MODULE III

List, Tuples, Set and Dictionaries .

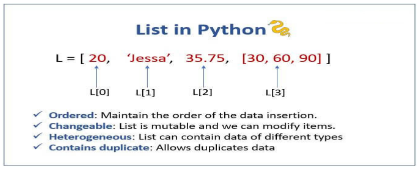
Python List: Introduction, accessing List, List operations, Working with Lists, List functions and methods

Python Tuple:- Introduction, accessing Tuple, operations on Tuple, Working with Tuple ,Functions and Methods,

Python Set - Introduction, accessing Set, Set operations, working with Set, Functions and Methods,

Python Dictionaries – Introduction, working with dictionaries, Properties, Functions. Dictionaries Operations, List Comprehension.

PYTHON LIST



List in Python

L = [20, 'Jessa', 35.75, [30, 60, 90]]

- ✓ **Ordered**: Maintain the order of the data insertion.
- ✓ **Changeable**: List is mutable and we can modify items.
- ✓ **Heterogeneous**: List can contain data of different types
- ✓ **Contains duplicate**: Allows duplicates data

Mutable: The elements of the list can be modified. We can add or remove items to the list after it has been created.

Ordered: The items in the lists are ordered. Each item has a unique index value. The new items will be added to the end of the list.

Heterogeneous: The list can contain different kinds of elements i.e; they can contain elements of string, integer, boolean, or any type.

Duplicates: The list can contain duplicates i.e., lists can have two items with the same values.

Why Use a List

•The **list data structure is very flexible** It has many unique inbuilt functionalities like **pop(), append()**, etc which makes it easier, where the data keeps changing.

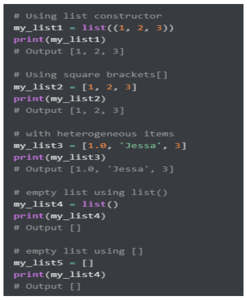
•The list can **contain duplicate elements** i.e two or more items can have the same values.

•Lists are **Heterogeneous i.e**, different kinds of objects/elements can be added.

•Lists are **mutable** it is used in applications where the values of the items change frequently.

List creation

- The list can be created using either the **list constructor list()** or using **square brackets [].**

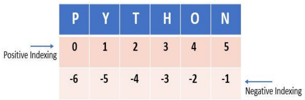


```
# Using list constructor
my_list1 = list((1, 2, 3))
print(my_list1)
# Output [1, 2, 3]

# Using square brackets[]
my_list2 = [1, 2, 3]
print(my_list2)
# Output [1, 2, 3]

# with heterogeneous items
my_list3 = [1.0, 'Jessa', 3]
print(my_list3)
# Output [1.0, 'Jessa', 3]

# empty list using list()
my_list4 = list()
print(my_list4)
# Output []

# empty list using []
my_list5 = []
print(my_list4)
# Output []
```

Length of a List : list1 = [1, 2, 3]

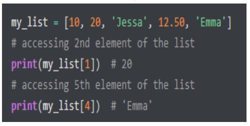
Print (**len** (list1))

Accessing items of a List : The items in a list can be **accessed through indexing** and **slicing**



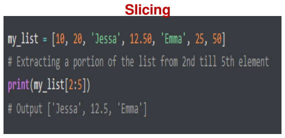
P	Y	T	H	O	N
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

indexing

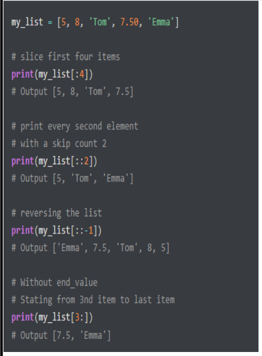


```
my_list = [10, 20, 'Jessa', 12.50, 'Emma']
# accessing 2nd element of the list
print(my_list[1])  # 20
# accessing 5th element of the list
print(my_list[4])  # 'Emma'
```

Slicing



```
my_list = [10, 20, 'Jessa', 12.50, 'Emma', 25, 50]
# Extracting a portion of the list from 2nd till 5th element
print(my_list[2:5])
# Output ['Jessa', 12.5, 'Emma']
```

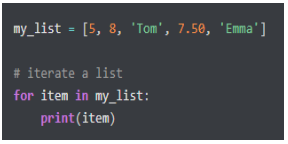


```
my_list = [5, 8, 'Tom', 7.50, 'Emma']

# slice first four items
print(my_list[:4])
# Output [5, 8, 'Tom', 7.5]

# print every second element
# with a skip count 2
print(my_list[::2])
# Output [5, 'Tom', 'Emma']

# reversing the list
print(my_list[::-1])
# Output ['Emma', 7.5, 'Tom', 8, 5]

# Without end_value
# Stating from 3nd item to last item
print(my_list[3:])
# Output [7.5, 'Emma']
```

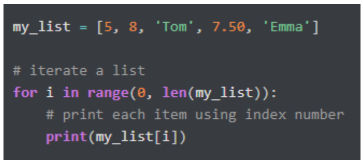
Examples of slicing a list:

- Extract a portion of the list
- Slicing with a step
- Reverse a list
- Slice without specifying start or end position

Iterating a List without and with index number



```
my_list = [5, 8, 'Tom', 7.50, 'Emma']

# iterate a list
for item in my_list:
    print(item)
```

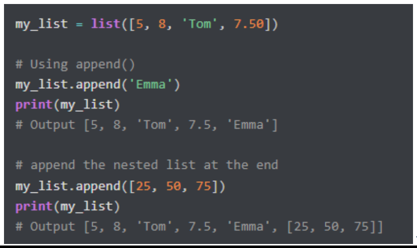


```
my_list = [5, 8, 'Tom', 7.50, 'Emma']

# iterate a list
for i in range(0, len(my_list)):
    # print each item using index number
    print(my_list[i])
```

Adding elements to the list

We can add a new **element / list** of elements to the list using the list methods such as **append() , insert() , and extend().**

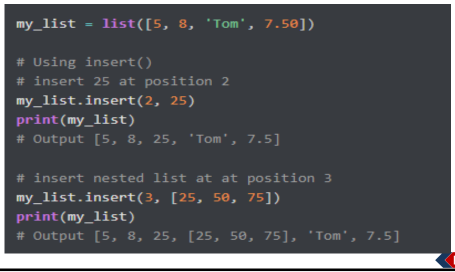
Append item at the end of the list: The **append()** method will accept **only one** parameter and add it at the **end o**f the list.



```
my_list = list([5, 8, 'Tom', 7.50])

# Using append()
my_list.append('Emma')
print(my_list)
# Output [5, 8, 'Tom', 7.5, 'Emma']

# append the nested list at the end
my_list.append([25, 50, 75])
print(my_list)
# Output [5, 8, 'Tom', 7.5, 'Emma', [25, 50, 75]]
```

Adding elements to the list

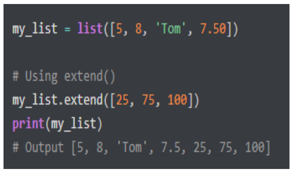
Add item at the specified position in the list: the **insert()** method to add the object / item at the **specified position** in the list. The insert method **accepts two parameters position and object.**



```
my_list = list([5, 8, 'Tom', 7.50])

# Using insert()
# insert 25 at position 2
my_list.insert(2, 25)
print(my_list)
# Output [5, 8, 25, 'Tom', 7.5]

# insert nested list at at position 3
my_list.insert(3, [25, 50, 75])
print(my_list)
# Output [5, 8, 25, [25, 50, 75], 'Tom', 7.5]
```

Adding elements to the list

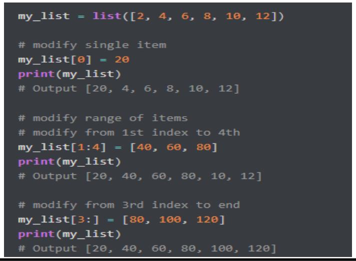
Using extend(): The **extend method** will accept the list of elements and add them at the **end of the list**. We can even add another list by using this method.



```
my_list = list([5, 8, 'Tom', 7.50])

# Using extend()
my_list.extend([25, 75, 100])
print(my_list)
# Output [5, 8, 'Tom', 7.5, 25, 75, 100]
```

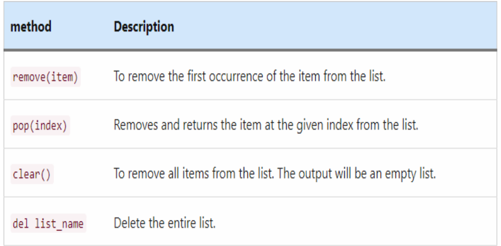
Modify the items of a List

- The list is a mutable sequence of **iterable objects**.
- It means we can modify the items of a list.
- Use the index number and assignment operator (=) to assign a new value to an item.
- **Modify the individual item** and **Modify the range of items**



```
my_list = list([2, 4, 6, 8, 10, 12])

# modify single item
my_list[0] = 20
print(my_list)
# Output [20, 4, 6, 8, 10, 12]

# modify range of items
# modify from 1st index to 4th
my_list[1:4] = [40, 60, 80]
print(my_list)
# Output [20, 40, 60, 80, 10, 12]

# modify from 3rd index to end
my_list[3:] = [80, 100, 120]
print(my_list)
# Output [20, 40, 60, 80, 100, 120]
```

Removing the items from the List

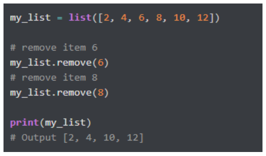


method	Description
`remove(item)`	To remove the first occurrence of the item from the list.
`pop(index)`	Removes and returns the item at the given index from the list.
`clear()`	To remove all items from the list. The output will be an empty list.
`del list_name`	Delete the entire list.

Removing Specific items from the List

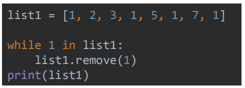
•Use the **remove() method** to remove **the first occurrence** of the item from the list.

• A **keyerror** is thrown if an item not present in the original list.

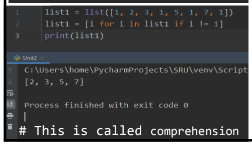


```
my_list = list([2, 4, 6, 8, 10, 12])

# remove item 6
my_list.remove(6)
# remove item 8
my_list.remove(8)

print(my_list)
# Output [2, 4, 10, 12]
```

Remove all occurrence of a specific item

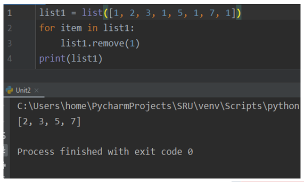


```
list1 = [1, 2, 3, 1, 5, 1, 7, 1]

while 1 in list1:
    list1.remove(1)
print(list1)
```



```
list1 = list([1, 2, 3, 1, 5, 1, 7, 1])
list1 = [i for i in list1 if i != 1]
print(list1)
```

```
C:\Users\home\PycharmProjects\SRU\venv\Script
[2, 3, 5, 7]

Process finished with exit code 0
|
# This is called comprehension
```



```
list1 = list([1, 2, 3, 1, 5, 1, 7, 1])
for item in list1:
    list1.remove(1)
print(list1)
```

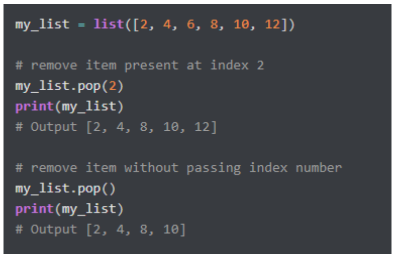
```
C:\Users\home\PycharmProjects\SRU\venv\Scripts\python
[2, 3, 5, 7]

Process finished with exit code 0
```

Removing items at a present index

•Use the **pop() method** to remove the item at the given index.

