Data Structures

CONTENTS

Lists

In Python programming, a list is created by placing all the items (elements) inside a square bracket [ ], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

A list is a collection of items in a particular order. You can make a list that includes the letters of the alphabet, the digits from 0–9, or the names of all the people in your family. You can put anything you want into a list the items in your list don’t have to be related in any particular way. Because a list usually contains more than one element, it’s a good idea to make the name of your list plural, such as letters, digits, or names. In Python, square brackets ([]) indicate a list, and individual elements in the list are separated by commas.

Creating and accessing Lists

Example:

```
family = ['liz', 'emma', 'mom', 'dad']
height = [1.73, 1.68, 1.71, 1.89 ]

print ("family[0]: ", family[0], end="\n\n")
print ("height: ", height[1:3])
```

```
family[0]:  liz

height:  [1.68, 1.71]
```

Iterating over a list

Example:

```
for i in family:
    print(i)
```

```
liz
emma
mom
dad
```

List of Lists

Example:

```
fam = [['liz', 1.73], ['emma', 1.68], ['mom', 1.71], ['dad', 1.89]]
print(fam)
```

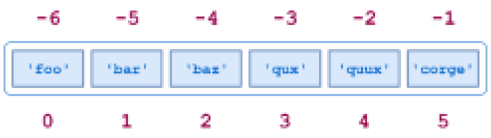
```
[['liz', 1.73], ['emma', 1.68], ['mom', 1.71], ['dad', 1.89]]
```

Example:

```
for i in fam :
    for j in i :
        print(j,end=" ")
    else:
        print()
```

```
liz 1.73
emma 1.68
mom 1.71
dad 1.89
```

Indexing and slicing



Example:

```
fam = ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

print("Print 1-3 index elements --> nos[1:4] : ", fam[1:4], end="\n\n")
print("Print all after index 2 --> nos[2:] :", fam[2:], end="\n\n")
print("Print all before 4 --> nos[:4] :", fam[:4], end="\n\n")
```

```
Print 1-3 index elements --> nos[1:4] :  [1.73, 'emma', 1.68]

Print all after index 2 --> nos[2:] : ['emma', 1.68, 'mom', 1.71, 'dad', 1.89 ]

Print all before 4 --> nos[:4] : ['liz', 1.73, 'emma', 1.68]
```

reverse indexing

Example:

```
fam = ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

print("Print last no in the list --> nos[-1] : ", fam[-1], end="\n\n")
print("Print last 4 elements in the list --> nos[-4:] :", fam[-4:], end="\n\n")
```

```
Print last no in the list --> nos[-1] :  1.89

Print last 4 elements in the list --> nos[-4:] : ['mom', 1.71, 'dad', 1.89]
```

Updating Lists

Example:

```
fam = ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
print ("Printing the original list : ",fam,end="\n\n")

fam[5] = 1.70
print ("Printing the updated list : ",fam)
```

```
Printing the original list : ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'da
d', 1.89]

Printing the updated list : ['liz', 1.73, 'emma', 1.68, 'mom', 1.7, 'dad',
1.89]
```

The [:] syntax works for lists. However, there is an important difference between how this operation works with a list and how it works with a string.

If s is a string, s[:] returns a reference to the same object:

```
>>> s = 'foobar'
>>> s[:]
'foobar'
>>> s[:] is s
True
```

Conversely, if a is a list, a[:] returns a new object that is a copy of a:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:]
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
>>> a[:] is a
False
```

Example:

```
fam = ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]

print(fam[2], type(fam[2])) # returns a single value

print(fam[:2], type(fam[:2])) # always returns a list
```

```
emma <class 'str'>
['liz', 1.73] <class 'list'>
```

Delete List Elements

Example:

```
fam = ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
print ("Printing the original list : ",fam,end="\n\n")

del fam[2]
del fam[:2]
del fam
print ("After deleting value at index 1 : ", fam)
```

```
Printing the original list : ['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'da
d', 1.89]

