2, 3, "Python", 3.14}

my_set.add(100)
print(my_set)                    # Output: {1, 2, 3, 'Python', 3.14, 100}


my_set.update([200, 300])
print(my_set)                    # Output: {1, 2, 3, 'Python', 3.14, 100, 200,
 300}
```

Review Activity : Modifying Set

Try this ! **Exercise 1**

Which method is used to add a single element to a set in Python?

A) append()

B) add()

C) insert()

D) extend()

Review Activity : Modifying Set

Try this !  **Exercise 2**

What is the purpose of the `update()` method in sets?

A) It replaces all elements in the set.

B) It adds a single element to the set.

C) It adds multiple elements from an iterable to the set.

D) It deletes elements from the set.

CRUD Operations

4 DELETE (D) – Removing Elements

There are multiple ways to remove elements from a set:

`remove( )` – Removes a specific element, raises an error if not found.

```
# Removing a specific element (Raises error if element not found)
set_name.remove(value)


my_set = {10, 20, 30, 40, 50}


# Remove an element (raises KeyError if not found)
my_set.remove(30)
print(my_set)   # Output: {10, 20, 40, 50}
```

CRUD Operations

4 DELETE (D) – Removing Elements

`discard( )` – Removes a specific element but does NOT raise an error if not found.

```
# Removing a specific element (No error if element not found)
set_name.discard(value)


my_set = {10, 20, 30, 40, 50}


# Discard an element (no error if not found)
my_set.discard(100)
print(my_set)   # Output: {10, 20, 40, 50}
```

CRUD Operations

4 DELETE (D) – Removing Elements

`pop()` – Removes a random element.

```
# Removing a random element
set_name.pop()


my_set = {10, 20, 30, 40, 50}


# Remove a random element
removed_element = my_set.pop()
print("Removed:", removed_element)
print(my_set)
```

CRUD Operations

4 DELETE (D) – Removing Elements

`clear( )` – Removes all elements.

```
# Clearing all elements from the set
set_name.clear()

my_set = {10, 20, 30, 40, 50}

# Clear all elements
my_set.clear()
print(my_set)   # Output: set()
```

Review Activity : Removing Elements

Try this ! **Exercise 1**

What will be the output of the following code?

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

A) {1, 2, 3}
B) {1, 3}
C) {2, 3}
D) Error

Review Activity : Removing Elements

Try this ! **Exercise 2**

What will be the output of the following code?

```
my_set = {1, 2, 3}
my_set.discard(4)
print(my_set)
```

A) {1, 2, 3}
B) {1, 2}
C) {1, 2, 3, 4}
D) Error

Review Activity : Removing Elements

Try this ! **Exercise 3**

What does the pop() method do in a set?

```
my_set = {10, 20, 30}

value = my_set.pop()

print(value)
```

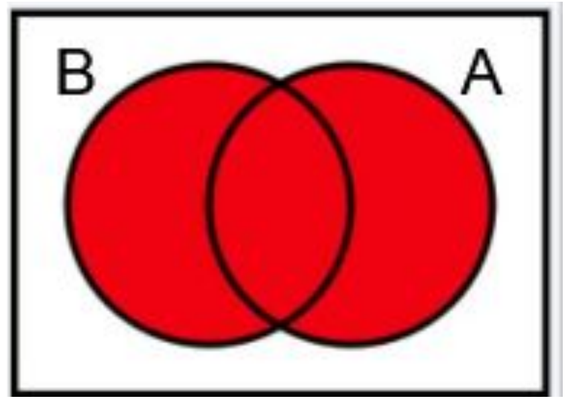
A) Removes and returns the smallest element
B) Removes and returns the largest element
C) Removes and returns an arbitrary element
D) Raises an error

Other Operations

METHODS	Description
`union()` `update()`	joins all items from both sets.
`intersection()`	keeps **ONLY** the duplicates.
`difference()`	keeps the items from the first set that are not in the other set(s).
`symmetric_difference()`	keeps all items EXCEPT the duplicates.

Mathematical Operations

1. Set Union (| or union( ))



- combines all elements from both sets without duplicates.

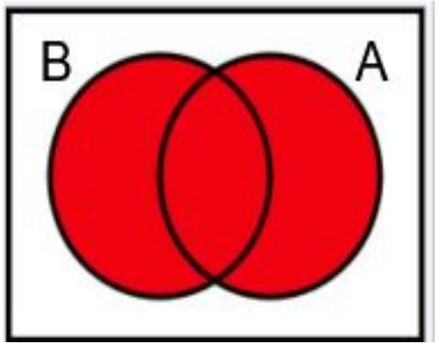
Syntax