•The **pop() method** removes and returns the **item present at the given index**.

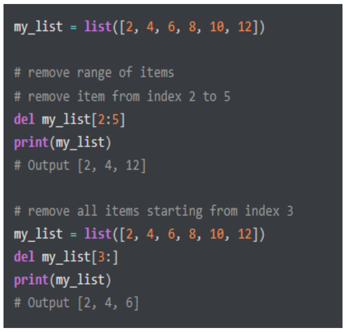
•Remove the **last item** from the list if the **index number is not passed.**



```
my_list = list([2, 4, 6, 8, 10, 12])

# remove item present at index 2
my_list.pop(2)
print(my_list)
# Output [2, 4, 8, 10, 12]

# remove item without passing index number
my_list.pop()
print(my_list)
# Output [2, 4, 8, 10]
```

Removing Range of items

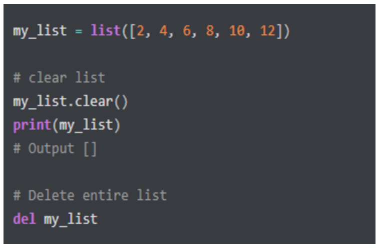
•Use **del keyword** along with **list slicing** to remove the range of items



```
my_list = list([2, 4, 6, 8, 10, 12])

# remove range of items
# remove item from index 2 to 5
del my_list[2:5]
print(my_list)
# Output [2, 4, 12]

# remove all items starting from index 3
my_list = list([2, 4, 6, 8, 10, 12])
del my_list[3:]
print(my_list)
# Output [2, 4, 6]
```

Removing All Items

•Use **clear() method** to remove all items from the list.

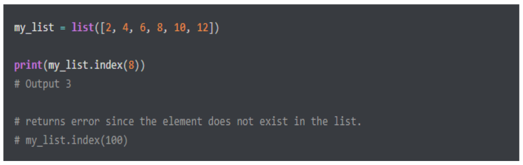


```
my_list = list([2, 4, 6, 8, 10, 12])

# clear list
my_list.clear()
print(my_list)
# Output []

# Delete entire list
del my_list
```

Finding Items in a list

Use the `index()` function to find an item in a list.

The `index()` function will accept the value of the element as a parameter and returns the first occurrence of the element or returns `ValueError` if the element does not exist.

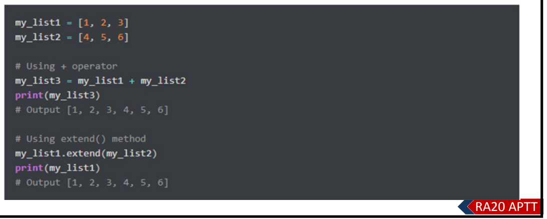


```
my_list = list([2, 4, 6, 8, 10, 12])

print(my_list.index(8))
# Output 3

# returns error since the element does not exist in the list.
# my_list.index(100)
```

Concatenation of two lists

The concatenation of two lists means merging of two lists. There are two ways to do that.

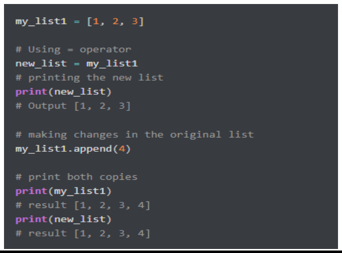
- Using the `+` operator.
- Using the `extend()` method. The `extend()` method appends the new list's items at the end of the calling list.



```
my_list1 = [1, 2, 3]
my_list2 = [4, 5, 6]

# Using + operator
my_list3 = my_list1 + my_list2
print(my_list3)
# Output [1, 2, 3, 4, 5, 6]

# Using extend() method
my_list1.extend(my_list2)
print(my_list1)
# Output [1, 2, 3, 4, 5, 6]
```

Copying a list

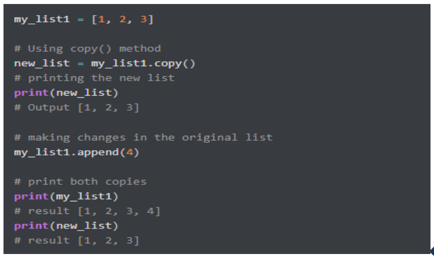
- There are two ways to copy of a list can be created . One is using **assignment operator (=)**
- This is a straightforward way of creating a copy and its called **deep copying.**
- The **changes made to the original list are reflected in the copied list as well.**
- When you set list1 = list2, you are making **them refer to the same list object**.



```
my_list1 = [1, 2, 3]

# Using = operator
new_list = my_list1
# printing the new list
print(new_list)
# Output [1, 2, 3]

# making changes in the original list
my_list1.append(4)

# print both copies
print(my_list1)
# result [1, 2, 3, 4]
print(new_list)
# result [1, 2, 3, 4]
```

Use Copy Method

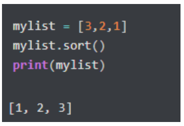
- The copy method can be used to create a copy of a list.
- This will create a new list and any **changes made in the original list** will **not reflect in the new list**. And This is **shallow copying.**



```
my_list1 = [1, 2, 3]

# Using copy() method
new_list = my_list1.copy()
# printing the new list
print(new_list)
# Output [1, 2, 3]

# making changes in the original list
my_list1.append(4)

# print both copies
print(my_list1)
# result [1, 2, 3, 4]
print(new_list)
# result [1, 2, 3]
```

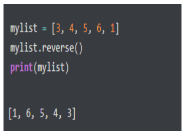
List Operations

Sort List using sort()



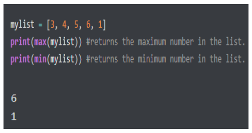
```
mylist = [3,2,1]
mylist.sort()
print(mylist)

[1, 2, 3]
```

Reverse a List using reverse()



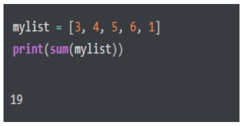
```
mylist = [3, 4, 5, 6, 1]
mylist.reverse()
print(mylist)

[1, 6, 5, 4, 3]
```

max() & min()



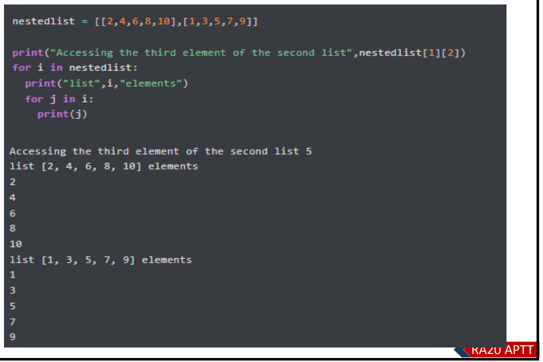
```
mylist = [3, 4, 5, 6, 1]
print(max(mylist)) #returns the maximum number in the list.
print(min(mylist)) #returns the minimum number in the list.

6
1
```

Using sum()



```
mylist = [3, 4, 5, 6, 1]
print(sum(mylist))

19
```

Nested List

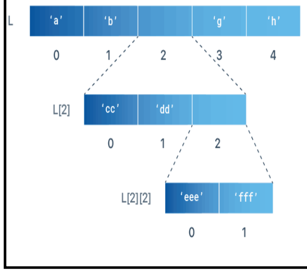


```
nestedlist = [[2,4,6,8,10],[1,3,5,7,9]]

print("Accessing the third element of the second list",nestedlist[1][2])
for i in nestedlist:
  print("list",i,"elements")
  for j in i:
    print(j)

Accessing the third element of the second list 5
list [2, 4, 6, 8, 10] elements
2
4
6
8
10
list [1, 3, 5, 7, 9] elements
1
3
5
7
9
```

Nested List

A nested list is created by placing a comma-separated sequence of sublists.

L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']

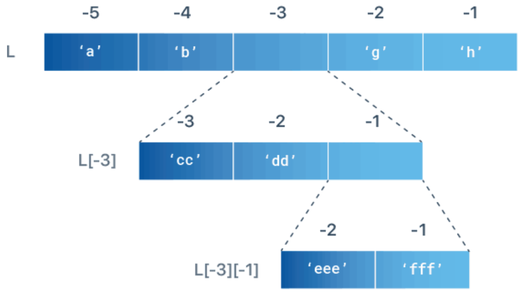


L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[2])
# Prints ['cc', 'dd', ['eee', 'fff']]

print(L[2][2])
# Prints ['eee', 'fff']

print(L[2][2][0])
# Prints eee

Negative Indexing In a Nested List



Iterate Through a Nested List

```
L = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
for list  in L:
    for number in list:
        print(number, end=' ')
# 1 2 3 4 5 6 7 8 9
```

Summary of List Operations

`l1` and `l2` are lists, `x`, `i`, `j`, `k`, `n` are integers.
`l1 = [10, 20, 30, 40, 50]` and `l2 = [60, 70, 80, 60]`

Operation	Description
X in l1	Check if the list l1 contains item x.
X not in l2	Check if list l1 does not contain item x.
l1 + l2	Concatenate the lists l1 and l2. Creates a new list containing the items from l1 and l2.
l1 * 5	Repeat the list l1 5 times.
l1[i]	Get the item at index i. Example l1[2] is 30.
l1[i:j]	List slicing. Get the items from index i up to index j (excluding j) as a List. An example l1[0:2] is [10, 20]
l1[i:j:k]	List slicing with step. Returns a List with the items from index i up to index j taking every k-th item. An example l1[0:4:2] is [10, 30].
len(l1)	Returns a count of total items in a list.