After deleting value at index 1 :  ['emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

Built-in Python functions

Example:

```
mylist = [10, 20, 30, 15]
print(len(mylist)," -- Gives the total length of the list.")

 # Applicable to numeric datatypes only:
print(max(mylist)," -- Returns item from the list with max value.")
print(min(mylist)," -- Returns item from the list with min value.")
print(sum(mylist)," -- Returns the summation of all numerical elements in the list")
```

```
4  -- Gives the total length of the list.
30  -- Returns item from the list with max value.
10  -- Returns item from the list with min value.
75  -- Returns the summation of all nunmerical elements in the list
```

List Methods

Example:

```
mylist = [40, 10, 20, 30, 15, 10]

print ("Printing the original list : ",mylist,end="\n\n")

# Returns count of how many times obj occurs in list
print("count of object 10 the list : ", mylist.count(10))

 # Returns the lowest index in list for that obj appears
print("Lowest index of object 10 : ", mylist.index(10))
```

```
Printing the original list :  [40, 10, 20, 30, 15, 10]
count of object 10 the list :  2
Lowest index of object 10 :  1
```

Append object to list

Example:

```
mylist = [40, 10, 20, 30, 15, 10]
print("Original list : ", mylist, end="\n\n")

mylist.append(70)
mylist.append("Hello")
mylist.append([1,2,3,4])
mylist[-1].append("abc")

print("List after appending 70 : ", mylist)
```

```
Original list :  [40, 10, 20, 30, 15, 10]

List after appending 70 : [40, 10, 20, 30, 15, 10, 70, 'Hello', [1, 2, 3,
4, 'abc']]
```

Extending Lists

Example:

```
fam_ext = fam + ["me", 1.79]
print(fam_ext)
```

```
['liz', 1.73, 'emma', 1.7, 'mom', 1.71, 'dad', 1.89, 'me', 1.79]
```

Append contents of sequence to list

Example:

```
mylist = [40, 10, 20]
print("Original list : ", mylist, end="\n\n")

# mylist.extend(70)
mylist.extend([1,2,5])
mylist.extend("Hello")
mylist.extend(range(50,100,10))
print( mylist)
```

```
Original list :  [40, 10, 20]

[40, 10, 20, 1, 2, 5, 'H', 'e', 'l', 'l', 'o', 50, 60, 70, 80, 90]
```

Inserts object into list at offset index

Example:

```
mylist = [40, 10, 20, 30, 15, 10]
print("Original list : ", mylist, end="\n\n")

# mylist[4]=85 # this will replace element at index 4

# insert() will add element at index 4
mylist.insert(4, 85)
print("List after inserting 85 at index 4 : ", mylist)
```

```
Original list :  [40, 10, 20, 30, 15, 10]

List after inserting 85 at index 4 :  [40, 10, 20, 30, 85, 15, 10]
```

Removing element from lists

Example:

```
mylist = [40, 10, 20, 30, 15, 10]
print("Original list : ", mylist, end="\n\n")

x = mylist.pop() # Removes and returns last object
print("List after pop : ", mylist, end="\n\n")

y = mylist.pop(2) # Removes the object at given index from list
print("List after pop value at index 2 : ", mylist, end="\n\n")

mylist.remove(30) # Removes object obj from list
print("List after removing 20 : ", mylist)
```

```
Original list :  [40, 10, 20, 30, 15, 10]

List after pop :  [40, 10, 20, 30, 15]

List after pop value at index 2 :  [40, 10, 30, 15]

List after removing 20 :  [40, 10, 15]
```

Sorting and Reversing

Example:

```
mylist = [40, 10, 20, 30, 15, 10]
print("Original list : ", mylist, end="\n\n")

mylist.reverse() # Reverses objects of list in place
print("List after reversing : ", mylist, end="\n\n")

mylist.sort(reverse=True) # Sorts objects of list, use compare func if given
print("List after sort : ", mylist, end="\n\n")
```

```
Original list :  [40, 10, 20, 30, 15, 10]

List after reversing :  [10, 15, 30, 20, 10, 40]

List after sort :  [40, 30, 20, 15, 10, 10]
```

Sorting a List Temporarily with the sorted() Function

To maintain the original order of a list but present it in a sorted order, you can use the sorted() function. The sorted() function lets you display your list in a particular order but doesn’t affect the actual order of the list.