```
A | B

A.union(B)  # alternative method
```

- A and B are sets.
- The | operator returns a new set containing all elements from A and B, **without duplicates**.
- Same as when write `.union( ).`

Mathematical Operations

1. Set Union (| or union( ))



Examples

Using `|`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A | B)
```

```
{1, 2, 3, 4, 5}
```

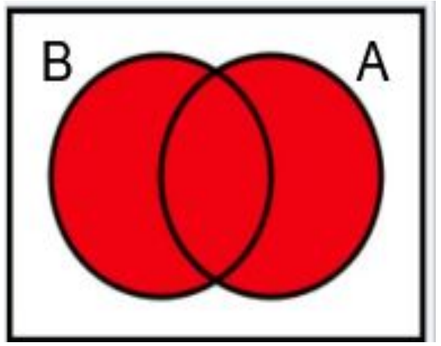
Using `.union( )`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A.union(B))
```

```
{1, 2, 3, 4, 5}
```

Mathematical Operations

1. Set Union (| or union( ))



Example

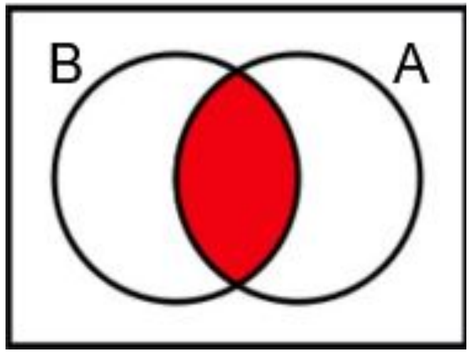
```
A = {1, 2, 3, "apple"}
B = {"banana", 3, 4, 5}

union_result = A|B
print("Union", union_result)
```

```
Union {1, 2, 3, 4, 5, 'banana', 'apple'}
```

Mathematical Operations

2 Set Intersection (& or intersection( ))

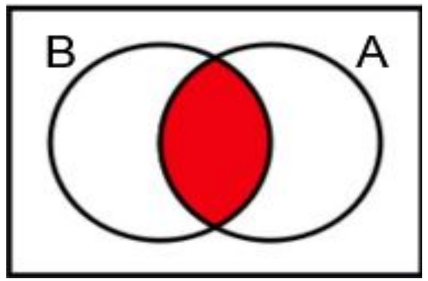
- The **intersection** of two sets includes only elements present in both sets.



Syntax

```
intersect_set = set1 & set2
# or
set1.intersection(set2)
```

Mathematical Operations



2 Set Intersection (& or intersection( ))

Examples

Using `**&**`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A & B)
```

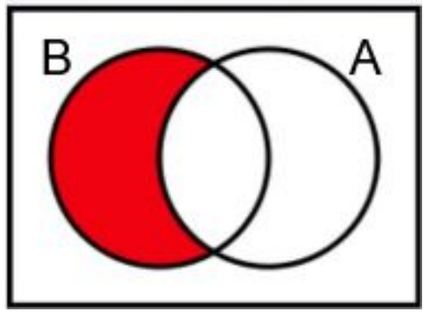
```
{3}
```

Using `**.intersection( )**`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A.intersection(B))
```