Summary of List Operations

l2.count(60)	Returns the number of times a particular item (60) appears in a list. The answer is 2.
l1.index(30)	Returns the index number of a particular item (30) in a list. The answer is 2.
l1.index(30, 2, 5)	Returns the index number of a particular item (30) in a list. But search Returns the item with maximum value from a list. The answer is 60 only from index number 2 to 5.
min(l1)	Returns the item with a minimum value from a list. The answer is 10.
max(l1)	Returns the item with maximum value from a list. The answer is 60.
l1.append(100)	Add item at the end of the list
l1.append([2, 5, 7])	Append the nested list at the end
l1[2] = 40	Modify the item present at index 2
l1.remove(40)	Removes the first occurrence of item 40 from the list.
pop(2)	Removes and returns the item at index 2 from the list.
l1.clear()	Make list empty
l3= l1.copy()	Copy l1 into l2

TUPLE in PYTHON

•Tuples are ordered collections of heterogeneous data that are unchangeable or immutable object

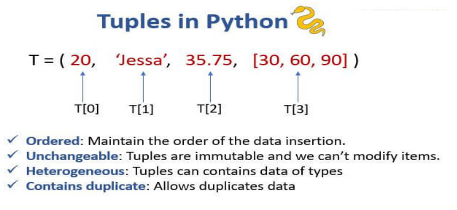
Ordered: Tuples are part of sequence data types, which means they hold the order of the data insertion. It maintains the index value for each item.

Unchangeable: Tuples are unchangeable, which means that we cannot add or delete items to the tuple after creation.

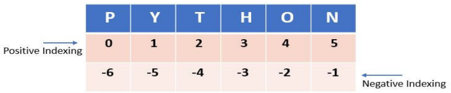
Heterogeneous: Tuples are a sequence of data of different data types (like integer, float, list, string, etc;) and can be accessed through indexing and slicing.

Contains Duplicates: Tuples can contain duplicates, which means they can have items with the same value.

TUPLE in PYTHON

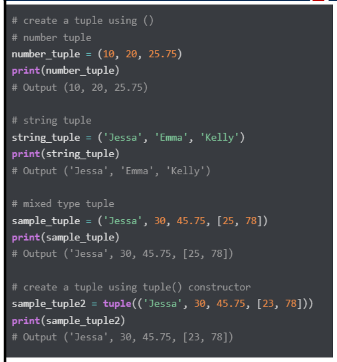


- ✓ **Ordered**: Maintain the order of the data insertion.
- ✓ **Unchangeable**: Tuples are immutable and we can't modify items.
- ✓ **Heterogeneous**: Tuples can contains data of types
- ✓ **Contains duplicate**: Allows duplicates data



P	Y	T	H	O	N
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Creating a Tuple

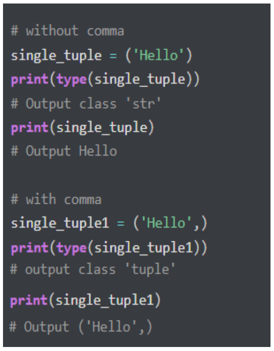


```
# create a tuple using ()
# number tuple
number_tuple = (10, 20, 25.75)
print(number_tuple)
# Output (10, 20, 25.75)

# string tuple
string_tuple = ('Jessa', 'Emma', 'Kelly')
print(string_tuple)
# Output ('Jessa', 'Emma', 'Kelly')

# mixed type tuple
sample_tuple = ('Jessa', 30, 45.75, [25, 78])
print(sample_tuple)
# Output ('Jessa', 30, 45.75, [25, 78])

# create a tuple using tuple() constructor
sample_tuple2 = tuple(('Jessa', 30, 45.75, [23, 78]))
print(sample_tuple2)
# Output ('Jessa', 30, 45.75, [23, 78])
```

•Using parenthesis (): A tuple is created by enclosing comma-separated items inside rounded brackets.

•Using a tuple() constructor: Create a tuple by passing the comma-separated items inside the tuple(). A tuple can have items of different data types.

Tuple With Single Item

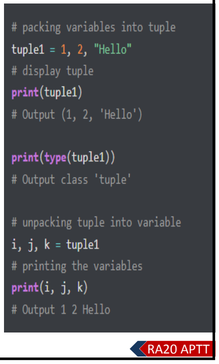


```
# without comma
single_tuple = ('Hello')
print(type(single_tuple))
# Output class 'str'
print(single_tuple)
# Output Hello

# with comma
single_tuple1 = ('Hello',)
print(type(single_tuple1))
# output class 'tuple'
print(single_tuple1)
# Output ('Hello',)
```

- A single item tuple is created by enclosing one item inside parentheses followed by a comma.
- If the tuple is a string enclosed within parentheses and not followed by a comma, Python treats it as a **str type**.

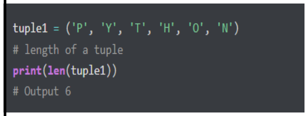
Packing and Unpacking

- A tuple can also be created without using a tuple() constructor or enclosing the items inside the parentheses and It is called the variable "Packing."
- we can create a tuple by packing a group of variables.
- Packing can be used when we want to collect multiple values in a single variable.
- This operation is referred to as tuple packing.

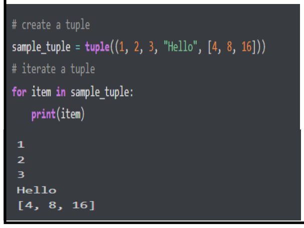


```
# packing variables into tuple
tuple1 = 1, 2, "Hello"
# display tuple
print(tuple1)
# Output (1, 2, 'Hello')

print(type(tuple1))
# Output class 'tuple'

# unpacking tuple into variable
i, j, k = tuple1
# printing the variables
print(i, j, k)
# Output 1 2 Hello
```

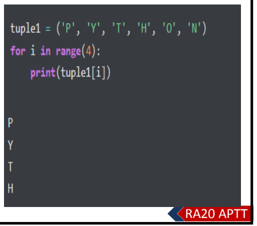
Iterating Through Tuple



```
tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')
# length of a tuple
print(len(tuple1))
# Output 6
```

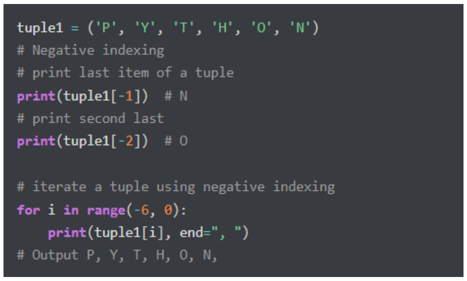


```
# create a tuple
sample_tuple = tuple((1, 2, 3, "Hello", [4, 8, 16]))
# iterate a tuple
for item in sample_tuple:
    print(item)

1
2
3
Hello
[4, 8, 16]
```



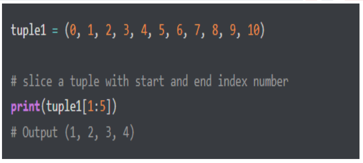
```
tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')
for i in range(4):
    print(tuple1[i])


P
Y
T
H
```

Negative Indexing

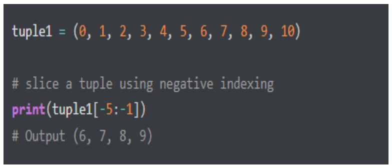


```
tuple1 = ('P', 'Y', 'T', 'H', 'O', 'N')
# Negative indexing
# print last item of a tuple
print(tuple1[-1])  # N
# print second last
print(tuple1[-2])  # O

# iterate a tuple using negative indexing
for i in range(-6, 0):
    print(tuple1[i], end=", ")
# Output P, Y, T, H, O, N,
```

Slicing a Tuple

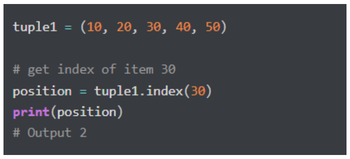


```
tuple1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# slice a tuple with start and end index number
print(tuple1[1:5])
# Output (1, 2, 3, 4)
```

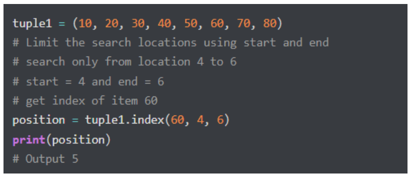


```
tuple1 = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# slice a tuple using negative indexing
print(tuple1[-5:-1])
# Output (6, 7, 8, 9)
```

Finding Item in Tuple



```
tuple1 = (10, 20, 30, 40, 50)

# get index of item 30
position = tuple1.index(30)
print(position)
# Output 2
```

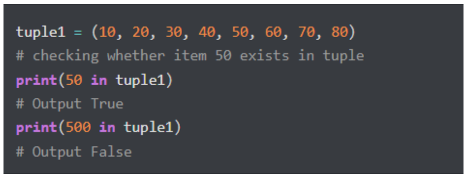


```
tuple1 = (10, 20, 30, 40, 50, 60, 70, 80)
# Limit the search locations using start and end
# search only from location 4 to 6
# start = 4 and end = 6
# get index of item 60
position = tuple1.index(60, 4, 6)
print(position)
# Output 5
```

Checking if an item exists

•We can check whether an item exists in a tuple by using the **in** operator.

•This will return a **Boolean True** if the item exists and **False** if it doesn't.



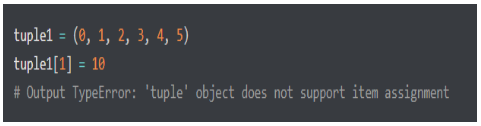
```
tuple1 = (10, 20, 30, 40, 50, 60, 70, 80)
# checking whether item 50 exists in tuple
print(50 in tuple1)
# Output True
print(500 in tuple1)
# Output False
```

Adding and changing items in a Tuple

•A list is a mutable type, which means we can add or modify values in it, but tuples are immutable, so they cannot be changed.

•Because a tuple is immutable there are no built-in methods to add items to the tuple.

•If you try to modify the value you will get an error.

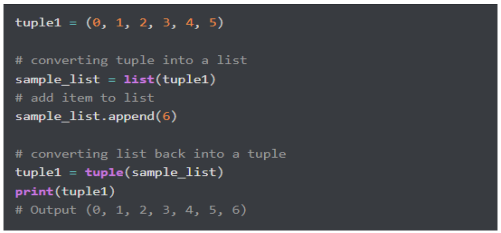


```
tuple1 = (0, 1, 2, 3, 4, 5)
tuple1[1] = 10
# Output TypeError: 'tuple' object does not support item assignment
```

Adding and changing items in a Tuple

•As a workaround solution, we can convert the tuple to a list, add items, and then convert it back to a tuple.

