UNIT 2 Lists, Dictionaries And Sets

Lists: List structures - Common list operations - List traversal - Lists in Python - Python list type –Tuples – sequences - Nested lists - Iterating over lists in python.
Dictionaries and Sets: Dictionary types in Python - Set data type- Strings and its operations.

2.1 List

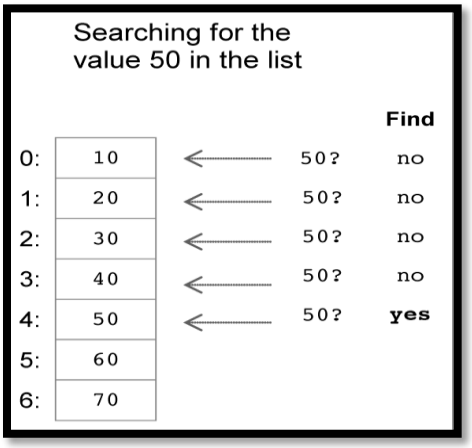
2.1.1 List structures:

- A **list** is a *linear data structure* , meaning that its elements have a linear ordering.
- Each item in the list is identified by its *index value* .
- To begin numbering sequences of items with an index value of 0 . This is referred to as *zero-based indexing* .
- 0 index has value 100, 1 index has value 200 etc.

Index	Value
0	100
1	200
2	300
3	400
4	500

2.1.2 List Traversal

- A **list traversal** is a means of accessing, one-by-one, the elements of a list.
- Each element can be accessed one-by-one, starting with the first,and ending with the last element.
- Similarly, the list could be traversed starting with the last element and ending with the first.



2.1.3 Python List Type

- A **list** in Python is a mutable, linear data structure of variable length and allowing mixed-type elements.
- *Mutable* means that the contents of the list may be altered.

- Lists in Python use zero based indexing.
  - Example; [1, 2, 3]
    ['one', 'two', 'three']
    ['apples', 50, True]
- Thus, all lists have index values 0 ... n-1, where n is the number of elements in the list.
- Lists are denoted by a comma-separated list of elements within square brackets.
- An **empty list** is denoted by an empty pair of square brackets, [].

2.1.4. List Operations

- Operations commonly performed on lists include retrieve, update, insert, delete (remove) and append.

1.Retrieve operation:

- Elements of a list are accessed by using an index value within square brackets.
- Example:

li = [1, 2, 3]

  - li [0] → 1 access of first element
  - li [1] → 2 access of second element
  - l i[2] → 3 access of third element
- The elements in list li can be summed as follows,
  - sum = li[0] + li[1] + li[2]
  - print(sum) -> o/p is 6

2.Replace operation:

- Elements of a list can be updated (replaced)
- Example:

li=[1, 2, 3]

  - li[2] = 4 o/p : [1, 2, 4] replacement of 3 with 4 at index 2

3. Delete operation:

- Elements of a list can be deleted (removed)
- Example:

li=[1, 2, 3]

  - del li[2] o/p : [1, 2] removal of 4 at index 2

4.Insert operation:

- Insert provide a means of inserting/adding elements in a list
- Example

li=[1, 2, 3]

  - li.insert(1, 3) o/p : [1, 3, 2,3] insertion of 3 at index 1

5. Append operation:

- Append also provide a means of altering a list

- Example:

li=[1, 2, 3]

- li.append(4) o/p : [1, 3, 2, 4] appending of 4 to end of list

6. Sort and reverse operation:

- methods sort() is used to arrange the elements in ascending order.
- Example:

li=[1, 3, 2]

- li.sort() o/p : [1,2,3] sort the list

7. Reverse operation:

- methods reverse() reorder the elements of a given list.
- Example:

li=[1, 3, 2]

- li.reverse() o/p : [2,3,1] reverse the element in the list

8.extend operation:

- The extend() method is used to add more than one element at the end of the list.
- Example:

li=[1, 2, 3]

- li.extend([4,5,6]) o/p : [1,2,3,4,5,6] extend the list

9.len operation:

- The len() method returns the length of the list, i.e. the number of elements in the list.
- Example:

li=[1, 2, 3]

- print(len(li)) o/p : 3 print length of the list

10.min and max operation:

- The min() method returns the minimum value in the list.
- The max() method returns the maximum value in the list. Both the methods accept only list having elements of similar type.
- Example:

li=[1, 2, 3]

- print(min(li)) o/p : 1 print minimum item of the list
- print(max(li)) o/p : 3 print maximum item of the list

Operation	fruit = ['banana', 'apple, 'cherry']	
Replace	fruit[2] = 'coconut'	['banana', 'apple', 'coconut']
Delete	del fruit[1]	['banana', 'cherry']
Insert	fruit.insert(2, 'pear')	['banana', 'apple', 'pear', 'cherry']
Append	fruit.append('peach')	['banana', 'apple', 'cherry', 'peach']
Sort	fruit.sort()	['apple', 'banana', 'cherry']
Reverse	fruit.reverse()	['cherry', 'banana', 'apple']

Example :

```
li=['apple','orange','mango','banana']
print(type(li))
#print all item in a list
print(li)
#print items in a list based on index start is 0
print(li[0])
print(li[1:])
#append() add an item in end of the list
li.append("Berry")
print(li)
#remove() delete an item
li.remove("apple")
print(li)
#Insert an item based on index
li.insert(2,"apple")
print(li)
#Crate mixed list
mi=['Hai',2,5.999]
print(mi)
print(li+mi)
```

Output:

```
<class 'list'>
['apple', 'orange', 'mango', 'banana']
apple
['orange', 'mango', 'banana']
['apple', 'orange', 'mango', 'banana', 'Berry']
['orange', 'mango', 'banana', 'Berry']
```

['orange', 'mango', 'apple', 'banana', 'Berry']
['Hai', 2, 5.999]
['orange', 'mango', 'apple', 'banana', 'Berry', 'Hai', 2, 5.999]

2.1.5 Tuples

- A **tuple** is an *immutable* linear data structure.
- Once a tuple is defined, it cannot be altered.
- Once a tuple is created, you cannot change its values. Tuples are **unchangeable**.
  Example : nums =(10, 20, 30)
  student = ('John Smith', 48, 'Computer Science', 3.42)
- Otherwise, tuples and lists are essentially the same.
- Tuples are denoted by parentheses
- An **empty tuple** is represented by a set of empty parentheses, ().
- delete, update, insert, and append operations are not defined on tuples.

Tuple Operations

1. Accessing Items in a Tuple:

- One can access the elements of a tuple in multiple ways, such as indexing, negative indexing, range, etc.
  tu =(1, 2, 3)
  - tu[0] → 1 access of first element
  - tu[1] → 2 access of second element
- We can find the use of negative indexing on tuples,
  - print(tu[-1]) -> 3
- Find a range of tuples,
  - print(tu[1:2]) -> 2

2. Concatenation Operation on Tuples:

- Concatenation simply means linking things together.
  - tu1 = (1, 3, 4)
    tu2 = (‘red’, ’green’, ‘blue’)
    print (tu1 + tu2)
  - Output: (1,3,4,’red’,’green’,’blue’)

3. Finding length of Tuples:

- It is used to find how many values are in a tuple.
  - tu = (1, 3, 4, ‘test’, ‘red’)
    print(len(tu))
  - Output: 5

4. **min and max operation:**

- The min() method returns the minimum value in the tuple.
- The max() method returns the maximum value in the tuple.
- Both the methods accept only tuple having elements of similar type.
    - tu = (1, 2, 3, 6)
      print(max(tu))
      print(min(tu))

    - Output:
      6
      1

Example:

```
tu=('apple','orange','mango','banana')
print(type(tu))
#print all item in a list
print(tu)
#print items in a tuple based on index start at 0
print(tu[0])
print(tu[1:3])
print(tu[:2])
#len() length of the tuple
print(len(tu))
```

