



# PYTHON

# LECTURE 24



# Today's Agenda



- **List -I**

- What Is A List ?
- Creating A List
- Accessing The List Elements
- Adding New Data In The List
- The Slice Operator With List



# What Is A List ?



- Unlike **C++** or **Java**, **Python** doesn't has **arrays**.
  
- So , to hold a **sequence of values**, Python provides us a special built in class called '**list**' .
  
- Thus a **list** in **Python** is defined as ***a collection of values***.

# Important Characteristics Of A List



- The important characteristics of **Python lists** are as follows:
  - **Lists are ordered.**
  - **Lists can contain any arbitrary objects.**
  - **List elements can be accessed by index.**
  - **Lists can be nested to arbitrary depth.**
  - **Lists are mutable.**
  - **Lists are dynamic.**



# How To Create A List ?



- In **Python**, a list is created by placing all the items (elements) inside a square bracket **[ ]**, separated by **commas**.
- It can contain **heterogeneous** elements also.

```
# empty list
```

```
my_list = []
```

```
# list of integers
```

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

```
# list with mixed datatypes
```

```
my_list = [1, "Hello", 3.4]
```



# Other Ways Of Creating List



- We also can create a list by using the **list( )** function

```
# Create an empty list
```

```
list1 = list()
```

```
# Create a list with elements 22, 31, 61
```

```
list2 = list([22, 31, 61])
```

```
# Create a list with strings
```

```
list3 = list(["tom", "jerry", "spyke"])
```

```
# Create a list with characters p, y, t, h, o, n
```

```
list4 = list("python")
```



# Printing The List



- We can print a list in **three** ways:
  - Directly passing it to the **print( )** function
  - Accessing individual elements using **subscript operator [ ]**
  - Accessing multiple elements using **slice operator [ : ]**



# Printing The List



```
mynums=[10,20,30,40,50]
print(mynums)
```

## Output:

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



# Accessing Individual Elements



- A list in Python has indexes running from **0** to **size-1**

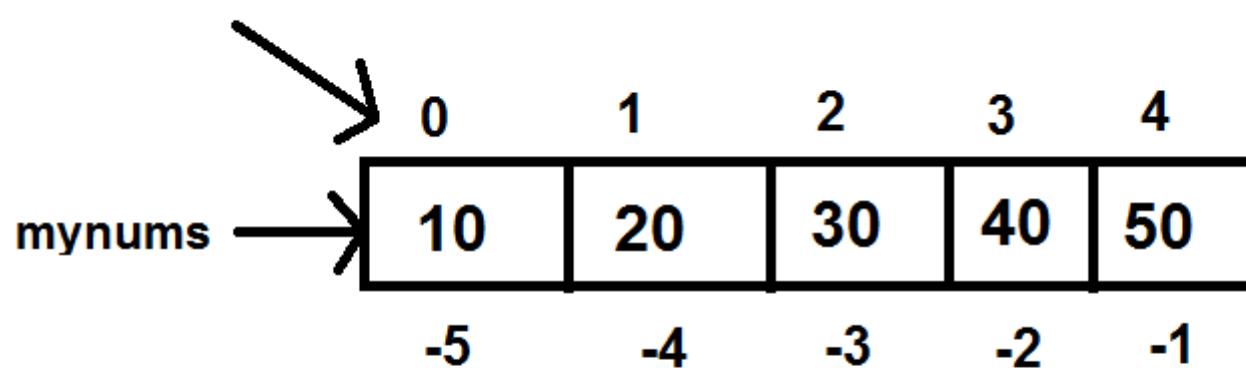
- **For example:**

- **mynums=[10,20,30,40,50]**
  - The above code will create a logical diagram in memory, where positive indexing will go from **0** to **4** and negative indexing from **-1** to **-5**



# Accessing Individual Elements

## Forward Indexing



Backward Indexing



# Accessing Individual Elements



```
mynums=[10,20,30,40,50]
print(mynums[0])
print(mynums[1])
print(mynums[-3])
print(mynums[-2])
```

## Output:

**10**

**20**

**30**

**40**

# Accessing List Elements Using While Loop



```
mynums=[10,20,30,40,50]  
n=len(mynums)  
i=0  
while i<n:  
    print(mynums[i])  
    i=i+1
```

## Output:

10  
20  
30  
40  
50

Just like **len()** works with **strings**, similarly it also works with **list** also and returns **number of elements in the list**

# Accessing List Elements Using For Loop



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

```
for x in mynums:
```

```
    print(x)
```

Output:

10

20

30

40

50

Since list is a sequence type , so for loop can iterate over individual elements of the list



## Exercise



- Redesign the previous code using for loop only to traverse the list in reverse order. Don't use slice operator



# Solution



```
mynums=[10,20,30,40,50]
n=len(mynums)
for i in range(n-1,-1,-1):
    print(mynums[i])
```

## Output:

**50**  
**40**  
**30**  
**20**  
**10**



# Adding New Data In The List



- The most common way of adding a new element to an existing list is by calling the **append( )** method.
- This method takes one argument and adds it at the end of the list

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

```
print(mynums)
```

```
mynums.append(60)
```

```
print(mynums)
```

Output:

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

```
[10,20,30,40,50,60]
```

Remember , lists are **mutable** . So **append()** method doesn't create a new list , rather it simply adds a new element to the existing list.

CAN YOU PROVE THIS ?



# Solution

```
mynums=[10,20,30,40,50]
print(mynums)
print(id(mynums))
mynums.append(60)
print(mynums)
print(id(mynums))
```

## Output:

```
[10, 20, 30, 40, 50]
35676744
[10, 20, 30, 40, 50, 60]
35676744
```

As we can see in the both the cases the **id()** function is returning the **same address** . This means that no new list was created.



## Exercise



- Write a program to accept five integers from the user , store them in a list . Display these integers and also display their sum
  
- Output:

```
Enter 1 element:10
Enter 2 element:20
Enter 3 element:30
Enter 4 element:40
Enter 5 element:50
The list is:
10
20
30
40
50
Sum is 150
```



# Solution



```
mynums=[]
i=1
while i<=5:
    x=int(input("Enter "+str(i)+" element:"))
    mynums.append(x)
    i=i+1
print("The list is:")
sum=0
for x in mynums:
    print(x)
    sum+=x
print("Sum is",sum)
```



# Slice Operator With List



- Just like we can apply slice operator with strings , similarly Python allows us to apply slice operator with lists also.
  
- **Syntax:** `list_var[x:y]`
  - **x** denotes the **start index** of slicing and **y** denotes the **end index** . But **Python** ends slicing at **y-1** index.



# The Slicing Operator

- **Example:**

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

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[3:5])
```

- **Output:**

[20,30,40]

- **Output:**

[40,50]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[0:4])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[0:10])
```

- **Output:**

[10,20,30,40]

- **Output:**

[10,20,3040,50]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[2:2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[6:10])
```

- **Output:**

[ ]

- **Output:**

[ ]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[1: ])
```

- **Output:**

[20,30,40,50 ]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[:3])
```

- **Output:**

[10,20,30 ]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[ :-2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-2:])
```

- **Output:**

[10, 20,30 ]

- **Output:**

[40,50]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-2:1])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-2:-2])
```

- **Output:**

[ ]

- **Output:**

[ ]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-2:2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-2:4])
```

- **Output:**

[ ]

- **Output:**

[40]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-4:2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[1:-2])
```

- **Output:**  
**[20 ]**

- **Output:**  
**[20,30]**



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-2: -1])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1:-2])
```

- **Output:**

[40]

- **Output:**

[]



# Using Step Value

- String slicing can accept a **third parameter** also after the two index numbers.
- The **third parameter** is called **step value**.
- So the complete syntax of slicing operator is:

**s[begin:end:step]**

- Step value indicates *how many characters to move forward after the first character is retrieved* from the string and it's default value is **1** , but can be changed as per our choice.



# Using Step Value

- Another point to understand is that if **step** is **positive** or **not mentioned** then
  - **Movement is in forward direction ( L → R)**
  - **Default for start is 0 and end is len**
- But if **step** is **negative** , then
  - **Movement is in backward direction ( R ← L)**
  - **Default for start is -1 and end is -(len+1)**



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[1:4:2])
```

- **Output:**  
**[20,40]**

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[1:4:0])
```

- **Output:**  
**ValueError: Slice  
step cannot be  
0**



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[4:1:-1])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[4:1:-1])
```

- **Output:**

[ ]

- **Output:**

[50,40,30]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[:])
```

- **Output:**

[10,20,30,40,50 ]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[::-1])
```

- **Output:**

[50,40,30,20,10]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[::-2])
```

- **Output:**

[50,30,10 ]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[::-2])
```

- **Output:**

[10,30,50]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-1:-4:])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1:-4:-1])
```

- **Output:**

```
[]
```

- **Output:**

```
[50,40,30]
```



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-4:-1:1])
```

- **Output:**

[20,30,40]

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-4:-1:-1])
```

- **Output:**

[]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-1: :-2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1::2])
```

- **Output:**

[50,30,10]

- **Output:**

[50]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[-1:4 :2])
```

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-3::-1])
```

- **Output:**

[]

- **Output:**

[30,20,10]



# The Slicing Operator

- **Example:**

```
mynums=[10,20,30,40,50]  
print(mynums[:-3:-1])
```

- **Output:**  
**[50,40]**

- **Example:**

```
mynums=[10,20,30,  
        40,50]  
print(mynums[-1:-1:-1])
```

- **Output:**  
**[]**



# PYTHON

# LECTURE 25



# Today's Agenda



## • **List -II**

- Modifying A List
- Deletion In A List
- Appending / Prepending Items In A List
- Multiplying A List
- Membership Operators On List



# Modifying A List

- Python allows us to **edit/change/modify** an element in a list by simply using it's **index** and assigning a **new value** to it

## Syntax:

```
list_var[index_no]=new_value
```

## Example

```
sports=["cricket","hockey","football"]
```

```
print(sports)
```

```
sports[1]="badminton"
```

```
print(sports)
```

## Output

```
['cricket', 'hockey', 'football']
['cricket', 'badminton', 'football']
```



# Guess The Output ?



```
sports=["cricket","hockey","football"]
print(sports)
sports[3]="badminton"
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football']
Traceback (most recent call last):
  File "listdemo9.py", line 3, in <module>
    sports[3]="badminton"
IndexError: list assignment index out of range
```



# Modifying Multiple Values

- Python allows us to modify **multiple continuous list values** in **a single statement**, which is done using the regular **slice operator**.

## Syntax:

`list_var[m:n]=[list of new value ]`

## Example:

```
sports=["cricket","hockey","football","snooker"]
```

```
print(sports)
```

```
sports[1:3]=["badminton","tennis"]
```

```
print(sports)
```

## Output:

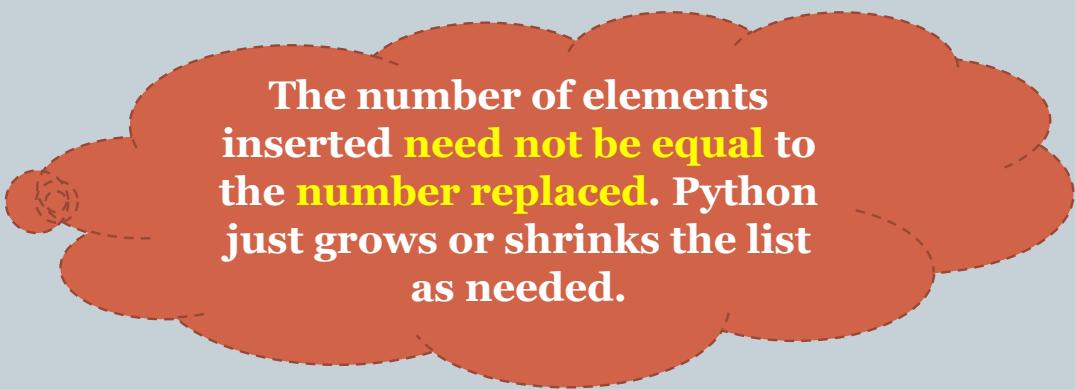
```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'snooker']
```



# Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:3]=["badminton","tennis","rugby","table
tennis"]
print(sports)
```

## Output:

A thought bubble graphic composed of three overlapping orange rounded rectangles. Inside the bubble, there is a small icon of a brain. The text within the bubble reads: "The number of elements inserted **need not be equal** to the **number replaced**. Python just grows or shrinks the list as needed."

The number of elements inserted **need not be equal** to the **number replaced**. Python just grows or shrinks the list as needed.

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'rugby', 'table tennis', 'snooker']
```



# Guess The Output ?



```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:2]=["badminton","tennis","rugby","table
tennis"]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'rugby', 'table tennis', 'football', 'snooker']
[]
```



# Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:1]=["badminton","tennis"]
print(sports)
```

## Output:

If we have **end index same or less than start index** , then Python **doesn't remove anything** . Rather it simply **inserts new elements** at the given index and **shifts the existing element**

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'hockey', 'football', 'snooker']
```



# Guess The Output ?



```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:0]=["badminton","tennis"]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'hockey', 'football', 'snooker']
```



# Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:-1]=["badminton","tennis"]
print(sports)
```

Since **-1** is present in the list ,  
Python **removed items** from **1** to  
**second last item** of the list and  
inserted new items there

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'snooker']
```



# Guess The Output ?



```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:-2]=["badminton","tennis"]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'badminton', 'tennis', 'football', 'snooker']
```



# Deleting Item From The List



- **Python** allows us to delete an item from the list by calling the **operator/keyword** called **del**

## Syntax:

**del list\_var[index\_no]**

## Example:

```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[3]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'hockey', 'football']
```



# Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[4]
print(sports)
```

Subscript operator will generate  
IndexError whenever we pass  
invalid index

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
Traceback (most recent call last):
  File "listdemo11.py", line 3, in <module>
    del sports[4]
IndexError: list assignment index out of range
```



# Deleting Multiple Items



- Python allows us to **delete** multiple items from the list in **2 ways:**
  
- By assigning empty list to the appropriate slice
  
- OR
  
- By passing slice to the del operator



# Example



## □ Assigning Empty List

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:3]=[]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'snooker']
```



# Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
sports[1:5]=[]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket']
```

Slice operator never generates IndexError , so the code will work fine and remove all the items from given start index to the end of the list



# Example

## □ Passing slice to del operator

```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[1:3]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket', 'snooker']
```



# Guess The Output ?

```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[1:5]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
['cricket']
```

Here also , since we have used the **slice operator** , no exception will arise



# Guess The Output ?



```
sports=["cricket","hockey","football","snooker"]
print(sports)
del sports[0:4]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
[]
```



# Deleting Entire List

- We can delete or remove the **entire list object** as well as it's **reference** from the memory by passing the **list object reference** to the **del** operator

## Syntax:

`del list_var`

## Example:

```
sports=["cricket","hockey","football","snooker"]
```

```
print(sports)
```

```
del sports
```

```
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'snooker']
Traceback (most recent call last):
  File "listdemo11.py", line 4, in <module>
    print(sports)
NameError: name 'sports' is not defined
```

# Appending Or Prepending Items To A List



- **Additional items** can be added to the **start** or **end** of a list using the **+** concatenation operator or the **`+ =`** compound assignment operator
- The only condition is that the item to be concatenated must be a **list**

## Example:

```
outdoor=["cricket","hockey","football"]
indoor=["carrom","chess","table-tennis"]
allsports=outdoor+indoor
print(allsports)
```

```
['cricket', 'hockey', 'football', 'carrom', 'chess', 'table-tennis']
```



# Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=["carrom","chess","table-tennis"]+sports
print(sports)
```

## Output:

```
['carrom', 'chess', 'table-tennis', 'cricket', 'hockey', 'football']
```



# Guess The Output ?



**evens=[4,6,8]**

**evens=2+evens**

**print(evens)**

## Output:

```
evens=2+evens
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```



# Guess The Output ?



**evens=[4,6,8]**

**evens=list(2)+evens**

**print(evens)**

## Output:

```
evens=list(2)+evens
```

```
TypeError: 'int' object is not iterable
```



# Guess The Output ?



**evens=[4,6,8]**

**evens=[2]+evens**

**print(evens)**

## Output:

```
[2, 4, 6, 8]
```



# Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports+"boxing"
print(sports)
```

## Output:

```
 sports=[cricket,hockey,football]
TypeError: can only concatenate list (not "str") to list
```



# Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports+list("boxing")
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'b', 'o', 'x', 'i', 'n', 'g']
```



# Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports+["boxing"]
print(sports)
```

## Output:

```
['cricket', 'hockey', 'football', 'boxing']
```



# Multiplying A List



- **Python** allows us to **multiply a list by an integer** and when we do so it makes copies of list items that many number of times, while preserving the order.
- **Syntax:**

**list\_var \* n**

**Example:**

```
sports=["cricket","hockey","football"]
```

```
sports=sports*3
```

```
print(sports)
```

**Output:**

```
['cricket', 'hockey', 'football', 'cricket', 'hockey', 'football', 'cricket', 'hockey', 'football']
```



# Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports*3.0
print(sports)
```

## Output:

```
sports=sports*3.0
TypeError: can't multiply sequence by non-int of type 'float'
```



# Guess The Output ?