•As tuples are ordered collection like lists the **items always get added in the end.**

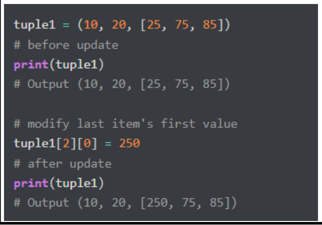


```
tuple1 = (0, 1, 2, 3, 4, 5)

# converting tuple into a list
sample_list = list(tuple1)
# add item to list
sample_list.append(6)

# converting list back into a tuple
tuple1 = tuple(sample_list)
print(tuple1)
# Output (0, 1, 2, 3, 4, 5, 6)
```

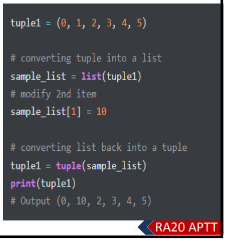
Modify nested items of a Tuple

•If one of the **items** is itself a **mutable** data type as a list, then we can change its values in the case of a **nested tuple.**

•For example, A tuple which has **a list as its last item** and you wanted to modify the list items.   tuple1 = (10, 20, **[25, 75, 85]**)



```
tuple1 = (10, 20, [25, 75, 85])
# before update
print(tuple1)
# Output (10, 20, [25, 75, 85])

# modify last item's first value
tuple1[2][0] = 250
# after update
print(tuple1)
# Output (10, 20, [250, 75, 85])
```



```
tuple1 = (0, 1, 2, 3, 4, 5)

# converting tuple into a list
sample_list = list(tuple1)
# modify 2nd item
sample_list[1] = 10

# converting list back into a tuple
tuple1 = tuple(sample_list)
print(tuple1)
# Output (0, 10, 2, 3, 4, 5)
```

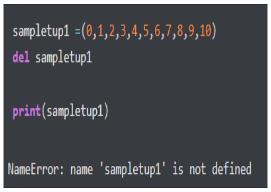
Removing items from a Tuple

•Tuples are immutable so there are no pop() or remove() methods for the tuple.

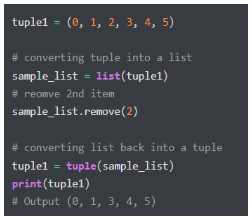
•We can remove the items from a tuple using the following two ways.

•Using del keyword

By converting it into a list

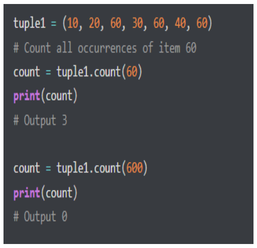


```
sampletup1 =(0,1,2,3,4,5,6,7,8,9,10)
del sampletup1


print(sampletup1)


NameError: name 'sampletup1' is not defined
```

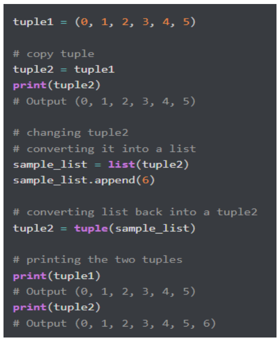


```
tuple1 = (0, 1, 2, 3, 4, 5)

# converting tuple into a list
sample_list = list(tuple1)
# reomve 2nd item
sample_list.remove(2)

# converting list back into a tuple
tuple1 = tuple(sample_list)
print(tuple1)
# Output (0, 1, 3, 4, 5)
```

Count the occurrence & Copying a tuple



```
tuple1 = (10, 20, 60, 30, 60, 40, 60)
# Count all occurrences of item 60
count = tuple1.count(60)
print(count)
# Output 3

count = tuple1.count(600)
print(count)
# Output 0
```

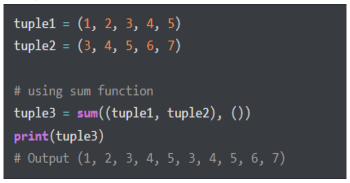


```
tuple1 = (0, 1, 2, 3, 4, 5)

# copy tuple
tuple2 = tuple1
print(tuple2)
# Output (0, 1, 2, 3, 4, 5)

# changing tuple2
# converting it into a list
sample_list = list(tuple2)
sample_list.append(6)

# converting list back into a tuple2
tuple2 = tuple(sample_list)

# printing the two tuples
print(tuple1)
# Output (0, 1, 2, 3, 4, 5)
print(tuple2)
# Output (0, 1, 2, 3, 4, 5, 6)
```

Concatenating two Tuples

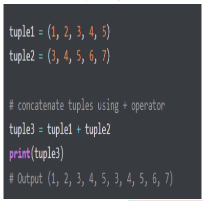
•The sum function of two iterables like tuples always needs to start with Empty Tuple.

•sum function takes an Empty tuple as an argument and it returns the items from both the tuples.



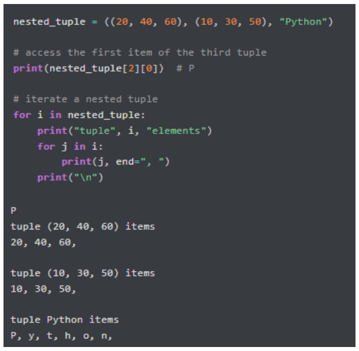
```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = (3, 4, 5, 6, 7)

# using sum function
tuple3 = sum((tuple1, tuple2), ())
print(tuple3)
# Output (1, 2, 3, 4, 5, 3, 4, 5, 6, 7)
```

Using + Operator



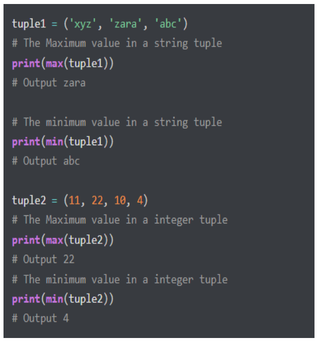
```
tuple1 = (1, 2, 3, 4, 5)
tuple2 = (3, 4, 5, 6, 7)


# concatenate tuples using + operator
tuple3 = tuple1 + tuple2
print(tuple3)
# Output (1, 2, 3, 4, 5, 3, 4, 5, 6, 7)
```

Nested Tuples



```
nested_tuple = ((20, 40, 60), (10, 30, 50), "Python")

# access the first item of the third tuple
print(nested_tuple[2][0])  # P

# iterate a nested tuple
for i in nested_tuple:
    print("tuple", i, "elements")
    for j in i:
        print(j, end=", ")
    print("\n")


P
tuple (20, 40, 60) items
20, 40, 60,

tuple (10, 30, 50) items
10, 30, 50,

tuple Python items
P, y, t, h, o, n,
```

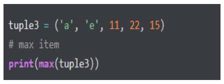
Built-in functions



```
tuple1 = ('xyz', 'zara', 'abc')
# The Maximum value in a string tuple
print(max(tuple1))
# Output zara

# The minimum value in a string tuple
print(min(tuple1))
# Output abc

tuple2 = (11, 22, 10, 4)
# The Maximum value in a integer tuple
print(max(tuple2))
# Output 22
# The minimum value in a integer tuple
print(min(tuple2))
# Output 4
```

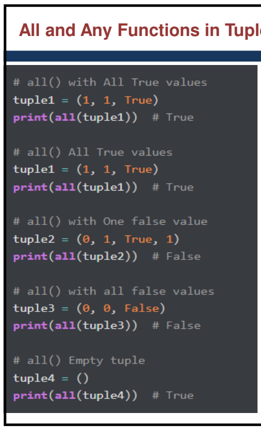
•We can't find the max() and min() fo a heterogeneous tuple (mixed types of items).

•It will throw Type Error

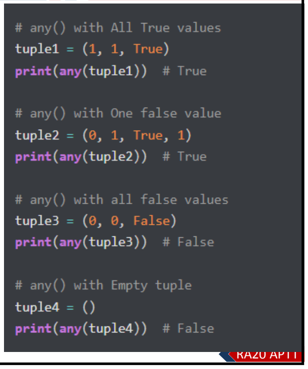


```
tuple3 = ('a', 'e', 11, 22, 15)
# max item
print(max(tuple3))
```

All and Any Functions in Tuple?



```
# all() with All True values
tuple1 = (1, 1, True)
print(all(tuple1))  # True

# all() All True values
tuple1 = (1, 1, True)
print(all(tuple1))  # True

# all() with One false value
tuple2 = (0, 1, True, 1)
print(all(tuple2))  # False

# all() with all false values
tuple3 = (0, 0, False)
print(all(tuple3))  # False

# all() Empty tuple
tuple4 = ()
print(all(tuple4))  # True
```



```
# any() with All True values
tuple1 = (1, 1, True)
print(any(tuple1))  # True

# any() with One false value
tuple2 = (0, 1, True, 1)
print(any(tuple2))  # True

# any() with all false values
tuple3 = (0, 0, False)
print(any(tuple3))  # False

# any() with Empty tuple
tuple4 = ()
print(any(tuple4))  # False
```

When to use Tuple?

The tuples are used for the following requirements instead of lists.

•There are no append() or extend() to add items and similarly no remove() or pop() methods to remove items.

• This ensures that the data is write-protected.

•As the tuples are Unchangeable, they can be used to represent read-only or fixed data that does not change.

•As they are immutable, they can be used as a key for the dictionaries.

•As they are immutable, the search operation is much faster than the lists.

•Tuples contain heterogeneous (all types) data that offers huge flexibility in data that contains combinations of data types like alphanumeric characters.

Summary of Tuples operations

t1 = (10, 20, 30, 40, 50)   t2 = (60, 70, 80, 60)

Operation	Description
x in t1	Check if the tuple t1 contains the item x.
x not in t2	Check if the tuple t1 does not contain the item x.
t1 + t2	Concatenate the tuples t1 and t2. Creates a new tuple containing the items from t1 and t2.
t1 * 5	Repeat the tuple t1 5 times.
t1[i]	Get the item at the index i. Example, t1[2] is 30
t1[i:j]	Tuple slicing. Get the items from index i up to index j (excluding j) as a tuple. An example t1[0:2] is (10, 20)
t1[i:j:k]	Tuple slicing with step. Return a tuple with the items from index i up to index j taking every k-th item. An example t1[0:4:2] is (10, 30)
len(t1)	Returns a count of total items in a tuple
t2.count(60)	Returns the number of times a particular item (60) appears in a tuple. Answer is 2
t1.index(30)	Returns the index number of a particular item(30) in a tuple. Answer is 2
t1.index(40, 2, 5)	Returns the index number of a particular item(30) in a tuple. But search only from index number 2 to 5.
min(t1)	Returns the item with a minimum value from a tuple
max(t1)	Returns the item with maximum value from a tuple

Sets in Python

- Set is an unordered collection of data items that are unique (no duplicate elements)
- Set doesn't maintain the order of elements.
- Elements cannot be accessed by their index

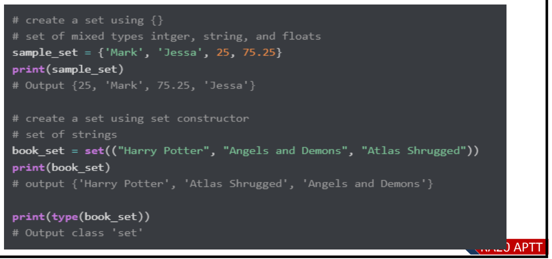
Set in Python

S = { 20, 'Jessa', 35.75}

✓ **Unordered**: Set doesn't maintain the order of the data insertion.
✓ **Unchangeable**: Set are immutable and we can't modify items.
✓ **Heterogeneous**: Set can contains data of all types
✓ **Unique**: Set doesn't allows duplicates items