Output

```
<class 'tuple'>
('apple', 'orange', 'mango', 'banana')
apple
('orange', 'mango')
('apple', 'orange')
4
```

2.1.6 Sequences

- A **sequence** in Python is a linearly ordered set of elements accessed by an index number.
- Lists, tuples, and strings are all sequences.
- Sequence operations are:

Operation		String s = 'hello' w = 'l'	Tuple s = (1,2,3,4) w = (5,6)	List s = [1,2,3,4] w = [5,6]
Length	len(s)	5	4	4
Select	s[0]	'h'	1	1
Slice	s[1:4] s[1:]	'ell' 'ello'	(2, 3, 4) (2, 3 ,4)	[2 ,3, 4] [2, 3, 4]
Count	s.count('e') s.count(4)	1 *error*	0 1	0 1
Index	s.index('e') s.index(3)	1 --	-- 2	-- 2
Membership	'h' in s	True	False	False
Concatenation	s + w	'hello!'	(1, 2, 3, 4, 5, 6)	[1, 2, 3, 4, 5, 6]
Minimum Value	min(s)	'e'	1	1
Maximum Value	max(s)	'o'	4	4
Sum	sum(s)	*error*	10	10

- For any sequence s, len(s)gives its length, and s[k] retrieves the element at index k.
- The s[index :] form of the slice operation returns a string containing all the list elements starting from the given index location to the end of the sequence.
- The count method returns how many instances of a given value occur within a sequence
- The + operator is used to denote concatenation.

2.1.7 Nested Lists

- Lists and tuples can contain elements of any type, including other sequences.
- Thus, lists and tuples can be nested to create arbitrarily complex data structures.

class_grades = [[85, 91, 89], [78, 81, 86], [62, 75, 77], ...]

- In this list, for example, class_grades[0] equals [85, 91, 89], and class_grades[1] equals [78, 81, 86].
- Thus, the following would access the first exam grade of the first student in the list,

class_grades[0][0] → [85, 91, 89][0] → 85

- To calculate the class average on the first exam, a while loop can be constructed that iterates over the first grade of each student's list of grades,

```
sum = 0
k = 0
while k < len(class_grades):
     sum = sum + class_grades[k][0]
     k = k + 1
average_exam1 = sum / float(len(class_grades))
```

2.1.8 Iterating Over Lists (Sequences) in Python

For Loops

- A for statement is an iterative control statement that iterates once for each element in a specified sequence of elements.
- Thus, for loops are used to construct definite loops.
- Syntax

```
for iterating_var in sequence:
  statement(s)
```

- In the for loop version, loop variable k *automatically* iterates over the provided sequence of values.
- Example:

```
li=[10,20,30,40,50]
for k in li:
    print(k)
```

- Variable k is referred to as a **loop variable** .
- Since there are 5 elements in the provided list, the for loop iterates exactly five times.
- Using while loop:

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

- The for statement can be applied to all sequence types, including strings.

```
for ch in 'Hello':
    print(ch)
```

The Built-in range Function

- Python provides a built-in **range function** that can be used for generating a sequence of integers that a for loop can iterate over, as shown below.

```
sum =0
for k in range(1, 11):
    sum = sum + k
```

- The values in the generated sequence include the starting value, up to *but not including* the ending value.

- For example, range(1, 11) generates the sequence [1, 2, 3, 4, 5, 6, 7, 8,9, 10].
- The range function is convenient when long sequences of integers are needed.
- Actually, range does not create a sequence of integers.
- It creates a *generator function* able to produce each next item of the sequence when needed.
- For example, range(0, 11, 2) produces the sequence [0, 2, 4, 6, 8, 10], with a step value of 2.
- A sequence can also be generated "backwards" when given a negative step value.
- For example, range(10, 0, -1) produces the sequence [10,9, 8, 7, 6, 5, 4, 3, 2, 1].

Iterating Over List Elements vs. List Index Values

- An **index variable** is a variable whose changing value is used to access elements of an indexed data structure.