```
sports=["cricket","hockey","football"]
sports=sports*["boxing"]
print(sports)
```

## Output:

```
sports=sports*["boxing"]
TypeError: can't multiply sequence by non-int of type 'list'
```



# Membership Operator On List



- We can apply **membership operators in** and **not in** on the **list** to **search** for a particular **item**

## Syntax:

**element in list\_var**

## Example:

```
sports=["cricket","hockey","football"]
```

```
print("cricket" in sports)
```

## Output:

```
True
```

# Exercise



- Write a program to accept **5 unique integers** from the user. Make sure if the integer being entered is **already present** in the list your code displays the message "**Item already present**" and ask the user to reenter the integer.

## Output:

```
Enter 5 unique integers:  
Enter element:1  
Enter element:2  
Enter element:1  
Item already present  
Enter element:2  
Item already present  
Enter element:3  
Enter element:4  
Enter element:4  
Item already present  
Enter element:5  
integers inputted by you are:  
1  
2  
3  
4  
5
```



# Solution



```
myints=[]
print("Enter 5 unique integers:")
i=0
while i<=4:
    item=int(input("Enter element:"))
    if item in myints:
        print("Item already present!")
        continue
    myints.append(item)
    i=i+1

print("integers inputted by you are:")
for x in myints:
    print(x)
```

# Exercise



- Write a program to accept 2 lists from the user of 5 nos each . Assume each list will have unique nos  
Now find out how many items in these lists are common .

## Output:

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:2  
Enter element:3  
Enter element:4  
Enter element:5  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have 2 items common
```

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:3  
Enter element:5  
Enter element:7  
Enter element:9  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have no common items
```

# Exercise



- Rewrite the previous code so that your code also displays the items which are common in both the lists

## Output:

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:2  
Enter element:3  
Enter element:4  
Enter element:5  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have 2 items common  
These items are: [2, 4]
```

```
Enter 5 unique nos for first list:  
Enter element:1  
Enter element:3  
Enter element:5  
Enter element:7  
Enter element:9  
Enter 5 unique nos for second list:  
Enter element:2  
Enter element:4  
Enter element:6  
Enter element:8  
Enter element:10  
These lists have no common items
```



# PYTHON

# LECTURE 26



# Today's Agenda



- **List -III**
  - Built In Functions For List



# Built In Functions For List



- There are some **built-in functions** in **Python** that we can use on **lists**.
- These are:
  - **len()**
  - **max()**
  - **min()**
  - **sum()**
  - **sorted()**
  - **list()**
  - **any()**
  - **all()**



# The **len()** Function

- Returns the **number of items** in the list

## Example:

```
fruits=["apple","banana","orange",None]  
print(len(fruits))
```

## Output:

4



# The **max()** Function



- Returns the **greatest** item present in the list

## Example:

```
nums=[5,2,11,3]
```

```
print(max(nums))
```

## Output:

11



# Guess The Output ?



```
months=["january","may","december"]  
print(max(months))
```

## Output:

may



# Guess The Output ?



```
booleans=[False,True]  
print(max(booleans))
```

**Output:**

**True**



# Guess The Output ?



```
mynums=[1.1,1.4,0.9]  
print(max(mynums))
```

## Output:

1.4



# Guess The Output ?



```
mynums=[True,5,False]  
print(max(mynums))
```

## Output:

5



# Guess The Output ?



```
mynums=[0.2,0.4,True,0.5]  
print(max(mynums))
```

## Output:

True



# Guess The Output ?



```
mynums=["True",False]  
print(max(mynums))
```

## Output:

```
print(max(mynums))  
TypeError: '>' not supported between instances of 'bool' and 'str'
```



# Guess The Output ?



```
values=[10,"hello",20,"bye"]  
print(max(values))
```

## Output:

```
print(max(values))  
TypeError: '>' not supported between instances of 'str' and 'int'
```



# Guess The Output ?



```
fruits=["apple","banana","orange"]  
print(max(fruits))
```

**Output:**

**orange**



# Guess The Output ?



```
fruits=["apple","banana","orange",None]  
print(max(fruits))
```

## Output:

```
print(max(fruits))  
TypeError: '>' not supported between instances of 'NoneType' and 'str'
```



# The **min()** Function



- Returns the **least** item present in the list

## Example:

```
nums=[5,2,11,3]
```

```
print(min(nums))
```

## Output:

2



# Guess The Output ?



```
months=["january","may","december"]  
print(min(months))
```

## Output:

december



# The **sum()** Function



- Returns the **sum** of all the **items** present in the list .  
However items must be of Numeric or boolean type

## Example:

```
nums=[10,20,30]  
print(sum(nums))
```

## Output:

60



# Guess The Output ?



```
nums=[10,20,30,True,False]  
print(sum(nums))
```

**Output:**

**61**



# Guess The Output ?



```
nums=['1','2','3']  
print(sum(nums))
```

## Output:

```
print(sum(nums))  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



# Guess The Output ?



```
nums=[2.5,3.5,4.5]  
print(sum(nums))
```

**Output:**

**10.5**



# The **sorted( )** Function



- Returns a **sorted version** of the **list** passed as argument.

## Example:

```
nums=[7,4,9,1]  
print(sorted(nums))  
print(nums)
```

## Output:

```
[1, 4, 7, 9]  
[7, 4, 9, 1]
```



# Guess The Output ?



```
months=["january","may","december"]  
print(sorted(months))
```

## Output:

[“december”, “january”, “may”]



# Guess The Output ?



```
months=["january","may","december",3]  
print(sorted(months))
```

## Output:

```
print(sorted(months))  
TypeError: '<' not supported between instances of 'int' and 'str'
```



# Guess The Output ?



```
values=[2.4,1.0,2,3.6]  
print(sorted(values))
```

## Output:

[1.0,2,2.4,3.6]



# Guess The Output ?



```
values=["bhupal","bhop","Bhopal"]  
print(sorted(values))
```

## Output:

["Bhopal", "bhop", "bhupal"]



# Sorting In Descending Order

- To **sort** the **list** in **descending order** , we can pass the **keyword argument reverse** with value set to **True** to the function **sorted( )**

## Example:

```
nums=[3,1,5,2]
```

```
print(sorted(nums,reverse=True))
```

## Output:

```
[5, 3, 2, 1]
```



# The **list()** Function

- The **list()** function converts an **iterable** i.e **tuple** , **range**, **set** , **dictionary** and **string** to a **list**.

## Syntax:

**list(iterable)**

## Example:

```
city="bhopal"  
x=list(city)  
print(x)
```

## Output:

```
['b', 'h', 'o', 'p', 'a', 'l']
```



# Guess The Output ?



**n=20**

**x=list(n)**

**print(x)**

## Output:

```
x=list(n)
```

```
TypeError: 'int' object is not iterable
```



# Guess The Output ?



```
n="20"
```

```
x=list(n)
```

```
print(x)
```

## Output:

```
[ '2', '0' ]
```



# Guess The Output ?

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

```
x=list(t)
```

```
print(x)
```

## Output:

```
[10, 20, 30]
```

This is a  
tuple



# The **any()** Function

- The **any()** function accepts a **List** as argument and returns **True** if atleast **one element** of the **List** is **True**. If not, this method returns **False**. If the **List** is empty, then also it returns **False**

## Syntax:

**list(iterable)**

## Example:

```
x = [1, 3, 4, 0]
print(any(x))
```

## Output:

**True**



# Guess The Output ?



**x = [0, False]**

**print(any(x))**

**Output:**

**False**



# Guess The Output ?



```
x = [0, False, 5]  
print(any(x))
```

**Output:**

**True**



# Guess The Output ?



**x= []**

**print(any(x))**

**Output:**

**False**



# The **all()** Function



- The **all()** function accepts a **List** as argument and returns **True** if **all the elements** of the **List** are **True** or if the **List** is **empty**. If not, this method returns **False**.

## Syntax:

**all(iterable)**

## Example:

```
x = [1, 3, 4, 0]  
print(all(x))
```

## Output:

**False**



# Guess The Output ?



```
x = [0, False]  
print(all(x))
```

**Output:**

**False**



# Guess The Output ?



```
x = [1,3,4,5]  
print(all(x))
```

**Output:**

**True**



# Guess The Output ?



```
x = [0, False, 5]  
print(all(x))
```

**Output:**

**False**



# Guess The Output ?



```
x= []
print(all(x))
```

## Output:

True



# PYTHON

# LECTURE 27



# Today's Agenda



- **List -IV**
  - Methods Of List



# List Methods



- There are some **methods** also in **Python** that we can use on **lists**.
- These are:
  - **append()**
  - **extend()**
  - **insert()**
  - **index()**
  - **count()**
  - **remove()**
  - **pop()**
  - **clear()**
  - **sort()**
  - **reverse()**



# The **append( )** Method



- Adds a **single element** to the **end** of the list . Modifies the list in place but doesn't return anything

## Syntax:

**list\_var.append(item)**

## Example:

**primes=[2,3,5,7]**

**primes.append(11)**

**print(primes)**

## Output:

**[2, 3, 5, 7, 11]**



# Guess The Output ?

```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.append(wild_animal)
print(animal)
```

## Output:

```
['cat', 'dog', 'rabbit', ['tiger', 'fox']]
```

It's important to notice that, **append()** adds entire list as a single element .

If we need to add items of a list to the another list (rather than the list itself), then we must use the **extend()** method .



# Guess The Output ?

```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.append(wild_animal)
print(animal[3])
print(animal[3][0])
print(animal[3][0][0])
print(animal[3][1][0])
```

## Output:

```
['tiger', 'fox']
tiger
t
f
```



## Exercise



- Write a program to accept an alphanumeric string from the user. Now extract only digits from the given input and add it to the list . Finally print the list as well as it's sum. **Make sure your list contains numeric representation of digits**

### Output:

```
Enter an alphanumeric string:a1b2c345de56
[1, 2, 3, 4, 5, 5, 6]
```



# Solution

```
text=input("Enter an alphanumeric string:")
nums=[]
for x in text:
    if x in "0123456789":
        nums.append(int(x))
print(nums)
```



# The **extend( )** Method



- **extend()** also adds to the **end of a list**, *but the argument is expected to be an iterable*.
- The items in **<iterable>** are added individually.
- Modifies the list in place but **doesn't return anything**

## Syntax:

`list_var.extend(iterable)`

## Output:

## Example:

`primes=[2,3,5,7]`

`primes.extend([11,13,17])`

`print(primes)`

```
[2, 3, 5, 7, 11, 13, 17]
```



# Guess The Output ?



```
primes=[2,3,5,7]
primes.extend(11)
print(primes)
```

## Output:

```
primes.extend(11)
TypeError: 'int' object is not iterable
```



# Guess The Output ?



```
primes=[2,3,5,7]  
primes.extend([11])  
print(primes)
```

## Output:

```
[2, 3, 5, 7, 11]
```



# Guess The Output ?

```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.extend(wild_animal)
print(animal)
```

## Output:

```
['cat', 'dog', 'rabbit', 'tiger', 'fox']
```



# Guess The Output ?

```
animal = ['cat', 'dog', 'rabbit']
wild_animal = ['tiger', 'fox']
animal.extend(wild_animal)
print(animal[3])
print(animal[3][0])
print(animal[3][1])
```

## Output:

```
tiger
t
i
```



# Guess The Output ?



```
colors=["red","green"]
colors.extend("blue")
print(colors)
```

## Output:

```
['red', 'green', 'b', 'l', 'u', 'e']
```



# Guess The Output ?



```
colors=["red","green"]
colors.extend(["blue"])
print(colors)
```

## Output:

```
['red', 'green', 'blue']
```



# Guess The Output ?



```
a = [1, 2, 3]  
b = [4, 5, 6].extend(a)  
print(b)
```

## Output:

None



# The **insert( )** Method



- The **insert()** method inserts the element to the list at the given index.
- Modifies the list in place but **doesn't return anything**

## Syntax:

`list_var.insert(index,item)`

## Example:

```
primes=[2,3,7,9]  
primes.insert(2,5)  
print(primes)
```

## Output:

```
[2, 3, 5, 7, 9]
```



# Guess The Output ?



```
primes=[2,3,7,9]  
primes.insert(-2,5)  
print(primes)
```

## Output:

```
[2, 3, 5, 7, 9]
```



# Guess The Output ?

```
primes=[2,3,5,7]  
primes.insert(5,9)  
print(primes)
```

## Output:

```
[2, 3, 5, 7, 9]
```

The method **insert( )** works like slicing operator . So even if the index given is beyond range , it will add the element at the end.



# Guess The Output ?



```
primes=[2,3,5,7]
primes.insert(-5,9)
print(primes)
```

## Output:

```
[9, 2, 3, 5, 7]
```



## Exercise



- Write a program to accept any 5 random integers from the user and add them in a list in such a way that list always remains sorted. **DO NOT USE THE FUNCTION sort()**

### Output:

```
Enter any 5 random integers:  
4  
1  
6  
2  
3  
Sorted list is:  
[1, 2, 3, 4, 6]
```



# Solution

```
i=1
sortednums=[]
print("Enter any 5 random integers:")
while i<=5:
    n=int(input())
    pos=0
    for x in sortednums:
        if x>n:
            break
        pos=pos+1
    sortednums.insert(pos,n)
    i=i+1
print("Sorted list is:")
print(sortednums)
```



# The **index( )** Method



- The **index()** method searches an element in the **list** and returns it's **index**.
- If the element occurs **more than once** it returns it's **smallest/first position**.
- If element is **not found**, it raises a **ValueError** exception

## Syntax:

**list\_var.index(item)**

## Example:

```
primes=[2,3,5,7]
pos=primes.index(5)
print("position of 5 is",pos)
```

## Output:

```
position of 5 is 2
```



# Guess The Output ?

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
pos = vowels.index('e')
print('The index of e:',pos)
pos = vowels.index('i')
print('The index of i:',pos)
```

## Output:

```
The index of e: 1
The index of i: 2
```



# Guess The Output ?



```
mynums = [10,20,30,40,50]
x = mynums.index(20)
print("20 occurs at",x,"position")
x = mynums.index(60)
print("60 occurs at",x,"position")
x = mynums.index(10)
print("10 occurs at",x,"position")
```

## Output:

```
20 occurs at 1 position
Traceback (most recent call last):
  File "listdemo30.py", line 4, in <module>
    x = mynums.index(60)
ValueError: 60 is not in list
```



# Guess The Output ?

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
pos = vowels.index('f')
print('The index of e:',pos)
pos = vowels.index('i')
print('The index of i:',pos)
```

## Output:

```
Traceback (most recent call last):
  File "listdemo30.py", line 2, in <module>
    pos = vowels.index('f')
ValueError: 'f' is not in list
```



# The **count( )** Method



- The **count()** method returns the **number of occurrences** of an element in a **list**
- In simple terms, it **counts** how many times an element has occurred in a **list** and returns it.

## Syntax:

**list\_var.count(item)**

## Output:

### Example:

```
country=['i','n','d','i','a']
```

```
x=country.count('i')
```

```
print("i occurs",x,"times in",country)
```

```
i occurs 2 times in '['i', 'n', 'd', 'i', 'a']'
```



# Guess The Output ?

```
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
x = vowels.count('i')
print("i occurs",x,"times")
x = vowels.count('e')
print("e occurs",x,"times")
x = vowels.count('j')
print("j occurs",x,"times")
```

## Output:

```
i occurs 2 times
e occurs 1 times
j occurs 0 times
```



# Guess The Output ?



```
points = [1, 4, 2, 9, 7, 8, 9, 3, 1]  
x = points.count(9)  
print("9 occurs",x,"times")
```

## Output:

```
9 occurs 2 times
```



# Guess The Output ?



```
strings = ['Cat', 'Bat', 'Sat', 'Cat', 'cat', 'Mat']
x=strings.count("Cat")
print("Cat occurs",x,"times")
```

## Output:

```
Cat occurs 2 times
```



# The **remove( )** Method



- The **remove()** method **searches** for the given element in the list and **removes the first matching element**.
- If the **element**(argument) passed to the **remove()** method doesn't exist, **ValueError** exception is thrown.

## Syntax:

`list_var.remove(item)`

## Output:

### Example:

```
vowels=['a','e','i','o','u']
vowels.remove('a')
print(vowels)
```

```
['e', 'i', 'o', 'u']
```



# Guess The Output ?

```
subjects=["phy","chem","maths"]
subjects.remove("chem")
print(subjects)
```

## Output:

```
['phy', 'maths']
```



# Guess The Output ?

```
subjects=["phy","chem","maths","chem"]
subjects.remove("chem")
print(subjects)
subjects.remove("chem")
print(subjects)
subjects.remove("chem")
print(subjects)
```

## Output:

```
['phy', 'maths', 'chem']
['phy', 'maths']
Traceback (most recent call last):
  File "listdemo33.py", line 6, in <module>
    subjects.remove("chem")
ValueError: list.remove(x): x not in list
```



# The **pop()** Method



- The **pop()** method **removes** and **returns** the element at the given index (passed as an argument) from the list.

## Syntax:

**list\_var.pop(index)**

- Important points about **pop()** method:
  - If the **index** passed to the **pop()** method is not in the range, it throws **IndexError: pop index out of range** exception.
  - The **parameter** passed to the **pop()** method is **optional**. If no parameter is passed, the **default index -1 is passed** as an argument which **returns the last element**
  - The **pop()** method returns the element present at the given index.
  - Also, the **pop()** method updates the list after removing the element



# Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(1))
print(fruits)
```

## Output:

```
['apple', 'banana', 'cherry']
banana
['apple', 'cherry']
```



# Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop())
print(fruits)
```

## Output:

```
['apple', 'banana', 'cherry']
cherry
['apple', 'banana']
```



# Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(3))
print(fruits)
```

## Output:

```
['apple', 'banana', 'cherry']
Traceback (most recent call last):
  File "listdemo34.py", line 3, in <module>
    print(fruits.pop(3))
IndexError: pop index out of range
```



# Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(-3))
print(fruits)
```

## Output:

```
['apple', 'banana', 'cherry']
apple
['banana', 'cherry']
```



# Guess The Output ?



```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
print(fruits.pop(-4))
print(fruits)
```

## Output:

```
['apple', 'banana', 'cherry']
Traceback (most recent call last):
  File "listdemo34.py", line 3, in <module>
    print(fruits.pop(-4))
IndexError: pop index out of range
```



# del v/s pop() v/s remove()



- **pop()** : Takes Index , removes element & returns it
- **remove()** : Takes value, removes first occurrence and returns nothing
- **del** : Takes index, removes value at that index and returns nothing
- Even their exceptions are also different if index is wrong or element is not present:
  - **pop()** : throws **IndexError**: pop index out of range
  - **remove()**: throws **ValueError**: list.remove(x): x not in list
  - **del**: throws **IndexError**: list assignment index out of range



# The **clear()** Method



- The **clear()** method removes all items from the list.
- It only empties the given **list** and doesn't return any value.

## Syntax:

**list\_var.clear()**

## Example:

```
fruits = ['apple', 'banana', 'cherry']
print(fruits)
fruits.clear()
print(fruits)
```

## Output:

```
['apple', 'banana', 'cherry']
[]
```



# The **sort()** Method



- The **sort()** method **sorts** the elements of a given list.
- The order can be **ascending** or **descending**

## Syntax:

`list_var.sort(reverse=True|False, key=name of func)`

## Parameter Values:

Parameter Name	Description
<b>reverse</b>	<b>Optional.</b> <code>reverse=True</code> will sort the list descending. Default is <code>reverse=False</code>
<b>key</b>	<b>Optional.</b> A function to specify the sorting criteria(s)



# Guess The Output ?



```
vowels = ['e', 'a', 'u', 'o', 'i']
```

```
vowels.sort()
```

```
print(vowels)
```

## Output:

```
['a', 'e', 'i', 'o', 'u']
```



# Guess The Output ?



```
vowels = ['e', 'a', 'u', 'o', 'i']
vowels.sort(reverse=True)
print(vowels)
```

## Output:

```
['u', 'o', 'i', 'e', 'a']
```



# Guess The Output ?



```
a = ["bee", "wasp", "moth", "ant"]  
a.sort()  
print(a)
```

## Output:

```
['ant', 'bee', 'moth', 'wasp']
```



# Guess The Output ?



```
a = ["bee", "wasp", "moth", "ant"]  
a.sort(reverse=True)  
print(a)
```

## Output:

```
['wasp', 'moth', 'bee', 'ant']
```



# Guess The Output ?



```
a = ["bee", "wasp", "moth", "ant"]  
a.sort(key=len)  
print(a)
```

## Output:

```
['bee', 'ant', 'wasp', 'moth']
```



# Guess The Output ?