Creating a Set

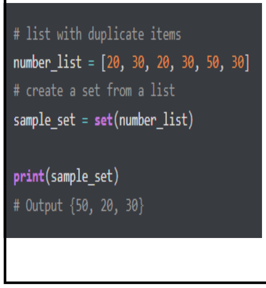
- Using curly brackets: The easiest way of creating a Set is by just enclosing all the data items inside the curly brackets {} and individual values are comma-separated.
- Using set() constructor: by calling the constructor of class 'set'. We can pass items to the set constructor inside double-rounded brackets.



```
# create a set using {}
# set of mixed types intger, string, and floats
sample_set = {'Mark', 'Jessa', 25, 75.25}
print(sample_set)
# Output {25, 'Mark', 75.25, 'Jessa'}

# create a set using set constructor
# set of strings
book_set = set(("Harry Potter", "Angels and Demons", "Atlas Shrugged"))
print(book_set)
# output {'Harry Potter', 'Atlas Shrugged', 'Angels and Demons'}

print(type(book_set))
# Output class 'set'
```

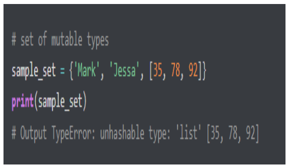
Creating a Set From Mutable Type

Set eliminating duplicate entries so if you try to create a set with duplicate items it will store an item only once and delete all duplicate items.



```
# list with duplicate items
number_list = [20, 30, 20, 30, 50, 30]
# create a set from a list
sample_set = set(number_list)

print(sample_set)
# Output {50, 20, 30}
```

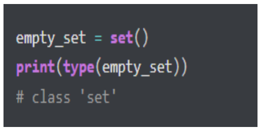
An **error will be generated** , if you try to **create a set with mutable elements** like lists or dictionaries as its elements.



```
# set of mutable types
sample_set = {'Mark', 'Jessa', [35, 78, 92]}
print(sample_set)
# Output TypeError: unhashable type: 'list' [35, 78, 92]
```

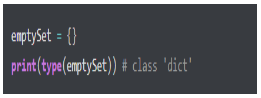
Empty SET

When we don't pass any item to the set constructor then it will create an **empty set**.



```
empty_set = set()
print(type(empty_set))
# class 'set'
```

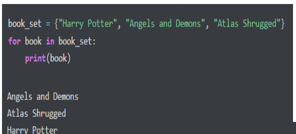
- When the same object is created without any items inside the curly brackets then **dictionary** is created.
- So whenever you wanted to create an **empty set** always use the set() constructor.



```
emptySet = {}
print(type(emptySet)) # class 'dict'
```

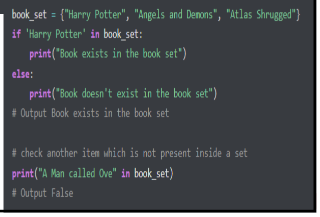
Accessing items of a set

To access the items of a set, we need to iterate through the set object using a for loop



```
book_set = {"Harry Potter", "Angels and Demons", "Atlas Shrugged"}
for book in book_set:
    print(book)

Angels and Demons
Atlas Shrugged
Harry Potter
```

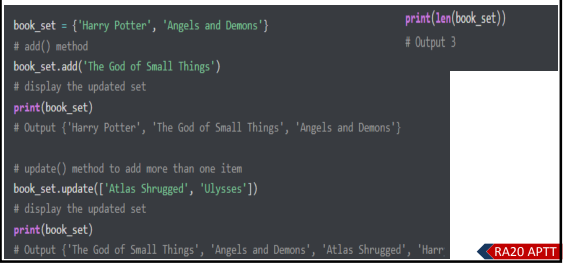
We can't find items using the index value. In order to check if an **item exists in the Set**, we can use the **in operator**.



```
book_set = {"Harry Potter", "Angels and Demons", "Atlas Shrugged"}
if 'Harry Potter' in book_set:
    print("Book exists in the book set")
else:
    print("Book doesn't exist in the book set")
# Output Book exists in the book set

# check another item which is not present inside a set
print("A Man called Ove" in book_set)
# Output False
```

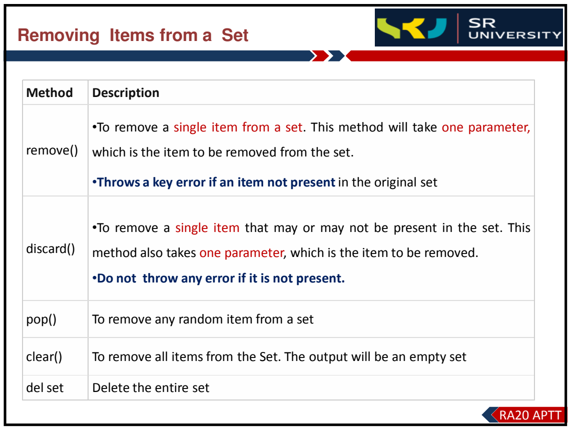
Adding Item to Set

- The add() method: The add() method is used to **add one** item to the set.
- Using update() Method: The update() method is used to **add multiple items** to the Set and list of items need to be passed to the update() method.
- The **length of set** is calculated by **len()** funtion



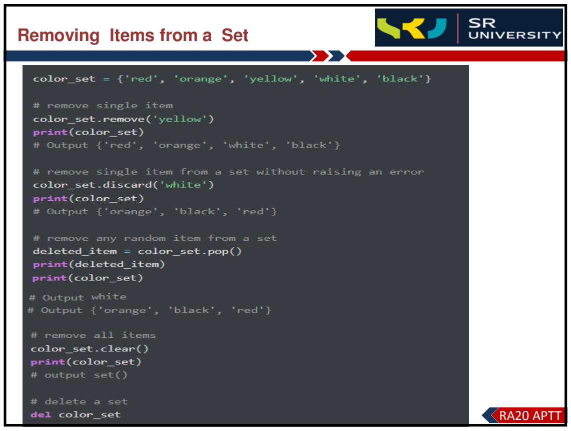
```
book_set = {'Harry Potter', 'Angels and Demons'}
# add() method
book_set.add('The God of Small Things')
# display the updated set
print(book_set)
# Output {'Harry Potter', 'The God of Small Things', 'Angels and Demons'}

# update() method to add more than one item
book_set.update(['Atlas Shrugged', 'Ulysses'])
# display the updated set
print(book_set)
# Output {'The God of Small Things', 'Angels and Demons', 'Atlas Shrugged', 'Harr
```

```
print(len(book_set))
# Output 3
```



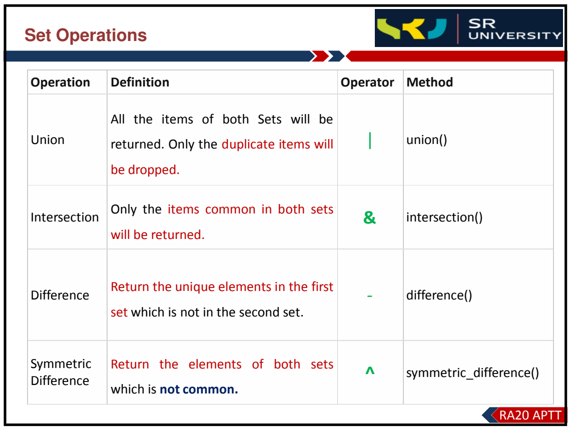
Removing Items from a Set

Method	Description
remove()	•To remove a single item from a set. This method will take one parameter, which is the item to be removed from the set. •Throws a key error if an item not present in the original set
discard()	•To remove a single item that may or may not be present in the set. This method also takes one parameter, which is the item to be removed. •Do not throw any error if it is not present.
pop()	To remove any random item from a set
clear()	To remove all items from the Set. The output will be an empty set
del set	Delete the entire set



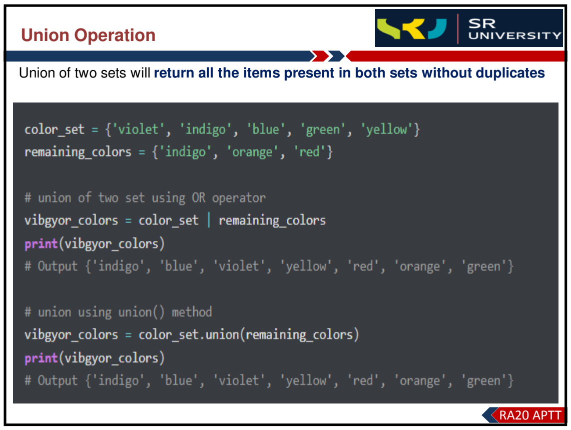
Removing Items from a Set

```
color_set = {'red', 'orange', 'yellow', 'white', 'black'}

# remove single item
color_set.remove('yellow')
print(color_set)
# Output {'red', 'orange', 'white', 'black'}

# remove single item from a set without raising an error
color_set.discard('white')
print(color_set)
# Output {'orange', 'black', 'red'}

# remove any random item from a set
deleted_item = color_set.pop()
print(deleted_item)
print(color_set)
# Output white
# Output {'orange', 'black', 'red'}

# remove all items
color_set.clear()
print(color_set)
# output set()

# delete a set
del color_set
```



Set Operations

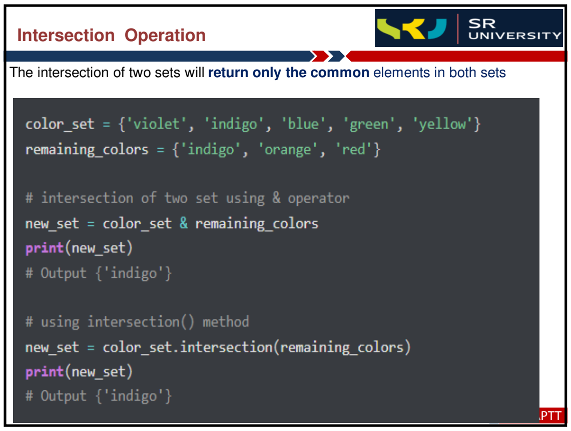
Operation	Definition	Operator	Method
Union	All the items of both Sets will be returned. Only the duplicate items will be dropped.	|	union()
Intersection	Only the items common in both sets will be returned.	&	intersection()
Difference	Return the unique elements in the first set which is not in the second set.	-	difference()
Symmetric Difference	Return the elements of both sets which is not common.	^	symmetric_difference()



Union Operation

Union of two sets will **return all the items present in both sets without duplicates**