```
{3}
```

Mathematical Operations



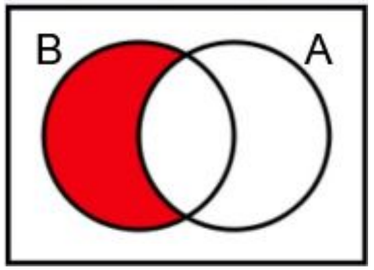
3 Set Difference (- or difference( ))

- The **difference** of two sets includes elements that are in the first set but **not** in the second set.

Syntax

```
set1 - set2

# or

set1.difference(set2)
```

Mathematical Operations



3 Set Difference (- or difference( ))

Using `-`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A - B)
```

```
{1, 2}
```

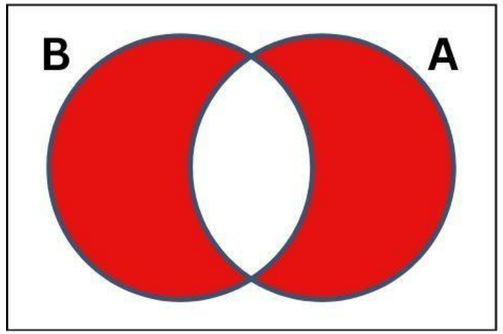
Example

Using `.difference( )`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A.difference(B))
```

```
{1, 2}
```

Mathematical Operations



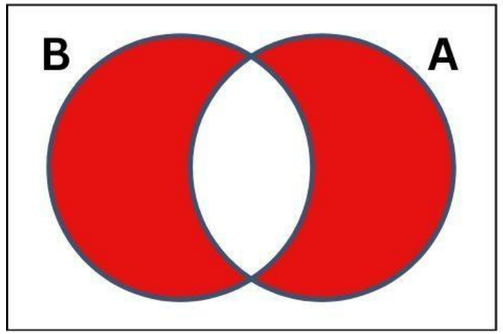
4. Symmetric Difference (^ or .symmetric_difference( ))

- includes elements that are in **either** of the sets but **not in both**.

Syntax

```
set1 ^ set2
# or
set1.symmetric_difference(set2)
```

Mathematical Operations



4 Symmetric Difference (^ or .symmetric_difference( ))

Example

Using `^`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A ^ B)
```

```
{1, 2, 4, 5}
```

Using `.symmetric_difference( )`

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A.symmetric_difference(B))
```

```
{1, 2, 4, 5}
```

Review Activity : Other Operations in a Set

Try this ! **Exercise 1**

What will be the output of the following code?

```
A = {2, 4, 6}
B = {1, 2, 3}
print(A | B)
```

A) {1, 2, 3, 4, 6}
B) {1, 2, 3, 4, 6, 2}
C) {2, 4, 6}
D) {1, 3}

Review Activity : Other Operations in a Set

Try this !  **Exercise 2**

Which of the following options correctly represents the output of this Python code?

```
X = {10, 20, 30, 40}

Y = {30, 40, 50, 60}

print(X & Y)
```

A) {10, 20, 30, 40, 50, 60}
B) {30, 40}
C) {50, 60}
D) { }

Review Activity : Other Operations in a Set

Try this ! **Exercise 3**

What will be printed when executing the following code?

```
A = {5, 10, 15, 20}

B = {10, 20, 30, 40}

print(A - B)
```

A) {5, 15}
B) {10, 20}
C) {30, 40}
D) {}

Review Activity : Other Operations in a Set

Try this !  **Exercise 4**

What will be the result of the following code?

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

B = {3, 4, 5}

print(A ^ B)
```

A) {1, 2, 3, 4, 5}
B) {1, 2, 4, 5}
C) {3}
D) { }

LECTURE NOTES

Operation	Tuple	Set
Accessing elements	Indexing ([ ])	No indexing (only iteration)
Adding elements	Not possible (immutable)	add( )
Removing elements	Not possible (immutable)	remove(), discard(), pop()
Concatenation	+ operator	Not supported
Repetition	* operator	Not supported
Checking existence	in, not in	in, not in
Mathematical operations	Not supported	union(), intersection(), difference(), symmetric_difference()

Table of Contents - Sets

Topic 3: Tuples, Sets and Dictionaries

Learning Outcomes:-

(a) Identify concept of tuples, sets and dictionaries

COMPUTER PROGRAMMING

LECTURE NOTES

CP125

Table of Contents – Dictionaries part 1

LECTURE NOTES

DICTIONARY

Definition of Dictionary:-

A collection of key and value pairs enclosed in curly brackets.