```
a = ["january", "february", "march"]  
a.sort(key=len,reverse=True)  
print(a)
```

## Output:

```
['february', 'january', 'march']
```



# Guess The Output ?



```
mylist = ["a",10,True]  
mylist.sort()  
print(mylist)
```

## Output:

```
Traceback (most recent call last):  
  File "listdemo34.py", line 2, in <module>  
    mylist.sort()  
TypeError: '<' not supported between instances of 'int' and 'str'
```



# Guess The Output ?



**mylist = [25,10,True,False]**

**mylist.sort()**

**print(mylist)**

## Output:

```
[False, True, 10, 25]
```

# Passing Our Own Function As Key



- We also can pass our own function name to be used as key but it should take only 1 argument and return some value based on that argument.
- **This return value will be used by Python as key to sorting**



# Guess The Output ?



```
def sortSecond(val):  
    return val[1]  
  
list1 = [(1,2),(3,3),(1,1)]  
list1.sort(key=sortSecond)  
print(list1)
```

## Output:

```
[(1, 1), (1, 2), (3, 3)]
```



# Guess The Output ?



```
def sortSecond(val):  
    return 0  
  
list1 = [(1,2),(3,3),(1,1)]  
list1.sort(key=sortSecond)  
print(list1)
```

## Output:

```
[(1, 2), (3, 3), (1, 1)]
```



# Guess The Output ?

```
student_rec = [['john', 'A', 12],['jane', 'B', 7],['dave',  
 'B', 10]]
```

```
student_rec.sort()  
print(student_rec)
```

## Output:

```
[['dave', 'B', 10], ['jane', 'B', 7], ['john', 'A', 12]]
```



# Guess The Output ?



```
def myFunc(val):  
    return val[2]
```

```
student_rec = [['john', 'A', 12],['jane', 'B', 7],['dave',  
 'B', 10]]
```

```
student_rec.sort(key=myFunc)  
print(student_rec)
```

## Output:

```
[['jane', 'B', 7], ['dave', 'B', 10], ['john', 'A', 12]]
```



# The **reverse( )** Method



- The **reverse()** method **reverses** the elements of a given list.
- It doesn't return any value. It only **reverses the elements** and **updates the list**.

## Syntax:

**list\_var.reverse( )**

## Example:

```
os = ['Windows', 'macOS', 'Linux']
print('Original List:', os)
os.reverse()
print('Updated List:', os)
```

## Output:

```
Original List: ['Windows', 'macOS', 'Linux']
Updated List: ['Linux', 'macOS', 'Windows']
```



# PYTHON

# LECTURE 28



# Today's Agenda



- **List -V**
  - List Comprehension



# What Is Comprehension ?



- Comprehensions are **constructs** that allow **sequences** to be built from other sequences.
  
- In simple words to build a **List** from another **List** or **Set** from another **Set** , we can use **Comprehensions**
  
- **Python 2.0** introduced **list comprehensions** and **Python 3.0** comes with **dictionary** and **set comprehensions**.

# Understanding List Comprehension



- To understand **List comprehensions** let's take a **programming challenge**.
- Suppose you want to take the letters in the word “**Bhopal**”, and put them in a **list**.
- **Can you tell in how many ways can you do this ?**
- Till now , we know **2 ways** to achieve this:
  - **Using for loop**
  - **Using lambda**



# Using “for” Loop



```
text="Bhopal"  
myList=[]  
for x in text:  
    myList.append(x)  
print(myList)
```

## Output:

```
[ 'B' , 'h' , 'o' , 'p' , 'a' , 'l' ]
```



# Using Lambda



```
myList=list(map(lambda x:x , "Bhopal"))
print(myList)
```

## Output:

```
[ 'B' , 'h' , 'o' , 'p' , 'a' , 'l' ]
```

# Understanding List Comprehension



- Now , we can solve the same problem by using **List Comprehension** also .
  
- The advantage is that ***List Comprehensions are 35% faster than FOR loop and 45% faster than map function***

# Understanding List Comprehension



- To understand this , look at the same code using **List Comprehension**:

```
myList=[x for x in "Bhopal"]  
print(myList)
```

## Output:

```
['B', 'h', 'o', 'p', 'a', 'l']
```

# Syntax For List Comprehension



## □ Syntax:

**list\_variable = [x for x in iterable <test\_cond>]**

## □ Explanation

- For a Python **List Comprehension**, we use the delimiters for a **list-square brackets**.
- Inside those, we use a **for-statement** on an **iterable**.
- Then there is an **optional test condition** we can apply on each member of **iterable**
- Finally we have our **output expression**



# Exercise



- Write a program to produce **square** of nos from **1 to 5** , store them in a **list** and **print** the list.

**Do the code using:**

1. **Normal for loop**
2. **List Comprehension**

**Output:**

**[1, 4, 9, 16, 25]**



# Using **for** Loop



```
squaresList=[]
for i in range(1,6):
    squaresList.append(i**2)

print(squaresList)
```



# Using List Comprehension



```
squaresList=[x**2 for x in range(1,6)]  
print(squaresList)
```



## Exercise



- Write a program to accept a string from the user and **convert** each word of the given string to **uppercase** , store it in a **list** and **print the list**

**Do the code using:**

1. **Normal for loop**
2. **List Comprehension**

**Output:**

```
Type a string:my name is sachin
['MY', 'NAME', 'IS', 'SACHIN']
```



# Using **for** Loop



```
text=input("Type a string:")
uppersList=[]
for x in text.split():
    uppersList.append(x.upper())
print(uppersList)
```



# Using List Comprehension



```
text=input("Type a string:")
uppersList=[x.upper()for x in text.split()]
print(uppersList)
```



# Exercise



- Write a program to accept 5 integers from the user and **store** them in a **list** . Now display the **list** and **sum** of the elements.

**Do the code using:**

1. **Normal for loop**
2. **List Comprehension**

**Output:**

```
Enter 5 integers:10 20 30 40 50
List is: [10, 20, 30, 40, 50]
Sum is: 150
```



# Using **for** Loop

```
myNums=[]
text=input("Enter 5 integers:")
for x in text.split():
    myNums.append(int(x))

print("List is:",myNums)
print("Sum is:",sum(myNums))
```



# Using List Comprehension

```
text=input("Enter 5 integers:")  
myNums=[int(x) for x in text.split()]
```

```
print("List is:",myNums)  
print("Sum is:",sum(myNums))
```

# Adding Conditions In List Comprehension



- As previously mentioned , it is possible to add a **test condition** in **List Comprehension**.
  
- When we do this , we get only those items from the **iterable** for which the condition is **True**.
  
- **Syntax:**

```
list_variable = [x  for x in iterable <test_cond>]
```



# Exercise



- Write a program to produce square of **only odd nos from 1 to 5** , store them in a **list** and **print** the list.

## Solution

```
squaresList=[x**2 for x in range(1,6) if x%2!=0]  
print(squaresList)
```

## Output:

```
[1, 9, 25]
```



# Exercise



- Create a function called **removevowels()** which accepts a **string as argument** and **returns a list** with **all the vowels removed** from that string

**Do this code using:**

- Normal for loop
- List Comprehension

**Output:**

```
Type a string:my name is sachin
['m', 'y', ' ', 'n', 'm', ' ', 's', ' ', 's', 'c', 'h', 'n']
```



# Using **for** Loop

```
def removevowels(text):
    myList=[]
    for x in text:
        if x not in "aeiou":
            myList.append(x)
    return myList
```

```
text=input("Type a string:")
finalList=removevowels(text)
print(finalList)
```



# Using List Comprehension

```
def removevowels(text):
    myList=[x for x in text if x not in "aeiou"]
    return myList
```

```
text=input("Type a string:")
finalList=removevowels(text)
print(finalList)
```



## Exercise

- Create a function called **get\_numbers()** which accepts a list of **strings , symbols and numbers as argument** and **returns a list containing only numbers from that list.**

### Output:

```
Actual List
['bhopal', 25, '$', 'hello', 34, 21, 'indore', 22]
List with numbers only
[25, 34, 21, 22]
```



# Solution

```
def get_numbers(myList):
    mynumberList=[x for x in myList if type(x) is int]
    return mynumberList

myList=["bhopal",25,"$","hello",34,21,"indore",22]
print("Actual List")
print(myList)
print("List with numbers only")
mynumberList=get_numbers(myList)
print(mynumberList)
```



## Exercise



- Create a function called **getlength()** which accepts a **string** as **argument** and **returns a list** containing the **length** of all the **words** of that **string** except the **word “the”**. Accept the **string** from the **user**.

### Output:

```
Type a string:the city of Bhopal is the second cleanest city in the country  
[4, 2, 6, 2, 6, 8, 4, 2, 7]
```



# Solution



```
def getlength(str):
    myList=[len(x) for x in str.split() if x!="the"]
    return myList
```

```
text=input("Type a string:")
myList=getlength(text)
print(myList)
```

# Adding Multiple Conditions In List Comprehension



- List Comprehension allows us to mention **more than one if condition**.
  
- To do this we simply have to mention the **next if** after the condition of **first if**
  
- **Syntax:**

```
list_variable = [x for x in iterable <test_cond_1> <test cond_2>]
```



## Exercise



- Write a program to produce square of only those numbers which are divisible by 2 as well as 3 from 1 to 20 , store them in a list and print the list.

### Solution

```
myList=[x**2 for x in range(1,21) if x%2==0 if x%3==0]  
print(myList)
```

### Output:

```
[36, 144, 324]
```



# Previous Code Using **for** Loop

```
myNums=[]
for x in range(1,21):
    if x%2==0:
        if x%3==0:
            myNums.append(x**2)
print(myNums)
```



## Exercise



- Write a function called **get\_upper( )** which accepts a **string** as **argument** and returns a **list** containing only **upper case letters but without any vowels** of that string. Accept the string from the user as input

### Output:

```
Type a string:I Live In Bhopal
['L', 'B']
```

```
Type a string:My Name Is Sachin
['M', 'N', 'S']
```



# Solution



```
def get_upper (text):  
    myList=[x for x in text if 65<=ord(x)<=90 if x not in "AEIOU"]  
    return myList
```

```
text=input("Type a string:")  
myList=get_upper(text)  
print(myList)
```

# What About Logical Operators ?



- **List Comprehension** allows us to use **logical or/and** operator also but we should use **only 1 if statement**

## Example:

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

```
b = [x for x in a if x % 2 == 0 or x % 3 == 0]
```

```
print(b)
```

## Output:

```
[2, 3, 4, 6, 8, 9, 10]
```



## Exercise

- Write a function called **remove\_min\_max()** which accepts a **list** as **argument** and removes the **minimum** and **maximum** elements from the **list** and returns it

### Output:

Original list

```
[10, 3, 15, 12, 24, 6, 1, 18]
```

After removing min and max element

```
[10, 3, 15, 12, 6, 18]
```



# Solution



```
def remove_min_max(myList):  
    myNewList=[x for x in myList if x!=min(myList) and x!=max(myList)]  
    return myNewList
```

```
a=[10,3,15,12,24,6,1,18]  
print("Original list")  
print(a)  
print("After removing min and max element")  
print(remove_min_max(a))
```

# If – else In List Comprehension



- **List Comprehension** allows us to put **if- else** statements also
  
- But since in a comprehension, the first thing we specify is the value to put in a list, so we put our **if-else** in place of value.
  
- **Syntax:**

```
list_variable = [expr1 if cond else expr2  for x in iterable ]
```



# Example



```
myList=["Even" if i%2==0 else "Odd" for i in range(1,11)]  
print(myList)
```

## Output:

```
['Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even']
```

# Nested List Comprehension



- **List Comprehension** can be nested also.
- To understand this , look at the code in the next slide and figure out it's output

# Nested List Comprehension



```
a=[20,40,60]
```

```
b=[2,4,6]
```

```
c=[]
```

```
for x in a:
```

```
    for y in b:
```

```
        c.append(x * y)
```

```
print(c)
```

## Explanation:

The code is multiplying every element of list **a** , with every element of list **b** and storing the product in list **c**

## Output:

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```

# Nested List Comprehension



- Now the same code can be rewritten using **Nested List Comprehension**.
  
- To do this , we will condense each of the lines of code into one line, beginning with the **x \* y** operation.
  
- This will be followed by the **outer for loop**, then the **inner for loop**.



## Example



```
a=[20,40,60]
```

```
b=[2,4,6]
```

```
c=[x * y for x in a for y in b]
```

```
print(c)
```

### Output:

```
[40, 80, 120, 80, 160, 240, 120, 240, 360]
```



## Exercise



- Write a function called **flatten()** which accepts a nested list as argument and returns a single list containing all the elements of the nested list

### Output:

```
Before calling flatten list is
[[1, 2, 3], [4, 5, 6], [7, 8]]
After calling flatten list is
[1, 2, 3, 4, 5, 6, 7, 8]
```



# Solution

```
def flatten(mylist):  
    newlist=[y for x in mylist for y in x]  
    return newlist
```

```
mylist = [[1,2,3],[4,5,6],[7,8]]  
print("Before calling flatten list is")  
print(mylist)  
newlist=flatten(mylist)  
print("After calling flatten list is")  
print(newlist)
```



# PYTHON

# LECTURE 29



# Today's Agenda



## • **Tuple-I**

- What Is A Tuple ?
- Differences With List
- Benefits Of Tuple
- Creating Tuple
- Packing / Unpacking A Tuple
- Accessing A Tuple



# What Is A Tuple ?



- Python provides another type that is an **ordered collection of objects**.
- This type is called a **tuple**.



# Differences With List



- **Tuples** are identical to **lists** in all respects, except for the following properties:
- **Tuples** are defined by enclosing the elements in **parentheses** `()` instead of **square brackets** `[]`.
- **Tuples** are **immutable**.



# Advantages Of Tuple Over List



- **Program execution is faster** when manipulating a **tuple** than it is for the equivalent **list**.
- They prevent **accidental modification**.
- There is another **Python** data type called a **dictionary**, which requires as one of its components a value that is of an **immutable** type. A **tuple** can be used for this purpose, whereas a **list** can't be.



# How To Create A Tuple ?



- As mentioned before a tuple is created by placing all the items (elements) inside a parenthesis ( ), separated by **commas**.
- It can contain **heterogeneous** elements also.

```
# empty tuple
```

```
my_tuple = ()
```

```
# tuple having integers
```

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

```
# tuple with mixed datatypes
```

```
my_tuple = (1, "Hello", 3.4)
```



# Guess The Output ?

```
my_tuple = (2)
print(my_tuple)
print(type(my_tuple))
```

## Output:

```
2
<class 'int'>
```

**Why did it happen ?**

**Since parentheses are also used to define operator precedence in expressions , so , Python evaluates the expression (2) as simply the integer 2 and creates an int object.**

# So , How To Create A 1 Element Tuple ?



- To tell **Python** that we really want to define a **singleton tuple**, include a **trailing comma** (,) just before the closing parenthesis.

```
my_tuple = (2,)  
print(my_tuple)  
print(type(my_tuple))
```

## Output:

```
(2,)  
<class 'tuple'>
```



# Guess The Output ?



```
my_tuple = ()  
print(my_tuple)  
print(type(my_tuple))
```

## Output:

```
()  
<class 'tuple'>
```

# Packing And Unpacking A Tuple



- **Packing** and **unpacking** is another thing which is commonly used with **tuples**.

## Packing:

- **Packing** is a simple syntax which lets us create tuples "on the fly" without using the **normal notation**:  
**a = 1, 2, 3**
- This creates a tuple of 3 values and assigned it to a. Comparing this to the "**normal**" way:  
**a = (1, 2, 3)**

# Packing And Unpacking A Tuple



## □ Unpacking:

- We can also go the other way, and **unpack** the values from a **tuple** into separate variables:

**a = 1, 2, 3**

**x, y, z = a**

- After running this code, we have **x = 1**, **y = 2** and **z = 3**.
- The value of the **tuple a** is unpacked into the 3 variables **x**, **y** and **z**.
- Note that the **number of variables** has to exactly match the **number of elements** in the **tuple**, or there will be an **exception**.

# Uses Of Packing And Unpacking



- In the previous code , the variable **a** is just used as a **temporary store** for the **tuple**.
  
- We also can leave the **middle-man** and do this:  
**x, y, z = 1, 2, 3**
  
- After running this code, as before, we have **x = 1**, **y =2** and **z = 3**.
  
- **But can you guess what Python is doing internally ?**

# Internal Working



- First the values are packed into a **tuple**:

$(1, 2, 3)$   
↑ ↑ ↑  
 $x, y, z = 1, 2, 3$

- Next, the **tuple** is assigned to the left hand side of the **= sign**:

$\xleftarrow{(1, 2, 3)} (1, 2, 3)$

$x, y, z = 1, 2, 3$

- Finally, the **tuple** is **unpacked** into the variables:

$\downarrow \downarrow \downarrow$   
 $x, y, z = 1, 2, 3$



# Swapping Of 2 Variables



- We have seen the following code many times before which **swaps** the variables **a** and **b** :

**a = 10**

**b = 20**

**b, a = a, b**

- **Now , can you tell what is happening internally ?**

- First, **a** and **b** get packed into a **tuple**.
- Then the **tuple** gets **unpacked** again, but this time into variables **b** then **a**.
- So the values get swapped!

# Returning Multiple Values From Function



```
def calculate(a,b):
    c=a+b
    d=a-b
    return c,d

x,y=calculate(5,3)
print("Sum is",x,"and difference is",y)

z=calculate(15,23)
print("Sum is",z[0],"and difference is",z[1])
```

Here **Python** will do the following:

1. It will **pack** the values of **c** and **d** in a **tuple**
2. Then it will **unpack** this **tuple** into the variables **x** and **y**
3. In the **second call** since **z** is a **single variable**, it is automatically converted into a **tuple**

## Output:

```
Sum is 8 and difference is 2
Sum is 38 and difference is -8
```



# Guess The Output ?