Example:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

print("Original list : ", cars,"\nSorted list : ",sorted(cars))

print("\nOriginal list : ", cars,"\nReversed list : ",list(reversed(cars)))
```

```
Original list :  ['bmw', 'audi', 'toyota', 'subaru']
Sorted list :  ['audi', 'bmw', 'subaru', 'toyota']

Original list :  ['bmw', 'audi', 'toyota', 'subaru']
Reversed list :  ['subaru', 'toyota', 'audi', 'bmw']
```

List works as object reference

Example:

```
list1 = [1, 2, 3]
list2 = list1
print("List1 = ", list1," List2 = ", list2, end="\n\n")

list1[0] = 5

print("List1 = ", list1," List2 = ", list2, end="\n\n")
```

```
List1 =  [1, 2, 3]  List2 =  [1, 2, 3]

List1 =  [5, 2, 3]  List2 =  [5, 2, 3]
```

To make to separate copy lists (Create a shadow copy)

Example:

```
list1 = [1, 2, 3]
list2 = list1[:]
print("List1 = ", list1, "List2 = ", list2, end="\n\n")
list1[0] = 5
print("List1 = ", list1, "List2 = ", list2, end="\n\n")
```

```
List1 =  [1, 2, 3] List2 =  [1, 2, 3]

List1 =  [5, 2, 3] List2 =  [1, 2, 3]
```

Tuples

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.

Creating a tuple

Example:

```
tup1 = ('physics', 'chemistry', [1997, 89], 2000)
tup2 = (1, 2, 3, 4, 5)
tup3 = "a", "b", "c", "d"

# The empty tuple is written as two parentheses containing nothing
- tup1 = ()
tup1 = tuple()

# A tuple containing a single value: include a comma, even though there is only one value
- tup1 = (50,)
```

Accessing Values in Tuples (Indexing and slicing)

Example:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7)

print("tup1[0]: ", tup1[0])
print("tup2[1:5]: ", tup2[1:5])
```

```
tup1[0]:  physics
tup2[1:5]:  (2, 3, 4, 5)
```

Updating Tuples

Example:

```
tup = ('physics', 'chemistry', 1997, 2000)
tup [0] = "Biology"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-69e604f2bb0a> in <module>()
      1 tup = ('physics', 'chemistry', 1997, 2000)
----> 2 tup [0] = "Biology"
TypeError: 'tuple' object does not support item assignment
```

Example:

```
tup = ('physics', 'chemistry', [1997, 89], 2000)
4 tup[2] = [1997, 1998] # error

4 In this case the list inside the tuple is getting updated
tup[2][1] = 1998
tup[2].append(1995)
print(tup)
```

```
('physics', 'chemistry', [1997, 1998, 1995], 2000)
```

Delete Tuple Elements

Example:

```
tup = ('physics', 'chemistry', 1997, 2000);

print("Print Tuple : ", tup)
del tup;
print("After deleting tup : ", tup)
```

```
Print Tuple :  ('physics', 'chemistry', 1997, 2000)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-ca940e5e630c> in <module>()
      3 print("Print Tuple : ", tup)
      4 del tup;
----> 5 print("After deleting tup : ", tup)

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

Built-in Python functions

Example:

```
tup = (10, 20, 30, 15)

print(len(tup)," -- Gives the total length of the tuple.")