- These pairs are sometimes called entries.
- The entire sequence of entries is enclosed in curly braces ({ and }).
- A colon (:) separates a key and its value.
- Each dictionary entry consists of a **key** (identifier) and its corresponding **value**.

Syntax

```
dictionary_name = {
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
}
```

COMPUTER PROGRAMMING

LECTURE NOTES

CP125

DICTIONARY

Example 1

```
A phone book: {"Savannah":"476-3321", "Nathaniel":"351-7743"}
```

Dictionary_name — key — value — key — value

Example 2

```
student = {
   "name": "Ali",
   "age": 20,
   "course": "Computer Science"
}

print(student["name"])   # Output: Ali
```

- Each key must be **unique and immutable** (e.g., strings, numbers, or tuples), while the values can be of any data type.

COMPUTER PROGRAMMING

LECTURE NOTES

CP125

DICTIONARY

Purpose of Use Dictionaries:-

a) Need **fast lookups** (O(1) average time complexity for accessing values by keys).

b) To store **associative data** (e.g., storing student records, product details).

c) Need a **flexible and mutable** data structure for data manipulation.

Characteristics of Dictionaries

1 Ordered

- Dictionaries now maintain **insertion order** by default.
- The order in which items are added is the same as the order when iterating over the dictionary.

```
my_dict = {"b": 2, "a": 1, "c": 3}

print(my_dict)

# Output: {'b': 2, 'a': 1, 'c': 3}  (Order is preserved)
```

Characteristics of Dictionaries

2 Mutable (Can be Modified)

- Unlike tuples, dictionaries **can be changed** after creation.
- You can **add**, **update**, and **delete** elements dynamically.

```
student = {"name": "Alice", "age": 20}
student["age"] = 21          # Update existing key
student["grade"] = "A"       # Add new key-value pair
print(student)               # {'name': 'Alice', 'age': 21, 'grade': 'A'}
```

Characteristics of Dictionaries

3 Heterogeneous Values (Different Data Types Allowed)

- Dictionary values can hold **integers, strings, lists, other dictionaries, or even functions**.

```
data = {
    "name": "Alice",
    "scores": [95, 88, 92],                          # List as a value
    "details": {"city": "New York", "age": 21}   # Nested dictionary
}
print(data["scores"][1])          # Output: 88
print(data["details"]["city"])    # Output: New York
```

Characteristics of Dictionaries

Characteristic	Description
Ordered (Python 3.7+)	Maintains the order in which items were inserted.
Mutable	Can be modified (add, update, delete elements).
Supports Various Data Types	Keys must be immutable (e.g., strings, numbers, tuples), values can be any data type.

Review Activity : Characteristics of Dictionary

Try this !  **Exercise 1**

Starting from Python 3.7, what characteristic of dictionaries ensures that items maintain the order in which they were inserted?

A) Immutable

B) Ordered

C) Unordered

D) Static

Review Activity : Characteristics of Dictionary

Try this !  **Exercise 2**

Which of the following data types can be used **as keys** in a Python dictionary?

A) Lists

B) Strings

C) Sets

D) Dictionaries

Review Activity : Characteristics of Dictionary

Try this !  **Exercise 3**

What does it mean when we say that dictionaries in Python are **mutable**?

A) Their keys cannot be changed after creation

B) Their values can be modified, added, or removed

C) They always maintain a fixed size

D) They cannot store different data types

CRUD Operations

1 CREATE (C) – Adding Elements to a Dictionary

There are multiple ways to create a dictionary and add elements to it.