```
def add(a,b,c,d):  
    print("Sum is",a+b+c+d)
```

```
mytuple=(10,20,30,40)  
add(mytuple)
```

## Output:

```
add(mytuple)  
TypeError: add() missing 3 required positional arguments: 'b', 'c', and 'd'
```

**Why did it happen ?**

**Since we are passing mytuple as a single argument , so Python is giving exception.**

**What is the solution ?**

**To overcome this we must unpack mytuple to 4 individual values. This can be done by prefixing mytuple with a \*. So the call will be add(\*mytuple)**



# Solution



```
def add(a,b,c,d):  
    print("Sum is",a+b+c+d)
```

```
mytuple=(10,20,30,40)  
add(*mytuple)
```

## Output:

```
Sum is 100
```



# Accessing The Tuple



- Similar to a **list** , we can access/print a **tuple** in **three** ways:
  - Directly passing it to the **print( )** function
  - Accessing individual elements using **subscript operator [ ]**
  - Accessing multiple elements using **slice operator [ : ]**



# Printing The Entire Tuple



```
values=(10,20,30,40)  
print(values)
```

## Output:

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

# Accessing Individual Elements



- Like a **list**, a **tuple** also has indexes running from **0** to **size-1**

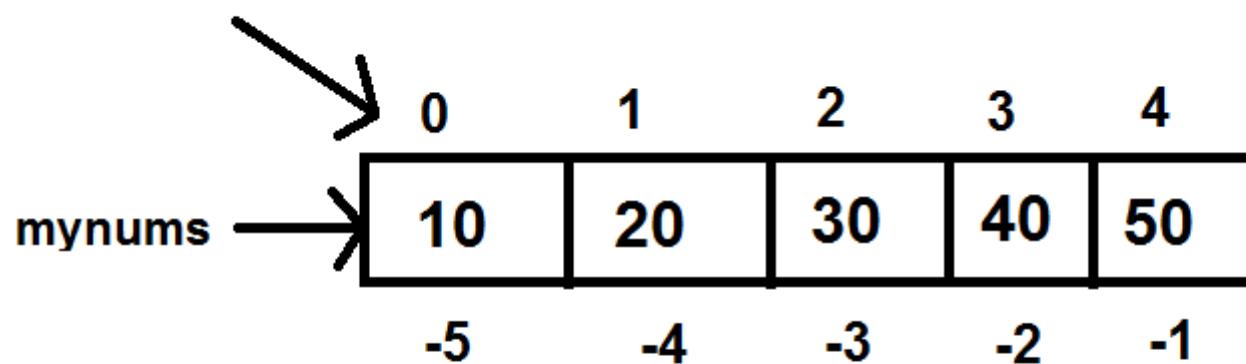
## For example:

- `mynums=(10,20,30,40,50)`
- The above code will create a logical diagram in memory, where **positive indexing** will go from **0** to **4** and **negative indexing** from **-1** to **-5**

# Accessing Individual Elements



## Forward Indexing



A diagram illustrating backward indexing. An arrow points from index -1 to the value 50. Another arrow points from index -1 to the text "Backward Indexing".

## Backward Indexing

# Accessing Individual Elements



```
mynums=(10,20,30,40,50)  
print(mynums[0])  
print(mynums[1])  
print(mynums[-3])  
print(mynums[-2])
```

## Output:

```
10  
20  
30  
40
```

Even though we create tuple using the symbol of () but when we access it's individual element , we still use the subscript or index operator [ ]

# Accessing Tuple Elements Using While Loop



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

```
n=len(mynums)
```

```
i=0
```

```
while i<n:
```

```
    print(mynums[i])
```

```
    i=i+1
```

## Output:

```
10
20
30
40
50
```

Just like **len()** works with **list**, similarly it also works with **tuple** and returns **number of elements in the tuple**

# Accessing Tuple Elements Using For Loop



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

```
for x in mynums:  
    print(x)
```

## Output:

```
10  
20  
30  
40  
50
```

Since **tuple** is a sequence type , so for loop can iterate over individual elements of the tuple



## Exercise

Given the tuple below that represents the **Arijit Singh's Aashiqui 2 songs**.

Write code to print the album details, followed by a listing of all the tracks in the album.

```
album="Aashiqui 2", 2013 , "Arijit Singh",((1,"Tum hi ho"),(2,"Chahun Mai Ya Na"),(3,"Meri Aashiqui"),(4,"Aasan Nahin Yahaan"))
```

### Output:

Title: Aashiqui 2

Year: 2013

Singer: Arijit Singh

    Song Number:1,Song Name:Tum hi ho

    Song Number:2,Song Name:Chahun Mai Ya Na

    Song Number:3,Song Name:Meri Aashiqui

    Song Number:4,Song Name:Aasan Nahin Yahaan



# Solution

```
album="Aashiqui 2","Arijit Singh",2013,((1,"Tum hi  
ho"),(2,"Chahun Mai Ya Na"),(3,"Meri  
Aashiqui"),(4,"Aasan Nahin Yahaan"))
```

```
title,singer,year,songs=album  
print("Title:",title)  
print("Year:",year)  
print("Singer:",singer)  
for info in songs:  
    print("\tSong Number:{0},Song  
Name:{1}".format(info[0],info[1]))
```



# Slice Operator With Tuple



- Just like we can apply slice operator with **lists** and **strings** , similarly **Python** allows us to apply slice operator with **tuple** also.
  
- **Syntax:** `tuple_var[x:y]`
  - **x** denotes the **start index** of slicing and **y** denotes the **end index** . But **Python** ends slicing at **y-1** index.



# The Slicing Operator

- **Example:**

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

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[3:5])
```

- **Output:**

(20,30,40)

- **Output:**

(40,50)



# The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[0:4])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[0:10])
```

- **Output:**

(10,20,30,40)

- **Output:**

(10,20,3040,50)



# The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[2:2])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[6:10])
```

- **Output:**

( )

- **Output:**

( )



# The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[1: ])
```

- **Output:**

(20,30,40,50 )

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[:3])
```

- **Output:**

(10,20,30)



# The Slicing Operator

- **Example:**

```
Mynums=(10,20,30,40,50)  
print(mynums[ :-2])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[-2:])
```

- **Output:**

(10, 20,30 )

- **Output:**

(40,50)



# Using Step Value

- The concept of **step value** in slicing with **tuple** is also same as that with **list**
  - Movement is in forward direction ( L  $\square$  R)
  - Default for start is **0** and end is **len**
- But if **step** is **negative** , then
  - Movement is in backward direction ( R  $\square$  L)
  - Default for start is **-1** and end is **-(len+1)**



# The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[1:4:2])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[1:4:0])
```

- **Output:**  
**(20,40)**

- **Output:**  
**ValueError: Slice  
step cannot be  
0**



# The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[4:1:-1])
```

- **Example:**

```
mynums=(10,20,30,  
        40,50)  
print(mynums[4:1:-1])
```

- **Output:**

()

- **Output:**

(50,40,30)



# The Slicing Operator

- **Example:**

```
mynums=(10,20,30,40,50)  
print(mynums[::-1])
```

- **Example:**

```
Mynums=(10,20,30,  
        40,50)  
print(mynums[::-1])
```

- **Output:**

(10,20,30,40,50 )

- **Output:**

(50,40,30,20,10)



# PYTHON

# LECTURE 30



# Today's Agenda



- **Tuple-II**
  - Changing The Tuple
  - Deleting The Tuple
  - Functions Used With Tuple



# Changing A Tuple



- Unlike **lists**, **tuples** are **immutable**.
- This means that elements of a **tuple** *cannot be changed* once it has been assigned.



# Guess The Output ?



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

`mynums[0]=15`

## Output:

```
mynums[0]=15
TypeError: 'tuple' object does not support item assignment
```



# Guess The Output ?



```
mynums=[10,20],30,40,50)
```

```
print(mynums)
```

```
mynums[0][0]=15
```

```
print(mynums)
```

**Why did the code run?**

**Although a tuple is immutable,  
but if it contains a mutable  
data then we can change  
it's value**

## Output:

```
([10, 20], 30, 40, 50)
([15, 20], 30, 40, 50)
```



# Guess The Output ?



```
myvalues=("hello",30,40,50)
print(myvalues)
myvalues[0]="hi"
print(myvalues)
```

## Output:

```
('hello', 30, 40, 50)
Traceback (most recent call last):
  File "tupledemo12.py", line 3, in <module>
    myvalues[0]="hi"
TypeError: 'tuple' object does not support item assignment
```



# Guess The Output ?



```
myvalues=[["hello",20],30,40,50)  
print(myvalues)  
myvalues[0][0]="hi"  
print(myvalues)
```

## Output:

```
(['hello', 20], 30, 40, 50)  
(['hi', 20], 30, 40, 50)
```



# Guess The Output ?



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

```
print(mynums)
```

```
mynums=(15,25,35,45,55)
```

```
print(mynums)
```

Why did the code run?

Tuple object **is immutable** ,  
but tuple reference **is mutable**.  
**So we can assign a new  
tuple object to the same reference**

## Output:

```
(10, 20, 30, 40, 50)
(15, 25, 35, 45, 55)
```



# Deleting A Tuple



- As we discussed previously, a **tuple** is **immutable**.
- This also means that we can't **delete** just a part of it.
- However we can **delete** an **entire tuple** if required.



# Guess The Output ?



```
mynums=(10,20,30,40,50)
print(mynums)
del mynums[0]
print(mynums)
```

## Output:

```
(10, 20, 30, 40, 50)
Traceback (most recent call last):
  File "tupledemo14.py", line 3, in <module>
    del mynums[0]
TypeError: 'tuple' object doesn't support item deletion
```



# Guess The Output ?



```
mynums=(10,20,30,40,50)
print(mynums)
del mynums[2:4]
print(mynums)
```

## Output:

```
Traceback (most recent call last):
  File "tupledemo14.py", line 3, in <module>
    del mynums[2:4]
TypeError: 'tuple' object does not support item deletion
```



# Guess The Output ?



```
mynums=(10,20,30,40,50)
print(mynums)
del mynums
print(mynums)
```

## Output:

```
(10, 20, 30, 40, 50)
Traceback (most recent call last):
  File "tupledemo14.py", line 4, in <module>
    print(mynums)
NameError: name 'mynums' is not defined
```



# Built In Functions For Tuple



- A lot of functions that work on **lists** work on **tuples** too.
- But only those functions work with **tuple** which do not **modify** it
- **Can you figure out which of these functions will work with tuple ?**

**Answer:**

- `len()`
- `max()`
- `min()`
- `sum()`
- `sorted()`
- `tuple()`
- `any()`
- `all()`

**All of them will work with tuple.**

**Will `sorted( )` also work ?**

**Yes, even `sorted( )` function will also work since it does not change the original tuple , rather it returns a sorted copy of it**



# The **len()** Function

- Returns the **number of items** in the **tuple**

## Example:

```
fruits=("apple","banana","orange",None)  
print(len(fruits))
```

## Output:

4



# The **max()** Function



- Returns the **greatest** item present in the **tuple**

## Example:

```
nums=(5,2,11,3)
```

```
print(max(nums))
```

## Output:

11



# Guess The Output ?



```
months=("january","may","december")
print(max(months))
```

## Output:

may



# Guess The Output ?



```
booleans=(False,True)  
print(max(booleans))
```

**Output:**

**True**



# Guess The Output ?



**Mynums=(True,5,False)**  
**print(max(mynums))**

**Output:**

**5**



# Guess The Output ?



```
mynums=("True",False)  
print(max(mynums))
```

## Output:

```
print(max(mynums))  
TypeError: '>' not supported between instances of 'bool' and 'str'
```



# Guess The Output ?



```
values=(10,"hello",20,"bye")
print(max(values))
```

## Output:

```
print(max(values))
TypeError: '>' not supported between instances of 'str' and 'int'
```



# Guess The Output ?



```
fruits=("apple","banana","orange")
print(max(fruits))
```

**Output:**

**orange**



# Guess The Output ?



```
fruits=("apple","banana","orange",None)  
print(max(fruits))
```

## Output:

```
print(max(fruits))  
TypeError: '>' not supported between instances of 'NoneType' and 'str'
```



# The **min()** Function



- Returns the **least** item present in the **tuple**

## Example:

```
nums=(5,2,11,3)
```

```
print(min(nums))
```

## Output:

2



# Guess The Output ?



```
months=("january","may","december")
print(min(months))
```

## Output:

december



# The **sum()** Function



- Returns the **sum** of all the **items** present in the **tuple** .
- As before , the items must be of **Numeric** or **boolean** type

## Example:

```
nums=(10,20,30)  
print(sum(nums))
```

## Output:

60



# Guess The Output ?



```
nums=(10,20,30,True,False)  
print(sum(nums))
```

**Output:**

**61**



# Guess The Output ?



```
nums=('1','2','3')  
print(sum(nums))
```

## Output:

```
print(sum(nums))  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



# Guess The Output ?



```
nums=(2.5,3.5,4.5)  
print(sum(nums))
```

**Output:**

**10.5**



# The **sorted( )** Function



- Returns a **sorted version** of the **tuple** passed as argument.

## Example:

```
nums=(7,4,9,1)  
print(sorted(nums))  
print(nums)
```

Did you notice a special point ?

Although **sorted( )** is working on a **tuple** , but it has returned a **list**

## Output:

```
[1, 4, 7, 9]  
(7, 4, 9, 1)
```



# Guess The Output ?



```
months=("january","may","december")
print(sorted(months))
```

## Output:

[“december”, “january”, “may”]



# Guess The Output ?



```
months=("january","may","december",3)  
print(sorted(months))
```

## Output:

```
print(sorted(months))  
TypeError: '<' not supported between instances of 'int' and 'str'
```



# Guess The Output ?



```
values=(2.4,1.0,2,3.6)  
print(sorted(values))
```

## Output:

[1.0,2,2.4,3.6]



# Guess The Output ?



```
values=("bhupal","bhop","Bhopal")
print(sorted(values))
```

## Output:

["Bhopal", "bhop", "bhupal"]



# Sorting In Descending Order

- To **sort** the **tuple** in **descending order** , we can pass the **keyword argument reverse** with value set to **True** to the function **sorted( )**

## Example:

```
nums=(7,4,9,1)  
print(sorted(nums,reverse=True))  
print(nums)
```

## Output:

```
[9, 7, 4, 1]  
(7, 4, 9, 1)
```



# The **tuple( )** Function



- The **tuple( )** function converts an **iterable** i.e **list** , **range**, **set** , **dictionary** and **string** to a **tuple**.

## Syntax:

**tuple(iterable)**

## Example:

```
city="bhopal"  
x=tuple(city)  
print(x)
```

## Output:

```
('b', 'h', 'o', 'p', 'a', 'l')
```



# Guess The Output ?



**n=20**

**x=tuple(n)**

**print(x)**

## Output:

```
x=tuple(n)
```

```
TypeError: 'int' object is not iterable
```



# Guess The Output ?



```
n="20"  
x=tuple(n)  
print(x)
```

## Output:

```
('2', '0')
```



# Guess The Output ?



```
l=[10,20,30]  
x=tuple(l)  
print(x)
```

## Output:

(10, 20, 30)



# The **any()** Function

- The **any()** function accepts a **Tuple** as argument and returns **True** if atleast **one element** of the **Tuple** is **True**. If not, this method returns **False**. If the **Tuple** is empty, then also it returns **False**

## Syntax:

**list(iterable)**

## Example:

```
x = (1, 3, 4, 0)  
print(any(x))
```

## Output:

**True**



# Guess The Output ?



```
x = (0, False)  
print(any(x))
```

**Output:**

**False**



# Guess The Output ?



```
x = (0, False, 5)  
print(any(x))
```

## Output:

True



# Guess The Output ?



**x= ()**

**print(any(x))**

**Output:**

**False**



# Guess The Output ?



```
x=("", "o", "")  
print(any(x))
```

## Output:

True



# Guess The Output ?



```
x=("",0, "")  
print(any(x))
```

**Output:**

**False**



# Guess The Output ?



```
x= (4)  
print(any(x))
```

## Output:

```
print(any(x))  
TypeError: 'int' object is not iterable
```



# Guess The Output ?



```
x= (4,)  
print(any(x))
```

**Output:**

**True**



# The **all()** Function



- The **all()** function accepts a **Tuple** as argument and returns **True** if **all the elements** of the **Tuple** are **True** or if the **Tuple** is **empty**. If not, this method returns **False**.

## Syntax:

**all(iterable)**

## Example:

```
x = (1, 3, 4, 0)  
print(all(x))
```

**False**



# Guess The Output ?



```
x = (0, False)  
print(all(x))
```

**Output:**

**False**



# Guess The Output ?



```
x = (1,3,4,5)  
print(all(x))
```

## Output:

True



# Guess The Output ?



```
x = (0, False, 5)  
print(all(x))
```

**Output:**

**False**



# Guess The Output ?



x= 0

print(all(x))

**Output:**

True



These Notes Have Python\_world\_InNotes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

# PYTHON

# LECTURE 31



# Today's Agenda



- **Tuple-III**
  - Methods Used With Tuple
  - Operations Allowed On Tuple



# Tuple Methods

- As mentioned previously a **Tuple** is **immutable**.
- So only those methods work with it which *do not change* the **tuple** data
- **Can you figure out which of these methods work with tuples ?**
- These are:
  - **append()**
  - **extend()**
  - **insert()**
  - **index()**
  - **count()**
  - **remove()**
  - **pop()**
  - **clear()**
  - **sort()**
  - **reverse()**

**Answer:**

**Only index( ) and count().**  
**Rest all the methods change the sequence object on which they have been called.**



# The **index( )** Method

- The **index()** method searches an element in the **tuple** and returns it's **index**.
- If the element occurs **more than once** it returns it's **smallest/first position**.
- If element is **not found**, it raises a **ValueError** exception

## Syntax:

`tuple_var.index(item)`

## Example:

`primes=(2,3,5,7)`

`pos=primes.index(5)`

`print("position of 5 is",pos)`

## Output:

position of 5 is 2



# Guess The Output ?

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

```
p = mynums.index(20)
```

```
print("20 occurs at",p,"position")
```

```
p = mynums.index(60)
```

```
print("60 occurs at",p,"position")
```

```
p = mynums.index(10)
```

```
print("10 occurs at",p,"position")
```

## Output:

```
20 occurs at 1 position
Traceback (most recent call last):
  File "tupledemo15.py", line 4, in <module>
    p = mynums.index(60)
ValueError: tuple.index(x): x not in tuple
```



# The **count()** Method



- The **count()** method returns the **number of occurrences** of an element in a **tuple**
- In simple terms, it **counts** how many times an element has occurred in a **tuple** and returns it.

## Syntax:

**tuple\_var.count(item)**

## Example:

```
country=('i','n','d','i','a')
x=country.count('i')
print("i occurs",x,"times in",country)
```

## Output:

```
i occurs 2 times in ('i', 'n', 'd', 'i', 'a')
```



# Guess The Output ?

```
vowels = ('a', 'e', 'i', 'o', 'i', 'u')
x = vowels.count('i')
print("i occurs",x,"times")
x = vowels.count('e')
print("e occurs",x,"times")
x = vowels.count('j')
print("j occurs",x,"times")
```

## Output:

```
i occurs 2 times
e occurs 1 times
j occurs 0 times
```



# Operations Allowed On Tuple



- We can apply **four** types of operators on **Tuple** objects
- These are:
  - **Membership Operators**
  - **Concatenation Operator**
  - **Multiplication**
  - **Relational Operators**



# Membership Operators



- We can apply the '**in**' and '**not in**' operators on **tuple**.
- This tells us whether an item **belongs / not belongs** to **tuple**.



# Guess The Output ?