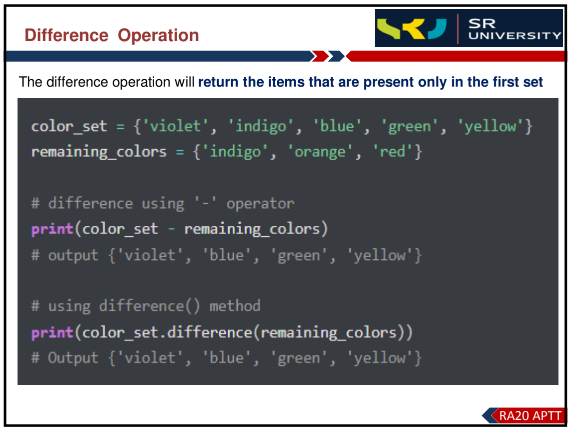
```
color_set = {'violet', 'indigo', 'blue', 'green', 'yellow'}
remaining_colors = {'indigo', 'orange', 'red'}

# union of two set using OR operator
vibgyor_colors = color_set | remaining_colors
print(vibgyor_colors)
# Output {'indigo', 'blue', 'violet', 'yellow', 'red', 'orange', 'green'}

# union using union() method
vibgyor_colors = color_set.union(remaining_colors)
print(vibgyor_colors)
# Output {'indigo', 'blue', 'violet', 'yellow', 'red', 'orange', 'green'}
```



Intersection Operation

The intersection of two sets will **return only the common** elements in both sets

```
color_set = {'violet', 'indigo', 'blue', 'green', 'yellow'}
remaining_colors = {'indigo', 'orange', 'red'}

# intersection of two set using & operator
new_set = color_set & remaining_colors
print(new_set)
# Output {'indigo'}

# using intersection() method
new_set = color_set.intersection(remaining_colors)
print(new_set)
# Output {'indigo'}
```



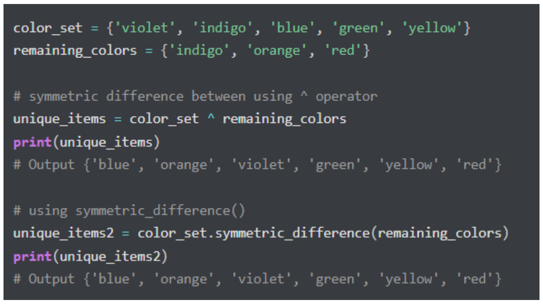
Difference Operation

The difference operation will **return the items that are present only in the first set**

```
color_set = {'violet', 'indigo', 'blue', 'green', 'yellow'}
remaining_colors = {'indigo', 'orange', 'red'}

# difference using '-' operator
print(color_set - remaining_colors)
# output {'violet', 'blue', 'green', 'yellow'}

# using difference() method
print(color_set.difference(remaining_colors))
# Output {'violet', 'blue', 'green', 'yellow'}
```

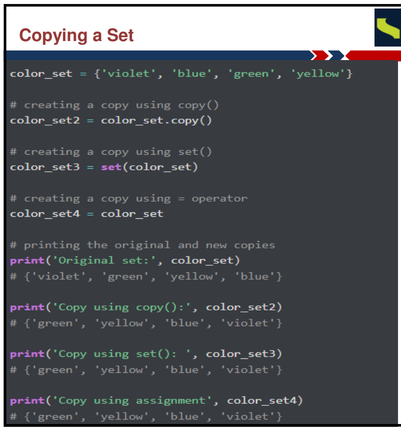
Symmetric Difference Operation

Symmetric difference operation **returns the elements that are unique in both sets**



```
color_set = {'violet', 'indigo', 'blue', 'green', 'yellow'}
remaining_colors = {'indigo', 'orange', 'red'}

# symmetric difference between using ^ operator
unique_items = color_set ^ remaining_colors
print(unique_items)
# Output {'blue', 'orange', 'violet', 'green', 'yellow', 'red'}

# using symmetric_difference()
unique_items2 = color_set.symmetric_difference(remaining_colors)
print(unique_items2)
# Output {'blue', 'orange', 'violet', 'green', 'yellow', 'red'}
```

Copying a Set



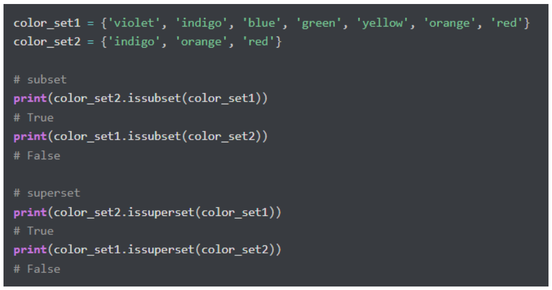
```
color_set = {'violet', 'blue', 'green', 'yellow'}

# creating a copy using copy()
color_set2 = color_set.copy()

# creating a copy using set()
color_set3 = set(color_set)

# creating a copy using = operator
color_set4 = color_set

# printing the original and new copies
print('Original set:', color_set)
# {'violet', 'green', 'yellow', 'blue'}

print('Copy using copy():', color_set2)
# {'green', 'yellow', 'blue', 'violet'}

print('Copy using set(): ', color_set3)
# {'green', 'yellow', 'blue', 'violet'}

print('Copy using assignment', color_set4)
# {'green', 'yellow', 'blue', 'violet'}
```

1. **copy() method,**
2. set() constructor.
3. **= operator**

Subset & Superset

issubset() : return true if a set is a subset of another set otherwise, it will return false.

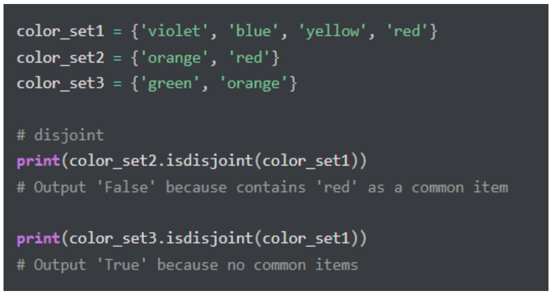
issuperset() : This method determines whether the set is a superset of another set.



```
color_set1 = {'violet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red'}
color_set2 = {'indigo', 'orange', 'red'}

# subset
print(color_set2.issubset(color_set1))
# True
print(color_set1.issubset(color_set2))
# False

# superset
print(color_set2.issuperset(color_set1))
# True
print(color_set1.issuperset(color_set2))
# False
```

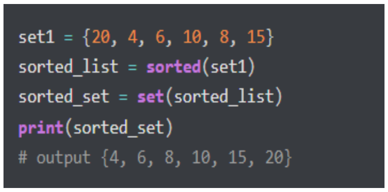
Sets are disjoint

The **isdisjoint()** method will find whether there are common elements or not.



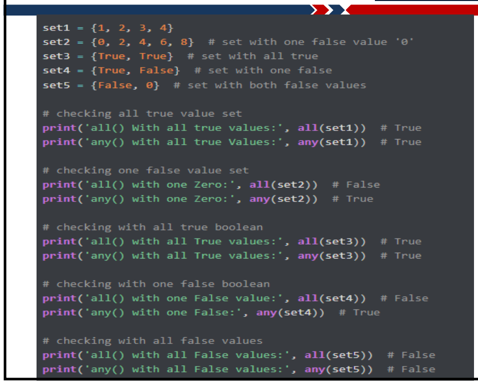
```
color_set1 = {'violet', 'blue', 'yellow', 'red'}
color_set2 = {'orange', 'red'}
color_set3 = {'green', 'orange'}

# disjoint
print(color_set2.isdisjoint(color_set1))
# Output 'False' because contains 'red' as a common item

print(color_set3.isdisjoint(color_set1))
# Output 'True' because no common items
```

Sort Sets



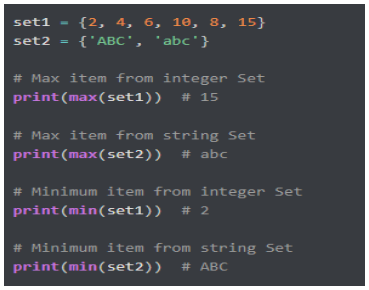
```
set1 = {20, 4, 6, 10, 8, 15}
sorted_list = sorted(set1)
sorted_set = set(sorted_list)
print(sorted_set)
# output {4, 6, 8, 10, 15, 20}
```

All & Any



```
set1 = {1, 2, 3, 4}
set2 = {0, 2, 4, 6, 8}  # set with one false value '0'
set3 = {True, True}  # set with all true
set4 = {True, False}  # set with one false
set5 = {False, 0}  # set with both false values

# checking all true value set
print('all() With all true values:', all(set1))  # True
print('any() with all true Values:', any(set1))  # True

# checking one false value set
print('all() with one Zero:', all(set2))  # False
print('any() with one Zero:', any(set2))  # True

# checking with all true boolean
print('all() with all True values:', all(set3))  # True
print('any() with all True values:', any(set3))  # True

# checking with one false boolean
print('all() with one False value:', all(set4))  # False
print('any() with one False:', any(set4))  # True

# checking with all false values
print('all() with all False values:', all(set5))  # False
print('any() with all False values:', any(set5))  # False
```

Max & Min



```
set1 = {2, 4, 6, 10, 8, 15}
set2 = {'ABC', 'abc'}

# Max item from integer Set
print(max(set1))  # 15

# Max item from string Set
print(max(set2))  # abc

# Minimum item from integer Set
print(min(set1))  # 2

# Minimum item from string Set
print(min(set2))  # ABC
```

Nested Sets

•A set **cannot have mutable objects as its elements**.

•So we **can't have another set inside a set.**

•When to use a Set Data structure?.

•Eliminating duplicate entries: In case a set is initialized with multiple entries of the same value, then the duplicate entries will be dropped in the actual set.

Performing arithmetic operations similar to Mathematical Sets: All the arithmetic operations like union, Intersection, finding the difference that we perform on the elements of two sets could be performed on this data structure.

Dictionaries

•Dictionaries are unordered collections of unique values stored in (Key-Value) pairs.

•Dictionary represents a **mapping between a key and a value**.

•Once stored in a dictionary, you can later obtain the value using just the key.

Characteristics of dictionaries

Unordered: The items in dictionaries are stored without any index value as Key-Value pairs, and the keys are their index, which will not be in any sequence.

Unique: the Keys in Dictionaries should be unique. If we store any value with a Key that already exists, then the most recent value will replace the old value.

Mutable: The dictionaries are collections that are changeable, which implies that we can add or remove items after the creation.

Creating Dictionaries

•Using curly brackets: The dictionaries are created by enclosing the **comma-separated Key: Value pairs inside the {} curly brackets**. The " **colon :** " is used to separate the key and value in a pair.

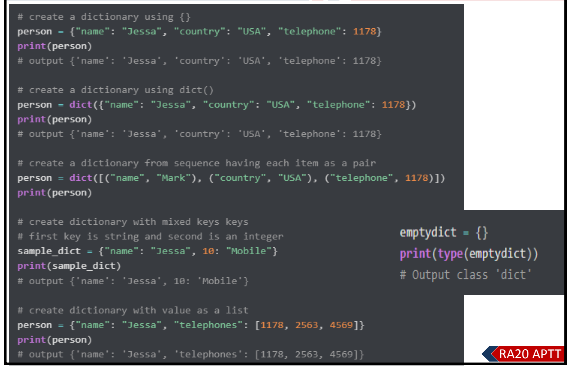
•Using dict() constructor: Create a dictionary by passing the comma-separated key: value pairs inside the **dict ().**

•Using sequence having each item as a **pair (key-value).**

•A dictionary value can be of **any type, and duplicates are allowed** in that.