Loop variable iterating over the elements of a sequence.	**Loop variable iterating over the index values of a sequence**

n=[10,20,30,40,50]
sum=0
for k in n:
  sum = sum + k
print(sum)
 	
O/P: 150
	n=[10,20,30,40,50]
	sum=0
	for k in range(len(n)):
	  sum = sum + n[k]
	print(sum)
	
	O/P: 150

- Note that the range function may be given only one argument.
- In that case, the starting value of the range defaults to 0. Thus, range(len(n)) is equivalent to range(0,len(n))

While Loops and Lists (Sequences)

- There are situations in which a sequence is to be traversed while a given condition is true.
- In such cases, a while loop is the appropriate control structure.
- To determine whether the value 40 occurs in list li.
- In this case, once the value is found, the traversal of the list is terminated.

```
n=[10,20,30,40,50]
k=0
item=int(input("Enter the item :"))
found_item=False
while k<len(n) and not found_item:
    if n[k]==item:
        found_item=True
    else:
        k=k+1
if found_item:
    print("Item found...")
else:
```

```
            print("Item not found...")
```

- Variable k is initialized to 0, and used as an index variable.
- Thus, the first time through the loop, k is 0, and n [0] (with the value 10) is compared to item. Since they are not equal, the second clause of the if statement is executed, incrementing k to 1.
- The loop continues until either the item is found, or the complete list has been traversed

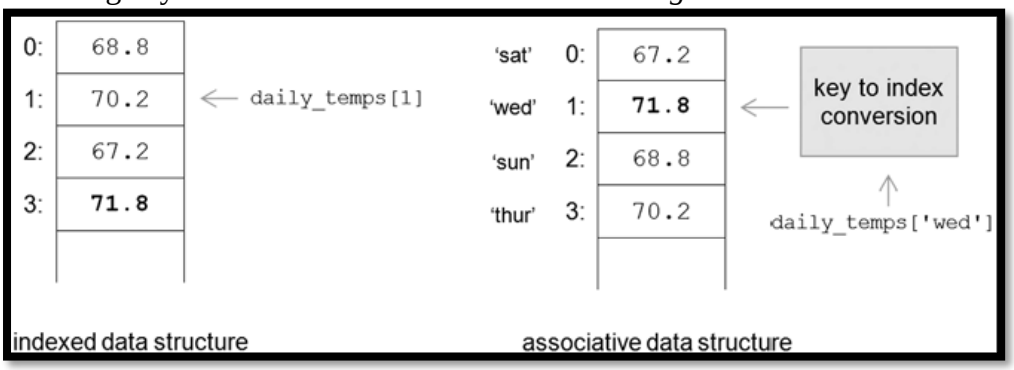
2.2 Dictionaries and Sets

2.2.1 Dictionary Type in Python

- A **dictionary** in Python is a mutable, associative data structure of variable length denoted by the use of curly braces..
- A dictionary is a collection which is unordered and changeable.
- In Python dictionaries are written with curly brackets, and they have keys and values.
- In Python, an associative data structure is provided by the *dictionary type* .
- A **dictionary** is a mutable, associative data structure of variable length.
- Syntax :

  { Key: Value }
- Example :

  daily_temps = {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8,'thur': 73.2, 'fri': 75.6, 'sat': 74.0}
- Dictionary daily_temps stores the average temperature for each day of the week
- Each temperature has associated with it a unique key value ('sun', 'mon', etc.).
- You can access the items of a dictionary by referring to its key name, inside square brackets:
- The syntax for accessing an element of a dictionary is the same as for accessing elements of sequence types, except that a key value is used within the square brackets instead of an index value: daily_temps['sun'].
- The specific location that a value is stored is determined by a particular method of converting key values into index values called *hashing* .