```
my_tuple = ('a','p','p','l','e',)  
print('a' in my_tuple)  
print('b' in my_tuple)  
print('g' not in my_tuple)
```

## Output:

True  
False  
True



# Concatenation On Tuple



- **Concatenation** is the act of joining.
- We can join two **tuples** using the **concatenation operator** ‘+’.
- All other arithmetic operators are not allowed to work on two tuples.
- However \* works but as a **repetition operator**



# Guess The Output ?



**odds=(1,3,5)**

**evens=(2,4,6)**

**all=odds+evens**

**print(all)**

## Output:

**(1, 3, 5, 2, 4, 6)**



# Guess The Output ?

```
ages=(10,20,30)
names=("amit","deepak","ravi")
students=ages+names
print(students)
```

## Output:

```
(10, 20, 30, 'amit', 'deepak', 'ravi')
```



# Multiplication On Tuple



- Python allows us to **multiply** a **tuple** by a **constant**
- To do this , as usual we use the operator \*



# Guess The Output ?



**a=(10,20,30)**

**b=a\*3**

**print(b)**

## **Output:**

```
(10, 20, 30, 10, 20, 30, 10, 20, 30)
```



# Guess The Output ?



**a=(10,20,30)**

**b=a\*3.0**

**print(b)**

## Output:

```
b=a*3.0
TypeError: can't multiply sequence by non-int of type 'float'
```

# Relational Operators On Tuples



- The **relational operators** work with **tuples** and other sequences.
- **Python** starts by comparing the **first element** from each sequence.
- If they are **equal**, it goes on to the **next element**, and so on, until it finds elements that **differ**.
- **Subsequent elements** are not considered (even if they are really big).



# Guess The Output ?



**a=(1,2,3)**

**b=(1,3,4)**

**print(a<b)**

## Output:

**True**



# Guess The Output ?



```
a=(1,3,2)  
b=(1,2,3)  
print(a<b)
```

**Output:**

**False**



# Guess The Output ?



```
a=(1,3,2)  
b=(1,3,2)  
print(a<b)
```

**Output:**

**False**



# Guess The Output ?



```
a=(1,2,3)  
b=(1,2,3,4)  
print(a<b)
```

## Output:

True



# Guess The Output ?



```
a=(5,2,7)  
b=(1,12,14)  
print(a>b)
```

**Output:**

**True**



# Guess The Output ?



a=()

b=(0)

print(a<b)

## Output:

**TypeError : < not supported between instances of  
'tuple' and 'int'**



# Guess The Output ?



```
a=0  
b=(0,)  
print(a<b)
```

## Output:

True



# Guess The Output ?



```
a=(1,2)  
b=("one","two")  
print(a<b)
```

## Output:

```
print(a<b)
TypeError: '<' not supported between instances of 'int' and 'str'
```



# Guess The Output ?



```
a=(1,"one")
b=(1,"two")
print(a<b)
```

## Output:

**True**



These Notes Have Python\_world™ Notes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

# PYTHON

# LECTURE 32



# Today's Agenda



## • **Strings -I**

- What Is A String ?
- Creating A String
- Different Ways Of Accessing Strings
- Operators Which Work On Strings



# What Is A String ?



- A Python **string** is a sequence of **zero or more** characters.
- It is an **immutable** data structure.
- This means that we although we **can access** the internal data elements of a string object but we **can not change** it's contents



# How Can Strings Be Created ?

- Python provides us 3 ways to create string objects:
  - By enclosing text in **single quotes**
  - By enclosing text in **double quotes**
  - By enclosing text in **triple quotes (generally used for multiline strings)**



# Example

```
my_string = 'Hello'  
print(my_string)  
my_string = "Hello"  
print(my_string)  
my_string = ""Hello""  
print(my_string)  
my_string = """Hello, welcome to  
the world of Python"""  
print(my_string)
```

Output:

```
Hello  
Hello  
Hello  
Hello, welcome to  
the world of Python
```

# How Can Strings Be Accessed ?



- **Python** provides us **3 ways** to access **string** objects:
  - Directly passing it to the **print( )** function
  - Accessing individual elements using **subscript operator [ ]**
  - Accessing multiple elements using **slice operator [ : ]**



# Printing The Whole String



```
city="Bhopal"  
print(city)
```

## Output:

Bhopal

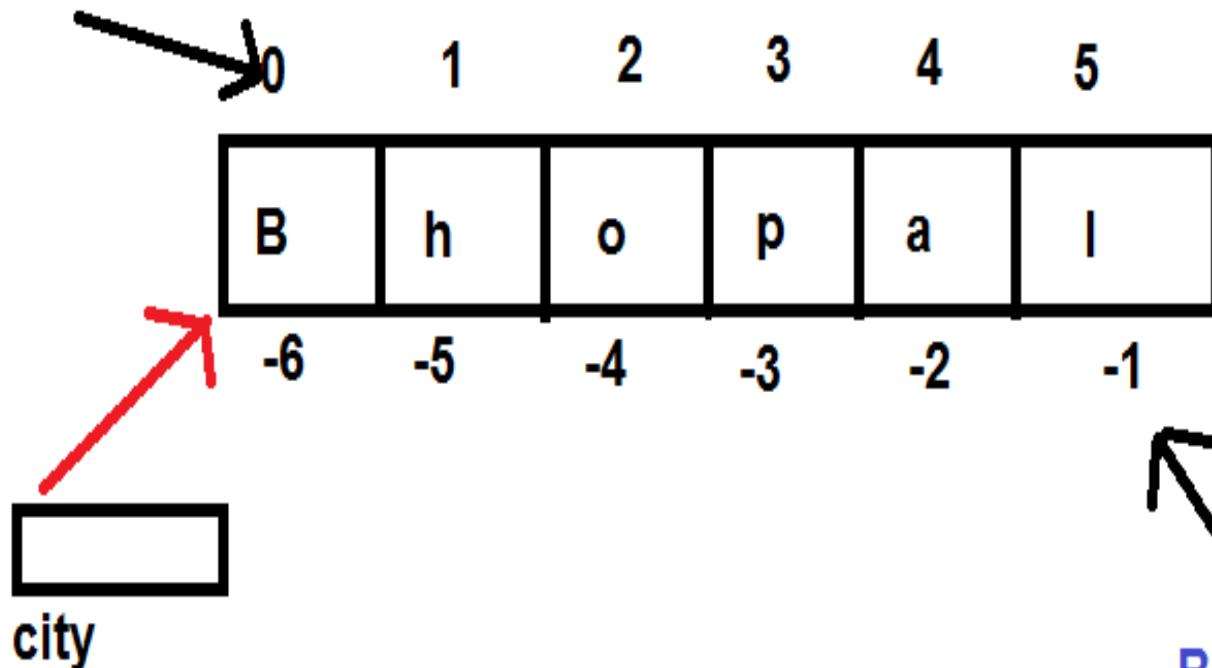
# Accessing Individual Elements



- A **string** in **Python** has indexes running from **0** to **size-1**
- **For example:**
  - **city=“Bhopal”**
- The above code will create a logical diagram in memory, where positive indexing will go from **0** to **5** and negative indexing from **-1** to **-6**

# Accessing Individual Element

## Forward Indexing



## Backward Indexing



# Accessing Individual Element

```
city="Bhopal"  
print(city[0])  
print(city[1])  
print(city[-1])  
print(city[-2])
```

## Output:

B  
h  
l  
a



# Guess The Output ?



```
city="Bhopal"  
print(city[6])
```

## Output:

**IndexError: String index out of range**



# Guess The Output ?



```
city="Bhopal"  
print(city[1.5])
```

## Output:

**TypeError: String indices must be integers**

# Accessing String Elements Using while Loop



```
city="Bhopal"
```

```
i=0
```

```
while i<len(city):
```

```
    print(city[i])
```

```
    i=i+1
```

## Output:

```
B  
h  
o  
p  
a  
l
```

Just like **len()** works with **lists** and **tuple** similarly it also works with **string** returns **number of elements** in the **string**

# Accessing String Elements Using for Loop



```
city="Bhopal"  
for ch in city:  
    print(ch)
```

## Output:

B  
h  
o  
p  
a  
l

Since **string** is a sequence type , so for loop can iterate over individual elements of the **string**



## Exercise



- Redesign the previous code using for loop only to traverse the **string** in reverse order. Don't use slice operator



# Solution



```
city="Bhopal"  
for i in range(len(city)-1,-1,-1):  
    print(city[i])
```

## Output:

l  
a  
p  
o  
h  
B



# Slice Operator With String

- Just like we can apply slice operator with **lists** and **tuples**, similarly **Python** allows us to apply slice operator with **strings** also.
- Syntax:** `string_var[x:y]`
  - x** denotes the **start index** of slicing and **y** denotes the **end index**. But **Python** ends slicing at **y-1** index.



# The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[1:4])
```

- **Example:**

```
city="Bhopal"  
print(city[3:5])
```

- **Output:**

hop

- **Output:**

pa



# The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[0:4])
```

- **Example:**

```
city="Bhopal"  
print(city[0:10])
```

- **Output:**

Bhop

- **Output:**

Bhopal



# The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[1:])
```

- **Example:**

```
city="Bhopal"  
print(city[:4])
```

- **Output:**

hopal

- **Output:**

Bhop



# The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[:-2])
```

- **Example:**

```
city="Bhopal"  
print(city[-2:])
```

- **Output:**

Bhop

- **Output:**

al



# Using Step Value

- String slicing can accept a **third parameter** also after the two index numbers.
- The **third parameter** is called **step value**.
- So the complete syntax of slicing operator is:

**s[begin:end:step]**

- Step value indicates *how many characters to move forward after the first character is retrieved* from the string and it's default value is **1** , but can be changed as per our choice.



# Using Step Value

- Another point to understand is that if **step** is **positive** or **not mentioned** then
  - **Movement is in forward direction ( L→R)**
  - **Default for start is 0 and end is len**
- But if **step** is **negative** , then
  - **Movement is in backward direction ( R→L)**
  - **Default for start is -1 and end is -(len+1)**



# The Slicing Operator

- Example:

```
city="Bhopal"  
print(city[1:4:2])
```
- Output:  
`hp`
- Example:

```
city="Bhopal"  
print(city[1:4:0])
```
- Output:  
`ValueError: Slice  
step cannot be  
0`



# The Slicing Operator

- **Example:**

```
city="Bhopal"  
print(city[4:1:-1])
```

- **Example:**

```
city="Bhopal"  
print(city[4:1:-1])
```

- **Output:**

apo



# The Slicing Operator

- Example:

```
city="Bhopal"  
print(city[::-1])
```
- Output:  
**Bhopal**
- Example:

```
city="Bhopal"  
print(city[::-1])
```
- Output:  
**lapohB**

# The Operators With Strings



- There are **6** operators which work on **Strings**:
  - **+** : For joining 2 strings
  - **\*** : For creating multiple copies of a string
  - **in** : For searching a substring in a string
  - **not in**: Opposite of in
  - **Relational Operator** : For comparing 2 strings
  - **Identity Operators** : For comparing addresses



# The Operator +

The **+** operator concatenates strings. It returns a **string** consisting of the operands joined together

## Example:

**a="Good"**

**b="Morning"**

**c="User"**

**print(a+b+c)**

## Output:

**GoodMorningUser**



# The Operator \*

- The \* operator creates multiple copies of a **string**.

## Example:

```
a="Bye"  
print(a*2)  
print(2*a)
```

## Output:

```
ByeBye  
ByeBye
```



# Guess The Output ?

```
a="Bye"  
print(a*0)  
print(a*-2)
```

**Output:**

The **\*** operator  
allows its  
operand to be  
**negative or 0** in  
which case it  
returns an **empty**  
**string**



# Guess The Output ?



```
x="Ba"+"na"*2  
print(x)
```

**Output:**  
**Banana**



# The Operator in



- The **in** operator returns **True** if the first operand is contained within the second, and **False** otherwise:

## Example:

a="banana"

print("nana" in a)

print("nani" in a)

## Output:

True

False



# The Operator not in

- The **not in** operator behaves opposite of **in** and returns **True** if the first operand is not contained within the second, and **False** otherwise:

## Example:

```
a="banana"
```

```
print("nana" not in a)
```

```
print("nani" not in a)
```

## Output:

False

True



# The Relational Operators

- We can use ( `>` , `<` , `<=` , `>=` , `==` , `!=` ) to compare two strings.
- **Python** compares string lexicographically i.e using Unicode value of the characters.

## Example:

```
"tim" == "tie"  
"free" != "freedom"  
"arrow" > "aron"  
"right" >= "left"  
"teeth" < "tee"  
"yellow" <= "fellow"  
"abc" > ""
```

## **Output:**

False  
True  
True  
True  
False  
False  
True



# The Identity Operators



- ‘**is**’ operator returns **True** if both the operand point to same memory location.
- ‘**is not**’ operator returns **True** if both the operand point to different memory location.

## Example:

```
a = 'London'  
b = 'London'  
c = 'Paris'  
print(a is b)  
print(a is c)  
print(b is c)  
print(b is not a)  
print(b is not c)
```

## Output:

True
False
False
False
True



# PYTHON

# LECTURE 33

These Notes Have Python \_world \_In Notes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



# Today's Agenda



- **Strings -II**

- Built In String Functions
- Printing string using f-string
- Modifying Strings



# Built In String Functions



- There are some **built-in functions** in **Python** that we can use on **strings**.
- These are:
  - **len()**
  - **max()**
  - **min()**
  - **chr()**
  - **ord()**



# The **len()** Function

- Returns the **number of characters** in the **string**

## Example:

```
name="Sachin"  
print(len(name))
```

## Output:

6



# The **max()** Function

- Returns a character which is **alphabetically the highest character** in the **string**.

## Example:

```
name="bhopal"
```

```
print(max(name))
```

## Output:

```
p
```



# Guess The Output ?



```
str="abc123#$.y@*"  
print(max(str))
```

## Output:

y



# Guess The Output ?



```
str="False,True"  
print(max(str))
```

## Output:

u



# Guess The Output ?



```
str="1.1,0.4,1.9"  
print(max(str))
```

Output:

9



# The **min()** Function



- Returns a character which is **alphabetically the lowest character** in the **string**.

## Example:

```
name="bhopal"
```

```
print(min(name))
```

## Output:

```
a
```



# Guess The Output ?



**str="Bhopal"**

**print(min(str))**

**Output:**

**B**



# Guess The Output ?



```
str="abc123#$.y@*"  
print(min(str))
```

## Output:

#



# Guess The Output ?



```
str="1.1,0.4,1.9"  
print(min(str))
```

## Output:

,



# The **chr()** Function



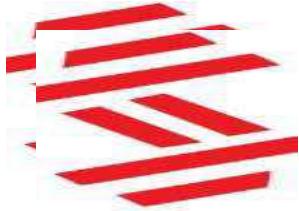
- Returns a **character** (a string) of the **unicode** value passed as an **integer**.
- The valid range of the argument is from **0** through **1,114,111**.

## Example:

```
print(chr(122))
```

## Output:

**z**



# Guess The Output ?



**print(chr(43))**

**Output:**

**+**



# Guess The Output ?



**print(chr(1))**

**Output:**





# Guess The Output ?



**print(chr(0))**

**Output:**



# Guess The Output ?

`print(chr(-1))`

Output:

`ValueError: chr() argument not in range`



# The `ord()` Function



- Returns an **integer** of the **unicode** value passed as an **character**.
- But the argument passed should be only 1 character in length.

## Example:

```
print(ord('a'))
```

## Output:

97



# Guess The Output ?

`print(ord("+"))`

Output:

43



# Guess The Output ?



**print(ord("5"))**

**Output:**

**53**



# The **str()** Function

- Returns a **string** representation of an **object**.
- We can pass object of any type and Python will convert it to string

## Example:

```
print(str(49.2))
```

## Output:

49.2



# Guess The Output ?



**print(str(True))**

**Output:**

**True**



# Guess The Output ?

`print(str(25))`

Output:

`25`



# String Interpolation



- In **Python** version **3.6**, a new string formatting mechanism was introduced.
- This feature is called **String Interpolation** , but is more usually referred to by its nickname **f-string**.



# String Interpolation



- To understand this feature , can you tell how can we print the following **2 variables** using **print( )**:

**name=“Sachin”**

**age=34**

- Till now , we know **4 ways**:

- 1. print(“My name is”,name,”and my age is”,age)**
- 2. print(“My name is ”+name+” and my age is ”+str(age))**
- 3. print(“My name is {0} and my age is {1}”.format(name,age))**
- 4. print(“My name is %s and my age is %d”%(name,age))**



# String Interpolation



- But , from **Python 3.6** onwards , there is much more simpler way to print them which is called **f-string**.
- **f-strings** have an **f** at the beginning and **curly braces containing expressions** that will be **replaced** with their values.
- The expressions are evaluated at runtime and then formatted

`name="Sachin"`

`age=34`

`print(f"My name is {name} and my age is {age}")`



# Arbitrary Expressions



- Because f-strings are evaluated at runtime, we can put any valid **Python expressions** in them

**a=10**

**b=20**

**print(f"sum is {a+b}")**

**Output:**

**sum is 30**



# Function Calls



- We could also call functions. Here's an example:

```
import math
```

```
a=10
```

```
b=20
```

```
print(f"max of {a} and {b} is {max(a,b)}")
```

```
print(f"Factorial of {a} is {math.factorial(10)}")
```

## Output:

```
max of 10 and 20 is 20
Factorial of 10 is 3628800
```



# Method Calls



- We could also call **methods**. Here's an example:

```
vowels=["a","e","i","o","u","a"]
```

```
ch="a"
```

```
print(f"{ch} is occurring in {vowels}  
{vowels.count(ch)} times")
```

## Output:

```
a is occurring in ['a', 'e', 'i', 'o', 'u', 'a'] 2 times
```



These Notes Have Python\_world\_In Notes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

# PYTHON

# LECTURE 34



# Today's Agenda



- **Strings -III**
  - String Methods



# String Methods



- A string object has a number of **method** or **member functions**.
- These can be grouped into different categories .
- These categories are:
  - **String conversion methods**
  - **String comparison methods**
  - **String searching methods**
  - **String replace methods**



# String Conversion Methods

- **capitalize()**

Returns a copy of the string with **first character capitalized** and rest of the characters in **lower case**.

## Example:

```
name="guido van rossum"  
newname=name.capitalize()  
print(f'Original name is {name}\nCapitalized name is {newname}')
```

## Output:

```
Original name is guido van rossum  
Capitalized name is Guido van rossum
```



# String Conversion Methods



```
text="python is awesome. java rocks"  
newtext=text.capitalize()  
print(f'Original text is {text}\nCapitalized text is {newtext}')
```

## Output:

```
Original text is python is awesome. java rocks  
Capitalized text is Python is awesome. java rocks
```



# String Conversion Methods

- **lower()** and **upper()**

Returns a copy of the string with all letters converted to **lowercase** and **uppercase** respectively

## Example:

```
name="Sachin Kapoor"  
lc=name.lower()  
uc=name.upper() print(f'Original  
name is {name}') print(f'Lower  
name is {lc}') print(f'Upper name  
is {uc}')
```

## Output:

```
Original name is Sachin Kapoor  
Lower name is sachin kapoor  
Upper name is SACHIN KAPOOR
```



# String Conversion Methods



- **swapcase()**

Returns a copy of the string with the **case** of every character **swapped**. Means that **lowercase characters** are changed to **uppercase** and **vice-versa**.

## Example:

```
name="Sachin Kapoor"  
newname=name.swapcase()  
print(f'Original name is {name}')  
print(f'Swapped name is {newname}')
```

## Output:

```
Original name is Sachin Kapoor  
Swapped name is SACHIN kAPOOR
```



# String Conversion Methods



- **title()**

Returns a copy of the string converted to **proper case** or **title case**. i.e., all **words** begin with **uppercase letter** and the **rest** are in **lowercase**.

**Example:**

```
text="we got independence in 1947"
newtext=text.title() print(f"Original
text is {text}") print(f"Title text is
{newtext}")
```

**Output:**

```
Original text is we got independence in 1947
Title text is We Got Independence In 1947
```



# String Conversion Methods



```
text = "i LOVE pYTHON"  
print(text.title())
```

## Output:

```
I Love Python
```



# String Conversion Methods

```
text = "physics,chemistry,maths"  
print(text.title())
```

## Output:

```
Physics,chemistry,Maths
```



# String Conversion Methods

```
text = "physics_chemistry_maths"  
print(text.title())
```

## Output:

```
Physics_Chemistry_Maths
```



# String Conversion Methods

```
text = "physics1chemistry2maths"  
print(text.title())
```

## Output:

```
Physics1Chemistry2Maths
```



# String Conversion Methods

```
text = "He's an engineer, isn't he?"  
print(text.title())
```

## Output:

```
He'S An Engineer, Isn'T He?
```



# String Comparison Methods

- **islower( ) and isupper()**

Returns **True** or **False** depending on whether all alphabets in the string are in **lowercase** and **uppercase** respectively



# String Comparison Methods



## Example:

```
s = 'this is good'
```

```
print(s.islower())
```

```
s = 'th!s is a1so g00d'
```

```
print(s.islower())
```

```
s = 'this is Not good'
```

```
print(s.islower())
```

## Output:

```
True
```

```
True
```

```
False
```



# String Comparison Methods



## Example:

```
s = "THIS IS GOOD!"
```

```
print(s.isupper())
```

```
s = "THIS IS ALSO GooD!"
```

```
print(s.isupper())
```

```
s= "THIS IS not GOOD!"
```

```
print(s.isupper())
```

## Output:

```
True  
True  
False
```



# String Comparison Methods



## Example:

```
s = ""
```

```
print(s.isupper())
print(s.islower())
```

## Output:

```
False
False
```



# String Comparison Methods



- **istitle()**

Returns **True** if the string is in **titlecase** or **empty** ,  
otherwise returns **False**



# String Comparison Methods

## Example:

```
s = 'Python Is Good.'  
print(s.istitle())
```

```
s = 'Python is good'  
print(s.istitle())
```

```
s = 'This Is @ Symbol.'  
print(s.istitle())
```

```
s = '99 Is A Number'  
print(s.istitle())
```

```
s = 'PYTHON'  
print(s.istitle())
```

## Output:

True  
False  
True  
True  
False



# String Comparison Methods



- **isalpha()**

Returns **True** if the string contains only **alphabets**,  
otherwise returns **False**



# String Comparison Methods

## Example:

```
name = "Monalisa"  
print(name.isalpha())
```

```
name = "M0nalisa"  
print(name.isalpha())
```

```
name = "Monalisa Shah"  
print(name.isalpha())
```

## Output:

True  
False  
False



# String Comparison Methods



- **isdigit()**

Returns **True** if the string contains only **digits** , otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "12345"
```

```
print(text.isdigit())
```

```
text = "012345"
```

```
print(text.isdigit())
```

```
text = "12345 6"
```

```
print(text.isdigit())
```

```
text = "a12345"
```

```
print(text.isdigit())
```

## Output:

True

True

False

False



# String Comparison Methods



- **isalnum()**

Returns **True** if the string contains only **alphanumeric** characters , otherwise returns **False**



# String Comparison Methods

## Example:

```
name = "M234onalisa"  
print(name.isalnum())
```

```
name = "M3ona Shah "  
print(name.isalnum())
```

```
name = "Mo3nalisaSha22ah"  
print(name.isalnum())
```

```
name = "133"  
print(name.isalnum())
```

## Output:

True  
False  
True  
True



# String Comparison Methods



- **isspace( )**

Returns **True** if the string contains only **whitespace** characters , otherwise returns **False**



# String Comparison Methods

## Example:

```
s = ' \t'
```

```
print(s.isspace())
```

```
s = ' a '
```

```
print(s.isspace())
```

```
s = ''
```

```
print(s.isspace())
```

```
s = "
```

```
print(s.isspace())
```

## Output:

True

False

True

False



# String Comparison Methods



- **startswith()**
- The **startswith()** method takes maximum of **three** parameters:
  - **prefix** - String to be checked
  - **start** (optional) - Beginning position where **prefix** is to be checked within the string.
  - **end** (optional) - Ending position where **prefix** is to be checked within the string.
- It returns **True** if the string **starts with** the specified **prefix**, otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "Python is easy to learn."
```

```
result = text.startswith('is easy')  
print(result)
```

```
result = text.startswith('Python is ')  
print(result)
```

```
result = text.startswith('Python is easy to learn.')  
print(result)
```

```
result = text.startswith('is easy',7)  
print(result)
```

## Output:

False

True

True

True



# String Comparison Methods



- **endswith()**
- The **endswith()** method takes maximum of **three** parameters:
  - **suffix** - String to be checked
  - **start** (optional) - Beginning position where **suffix** is to be checked within the string.
  - **end** (optional) - Ending position where **suffix** is to be checked within the string.
- It returns **True** if the string **ends with** the specified **suffix**, otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "Python is easy to learn."
```

```
result = text.endswith('to learn')
print(result)
```

```
result = text.endswith('to learn.')
print(result)
```

```
result = text.endswith('learn.', 7)
print(result)
```

```
result = text.endswith('is', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 14)
print(result)
```

## Output:

False

True

True

False

False

True



# String Searching Methods



- **index()**

Returns the **index** of **first occurrence** of a **substring** inside the string (if found).

If the substring is not found, it raises an **exception**.

**Syntax:** The **index()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



# String Searching Methods



## Example:

```
text= 'Sunday is a fun day'
```

```
result = text.index('is a fun')
print(result)
```

```
result = text.index('day')
print(result)
```

```
result = text.index('day',7)
print(result)
```

```
result = text.index('night')
print(result)
```

## Output:

7

3

16

ValueError



# String Searching Methods



- **find()**

Returns the **first index** of a **substring** inside the string (if found).

If the substring is not found, it returns **-1**

**Syntax:** The **find()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



# String Searching Methods

## Example:

```
text= 'Sunday is a fun day'
```

```
result = text.find('is a fun')
print(result)
```

```
result = text.find('day')
print(result)
```

```
result = text.find('day',7)
print(result)
```

```
result = text.find('night')
print(result)
```

## Output:

7

3

16

-1



# String Searching Methods



- **count()**

Returns the **number of occurrences** of a **substring** in the given **string**

If the substring is not found, it returns 0

**Syntax:** The **count()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



# String Searching Methods

## Example:

```
text = "Python is awesome, isn't it?"
```

```
substring = "is"
```

```
count = text.count(substring)
```

```
print(count)
```

```
substring = "i"
```

```
count = text.count(substring, 8, 25)
```

```
print(count)
```

```
substring = "ton"
```

```
count = text.count(substring)
```

```
print(count)
```

## Output:

2

1

0



# String Replacement Methods

- **replace( )**

Returns a copy of the string where all occurrences of a **substring** is replaced with **another substring**.

**Syntax:** The **replace()** method takes **three** parameters:

- **old** - old substring we want to replace
  - **new** - new substring which would replace the old substring
  - **count** (optional) - the number of times we want to replace the old substring with the new substring
- 
- The **replace()** method returns a copy of the string where **old substring** is replaced with the **new substring**. The original string is unchanged.
  - If the **old substring** is not found, it returns the copy of the original string.



# String Replacement Methods

## Example:

```
text = "Blue Blue Blue"  
newtext= text.replace("ue","ack")  
print(newtext)  
  
newtext= text.replace("ue","ack",2)  
print(newtext)  
  
newtext= text.replace("eu","ack")  
print(newtext)
```

## Output:

```
Black Black Black  
Black Black Blue  
Blue Blue Blue
```



# String Replacement Methods



- **strip()**

Returns a copy of the string with both **leading** and **trailing** characters removed (based on the string argument passed).

**Syntax:** The **strip()** method takes **one** optional parameter:

- **chars** (optional) - a string specifying the set of characters to be removed.
- If the **chars** argument is not provided, all leading and trailing whitespaces are removed from the string.



# String Replacement Methods

## Example:

```
text = " Good Morning"  
newtext= text.strip()  
print("Original text:[ "+text+"]")  
print("New text:[ "+newtext+"]")
```

## Output:

```
Original text:[ Good Morning]  
New text:[Good Morning]
```



# Exercise

**Write a program to simulate user registration process. Your code should do the following:**

- 1. It should first ask the user to input his full name. If he doesn't enter his full name then program should display the error message and again ask the user to enter full name . Repeat the process until the user types his full name.[ full name means a string with atleast 2 words separated with a space]**
- 2. Then it should ask the user to input his password. The rules for password are:**
  - 1. It should contain atleast 8 characters**
  - 2. It should contain atleast 1 digit and 1 upper case letter**

**Repeat the process until the user correctly types his Password.**

**Finally , display the user's first name with a THANK YOU message. Create separate functions for accepting fullname , password and returning firstname**



# Sample Output

```
Type your full name:Sachin
Please enter your full name!
Type your full name:Sachin Kapoor
Type your password:Admin
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Administrator1
Hello Sachin
Thank you for joining us!
```



# Solution

```
def get_full_name():
    pass
def get_password():
    pass
def get_first_name(fullname):
    pass
fullname=get_full_name()
pwd=get_password()
firstname=get_first_name(fullname)
print("Hello",firstname,"\\nThank you for joining us!")
```



# String Replacement Methods

- **split()**

The **split()** method breaks up a string at the specified separator and returns a list of strings.

**Syntax:** The **split()** method takes **two** parameters:

- **separator** (optional)- This is the **delimiter**. The string splits at the specified **separator**. If the **separator** is not specified, any whitespace (space, newline etc.) string is a **separator**.
- **maxsplit** (optional) - The **maxsplit** defines the maximum number of splits. The default value of **maxsplit** is **-1**, meaning, no limit on the number of splits.

The **split()** breaks the string at the separator and returns a list of strings.



# String Replacement Methods

## Example:

```
text= 'Live and let live'  
print(text.split())  
grocery = 'Milk, Butter, Bread'  
print(grocery.split(','))  
print(grocery.split(':'))
```

## Output:

```
['Live', 'and', 'let', 'live']  
['Milk', ' Butter', ' Bread']  
['Milk, Butter, Bread']
```

## Exercise

**Write a program to which takes a **string** from the user and displays a **list** of all of the **words** that the string contains **removing all special characters except **space****. It should also display **number of words** in the string.**

**For example:**

**Input:**

**Let's learn C++, Python and then Java.**

**Output:**

**['Lets', 'learn', 'C', 'Python', 'and', 'then', 'Java']**



# String Replacement Methods

- **join()**

The **join()** method returns a **string** concatenated with the elements of an **iterable**.

But the **iterable** should only contain strings

The **join()** method takes an **iterable** - objects capable of returning its members one at a time

**Syntax:** `<str>.join(<iterable>)`



# String Replacement Methods

## Example:

```
mylist = ["C","C++","Java","Python"]
```

```
s = "->"
```

```
print(s.join(mylist))
```

## Output:

C->C++->Java->Python



# String Replacement Methods

## Example:

```
letters = 'PYTHON'
```

```
letters_spaced = ''.join(letters)
```

```
print(letters_spaced)
```

## Output:

```
P Y T H O N
```



These Notes Have Python\_world\_In Notes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

PYTHON

**LECTURE 35**



# Today's Agenda



## • **Dictionary-I**

- What Is A Dictionary ?
- What Is Key-Value Pair ?
- Creating A Dictionary
- Important Characteristics Of A Dictionary
- Different Ways To Access A Dictionary
- An Important Point



# What Is A Dictionary ?



- Python **dictionary** is an **unordered collection** of **items**.
- The collections we have studied till now like **list** , **tuple** and **string** are all **ordered collections** as well as can hold only one value as their element
- On the other hand **dictionary** is an **unordered collection** which holds the data in a **key: value** pair.



# What Is Key-Value Pair?



- Sometimes we need to store the data so that one piece of information is **connected** to another piece of information.
- For example **RollNo**→**Student Name** or **Customer Name**→**Mobile Number**
- In these examples **RollNo** will be called a **key** while it's associated **Student Name** will be called **value**
- To store such paired data **Python** provides us the data type called **dictionary**



# How To Create A Dictionary?



- Creating a dictionary is as simple as placing **items** inside **curly braces { }** separated by **comma**.
- Every **item** has a **key** and the corresponding **value** expressed as a **pair, key: value**.
- While **values** can be of **any data type** and **can repeat**, but **keys** must be of **immutable type** and must be **unique**.

# General Syntax Of Creating A Dictionary



## Syntax:

```
d = {  
    <key>: <value>,  
    <key>: <value>,  
    ...  
    <key>: <value>  
}
```



# How To Create A Dictionary?



```
# empty dictionary  
my_dict = {}
```

```
# dictionary with integer keys  
my_dict = {1: 'Amit', 2: 'Brajesh', 3:'Chetan'}
```

```
# dictionary with mixed keys  
my_dict = {1: 'John', 'a':'vowel'}
```

```
# dictionary with list as values  
my_dict = {'Rahul':['C', 'C++'], 'Ajay':['Java', 'C', 'Python'],  
          'Neeraj':['Oracle', 'Python']}
```

# Important Characteristics Of Dictionaries



- The important characteristics of **Python dictionaries** are as follows:
  - They can be **nested**.  
Click to add text
  - They are **mutable**.
  - They are **dynamic**.
  - They are **unordered**.
  - Unlike **Lists** and **tuples**, a **dictionary item is accessed by its corresponding key not index**

# Other Ways Of Creating Dictionary



- We also can create a list by using the **dict( )** function

```
# Create an empty dictionary
```

```
my_dict = dict()
```

```
# Create a dictionary with elements
```

```
my_dict = dict({1:'apple', 2:'ball'})
```

```
# Create a dictionary with other sequences
```

```
my_dict = dict([(1,'apple'), (2,'ball')])
```



# PYTHON

# LECTURE 35

These Notes Have Python \_ world \_ In Notes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



# Today's Agenda



- **Strings -III**
  - String Methods



# String Methods



- A string object has a number of **method** or **member functions**.
- These can be grouped into different categories .
- These categories are:
  - **String conversion methods**
  - **String comparison methods**
  - **String searching methods**
  - **String replace methods**



# String Conversion Methods



- **capitalize()**

Returns a copy of the string with **first character capitalized** and rest of the characters in **lower case**.

## Example:

```
name="guido van rossum"  
newname=name.capitalize()  
print(f'Original name is {name}\nCapitalized name is {newname}')
```

## Output:

```
Original name is guido van rossum  
Capitalized name is Guido van rossum
```



# String Conversion Methods

```
name="Guido Van Rossum"  
newname=name.capitalize()  
print(f'Original name is {name}\nCapitalized name is {newname}')
```

## Output:

```
Original name is Guido Van Rossum  
Capitalized name is Guido van rossum
```



# String Conversion Methods

```
text="python is awesome. java rocks"  
newtext=text.capitalize()  
print(f'Original text is {text}\nCapitalized text is {newtext}')
```

## Output:

```
Original text is python is awesome. java rocks  
Capitalized text is Python is awesome. java rocks
```



# String Conversion Methods

- **lower()** and **upper()**

Returns a copy of the string with all letters converted to **lowercase** and **uppercase** respectively

## Example:

```
name="Sachin Kapoor"  
lc=name.lower()  
uc=name.upper() print(f'Original  
name is {name}') print(f'Lower  
name is {lc}') print(f'Upper name  
is {uc}')
```

## Output:

```
Original name is Sachin Kapoor  
Lower name is sachin kapoor  
Upper name is SACHIN KAPOOR
```



# String Conversion Methods



- **swapcase()**

Returns a copy of the string with the **case** of every character **swapped**. Means that **lowercase characters** are changed to **uppercase** and **vice-versa**.

## Example:

```
name="Sachin Kapoor"  
newname=name.swapcase()  
print(f'Original name is {name}')  
print(f'Swapped name is {newname}')
```

## Output:

```
Original name is Sachin Kapoor  
Swapped name is SACHIN kAPOOR
```



# String Conversion Methods



- **title()**

Returns a copy of the string converted to **proper case** or **title case**. i.e., all **words** begin with **uppercase letter** and the **rest** are in **lowercase**.