# Applicable only to numeric datatype.
print(max(tup)," -- Returns item from the tuple with max value.")
print(min(tup)," -- Returns item from the tuple with min value.")
print(sum(tup)," -- Returns the summation of all nunmerical elements in the tuple")
```

```
31 -- Gives the total length of the tuple.
11  -- Returns item from the tuple with max value.
76  -- Returns item from the tuple with min value.
#   -- Returns the summation of all nunmerical elements in the tuple
```

Example:

```
tup = (10, 20, 30, 15)
tup.count(10)
```

```
1
```

Example:

```
tup.index(30)
```

```
2
```

Why use a tuple instead of a list?

- ❑ Program execution is faster when manipulating a tuple than it is for the equivalent list. (This is probably not going to be noticeable when the list or tuple is small.)
- ❑ If the values in the collection are meant to remain constant for the life of the program, using a tuple instead of a list guards against accidental modification.

Sets

Python includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set().

Creating a set

Example:

```
s = set() # empty set
type(s)
```

Out[1]:

```
set
```

Updating sets

Example:

```
a= {1,2,3,4, 'a', (1,2,3)} # Sets can contain only immutable
objects print (a)
```

```
{1, 2, 3, 4, (1, 2, 3), 'a'}
```

Example: Adding an object to set

```
a.add(5)
print (a)
```

```
{1, 2, 3, 4, 5, (1, 2, 3), 'a'}
```

Example: Adding a collection of objects to set

```
a.update([6,7])
print (a)
```

```
{1, 2, 3, 4, 5, 6, 7, (1, 2, 3), 'a'}
```

Example: Removing an object from set

```
a.pop()
print (a)
```

```
{2, 3, 4, 5, 6, 7, (1, 2, 3), 'a'}
```

Example: Removing desired object from the set

```
a.remove(4)
print (a)
```

```
{2, 3, 5, 6, 7, (1, 2, 3), 'a'}
```

Operations on sets

Example:

```
a = [1, 1, 2, 3, 5, 8, 13]
b = [1, 2, 3, 4, 5, 6, 7]

# Converting a list into set (this will also remove the duplicates inside the
list) set_a = set(a)
set_b = set(b)
```

Example:

```
# Membership Operator :
1 in set_a
```

```
True
```

Example:

```
# set intersection :
print(set_a & set_b)

# OR

print (set_a.intersection(set_b))
```

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

Example:

```
# Set Union
print(set_a | set_b)

# OR

print (set_a.union(set_b))
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 13}
{1, 2, 3, 4, 5, 6, 7, 8, 13}
```

Example:

```
# symmetric difference or uncommon elements
print(set_a ^ set_b)

# OR

print (set_a.symmetric_difference(set_b))
```

```
{4, 6, 7, 8, 13}
{4, 6, 7, 8, 13}
```

Example:

```
# set difference
print(set_a - set_b)

# OR

print (set_a.difference(set_b))
```

```
{8, 13}
{8, 13}
```

Some more Set functions

Example:

```
a= {1,2,3,4}
print (a)

b=set([1,4])
print (b)

print (a.issubset(b))
print (b.issubset(a))
print (a.issuperset(b))
print (b.issuperset(a))
```

```
{1, 2, 3, 4}
{1, 4}
False
True
True
False
```

Example:

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
for f in sorted(set(basket)):
    print(f)
```

```
apple
banana
orange
pear
```

Immutable Sets

Frozensets can be created using the function frozenset(). This datatype supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have method that add or remove elements.

Example:

```
x = frozenset([1, 2, 3, 4, 5])
x
```

Out[2]:

```
frozenset({1, 2, 3, 4, 5})
```

Dictionaries

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds **key:value** pair. Key value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon **:**, whereas each key is separated by a 'comma'.

A Dictionary in Python works similar to the Dictionary in a real world. Keys of a Dictionary must be unique and of *immutable* data type such as Strings, Integers and tuples, but the key-values can be repeated and be of any type.
Note – Keys in a dictionary doesn't allows Polymorphism.