•Keys in dictionary must be **unique and of immutable types**

Creating Dictionaries



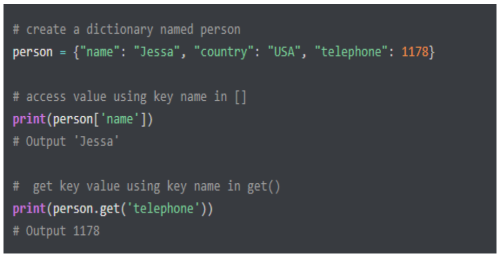
```
# create a dictionary using {}
person = {"name": "Jessa", "country": "USA", "telephone": 1178}
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178}

# create a dictionary using dict()
person = dict({"name": "Jessa", "country": "USA", "telephone": 1178})
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178}

# create a dictionary from sequence having each item as a pair
person = dict([("name", "Mark"), ("country", "USA"), ("telephone", 1178)])
print(person)

# create dictionary with mixed keys keys
# first key is string and second is an integer
sample_dict = {"name": "Jessa", 10: "Mobile"}
print(sample_dict)
# output {'name': 'Jessa', 10: 'Mobile'}

# create dictionary with value as a list
person = {"name": "Jessa", "telephones": [1178, 2563, 4569]}
print(person)
# output {'name': 'Jessa', 'telephones': [1178, 2563, 4569]}
```

```
emptydict = {}
print(type(emptydict))
# Output class 'dict'
```

Accessing elements of a Dictionaries

•Retrieve value using the **key name inside the [] square brackets.**

•Retrieve value by **passing key name as a parameter** to the **get() method.**



```
# create a dictionary named person
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# access value using key name in []
print(person['name'])
# Output 'Jessa'

#  get key value using key name in get()
print(person.get('telephone'))
# Output 1178
```

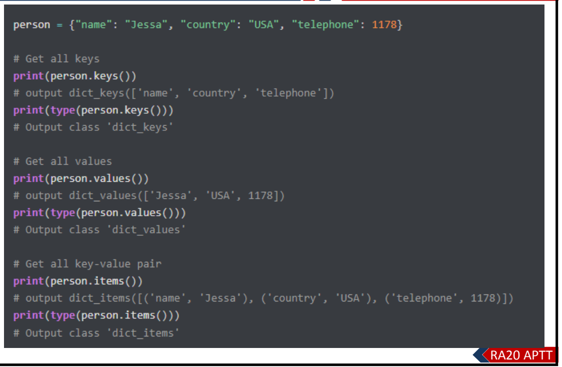
Get all keys and values

•Retrieve value using the **key name inside the [] square brackets.**

•Retrieve value by **passing key name as a parameter** to the **get() method.**

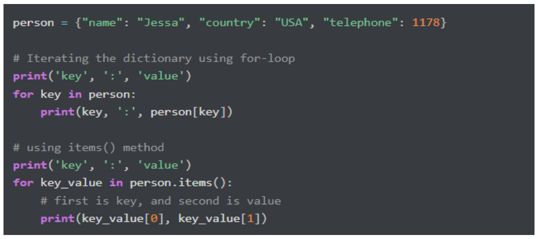
Method	Description
keys()	**Returns the list of all keys** present in the dictionary.
values()	**Returns the list of all values** present in the dictionary
items()	• **Returns all the items present in the dictionary.** • Each item will be inside a **tuple as a key-value pair**.

Get all keys and values

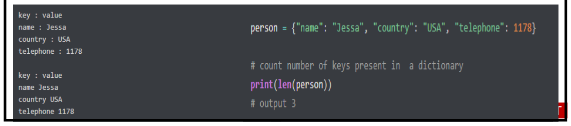


```
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# Get all keys
print(person.keys())
# output dict_keys(['name', 'country', 'telephone'])
print(type(person.keys()))
# Output class 'dict_keys'

# Get all values
print(person.values())
# output dict_values(['Jessa', 'USA', 1178])
print(type(person.values()))
# Output class 'dict_values'

# Get all key-value pair
print(person.items())
# output dict_items([('name', 'Jessa'), ('country', 'USA'), ('telephone', 1178)])
print(type(person.items()))
# Output class 'dict_items'
```

Iterating a Dictionary



```
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# Iterating the dictionary using for-loop
print('key', ':', 'value')
for key in person:
    print(key, ':', person[key])

# using items() method
print('key', ':', 'value')
for key_value in person.items():
    # first is key, and second is value
    print(key_value[0], key_value[1])
```



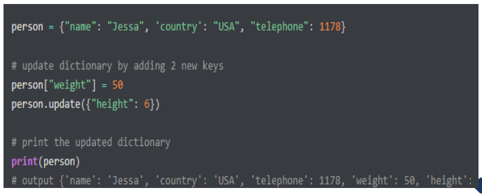
```
key : value
name : Jessa
country : USA
telephone : 1178

key : value
name Jessa
country USA
telephone 1178
```

```
person = {"name": "Jessa", "country": "USA", "telephone": 1178}

# count number of keys present in  a dictionary
print(len(person))
# output 3
```

Adding Items to the Dictionary

Using key-value assignment: Using a simple assignment statement where value can be assigned directly to the new key.

Using update() Method: The item passed inside the update() method will be inserted into the dictionary. The item can be another dictionary or any iterable like a tuple of key-value pairs.

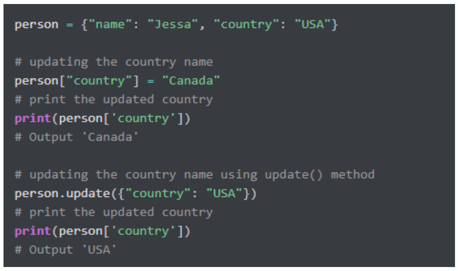


```
person = {"name": "Jessa", 'country': "USA", "telephone": 1178}

# update dictionary by adding 2 new keys
person["weight"] = 50
person.update({"height": 6})

# print the updated dictionary
print(person)
# output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178, 'weight': 50, 'height':
```

Modifying Values in Dictionary

Using key name: We can directly assign new values by using its key name.

Using update() method: by passing the key-value pair to change the value.

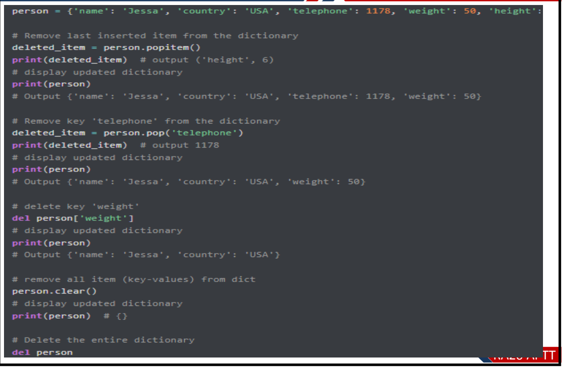


```
person = {"name": "Jessa", "country": "USA"}

# updating the country name
person["country"] = "Canada"
# print the updated country
print(person['country'])
# Output 'Canada'

# updating the country name using update() method
person.update({"country": "USA"})
# print the updated country
print(person['country'])
# Output 'USA'
```

Removing Items From Dictionary

Method	Description
pop(key[,d])	•Return and removes the item with the key and return its value. •If the key is not found, it raises Key Error.
Pop item()	•Return and removes the last inserted item from the dictionary. •If the dictionary is empty, it raises Key Error.
del key	The del keyword will delete the item with the key that is passed
clear()	Removes all items from the dictionary. Empty the dictionary
del dict_name	Delete the entire dictionary

Removing Items From Dictionary

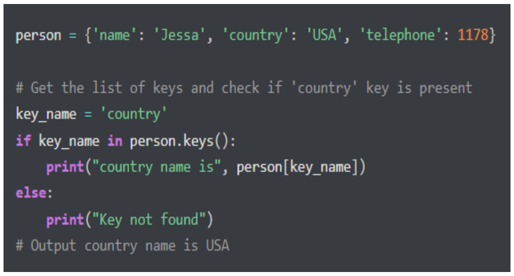


```
person = {'name': 'Jessa', 'country': 'USA', 'telephone': 1178, 'weight': 50, 'height':

# Remove last inserted item from the dictionary
deleted_item = person.popitem()
print(deleted_item)  # output ('height', 6)
# display updated dictionary
print(person)
# Output {'name': 'Jessa', 'country': 'USA', 'telephone': 1178, 'weight': 50}

# Remove key 'telephone' from the dictionary
deleted_item = person.pop('telephone')
print(deleted_item)  # output 1178
# display updated dictionary
print(person)
# Output {'name': 'Jessa', 'country': 'USA', 'weight': 50}

# delete key 'weight'
del person['weight']
# display updated dictionary
print(person)
# Output {'name': 'Jessa', 'country': 'USA'}

# remove all item (key-values) from dict
person.clear()
# display updated dictionary
print(person)  # {}

# Delete the entire dictionary
del person
```

Checking if a key exists

In order to check whether a particular key exists in a dictionary.

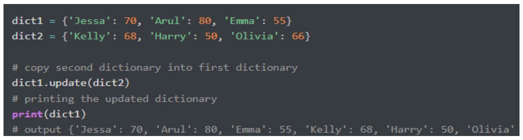
•We can use the keys() method and **in operator** to check whether the key is present.



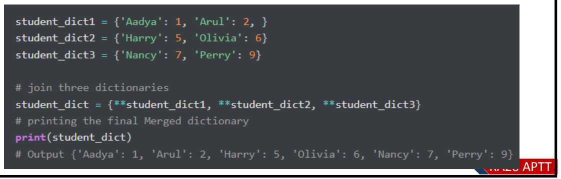
```
person = {'name': 'Jessa', 'country': 'USA', 'telephone': 1178}

# Get the list of keys and check if 'country' key is present
key_name = 'country'
if key_name in person.keys():
    print("country name is", person[key_name])
else:
    print("Key not found")
# Output country name is USA
```

Join two Dictionary

•We can add two dictionaries using the update() method or unpacking arbitrary keywords operator **.