Dictionary Operations

1. copy()

- The copy() method is used for copying the contents of one dictionary to another dictionary
    - bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}
      bikes1 = bikes.copy()
      print(bikesa)
    - Output: {'B1': 'CT100', 'B2': 'HeroHonda', 'B3': 'Yamaha'}

2. get()

- The get() method is used to get the value of the given key in a dict variable.
    - bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}
      print('The second bike:',bikes.get('B2'))
    - Output: The second bike: HeroHonda

3. update()

- Two key processes can be achieved by using this update() method.
- First process of revising an existing key-value pair in the dictionary
- Other process of inserting a fresh entry into the dictionary.

```
bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}
bikes.update({'B4': 'TVS'})
print(bikes)
bikes.update({'B1':'Suzuki'})
print(bikes)
```

- Output:

```
{'B1': 'CT100', 'B2': 'HeroHonda', 'B3': 'Yamaha', 'B4': 'TVS'}
{'B1': 'Suzuki', 'B2': 'HeroHonda', 'B3': 'Yamaha', 'B4': 'TVS'}
```

4. keys()

- For displaying the entire set of keys in the dictionary the keyes() method is used.

```
bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}
print(bikes.keys())
```

- Output:

```
dict_keys(['B1', 'B2', 'B3'])
```

5. values()

- The values() method is used to display all the values in the dictionary
    - bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}
      print(bikes.values())
    - Output:
      dict_values(['CT100', 'HeroHonda', 'Yamaha'])

6. len()

- len() is used to count of total key value pairs in a dictionary data type.
    - bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}
      print(len(bikes))
    - Output: 3

7. del

- del keyword is used to delete a value from dictionary.
    - bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}

```
del bikes['B2']
print(bikes)
```

- Output: {'B1': 'CT100', 'B3': 'Yamaha'}

8. Insert or update the item

- d[key]=value. Set the associated value for key to value. Used to add a new key/value pair or replace the existing key/value pair.
  - bikes={'B1':'CT100','B2':'HeroHonda','B3':'Yamaha'}
    bikes['B4']='Pulsar'
    print(bikes)
    bikes['B2']='yamaha'
    print(bikes)
  - Output:
    {'B1': 'CT100', 'B2': 'HeroHonda', 'B3': 'Yamaha', 'B4': 'Pulsar'}
    {'B1': 'CT100', 'B2': 'yamaha', 'B3': 'Yamaha', 'B4': 'Pulsar'}