In Python, a Dictionary can be created by placing sequence of elements within curly **{}** braces, separated by 'comma'. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be *immutable*.
Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

Note – Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.

Creating a dictionary and accessing values

Example:

```
d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'marks': [70,40,80,89,95]}

print(d['Name'], "is", d['Age'], "years old and lives in", d['City'], end = "\n\n")

print(d)
```

```
Jia is 25 years old and lives in Mumbai

{'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'marks': [70, 40, 80, 89, 95]}
```

Updating values

Example:

```
d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai'}

d['Age'] = 8;  # update existing entry
print(d, end = "\n\n")

d['School'] = "DPS School"  # Add new entry
print(d)
```

```
{'Name': 'Jia', 'Age': 8, 'City': 'Mumbai'}

{'Name': 'Jia', 'Age': 8, 'City': 'Mumbai', 'School': 'DPS School'}
```

Deleting values

Example:

```
d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai'}
print("Original dictionary : ", d, end = "\n\n")

del d['Name']  # remove entry with key 'Name'
print("Printing dictionary after deleting 'name'", d, end = "\n\n")

d.clear()  # remove all entries in dict
print("Printing dictionary after clear()", d, end = "\n\n")

d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai'}
del d  # delete entire dictionary
print("Printing dictionary after del command :",d)
```

```
Original dictionary :  {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai'}

Printing dictionary after deleting 'name' {'Age': 25, 'City': 'Mumbai'}

Printing dictionary after clear() {}

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-f81aa520945f> in <module>()
     10 d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai'}
     11 del d  # delete entire dictionary
---> 12 print("Printing dictionary after del command :",d)

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

Built-in Dictionary functions

Example:

```
d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'School': 'DPS School', 'marks': [70,40,80

# Returns a list of dict's (key, value) tuple pairs
print(list(d.items()),"--> Key - value pairs in the dictionary", end = "\n\n")

# Returns list of dictionary dict's keys
print(list(d.keys()),"--> list of keys in dictionary", end = "\n\n")

# Returns list of dictionary dict's values
print(list(d.values()),"--> list of all values in the dictionary")
```

```
[('Name', 'Jia'), ('Age', 25), ('City', 'Mumbai'), ('School', 'DPS School'),
('marks', [70, 40, 80, 89, 95]), (10, 'abc')] --> Key - value pairs in the
dictionary

['Name', 'Age', 'City', 'School', 'marks', 10] --> list of keys in dictionary
```

```
['Jia', 25, 'Mumbai', 'DPS School', [70, 40, 80, 89, 95], 'abc'] --> list of
all values in the dictionary
```

Iterating over dictionary elements:

Example:

```
for i,j in d.items():
    print(i,j)
```

```
Name Jia
Age 25
City Mumbai
School DPS School
marks [70, 40, 80, 89, 95]
10 abc
```

Example:

```
d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'School': 'DPS School'}

# For key key, returns value or default if key not in dictionary
print(d.get('Name1', 'Invalid Key'), end = "\n\n")

# sets dict[key]=default if key is not already in dict
print( d.setdefault('Name1','abc'), end = "\n\n")

print("Printing d : ",d)
```

```
Invalid Key

abc

Printing d :  {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'School': 'DPS Sc
hool', 'Name1': 'abc'}
```

Example:

```
d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'School': 'DPS School'}

d2 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print("Original d : ", d, end = "\n\n")

d.update(d2) # Adds dictionary d2's key-values pairs to d
print("Updated d : ", d, end = "\n\n")
```

```
Original d : {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'School': 'DPS Sc
hool'}

Updated d :  {'Name': 'Zara', 'Age': 7, 'City': 'Mumbai', 'School': 'DPS Sch
ool', 'Class': 'First'}
```

Sorted Dictionary

In [1]:

```
d = {'Name': 'Jia', 'Age': 25, 'City': 'Mumbai', 'School': 'DPS School'}

print(sorted(d))
```

```
['Age', 'City', 'Name', 'School']
```