## Example:

```
text="we got independence in 1947"
newtext=text.title() print(f"Original
text is {text}") print(f"Title text is
{newtext}")
```

## Output:

```
Original text is we got independence in 1947
Title text is We Got Independence In 1947
```



# String Conversion Methods



```
text = "i LOVE pYTHON"  
print(text.title())
```

## Output:

I Love Python



# String Conversion Methods

```
text = "physics,chemistry,maths"  
print(text.title())
```

## Output:

```
Physics,chemistry,Maths
```



# String Conversion Methods

```
text = "physics_chemistry_maths"  
print(text.title())
```

## Output:

```
Physics_Chemistry_Maths
```



# String Conversion Methods

```
text = "physics1chemistry2maths"  
print(text.title())
```

## Output:

```
Physics1Chemistry2Maths
```



# String Conversion Methods

```
text = "He's an engineer, isn't he?"  
print(text.title())
```

## Output:

```
He'S An Engineer, Isn'T He?
```



# String Comparison Methods

- **islower( ) and isupper( )**

Returns **True** or **False** depending on whether all alphabets in the string are in **lowercase** and **uppercase** respectively



# String Comparison Methods



## Example:

```
s = 'this is good'
```

```
print(s.islower())
```

```
s = 'th!s is a1so g00d'
```

```
print(s.islower())
```

```
s = 'this is Not good'
```

```
print(s.islower())
```

## Output:

```
True  
True  
False
```



# String Comparison Methods



## Example:

```
s = "THIS IS GOOD!"
```

```
print(s.isupper())
```

```
s = "THIS IS ALSO GooD!"
```

```
print(s.isupper())
```

```
s= "THIS IS not GOOD!"
```

```
print(s.isupper())
```

## Output:

```
True  
True  
False
```



# String Comparison Methods



## Example:

```
s = ""
```

```
print(s.isupper())
print(s.islower())
```

## Output:

```
False
False
```



# String Comparison Methods



- **istitle()**

Returns **True** if the string is in **titlecase** or **empty** ,  
otherwise returns **False**



# String Comparison Methods

## Example:

```
s = 'Python Is Good.'  
print(s.istitle())
```

```
s = 'Python is good'  
print(s.istitle())
```

```
s = 'This Is @ Symbol.'  
print(s.istitle())
```

```
s = '99 Is A Number'  
print(s.istitle())
```

```
s = 'PYTHON'  
print(s.istitle())
```

## Output:

True  
False  
True  
True  
False



# String Comparison Methods



- **isalpha()**

Returns **True** if the string contains only **alphabets**,  
otherwise returns **False**



# String Comparison Methods

## Example:

```
name = "Monalisa"  
print(name.isalpha())
```

```
name = "M0nalisa"  
print(name.isalpha())
```

```
name = "Monalisa Shah"  
print(name.isalpha())
```

## Output:

True  
False  
False



# String Comparison Methods



- **isdigit()**

Returns **True** if the string contains only **digits** , otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "12345"
```

```
print(text.isdigit())
```

```
text = "012345"
```

```
print(text.isdigit())
```

```
text = "12345 6"
```

```
print(text.isdigit())
```

```
text = "a12345"
```

```
print(text.isdigit())
```

## Output:

True

True

False

False



# String Comparison Methods



- **isdecimal()**

Returns **True** if the string contains only **decimal characters**, otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "12345"
```

```
print(text.isdecimal())
```

```
text = "012345"
```

```
print(text.isdecimal())
```

```
text = "12345 6"
```

```
print(text.isdecimal())
```

```
text = "a12345"
```

```
print(text.isdecimal())
```

## Output:

True

True

False

False



# String Comparison Methods



- **isnumeric()**

Returns **True** if the string contains only  
**numeric characters**, otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "12345"
```

```
print(text.isnumeric())
```

```
text = "012345"
```

```
print(text.isnumeric())
```

```
text = "12345 6"
```

```
print(text.isnumeric())
```

```
text = "a12345"
```

```
print(text.isnumeric())
```

## Output:

True

True

False

False

# **isdigit() V/s isdecimal() V/s isnumeric()**



- To understand the difference between **isdigit()** , **isdecimal()** and **isnumeric()** , we will first have to understand what **Python** considers as **digits** , **decimal** or **numerics** for **special symbols**
- In Python:
  - **superscript** and **subscripts** (usually written using unicode) are also considered **digit** characters , **numeric** characters but not **decimals**.
  - **roman numerals**, **currency numerators** and **fractions** (usually written using unicode) are considered **numeric** characters but not **digits** or **decimals**



# String Comparison Methods

## Example:

```
s = '\u00B23455'  
print(s)  
print(s.isdigit())  
print(s.isdecimal())  
print(s.isnumeric())  
  
s = '\u00BD'  
print(s)  
print(s.isdigit())  
print(s.isdecimal())  
print(s.isnumeric())
```

## Output:

```
'²3455'  
isdigit(): True  
isdecimal(): False  
isnumeric(): True  
½  
isdigit(): False  
isdecimal(): False  
isnumeric(): True
```



# String Comparison Methods



- **isalnum()**

Returns **True** if the string contains only **alphanumeric** characters , otherwise returns **False**



# String Comparison Methods

## Example:

```
name = "M234onalisa"  
print(name.isalnum())
```

```
name = "M3ona Shah "  
print(name.isalnum())
```

```
name = "Mo3nalisaSha22ah"  
print(name.isalnum())
```

```
name = "133"  
print(name.isalnum())
```

## Output:

True  
False  
True  
True



# String Comparison Methods



- **isspace( )**

Returns **True** if the string contains only **whitespace** characters , otherwise returns **False**



# String Comparison Methods

## Example:

```
s = ' \t'  
print(s.isspace())
```

```
s = ' a '  
print(s.isspace())
```

```
s = ''  
print(s.isspace())
```

```
s = "  
print(s.isspace())
```

## Output:

True  
False  
True  
False



# String Comparison Methods



- **startswith()**
- The **startswith()** method takes maximum of **three** parameters:
  - **prefix** - String to be checked
  - **start** (optional) - Beginning position where **prefix** is to be checked within the string.
  - **end** (optional) - Ending position where **prefix** is to be checked within the string.
- It returns **True** if the string **starts with** the specified **prefix** , otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "Python is easy to learn."
```

```
result = text.startswith('is easy')  
print(result)
```

```
result = text.startswith('Python is ')  
print(result)
```

```
result = text.startswith('Python is easy to learn.')  
print(result)
```

```
result = text.startswith('is easy',7)  
print(result)
```

## Output:

False

True

True

True



# String Comparison Methods



- **endswith()**
- The **endswith()** method takes maximum of **three** parameters:
  - **suffix** - String to be checked
  - **start** (optional) - Beginning position where **suffix** is to be checked within the string.
  - **end** (optional) - Ending position where **suffix** is to be checked within the string.
- It returns **True** if the string **ends with** the specified **suffix**, otherwise returns **False**



# String Comparison Methods

## Example:

```
text = "Python is easy to learn."
```

```
result = text.endswith('to learn')
print(result)
```

```
result = text.endswith('to learn.')
print(result)
```

```
result = text.endswith('learn.', 7)
print(result)
```

```
result = text.endswith('is', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 13)
print(result)
```

```
result = text.endswith('easy', 7, 14)
print(result)
```

## Output:

False

True

True

False

False

True



# String Searching Methods



- **index()**

Returns the **index** of **first occurrence** of a **substring** inside the string (if found).

If the substring is not found, it raises an **exception**.

**Syntax:** The **index()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



# String Searching Methods



## Example:

```
text= 'Sunday is a fun day'
```

```
result = text.index('is a fun')
print(result)
```

```
result = text.index('day')
print(result)
```

```
result = text.index('day',7)
print(result)
```

```
result = text.index('night')
print(result)
```

## Output:

7

3

16

ValueError



# String Searching Methods



- **find()**

Returns the **first index** of a **substring** inside the string (if found).

If the substring is not found, it returns **-1**

**Syntax:** The **find()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



# String Searching Methods



## Example:

```
text= 'Sunday is a fun day'
```

```
result = text.find('is a fun')
print(result)
```

```
result = text.find('day')
print(result)
```

```
result = text.find('day',7)
print(result)
```

```
result = text.find('night')
print(result)
```

## Output:

7  
3  
16  
-1



# String Searching Methods



- **rfind()**

Returns the **highest index** of a **substring** inside the string (if found).

If the substring is not found, it returns -1

**Syntax:** The **rfind()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



# String Searching Methods



## Example:

```
text= 'Sunday is a fun day'
```

```
result = text.rfind('is a fun')
print(result)
```

```
result = text.rfind('day')
print(result)
```

```
result = text.rfind('day',0)
print(result)
```

```
result = text.rfind('night')
print(result)
```

## Output:

7

16

16

-1



# String Searching Methods



- **count()**

Returns the **number of occurrences** of a **substring** in the given **string**

If the substring is not found, it returns 0

**Syntax:** The **count()** method takes **three** parameters:

- **sub** - substring to be searched in the string str.
- **start** and **end**(optional) - substring is searched within **str[start:end]**



# String Searching Methods

## Example:

```
text = "Python is awesome, isn't it?"
```

```
substring = "is"
```

```
count = text.count(substring)
```

```
print(count)
```

```
substring = "i"
```

```
count = text.count(substring, 8, 25)
```

```
print(count)
```

```
substring = "ton"
```

```
count = text.count(substring)
```

```
print(count)
```

## Output:

```
2
```

```
1
```

```
0
```



# String Replacement Methods

- **replace( )**

Returns a copy of the string where all occurrences of a **substring** is replaced with **another substring**.

**Syntax:** The **replace()** method takes **three** parameters:

- **old** - old substring we want to replace
  - **new** - new substring which would replace the old substring
  - **count** (optional) - the number of times we want to replace the old substring with the new substring
- 
- The **replace()** method returns a copy of the string where **old substring** is replaced with the **new substring**. The original string is unchanged.
  - If the **old substring** is not found, it returns the copy of the original string.



# String Replacement Methods

## Example:

```
text = "Blue Blue Blue"  
newtext= text.replace("ue","ack")  
print(newtext)  
  
newtext= text.replace("ue","ack",2)  
print(newtext)  
  
newtext= text.replace("eu","ack")  
print(newtext)
```

## Output:

```
Black Black Black  
Black Black Blue  
Blue Blue Blue
```



# String Replacement Methods



- **strip()**

Returns a copy of the string with both **leading** and **trailing** characters removed (based on the string argument passed).

**Syntax:** The **strip()** method takes **one** optional parameter:

- **chars** (optional) - a string specifying the set of characters to be removed.
- If the **chars** argument is not provided, all leading and trailing whitespaces are removed from the string.



# String Replacement Methods

## Example:

```
text = " Good Morning"  
newtext= text.strip()  
print("Original text:[ "+text+"]")  
print("New text:[ "+newtext+"]")
```

## Output:

```
Original text:[ Good Morning]  
New text:[Good Morning]
```



# Exercise

**Write a program to simulate user registration process. Your code should do the following:**

- 1. It should first ask the user to input his full name. If he doesn't enter his full name then program should display the error message and again ask the user to enter full name . Repeat the process until the user types his full name.[ full name means a string with atleast 2 words separated with a space]**
- 2. Then it should ask the user to input his password. The rules for password are:**
  - 1. It should contain atleast 8 characters**
  - 2. It should contain atleast 1 digit and 1 upper case letter**

**Repeat the process until the user correctly types his Password.**

**Finally , display the user's first name with a THANK YOU message. Create separate functions for accepting fullname , password and returning firstname**



# Sample Output

```
Type your full name:Sachin
Please enter your full name!
Type your full name:Sachin Kapoor
Type your password:Admin
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Admin1
Password must be of 8 or more characters in length
with atleast 1 digit and 1 capital letter
Type your password:Administrator1
Hello Sachin
Thank you for joining us!
```



# Solution



```
def get_full_name():
    while True:
        name=input("Type your full name:").strip()
        if name.find(" ")!=-1:
            return name
        print("Please enter your full name!")

def get_password():
    while True:
        pwd=input("Type your password:")
        cap_letter_present=False
        digit_present=False
        for x in pwd:
            if x.isdigit():
                digit_present=True
            elif x.isupper():
                cap_letter_present=True
        if digit_present==False or cap_letter_present==False or len(pwd)<8:
            print("Password must be of 8 or more characters in length\nwith atleast 1 digit and 1 capital letter")
        else:
            return pwd

def get_first_name(fullname):
    spacepos=fullname.find(" ")
    return fullname[0:spacepos]

fullname=get_full_name()
pwd=get_password()
firstname=get_first_name(fullname)
print("Hello",firstname,"\\nThank you for joining us!")
```



# String Replacement Methods

- **split()**

The **split()** method breaks up a string at the specified separator and returns a list of strings.

**Syntax:** The **split()** method takes **two** parameters:

- **separator** (optional)- This is the **delimiter**. The string splits at the specified **separator**. If the **separator** is not specified, any whitespace (space, newline etc.) string is a **separator**.
- **maxsplit** (optional) - The **maxsplit** defines the maximum number of splits. The default value of **maxsplit** is **-1**, meaning, no limit on the number of splits.

The **split()** breaks the string at the separator and returns a list of strings.



# String Replacement Methods

## Example:

```
text= 'Live and let live'  
print(text.split())  
grocery = 'Milk, Butter, Bread'  
print(grocery.split(','))  
print(grocery.split(':'))
```

## Output:

```
['Live', 'and', 'let', 'live']  
['Milk', 'Butter', 'Bread']  
['Milk,Butter,Bread']
```



# String Replacement Methods

- **join()**

The **join()** method returns a **string** concatenated with the elements of an **iterable**.

But the **iterable** should only contain strings

The **join()** method takes an **iterable** - objects capable of returning its members one at a time

**Syntax:** `<str>.join(<iterable>)`



# String Replacement Methods

## Example:

```
mylist = ["C","C++","Java","Python"]
```

```
s = "->"
```

```
print(s.join(mylist))
```

## Output:

C->C++->Java->Python



# String Replacement Methods

## Example:

```
letters = 'PYTHON'  
letters_spaced = ' '.join(letters)  
print(letters_spaced)
```

## Output:

P Y T H O N



These Notes Have Python\_world\_In Notes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited

# PYTHON

# LECTURE 36



# Today's Agenda



- **Dictionary-II**
  - Updating Elements In Dictionary
  - Removing Elements From Dictionary
  - Functions Used In Dictionary



# Updating A Dictionary



- Since dictionary is mutable, so we can **add new items** or **change the value** of **existing items** using either of two ways.
- These are:
  - **assignment operator** or
  - **update( )** method of **dictionary** object

# Updating Using Assignment Operator



- **Syntax Of Assignment Operator:**

`dict_var[key]=value`

- When we use **assignment operator** , Python simply searches for the **key** in the **dictionary object**.
- If the **key** is found , it's **value** is replaced with the **value** we have passed, otherwise a ***new key-value pair entry is created***



# Guess The Output ?

```
student_data = {1:'Amit', 2:'Brajesh',3:'Chetan',
4:'Deepak',5:'Neeraj'}
print("Before updating:")
print(student_data)
student_data[2]='Brajendra'
print("After updating:")
print(student_data)
```

## Output:

```
Before updating:
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}
After updating:
{1: 'Amit', 2: 'Brajendra', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}
```



# Guess The Output ?

```
student_data = {1:'Amit', 2:'Brajesh',3:'Chetan',  
 4:'Deepak',5:'Neeraj'}  
print("Before updating:")  
print(student_data)  
student_data[8]='Ankit'  
print("After updating:")  
print(student_data)
```

## Output:

```
Before updating:  
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}  
After updating:  
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj', 8: 'Ankit'}
```

# Updating Using **update()** Method



- **Syntax Of update() Method:**

**dict\_var.update(dict\_var2)**

- The **update()** method **merges** the **keys** and **values** of one **dictionary/iterable** into another, **overwriting** values of the **same key**



# Guess The Output ?

```
student_data = {1:'Amit', 2:'Brajesh',3:'Chetan',
4:'Deepak',5:'Neeraj'}
student_data2={2:'Brajendra',8:'Ankit'}
print("Before updating:")
print(student_data)
student_data.update(student_data2)
print("After updating:")
print(student_data)
```

## Output:

```
Before updating:
{1: 'Amit', 2: 'Brajesh', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj'}
After updating:
{1: 'Amit', 2: 'Brajendra', 3: 'Chetan', 4: 'Deepak', 5: 'Neeraj', 8: 'Ankit'}
```



## Exercise

**Write a program to create a dictionary called **accounts** containing **account id** and **balance** of account holders . Initialize it with the following data:**

**101: 50000**

**102:45000**

**103:55000**

**Now ask the user to input an **account id** and **amount** . If the **account id** is present in the dictionary then **update** the **balance** by adding the **amount** given otherwise add a new entry of **account id** and **balance** in the dictionary. Finally print all the **accounts** details.**



## Sample Output



```
Current Accounts Present
{101: 50000, 102: 45000, 103: 55000}
Enter account id:101
Enter amount:4000
Account updated
Account Details:
{101: 54000, 102: 45000, 103: 55000}
```

```
Current Accounts Present
{101: 50000, 102: 45000, 103: 55000}
Enter account id:104
Enter amount:60000
New Account Created
Account Details:
{101: 50000, 102: 45000, 103: 55000, 104: 60000}
```

# Solution

```
accounts={101:50000,102:45000,103:55000}
print("Current Accounts Present")
print(accounts)
id=int(input("Enter account id:"))
amt=int(input("Enter amount:"))
balance=accounts.get(id)
if balance == None:
    accounts[id]=amt
    print("New Account Created")
else:
    newbalance=balance+amt
    accounts[id]=newbalance
    print("Account updated")
print("Account Details:")
print(accounts)
```

These Notes Have Python\_wno\_World\_India. So commercial Use of this notes is Strictly Prohibited

# Removing Data From Dictionary



- Since dictionary is **mutable**, so we can **remove items** from the dictionary.
- There are certain **methods** available in **Python** to **delete** the **items** or **delete entire** dictionary.
- These are:
  - **pop(key)**: removes the entry with provided **key** and returns its **value**
  - **del**: deletes a single item or the dictionary entirely
  - **clear()**: clears all the items in the dictionary and returns an empty dictionary



# The `pop()` Method



- **Syntax Of `pop()` Method:**

`dict_var.pop(key,[default])`

- The `pop()` method takes **two** parameters:
  - **key** - key which is to be searched for removal
  - **default** - (optional) value which is to be returned when the key is not in the dictionary
- **Return Value From `pop()`**
  - If **key** is found – it returns **value** of removed/popped element from the dictionary
  - If **key** is not found – it returns the value specified as the **second argument (default)**
  - If key is not found and default argument is not specified – it raises **KeyError** exception



# Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(sixMonths)
```

```
print(sixMonths.pop("Jun"))
```

```
print(sixMonths)
```

```
print(sixMonths.pop("Jul",-1))
```

```
print(sixMonths.pop("Jul","Not found"))
```

```
print(sixMonths.pop("Jul"))
```

## Output:

```
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30}
```

```
30
```

```
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31}
```

```
-1
```

```
Not found
```

```
Traceback (most recent call last):
```

```
  File "dictdemo4.py", line 8, in <module>  
    print(sixMonths.pop("Jul"))
```

```
KeyError: 'Jul'
```



# The **del** Operator

- Just like **list** , **Python** also allows us to **delete** an item from the dictionary by calling the **operator/keyword del**
- **Syntax Of `del` Operator:**

**`del dict_var[key]`**

- It removes the **entry** from the dictionary whose **key** is passed as argument
- If the **key** is **not found** or **dictionary** is **empty** it raises **KeyError** exception
- If we do not pass the key then **del** deletes the **entire dictionary object**



# Guess The Output ?

```
threeMonths = {"Jan":31, "Feb":28, "Mar":31}  
print(threeMonths)  
del threeMonths  
print(threeMonths)
```

## Output:

```
{'Jan': 31, 'Feb': 28, 'Mar': 31}  
Traceback (most recent call last):  
  File "dictdemo4.py", line 5, in <module>  
    print(threeMonths)  
NameError: name 'threeMonths' is not defined
```



# The **clear()** Method



- The **clear()** method **removes all items** from the dictionary.
- **Syntax Of clear() Method:**

**dict\_var.clear()**

- The **clear()** method doesn't take any argument
- It returns nothing ( None )



# Guess The Output ?