Creating an Empty Dictionary

```
my_dict = {}          # Empty dictionary
my_dict = dict()      # Using the dict() constructor
print(my_dict)        # Output: {}
```

CRUD Operations

1 CREATE (C) – Adding Elements to a Dictionary

There are multiple ways to create a dictionary and add elements to it.

Creating a Dictionary with Initial Values

Syntax

```
dictionary_name = {key1: value1, key2: value2, ...}
```

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}
print(student)   # Output: {'name': 'Alice', 'age': 20, 'grade': 'A'}
```

CRUD Operations

1 CREATE (C) – Adding Elements to a Dictionary

There are multiple ways to create a dictionary and add elements to it.

Adding New Key-Value Pairs

Example

```
student = {"name": "Alice", "age": 20, "course": "CS"}

student["grade"] ="A" # Adds a new key-value pair
print(student)
```

```
{'name': 'Alice', 'age': 20, 'course': 'CS', 'grade': 'A'}
```

CRUD Operations

1 CREATE (C) – Adding Elements to a Dictionary

There are multiple ways to create a dictionary and add elements to it.

Using the update( ) Method to Add Multiple Entries

Syntax

```
dictionary_name.update({key: value})
print(dictionary_name)
```

CRUD Operations

1 CREATE (C) – Adding Elements to a Dictionary

There are multiple ways to create a dictionary and add elements to it.

Using the update( ) Method to Add Multiple Entries

Example

```
student = {"name": "Alice", "age": 20, "course": "CS"}
student.update ({"city": "New York", "GPA": 3.8})
print(student)
```

```
{'name': 'Alice', 'age': 20, 'course': 'CS', 'city': 'New York', 'GPA': 3.8}
```

Review Activity : Adding Elements to a Dictionary

Try this !  **Exercise 1**

1. What is the correct way to add a new key-value pair to an existing dictionary in Python?

   a) `my_dict.append("key", "value")`
   b) `my_dict["key"] = "value"`
   c) `my_dict.add("key": "value")`
   d) `my_dict.insert("key", "value")`

Review Activity : Adding Elements to a Dictionary

Try this !  **Exercise 2**

2. Given the following dictionary:

```
student_scores = {"Alice": 85, "Bob": 90}
```

Write a Python statement to add a new student, `"Charlie"`, with a score of `88` to the `student_scores` dictionary.

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Accessing Values Using Keys

- If the key does not exist, it raises a `KeyError`

Syntax

```
value = dictionary_name[key]    # Direct access
```

Example

```
student = {"name": "Alice", "age": 20, "course": "CS"}
value = student["name"]
print(value)
```

```
Alice
```

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Using get( ) to Avoid Errors

Syntax

```
value = dictionary_name.get(key) # Safe access
```

Example

```
student = {"name": "Alice", "age": 20, "course": "CS"}
info = student.get("age")
print(info)
```

```
20
```

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Using get( ) to Avoid Errors

Can use any suitable message

Syntax

```
value = dictionary_name.get(key, default_value) # Safe access
```

Example

```
student = {"name": "Alice", "age": 20, "course": "CS"}
info = student.get("status","Data not available")
print(info)
```

```
Data not available
```

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Getting All Keys, Values, and Items

Syntax

```
print(dictionary_name.keys())    # Output: display all dict_keys
print(dictionary_name.values())  # Output: display all dict_values
print(dictionary_name.items())   # Output: display all dict_items
```

Refer to both key and value

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Getting All Keys, Values, and Items (both keys & values)

Example

```
print(student.keys())
```

```
dict_keys(['name', 'age', 'course'])
```

```
print(student.values())
```

```
dict_values(['Alice', 20, 'CS'])
```

```
print(student.items())
```

```
dict_items([('name', 'Alice'), ('age', 20), ('course', 'CS')])
```

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Iterating Over a Dictionary

Syntax – Iterating Over Key

```
for key in dictionary_name:
    print(key)
```

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}

for key in student:
    print(key)
```

```
name
age
grade
```