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()   #copy
print(mydict)
x = thisdict["model"]       #print model value
print(x)
thisdict["year"] = 2018     #change year value
print(thisdict)
thisdict["color"] = "red"     #add new key & value
print(thisdict)
del thisdict["model"]
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
Mustang
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018, 'color': 'red'}
{'brand': 'Ford', 'year': 2018, 'color': 'red'}
```

2.2.2 Set Data type

- A **set** is a mutable data type with nonduplicate, unordered values, providing the usual mathematical set operations.
- Set is a collection which is unordered and unindexed. No duplicate members.
- Example: s={1,2,3,4}
  s={‘red’,’blue’green’}
- These data can be of any type.It can be an integer, float, string or both.
- In Sets, values are stored inside the curly brackets.
- Values stored in the sets cannot be duplicated. They are unique.

- The values stored in the sets are mutable means they can be changed.

Set Operations

1. Add Method

- This method is used to add the element in the set.
- The element will be added to the unspecific location as it is unordered.
- Example:
  - sample = {1,2,3,4,5,6,5}
    sample.add(7)
    print(sample)
  - Output: {1, 2, 3, 4, 5, 6, 7}

2. Remove Method

- This method is used to remove any value from the set.
- As there is no indexing so we have to pass the value that we want to remove from the set.
- Example:
  - sample = {1,2,3,4,5}
    sample.remove(5)
    print(sample)
  - Output: {1, 2, 3, 4}

3. Discard Method

- This method is also used to remove an element from the set, but the remove method will generate an error if the passed value does not exist in the set.
- This method won't return any error.
- Example:
  - sample = {1,2,3,4,5}
    sample.discard(5)
    print(sample)
  - Output: {1, 2, 3, 4}

4. Clear Method

- This method is used to remove all the elements from the set.
- Example:
  - sample = {1,2,3,4,5}
    sample.clear()
    print(sample)
  - Output: set()

5. Copy Method

- This method is used to make a copy of the Set.
- Example:
  - sample = {1,2,3,4,5}
    new_sample = sample.copy()
    print(new_sample)
  - Output: {1, 2, 3, 4, 5}

6. Pop Method

- It will remove the first element of the set.
- Example:
  - sample = {1,2,3,4,5}
    sample.pop()
    print(sample)
  - Output: {2, 3, 4, 5}

7. Update Method

- We can use this method to add multiple values to set, we can also pass new sets or list or both and it will be added to the set.
- Example:
  - sample = {1,2,3,4,5}
    sample.update([6,7,8])
    print(sample)
  - Output: {1, 2, 3, 4, 5, 6, 7, 8}

Python Set Operation

- We can perform mathematical operations like Union, Intersection, Difference on sets.
- This can be done in two ways using methods or operators.

1. Union

- This function used to merge the elements of two sets into one set.
- Example:
  - sample = {1,2,3,4,5}
    new_sample = {5,6,7,8}
    sample_union = (sample | new_sample)
    sample_union1 = sample.union(new_sample)
    print(sample_union)
    print(sample_union1)
  - Output: {1, 2, 3, 4, 5, 6, 7, 8} {1, 2, 3, 4, 5, 6, 7, 8}

2. Intersection

- This method is used to find out the common elements from two sets.
- Example:
  - sample = {1,2,3,4,5}
    new_sample = {1,2,3}
    sample_intersection = (sample & new_sample)
    sample_intersection1 = sample.intersection(new_sample)
    print(sample_intersection)
    print(sample_intersection1)
  - Output: {1, 2, 3} {1, 2, 3}

3. Difference

- This method is used to find the difference of the element from one set to the second set.
- It means it will return the element that doesn't exist in the set first as compared to set second.
- Example:

- sample = {1,2,3,4,5}
  new_sample = {4,5,6,7,8,9}
  sample_difference = (sample - new_sample)
  sample_difference1 = sample.difference(new_sample)
  print(sample_difference)
  print(sample_difference)
- Output: {1, 2, 3} {1, 2, 3}

Example:

```
s={1,2,3,4,5,5}
print(s)
s.add(6)  # add new element
print(s)
s.update([7,8])  #add set of elements

print(s)
s.discard(7)  #delete an item
print(s)
s.remove(8)     #delete an  item
print(s)
s.clear()       #clear all the item in a set
print(s)
```

Output:

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 4, 5, 6, 8}
{1, 2, 3, 4, 5, 6}
set()
```

2.2.3 Strings

- String is a sequence of characters.
- Python Strings are immutable, it means once we declare a string we can't modify it.
- Python provides a built-in class "str" for handling text as the text is the most common form of data that a Python program handles.
- Example : s='hello' or s="hello"

Sequence operations on string

1.length

- len(str) is used to find the length of the string.
- Example:

```
s='hello'
print(len(s))
```

O/P : 5

2.select

- s[index_no] is used to select the charter of the string based in index number.
- Example:

```
s='hello'
print(s[1])
O/P : e
```

3.slice

- s[start:end] returns the substring starting with index start, *up to but not including* index end.
- Example:

```
s='hello'
print(s[0:3))
O/P : hel
```

4.count

- s.count(char) is used to count the number of character on a string.
- Example:

```
s='hello'
print(s.count('l'))
O/P : 2
```

5.Index

- s.index(char) returns the index of the first occurrence of character in s.
- Example:

```
s='hello'
print(s.index('o'))
O/P : 4
```

6.Minimum and maximum value

- min(s) and max(s) as applied to strings return the smallest (largest) character based on the underlying Unicode encoding.
- Example:

```
s='hello'
print(min(s))
print(max(s))
O/P : e  o
```

7.Concatenation

- s1+s2 is used to concatenate two strings like s1 and s2.
- Example:

```
s='hello'
s1='world'
print(s+s1)
O/P : helloworld
```

8.Membership

- 'char' in s - is used to find whether the particular characters in a string or not.
- Example:

```
s='hello'
print('ho' in s)
print('he' in s)
O/P : False          True
```