```
threeMonths = {"Jan":31, "Feb":28, "Mar":31}  
print(threeMonths)  
threeMonths.clear()  
print(threeMonths)
```

## Output:

```
{'Jan': 31, 'Feb': 28, 'Mar': 31}  
}
```

# Functions Used With Dictionary



- Like **list** , Python allows us to use the following functions with **dictionary** object
  - **len()**
  - **max()**
  - **min()**
  - **any()**
  - **all()**
  - **sorted()**



# The **len()** Function

- Returns the **number of items** in the dictionary

## Example:

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(len(sixMonths))
```

## Output:

6



# The **max()** Function

- Returns the **greatest key** present in the dictionary

## Example:

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(max(sixMonths))
```

## Output:

May



# Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, 3:31, "Apr":30, 5:31, 6:30}  
print(max(sixMonths))
```

## Output:

```
print(max(sixMonths))  
TypeError: '>' not supported between instances of 'int' and 'str'
```



# Guess The Output ?



```
sixMonths = {1:31, 2:28, 3:31, 4:30, 5:31, 6:30}  
print(max(sixMonths))
```

**Output:**

**6**



# Guess The Output ?



```
data = {False: 10, True: 5}  
print(max(data))
```

**Output:**

**True**



# Guess The Output ?



```
data={False:0,True:1,None:2}  
print(max(data))
```

## Output:

```
print(max(data))  
TypeError: '>' not supported between instances of 'NoneType' and 'bool'
```



## Exercise



**Write a program to create a dictionary called `players` and accept names of `5 players` and their `runs` from the user. Now find out the `highest score`**



## Sample Output

```
Enter player name:Virat
Enter runs:100
Enter player name:Dhoni
Enter runs:200
Enter player name:Shikhar
Enter runs:45
Enter player name:Kedar
Enter runs:50
Enter player name:Raina
Enter runs:75
{'Virat': 100, 'Dhoni': 200, 'Shikhar': 45, 'Kedar': 50, 'Raina': 75}
Highest runs are : 200
```



# Solution

```
players={}
i=1
while i<=5:
    name=input("Enter player name:")
    runs=int(input("Enter runs:"))
    players[name]=runs
    i=i+1
print(players)
runs=max(players.values())
print("Highest runs are :",runs)
```



## Exercise



**Modify the previous code so that you are able to find out the name of the player also who has scored the highest score**



## Sample Output

```
Enter player name:Virat
Enter runs:100
Enter player name:Shikar
Enter runs:45
Enter player name:Dhoni
Enter runs:120
Enter player name:Kedar
Enter runs:90
Enter player name:Raina
Enter runs:50
{'Virat': 100, 'Shikar': 45, 'Dhoni': 120, 'Kedar': 90, 'Raina': 50}
Player with top score is Dhoni with score of 120
```



# Solution

```
players={}
i=1
while i<=5:
    name=input("Enter player name:")
    runs=int(input("Enter runs:"))
    players[name]=runs
    i=i+1
print(players)
max=0
pl=""
for name,runs in players.items():
    if runs>max:
        max=runs
        pl=name
print("Player with top score is",pl,"with score of",max)
```



# The **min()** Function



- Returns the **least** item present in the dictionary

## Example:

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
"Jun":30}
```

```
print(min(sixMonths))
```

## Output:

Apr



# The **any()** Function

- Like list and tuple , **any()** function accepts a **dict** as argument and returns **True** if atleast **one element** of the **dict** is **True**. If not, this method returns **False**. If the **dict** is empty, then also it returns **False**

## Example:

```
data={1:31,2:28,3:30}
```

```
print(any(data))
```

## Output:

```
True
```



# Guess The Output ?



```
data={0:1,False:2,"":3}  
print(any(data))
```

**Output:**

**False**



# Guess The Output ?



```
data={'o':1,False:2,'':3}  
print(any(data))
```

## Output:

True



# Guess The Output ?



```
data= {}  
print(any(data))
```

**Output:**

**False**



# The **all()** Function



- The **all()** function accepts a **dict** as argument and returns **True** if **all the keys** of the **dict** are **True** or if the **List** is **empty**. If not, this method returns **False**.

## Example:

```
data={1:31,2:28,3:30,0:10}  
print(all(data))
```

## Output:

False



# Guess The Output ?

```
data={1:31,2:28,3:30}  
print(all(data))
```

**Output:**

True



# Guess The Output ?

```
data={'o':1,True:2,' ':3}  
print(all(data))
```

## Output:

True



# Guess The Output ?



```
data= {}  
print(all(data))
```

**Output:**

**True**



# The **sorted()** Function



- Like it is with **lists** and **tuples**, the **sorted()** function returns a **sorted list** of only **keys** in the **dictionary**.
- The sorting is in **ascending order**, and doesn't modify the original **dictionary**.



# Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30, "May":31,  
             "Jun":30}  
print(sorted(sixMonths))  
print(sixMonths)
```

## Output:

```
['Apr', 'Feb', 'Jan', 'Jun', 'Mar', 'May']  
{'Jan': 31, 'Feb': 28, 'Mar': 31, 'Apr': 30, 'May': 31, 'Jun': 30}
```



# Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30,  
             4:31, 5:30}  
  
print(sorted(sixMonths))
```

## Output:

```
print(sorted(sixMonths))  
TypeError: '<' not supported between instances of 'int' and 'str'
```



# Guess The Output ?

```
sixMonths = {"Jan":31, "Feb":28, "Mar":31, "Apr":30,  
"May":31, "Jun":30}  
print(sorted(sixMonths,reverse=True))
```

## Output:

```
['May', 'Mar', 'Jun', 'Jan', 'Feb', 'Apr']
```

# The **kwargs** Function Argument



- To understand **kwargs**, try to figure out the output of the code below

## Example:

```
def addnos(x,y,z):
    print("sum:",x+y+z)
addnos(10,20,30)
addnos(10,20,30,40,50)
```

## Output:

```
sum: 60
Traceback (most recent call last):
  File "inp_5.py", line 4, in <module>
    addnos(10,20,30,40,50)
TypeError: addnos() takes 3 positional arguments but 5 were given
```

# The **kwargs** Function Argument



- To overcome this problem , we used the technique of **variable length arguments**, where we prefix the function parameter with an asterisk
- This allows us to **pass any number of arguments** to the function and inside the function they are received as **tuple**

# The **args** Function Argument



```
def addnos(*args):  
    sum=0  
    for x in args:  
        sum=sum+x  
    print("sum:",sum)  
addnos(10,20,30)  
addnos(10,20,30,40,50)
```

## Output

```
sum: 60  
sum: 150
```

# The **args** Function Argument



- Using **\*args**, we cannot pass **keyword arguments**
- So, **Python** has given us a solution for this, called **\*\*kwargs**, which allows us to pass the **variable length of keyword arguments** to the function.

# The **kwargs** Function Argument



- In the function, we use the **double asterisk \*\*** before the **parameter name** to denote this type of argument.
- The arguments are passed as a **dictionary** and the name of the dictionary is the name of parameter
- The **keywords** become **keys** and the **actual data** passed becomes **values**

# The **kwargs** Function Argument



```
def show_details(**data):
```

```
    print("\nData type of argument:", type(data))
```

```
    for key, value in data.items():
```

```
        print("{} is {}".format(key, value))
```

```
show_details(Firstname="Sachin", Lastname="Kapoor", Age=38,  
Phone=9826012345)
```

```
show_details(Firstname="Amit", Lastname="Sharma",  
Email="amit@gmail.com", Country="India", Age=25,  
Phone=9893198931)
```

# The **kwargs** Function Argument



```
Data type of argument: <class 'dict'>
Firstname is Sachin
Lastname is Kapoor
Age is 38
Phone is 9826012345
```

```
Data type of argument: <class 'dict'>
Firstname is Amit
Lastname is Sharma
Email is amit@gmail.com
Country is India
Age is 25
Phone is 9893198931
```



# PYTHON

# LECTURE 37

These Notes Have Python\_world\_In Notes Copy rights under copy right  
Acts of Government of India. So commercial Use of this notes is Strictly prohibited



# Today's Agenda



- **Dictionary-III**
  - Dictionary Methods
  - Dictionary Comprehension
  - Restrictions On Keys



# Dictionary Methods



- Python provides us following methods to work upon dictionary object:
  - **clear()**
  - **copy()**
  - **get()**
  - **items()**
  - **keys()**
  - **pop()**
  - **update()**
  - **values()**



# The **copy()** Method

- This method returns a **shallow copy** of the **dictionary**.

## Syntax:

`dict.copy()`

## Example:

```
original = {1:'one', 2:'two'}  
new = original.copy()  
print('new:', new)  
print('original:', original)
```

## Output:

```
new:  {1: 'one', 2: 'two'}  
original:  {1: 'one', 2: 'two'}
```



## copy() v/s =

- When **copy()** method is used, a **new dictionary** is created which is filled with a **copy of the data** from the original dictionary.
- When **=** operator is used, a **new reference** to the **original dictionary** is created.



# Guess The Output ?

```
original = {1:'one', 2:'two'}  
new = original  
new.clear()  
print('new: ', new)  
print('original: ', original)
```

## Output:

```
new:  {}  
original:  {}
```

```
original = {1:'one', 2:'two'}  
new = original.copy()  
new.clear()  
print('new: ', new)  
print('original: ', original)
```

## Output:

```
new:  {}  
original:  {1: 'one', 2: 'two'}
```

# Using **in** And **not in** With Dictionary



- We can apply the '**in**' and '**not in**' operators on a dictionary to check whether it contains a certain **key**.
- If the **key** is **present** then **in** returns **True**, otherwise it returns **False**.
- Similarly , if the **key** is **not present** **not in** returns **True** , otherwise it returns **False**



# Guess The Output ?

```
cars = {"Maruti":"Ciaz","Hyundai":"Verna","Honda":"Amaze"}  
print(cars)  
print("Hyundai is present:", "Hyundai" in cars)  
print("Audi is present:", "Audi" in cars)  
print("Renault not present:", "Renault" not in cars)
```

## Output:

```
{'Maruti': 'Ciaz', 'Hyundai': 'Verna', 'Honda': 'Amaze'}  
Hyundai is present: True  
Audi is present: False  
Renault not present: True
```



# Dictionary Comprehension



- Just like we have **list comprehension**, we also have **dictionary comprehension**
- **Dictionary Comprehension** is a mechanism for transforming **one dictionary** into **another dictionary**.
- During this **transformation**, items within the **original dictionary** can be **conditionally** included in the **new dictionary** and each item can be transformed as needed.

# Syntax For Dictionary Comprehension



- **Syntax:**

```
dict_variable = { key:value for (key,value) in iterable}
```

- **Explanation**

- **Iterable** can be any object on which iteration is possible
- **(key,value)** is the **tuple** which will receive these **key-value** pairs one at a time
- **key:value** is the expression or **key-value** pair which will be assigned to **new dictionary**



## Exercise



- Write a program to produce a **copy** of the dictionary **cars** using **dictionary comprehension**

```
cars = {"Maruti": "Ciaz", "Hyundai": "Verna", "Honda": "Amaze"}  
newcars={ k:v for (k,v) in cars.items()}  
print(newcars)
```

### Output:

```
{'Maruti': 'Ciaz', 'Hyundai': 'Verna', 'Honda': 'Amaze'}
```



## Exercise

- Write a program to produce a **new dictionary** from the given dictionary with the **values** of each key **getting doubled**

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}  
double_dict1 = {k:v*2 for (k,v) in dict1.items()}  
print(double_dict1)
```

### Output:

```
{'a': 2, 'b': 4, 'c': 6, 'd': 8, 'e': 10}
```

## Exercise

- Write a program to produce a **new dictionary** from the given dictionary with the **keys** of each key **getting doubled**

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}  
double_dict1 = {k*2:v for (k,v) in dict1.items()}  
print(double_dict1)
```

### Output:

```
{'aa': 1, 'bb': 2, 'cc': 3, 'dd': 4, 'ee': 5}
```

## Exercise



- Write a program to accept a string from the user and print the frequency count of it's letters , i.e. how many times each letter is occurring in the string

### **Output:**

```
Type a string:WE LOVE INDIA
W : 1
E : 2
E : 2
L : 1
O : 1
V : 1
I : 2
N : 1
D : 1
I : 1
A : 1
```



# Solution

```
str=input("Type a string:")  
mydict={ch:str.count(ch) for ch in str}  
for k,v in mydict.items():  
    print(k,":",v)
```

# Adding Conditions To Dictionary Comprehension



- Like list comprehension , **dictionary comprehension** also allows us to add **conditions** to make it more powerful.
- **Syntax:**  
`dict_variable = { key:value for (key,value) in iterable <test_cond> }`
- As usual , only those **key-value** pairs will be returned by **dictionary comprehension** which satisfy the condition

## Exercise

- Write a program to produce a **new dictionary** from the given dictionary but with the **values** that are greater than **2** and store their doubles

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
dict2 = {k:v*2 for (k,v) in dict1.items() if v>2}
```

```
print(dict2)
```

### Output:

```
{'c': 6, 'd': 8, 'e': 10}
```

## Exercise



- Write a program to produce a **new dictionary** from the given dictionary but with the double of the **values that are greater than 2 as well as multiple of 2**

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
dict2 = {k:v*2 for (k,v) in dict1.items() if v>2 if v%2==0}
```

```
print(dict2)
```

### Output:

```
{'d': 8}
```

## Exercise

- Write a program to produce a **new dictionary** from the given dictionary but the value should be the string “**EVEN**” for even values and “**ODD**” for odd values

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

```
dict2= {k:'even' if v%2==0 else 'odd' for (k,v) in dict1.items()}
```

```
print(dict2)
```

### Output:

```
{'a': 'odd', 'b': 'even', 'c': 'odd', 'd': 'even', 'e': 'odd'}
```

# Restrictions On Dictionary Keys



1. Almost any type of value can be used as a dictionary key in **Python**, like **integer**, **float**, **Boolean** etc

## Example:

```
d={65:"A", 3.14:"pi", True:1}
```

```
print(d)
```

## Output:

```
{65: 'A', 3.14: 'pi', True: 1}
```

# Restrictions On Dictionary Keys



2. We can even use class names as keys.

## Example:

```
d={int:1, float:2, bool:3}
```

```
print(d)
```

```
print(d[float])
```

## Output:

```
{<class 'int'>: 1, <class 'float'>: 2, <class 'bool'>: 3}  
2
```

# Restrictions On Dictionary Keys



3. Duplicate keys are not allowed. If we assign a value to an already existing dictionary key, it does not add the key a second time, but replaces the existing value:

## Example:

```
d={"MP":"Indore","UP":"Lucknow","RAJ":"Jaipur"}  
print(d)  
d["MP"]="Bhopal"  
print(d)
```

## Output:

```
{'MP': 'Indore', 'UP': 'Lucknow', 'RAJ': 'Jaipur'}  
{'MP': 'Bhopal', 'UP': 'Lucknow', 'RAJ': 'Jaipur'}
```

# Restrictions On Dictionary Keys



4. If we specify a key a second time during the initial creation of a dictionary, the second occurrence will override the first:

## Example:

```
d={"MP":"Indore","UP":"Lucknow","RAJ":"Jaipur","MP":"Bhopal"}  
print(d)
```

## Output:

```
{'MP': 'Bhopal', 'UP': 'Lucknow', 'RAJ': 'Jaipur'}
```

# Restrictions On Dictionary Keys



5. A dictionary key must be of a type that is **immutable**. Like **integer**, **float**, **string** and **Boolean**—can serve as dictionary keys. Even a **tuple** can also be a dictionary key, because **tuples** are **immutable**:

## Example:

```
d = {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}  
print(d)  
print(d[(1,2)])
```

## Output:

```
{(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}  
b
```

# Restrictions On Dictionary Keys



6. However, neither a list nor another dictionary can serve as a dictionary key, because lists and dictionaries are mutable:

## Example:

```
d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}  
print(d)
```

## Output:

```
d = {[1, 1]: 'a', [1, 2]: 'b', [2, 1]: 'c', [2, 2]: 'd'}  
TypeError: unhashable type: 'list'
```

# Restrictions On Dictionary Values



- There are **no restrictions** on dictionary **values**.
- A dictionary value can be any type of object Python supports, including **mutable types** like **lists** and **dictionaries**, and **user-defined objects**
- There is also no restriction against a particular value appearing in a dictionary multiple times:



## Exercise



- Write a complete **COUNTRY MANAGEMENT APP**. Your code should store **COUNTRY CODE** and **COUNTRY NAME** as **key-value** pair in a **dictionary** and allow perform following operations on the dictionary :
  - **View**
  - **Add**
  - **Delete**

To start the program initialize your dictionary with the following set of key-value pairs:

**IN→India**

**US→America**

**AU→Australia**

**CA→Canada**



# Sample Output

```
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IN  
Country is India  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IT  
There is no country for country code IT  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:add
```

```
Your choice:add  
Enter country code:IT  
Enter country name:Italy  
Italy added to the list  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN IT US  
Enter country code:IT  
Country is Italy  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:add  
Enter country code:IN  
IN is already used by India  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```



# Solution



```
def show_menu():
```

```
    print("SELECT AN OPTION:")
```

```
    print("view: View country names")
```

```
    print("add: Add a country")
```

```
    print("del: Delete a country")
```

```
    print("exit- Exit the program")
```

```
    print()
```

```
def show_codes(countries):
```

```
    pass
```



# Solution



```
def view_country(countries):  
    pass
```

```
def add_country(countries):  
    pass
```

```
def del_country(countries):  
    pass
```



# Solution

```
countries={"IN":"India","US":"America","AU":"Australia","CA  
":"Canada"}
```

**while True:**

**show\_menu()**

**choice=input("Your choice:")**

**if choice=="view":**

**view\_country(countries)**

**elif choice=="add":**

**add\_country(countries)**

**elif choice=="del":**

**del\_country(countries)**

**elif choice=="exit":**

**break;**

**else:**

**print("Wrong choice ! Try again!")**



## Exercise



- Modify the previous code , so that now you are able to store 3 values for each key . These are COUNTRY NAME , CAPITAL CITY and POPULATION. Provide same options to the user and start with the following data

**IN→India , Delhi,1320000000**

**US→America,Washington,320000000**

**AU→Australia,Canberra,24000000**

**CA→Canada,Ottawa,940000**



# Sample Output

```
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IN  
Country name is: India  
Country capital is: Delhi  
Country population is: 1320000000  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN US  
Enter country code:IT  
There is no country for country code IT  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:add  
Enter country code:IT  
Enter country name:Italy  
Enter capital city:Rome  
Enter population:2870000  
Italy added to the list  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:view  
Country Codes:AU CA IN IT US  
Enter country code:IT  
Country name is: Italy  
Country capital is: Rome  
Country population is: 2870000  
SELECT AN OPTION:  
view: View country names  
add: Add a country  
del: Delete a country  
exit- Exit the program
```

```
Your choice:exit
```