** The loop goes through each key in the dictionary and prints it.*

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Iterating Over a Dictionary

Syntax – Iterating Over Value

```
for value in dictionary_name.values():
    print(value)
```

Example

** The `.values()` method retrieves only the values from the dictionary*

```
student = {"name": "Alice", "age": 20, "grade": "A"}

for value in student.values():
    print(value)
```

```
Alice
20
A
```

CRUD Operations

2 READ (R) – Accessing Values from a Dictionary

Iterating Over a Dictionary

Syntax – Iterating Over Key-Value Pair

```
for key, value in dictionary_name.items():
    print(key, value)
```

Example

** The `.items()` method returns both keys and values, allowing iteration over key-value pairs.*

```
student = {"name": "Alice", "age": 20, "grade": "A"}

for key, value in student.items():
    print(key,value)
```

```
name Alice
age 20
grade A
```

Review Activity : Accessing Values from a Dictionary

Try this !  **Exercise 1**

What will be the output of the following code?

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

Review Activity : Accessing Values from a Dictionary

Try this !  **Exercise 2**

Which of the following methods is used to safely access a dictionary value without causing an error if the key does not exist?

A) `dict.fetch()`
B) `dict.access()`
C) `dict.get()`
D) `dict.value()`

Review Activity : Accessing Values from a Dictionary

Try this !  **Exercise 3**

What will be the output of the following code?

```
student = {"name": "Alice", "age": 20, "grade": "A"}
print(student.get("height", "Not Available"))
```

Review Activity : Accessing Values from a Dictionary

Try this !  **Exercise 4**

What will be the output of the following code?

```
student = {"name": "Alice", "age": 20, "grade": "A"}

for key in student:
    print(key)
```

Review Activity : Accessing Values from a Dictionary

Try this !  **Exercise 5**

Which of the following statements about iterating over a dictionary is **False**?

A) Using for key in dictionary: iterates over keys.
B) dictionary.items() returns key-value pairs as tuples.
C) dictionary.values() allows modifying values directly while iterating.
D) Using for key, value in dictionary.items(): iterates over both keys and values.

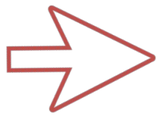
COMPUTER PROGRAMMING

LECTURE NOTES

CP125

Table of Contents - Dictionaries

LECTURE NOTES

Next Lecture  **Dictionary** – part 2

Topic 3: Tuples, Sets and Dictionaries

Learning Outcomes:-

(a) Identify concept of tuples, sets and dictionaries

Table of Contents - Dictionaries

LECTURE NOTES

- Recap

Definition of Dictionary:-
A collection of key and value pairs enclosed in curly brackets.

- These pairs are sometimes called entries.
- The entire sequence of entries is enclosed in curly braces ({ and }).
- A colon (:) separates a key and its value.
- Each dictionary entry consists of a **key** (identifier) and its corresponding **value**.

Syntax

```
dictionary_name = {
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
}
```

LECTURE NOTES

- Recap

Example 1

```
A phone book: {"Savannah":"476-3321", "Nathaniel":"351-7743"}
```

Dictionary_name — key — value — key — value

Example 2

```
student = {
   "name": "Ali",
   "age": 20,
   "course": "Computer Science"
}

print(student["name"])   # Output: Ali
```

- Each key must be **unique and immutable** (e.g., strings, numbers, or tuples), while the values can be of any data type.

CRUD Operations

3 UPDATE (U) – Modifying Existing Elements in a Dictionary

- **Updating a Single Value**

Syntax

```
dictionary_name[key] = new_value   # Updating a single value
```

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}

student ["age"] = 21 # modify an existing key-value
print(student ["age"])
```

```
21
```

CRUD Operations

3 UPDATE (U) – Modifying Existing Elements in a Dictionary

- **Adding a new key-value pair**

Syntax

```
dictionary_name[key] = value
# Adding a new key-value
```

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}

student ["city"] = "Putrajaya" # Adding a new key-value pair
print(student)
```

```
{'name': 'Alice', 'age': 20, 'grade': 'A', 'city': 'Putrajaya'}
```

CRUD Operations

3 **UPDATE (U) –** Modifying Existing Elements in a Dictionary

- **Using `update()` to Modify Multiple Values**

Syntax

```
dictionary_name.update({key1: new_value1, key2: new_value2})
```

- This updates an existing key's value or adds a new key-value pair.
- It can also update multiple values at once.