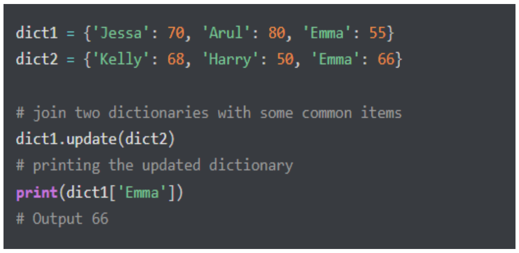
```
dict1 = {'Jessa': 70, 'Arul': 80, 'Emma': 55}
dict2 = {'Kelly': 68, 'Harry': 50, 'Olivia': 66}

# copy second dictionary into first dictionary
dict1.update(dict2)
# printing the updated dictionary
print(dict1)
# output {'Jessa': 70, 'Arul': 80, 'Emma': 55, 'Kelly': 68, 'Harry': 50, 'Olivia'
```



```
student_dict1 = {'Aadya': 1, 'Arul': 2, }
student_dict2 = {'Harry': 5, 'Olivia': 6}
student_dict3 = {'Nancy': 7, 'Perry': 9}

# join three dictionaries
student_dict = {**student_dict1, **student_dict2, **student_dict3}
# printing the final Merged dictionary
print(student_dict)
# Output {'Aadya': 1, 'Arul': 2, 'Harry': 5, 'Olivia': 6, 'Nancy': 7, 'Perry': 9}
```

Join two dictionaries having few items in common

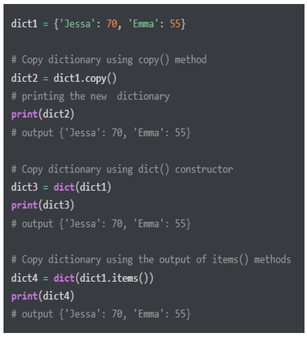
•if **both the dictionaries** have a common key then the **first dictionary value** will be **overridden with the second dictionary value**..



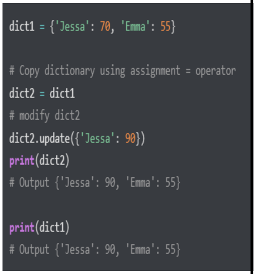
```
dict1 = {'Jessa': 70, 'Arul': 80, 'Emma': 55}
dict2 = {'Kelly': 68, 'Harry': 50, 'Emma': 66}

# join two dictionaries with some common items
dict1.update(dict2)
# printing the updated dictionary
print(dict1['Emma'])
# Output 66
```

Copy a Dictionary

•Using copy() method , the dict() constructor and = Operator



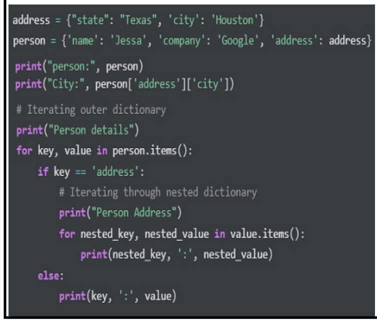
```
dict1 = {'Jessa': 70, 'Emma': 55}

# Copy dictionary using copy() method
dict2 = dict1.copy()
# printing the new  dictionary
print(dict2)
# output {'Jessa': 70, 'Emma': 55}

# Copy dictionary using dict() constructor
dict3 = dict(dict1)
print(dict3)
# output {'Jessa': 70, 'Emma': 55}

# Copy dictionary using the output of items() methods
dict4 = dict(dict1.items())
print(dict4)
# output {'Jessa': 70, 'Emma': 55}
```



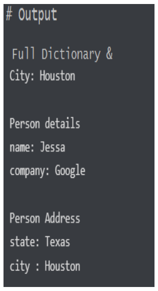
```
dict1 = {'Jessa': 70, 'Emma': 55}

# Copy dictionary using assignment = operator
dict2 = dict1
# modify dict2
dict2.update({'Jessa': 90})
print(dict2)
# Output {'Jessa': 90, 'Emma': 55}

print(dict1)
# Output {'Jessa': 90, 'Emma': 55}
```

Nested Dictionary

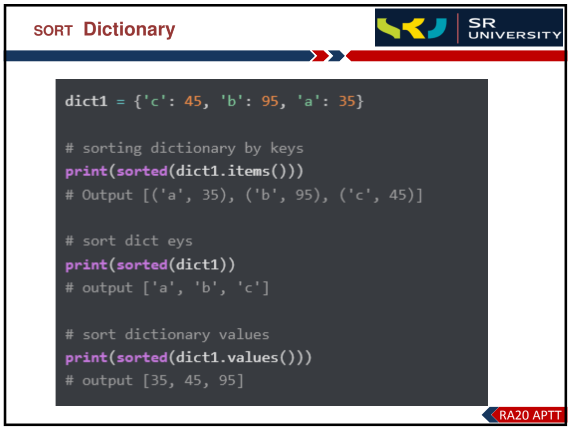
•Nested dictionaries are dictionaries that have one or more dictionaries as their members. It is a collection of many dictionaries in one dictionary.



```
address = {"state": "Texas", 'city': 'Houston'}
person = {'name': 'Jessa', 'company': 'Google', 'address': address}

print("person:", person)
print("City:", person['address']['city'])

# Iterating outer dictionary
print("Person details")
for key, value in person.items():
    if key == 'address':
        # Iterating through nested dictionary
        print("Person Address")
        for nested_key, nested_value in value.items():
            print(nested_key, ':', nested_value)
    else:
        print(key, ':', value)
```



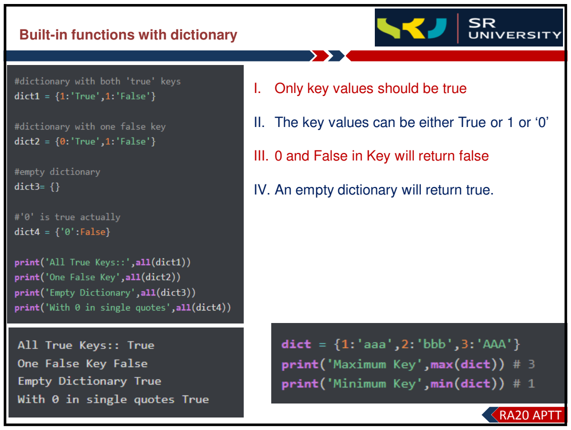
```
# Output

 Full Dictionary &
City: Houston


Person details
name: Jessa
company: Google


Person Address
state: Texas
city : Houston
```



SORT Dictionary

```
dict1 = {'c': 45, 'b': 95, 'a': 35}

# sorting dictionary by keys
print(sorted(dict1.items()))
# Output [('a', 35), ('b', 95), ('c', 45)]

# sort dict eys
print(sorted(dict1))
# output ['a', 'b', 'c']

# sort dictionary values
print(sorted(dict1.values()))
# output [35, 45, 95]
```

RA20 APTT



Built-in functions with dictionary

```
#dictionary with both 'true' keys
dict1 = {1:'True',1:'False'}

#dictionary with one false key
dict2 = {0:'True',1:'False'}

#empty dictionary
dict3= {}

#'0' is true actually
dict4 = {'0':False}

print('All True Keys::',all(dict1))
print('One False Key',all(dict2))
print('Empty Dictionary',all(dict3))
print('With 0 in single quotes',all(dict4))
```

I. Only key values should be true

II. The key values can be either True or 1 or ‘0’

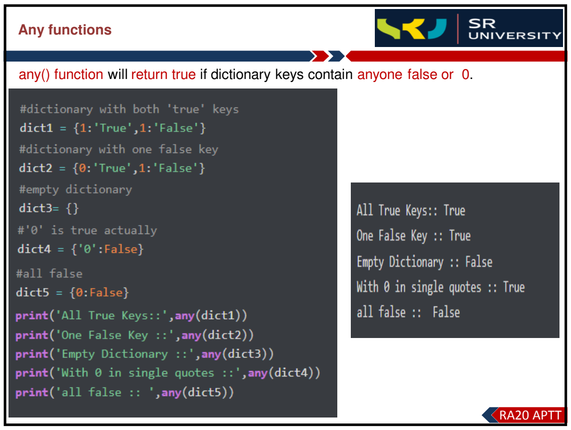
III. 0 and False in Key will return false

IV. An empty dictionary will return true.

```
All True Keys:: True
One False Key False
Empty Dictionary True
With 0 in single quotes True
```

```
dict = {1:'aaa',2:'bbb',3:'AAA'}
print('Maximum Key',max(dict)) # 3
print('Minimum Key',min(dict)) # 1
```

RA20 APTT



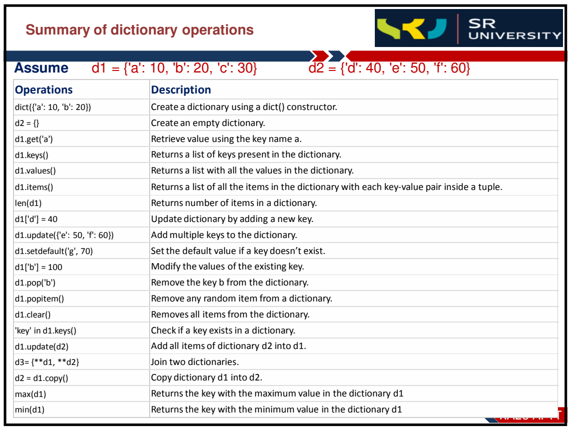
Any functions

any() function will return true if dictionary keys contain anyone false or 0.

```
#dictionary with both 'true' keys
dict1 = {1:'True',1:'False'}
#dictionary with one false key
dict2 = {0:'True',1:'False'}
#empty dictionary
dict3= {}
#'0' is true actually
dict4 = {'0':False}
#all false
dict5 = {0:False}
print('All True Keys::',any(dict1))
print('One False Key ::',any(dict2))
print('Empty Dictionary ::',any(dict3))
print('With 0 in single quotes ::',any(dict4))
print('all false :: ',any(dict5))
```

```
All True Keys:: True
One False Key :: True
Empty Dictionary :: False
With 0 in single quotes :: True
all false ::  False
```

RA20 APTT



Summary of dictionary operations

Assume d1 = {'a': 10, 'b': 20, 'c': 30} d2 = {'d': 40, 'e': 50, 'f': 60}

Operations	Description
dict({'a': 10, 'b': 20})	Create a dictionary using a dict() constructor.
d2 = {}	Create an empty dictionary.
d1.get('a')	Retrieve value using the key name a.
d1.keys()	Returns a list of keys present in the dictionary.
d1.values()	Returns a list with all the values in the dictionary.
d1.items()	Returns a list of all the items in the dictionary with each key-value pair inside a tuple.
len(d1)	Returns number of items in a dictionary.
d1['d'] = 40	Update dictionary by adding a new key.
d1.update({'e': 50, 'f': 60})	Add multiple keys to the dictionary.
d1.setdefault('g', 70)	Set the default value if a key doesn’t exist.
d1['b'] = 100	Modify the values of the existing key.
d1.pop('b')	Remove the key b from the dictionary.
d1.popitem()	Remove any random item from a dictionary.
d1.clear()	Removes all items from the dictionary.
'key' in d1.keys()	Check if a key exists in a dictionary.
d1.update(d2)	Add all items of dictionary d2 into d1.
d3= {**d1, **d2}	Join two dictionaries.
d2 = d1.copy()	Copy dictionary d1 into d2.
max(d1)	Returns the key with the maximum value in the dictionary d1
min(d1)	Returns the key with the minimum value in the dictionary d1