String Methods

1.isalpha()

- str.isalpha() – Returns true if str contains only letters.
- Example:

```
s='hello'
print(s.isalpha())
O/P : True
```

2.isdigit()

- str.isdigit() – Returns true if str contains only digits.

```
s='123'
print(s.isdigit())
O/P : True
```

3.islower()

- str.islower() – Returns true if str contains only lower case letters.
- Example:

```
s='hello'
print(s.islower())
O/P : True
```

4.isupper()

- str.isupper() – Returns true if str contains only upper case letters.
- Example:

```
s='hello'
print(s.isupper())
O/P : False
```

5. lower()

- str.lower() – Returns lower case version of str.

- Example:

```
s='HAI'
print(s. lower())
O/P : hai
```

6. upper()

- str. upper() – Returns upper case version of str.
- Example:

```
s='hai'
print(s.upper())
O/P : HAI
```

7. find()

- str. find(w) – Returns the index of the first occurrence of w in str.
- Example:

```
s='hai'
print(s.find('a'))
O/P : 1
```

8. replace()

- str. replace(w,t) – All occurrence of w replace with t.
- Example:

```
s='hai'
print(s.replace('h','c'))
O/P : cai
```

9. strip()

- str. strip(w) – All leading and trailing characters that appear in w removed.
- Example:

```
s='    hai!'
print(s.strip(' !'))
O/P : hai
```

Example:

```
s='hello'
s1='WORLD'
s2='123a'

print(s.replace('l','c'))
print(s.find('e'))
print(s.upper())
print(s1.lower())
print(s.islower())
print(s.isupper())
print(s.isalpha())
print(s2.isdigit())
```

```
print('ho' in s)
print(s+s1)
print(min(s))
print(max(s))
print(s.index('o'))
print(s.count('l'))
print(s[0:3])
print(s[0])
print(len(s))
```

Output:

```
hecco
1
HELLO
world
True
False
True
False
False
helloWORLD
e
o
4
2
hel
h
5
```

String formatting

- String formatting can be done in three ways:

    1. Using f-strings
    2. By format() method
    3. Using % operator

- **f-string:** Letter "f" is placed before the beginning of the string, and the variables mentioned in curly braces will refer to the variables declared above. For example {name}.
- **format() method:** format() method is called on a string object. Inside the string we use curly braces {} that will refer to the format() method arguments. Number of {} should match number of arguments inside format()

- **% operator:** "%" operator will be replaced by variables defined in parenthesis/in tuple. %s means a string variable will come to this place, %d is an integer, %f is a floating-point value.

- **Example:**

```
# string formatting using f strings
s=input("enter the name : ")
print(f"Hai {s} !")

# String formatting using format() method
s1 = "{} {} {}".format("how", 'are', 'you?')
print(s1)

# String formatting using % operator
item = int(input("Enter number of items"))
print("%s is carrying %d items"%(s, item))
```

- **Output:**

```
enter the name : cathy
Hai cathy !
how are you?
Enter number of items5
cathy is carrying 5 items
```

Difference between list, tuple, dictionary and set

List	**Tuple**	**Dictionary**	**Set**
It can be represented by [ ]	It can be represented by ( )	It can be represented by { }	It can be represented by { }
Example: [1, 2, 3, 4, 5]	Example: (1, 2, 3, 4, 5)	Example: {'one':1, 'two':2, 'three':3}	Example: {1, 2, 3, 4, 5}
It is mutable	It is immutable	It is mutable	It is mutable.
It is ordered	It is ordered	It is unordered	It is unordered
List allows duplicate elements	Tuple allows duplicate elements	It allow duplicate values but keys are not duplicated	Set will not allow duplicate elements
Items in list can be replaced or changed	Items in tuple cannot be changed or repla ced	Items in dictionary can be replaced or changed	Items in set can be changed or replaced
Creating an empty list l=[]	Creating an empty Tuple t=()	Creating an empty dictionary d={}	Creating a empty set a=set()