CRUD Operations

3 **UPDATE (U) –** Modifying Existing Elements in a Dictionary

- **Using `update()` to Modify Multiple Values**

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}

student.update({"grade": "A+", "country": "USA"})
print(student)
```

```
{'name': 'Alice', 'age': 20, 'grade': 'A+', 'country': 'USA'}
```

Review Activity : Modifying Existing Elements in a Dictionary

Try this ! **Exercise 1**

What will be the output of the following code?

```
student = {"name": "Alice", "age": 20, "grade": "B"}

# Updating an existing value
student["grade"] = "A"

# Adding a new key-value pair
student["city"] = "New York"

print(student)
```

Review Activity : Modifying Existing Elements in a Dictionary

Try this !  **Exercise 2**

Which of the following correctly **adds multiple key-value pairs** to a dictionary?

A.
```
student["age", "grade"] = 21, "A"
```

B.
```
student.update({"age": 21, "grade": "A"})
```

C.
```
student.add("age", 21)
  student.add("grade", "A")
```

D.
```
student["age": 21, "grade": "A"]
```

Review Activity : Modifying Existing Elements in a Dictionary

Try this !  **Exercise 3**

You have the following dictionary representing a book:

```
book = { "title": "Python Basics", "author": "John Doe",
"price": 50 }
```

a) Update the price of the book to **55**.

b) Print the updated dictionary.

Review Activity : Modifying Existing Elements in a Dictionary

Try this !  **Exercise 4**

Given the dictionary below:

```
employee = {"name": "Aisha","department": "IT","salary": 5000}
```

a) Add a new key "position" with the value "Software Engineer".

b) Print the updated dictionary.

Review Activity : Modifying Existing Elements in a Dictionary

Try this ! **Exercise 5**

You have the following dictionary representing a student:

```
student = {"name": "Ali","age": 20,"course": "Mathematics‟ }
```

a) Update the "age" to **21** and the "course" to "Computer Science" in one operation.

b) Print the updated dictionary.

CRUD Operations

4 DELETE (D) – Removing Elements from a Dictionary

- **Using `del` to Remove a Specific Key**

Syntax

```
del dictionary_name[key]    # Deletes a specific key
```

- Removes the specified key and its corresponding value.
- Raises a KeyError if the key does not exist.

CRUD Operations

4 DELETE (D) – Removing Elements from a Dictionary

- **Using `del` to Remove a Specific Key**

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}

del student ["grade"]
print(student)
```

```
{'name': 'Alice', 'age': 20}
```

***'grade' key is removed*

CRUD Operations

4 DELETE (D) – Removing Elements from a Dictionary

- **Using `pop( )` to Remove a Key and Return Its Value**

Syntax

```
removed_value = dictionary_name.pop(key, default_value)
```

- Removes the key and returns its value.
- If the key does not exist, it returns default_value (if provided); otherwise, it raises a KeyError.

CRUD Operations

4 DELETE (D) – Removing Elements from a Dictionary

- **Using `pop( )` to Remove a Key and Return Its Value**

Example

```
student = {"name": "Alice", "age": 20, "grade": "A", "city": "New York"}

removed_value = student.pop("age") # Removes "age" and returns its value
print(removed_value)
print(student)
```

```
20
{'name': 'Alice', 'grade': 'A', 'city': 'New York'}
```

CRUD Operations

4 DELETE (D) – Removing Elements from a Dictionary

- **Using `clear( )` to Remove All Elements**

Syntax

```
dictionary_name.clear()    # Removes all elements
```

- Removes all key-value pairs from the dictionary, making it an empty dictionary { }.

CRUD Operations

4 DELETE (D) – Removing Elements from a Dictionary

- **Using `clear( )` to Remove All Elements**

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}

student.clear()
print(student)
```

```
{}
```

Review Activity : Removing Elements from a Dictionary

Try this !  **Exercise 1**

What will be the output of the following code?

```
student = {"name": "Ali", "age": 20, "course": "Math"}

del student["age"]
print(student)
```

A) `{'name': 'Ali', 'age': 20, 'course': 'Math'}`
B) `{'name': 'Ali', 'course': 'Math'}`
C) `KeyError`
D) `{'name': 'Ali'}`

Review Activity : Removing Elements from a Dictionary

Try this !  **Exercise 2**

What will be stored in the variable removed_value after executing the following code?

```
car = {"brand": "Toyota", "model": "Camry", "year": 2022}
removed_value = car.pop("model")
print(removed_value)
```

A) `"Toyota"`
B) `"Camry"`
C) `{"brand": "Toyota", "year": 2022}`
D) `KeyError`

Review Activity : Removing Elements from a Dictionary

Try this !  **Exercise 3**

What happens when you apply the clear() method to a dictionary?

```
data = {"id": 101, "name": "John", "age": 25}

data.clear()

print(data)
```

A) `{'id': 101, 'name': 'John', 'age': 25}`
B) `{}`
C) `None`
D) `KeyError`

Other Operations

1. Checking Key Existence

Syntax

```
if key in dictionary_name:
    print("Key exists!")
```

- Checks if the key is present in the dictionary.
- Returns True if the key exists, otherwise False.

Other Operations

1. Checking Key Existence

Example

```
student = {"name": "Alice", "age": 20, "grade": "A"}

if "age" in student:
    print("Key 'age' exists in the dictionary!")
```

```
Key 'age' exists in the dictionary!
```

Review Activity : Checking Key Existence

Try this !  **Exercise 1**

What will be the output of the following code?

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

if "age" in person:
    print("Key exists!")
else:
    print("Key does not exist!")
```

Comparison of Dictionary, Tuple, and Set in Python

Feature	Dictionary (dict)	Tuple (tuple)	Set (set)
Definition	Key-value pairs	Ordered collection of elements	Unordered collection of unique elements
Mutable?	✅ Yes (values can change, keys cannot)	❌ No (immutable)	✅ Yes (elements can be added/removed)
Ordered?	✅ Yes (Python 3.7+)	✅ Yes	❌ No
Duplicates Allowed?	❌ No (keys must be unique)	✅ Yes	❌ No
Access Method	By key (dict[key])	By index (tuple[index])	Cannot access by index (only iteration)
Common Use Cases	Storing structured data (e.g., JSON, records)	Fixed data that should not change (e.g., coordinates, database records)	Unique collections (e.g., removing duplicates, set operations)

Review Activity : Comparison of Dictionary, Tuple, and Set

Try this !  **Exercise 1**

Which of the following statements is **TRUE** based on the characteristics of Dictionary, Tuple, and Set?

A) A tuple is mutable and allows modification after creation.
B) A dictionary allows duplicate keys.
C) A set is ordered and supports indexing.
D) A dictionary allows value modification, but keys must remain unique.

Review Activity : Comparison of Dictionary, Tuple, and Set

Try this !  **Exercise 2**

Which of the following statements correctly describes how elements are accessed in different data structures?

A) A dictionary accesses values using keys (dict[key]).
B) A tuple accesses values using keys.
C) A set allows direct access using an index like a list.
D) A dictionary and a set both support accessing elements by index.

Review Activity : Comparison of Dictionary, Tuple, and Set

Try this !  **Exercise 3**

Which of the following data structures allows duplicate elements?

A) Dictionary (dict)
B) Tuple (tuple)
C) Set (set)
D) Both Dictionary and Set

Table of Contents - Dictionaries