Date June 9, 2021

WEEK 6

Python Collections

LISTS

Notation : [ ]

Creation : list ( )
or l = [ ]

Mutability : Mutable (It means we can change the order of items in a list or reassign an item in a list)

Types of Elements which can be stored : Any

Order of Elements : Ordered

Duplicate Elements : Allowed

List Operations : + , *

\+ → appends lists
* → replicates list

EXAMPLE :

```
l1 = [1,2,3]
l2 = [4,5]
l3 = l1 + l2
l4 = l1 * 3
print (l3)
print (l4)
```

OUTPUT : [1,2,3,4,5]
[4,5,4,5,4,5]

Sorting : Possible

List Methods :

append()	count()	insert()	reverse()
clear()	extend()	pop()	sort()
copy()	index()	remove()	

Let us understand the list methods by writing a code :

```
l = [1, 2, 3]
l.append(4) # adds '4' at last
print(l)

x = [4,5]
l.extend(x)
print(l)

y = [8, 4, 9, 6]
y.sort()
print(y)

y.pop()    # deletes last element
print(y)

y.pop(0)   # deletes element at index 0
print(y)

l.remove(2) # remove '2' from list
print(l)
```

OUTPUT

```
[1, 2, 3, 4]
[1, 2, 3, 4, 4, 5]
[4, 6, 8, 9]
[4, 6, 8]
[6, 8]
[1, 3, 4, 4, 5]
[4, 1, 3, 4, 4, 5]
3
[5, 4, 4, 3, 1, 4]
3
[6, 8]
[]
```

```
l.insert(4,0)  # inserts '4' at index 0
print(l)

#l.count(4)
z = l.count(4)  # counts '4' item in list
print(z)

l.reverse()
print(l)

n = l.index(3)  # gives index of element '3'
print(n)

l1 = y.copy()  # copies list y to l1
print(l1)

l.clear()  # clears the list
print.(l)
```

Aliasing of List

If 'a' refers to a list and you assign b = a, then both variables refer to the same list.

CODE 1:
```
a = [1, 2, 3]
b = a
print(b is a)
```

OUTPUT: True

CODE 2:
```
a = [1, 2, 3]
c = a.copy()
print(c is a)
```

OUTPUT: False

→ The association of a variable with an object is called a reference.
In code1, there are two references to the same object (list)

→ An object (list) with more than one reference has more than one name, so we say that the object is aliased.

→ Since, we know list is mutable, changes made with one alias affects the other:

CODE:
```
a = [1, 2, 3, 4]
b = a
b[0] = 17
print (a)
```

OUTPUT:
```
[17, 2, 3, 4]
```

→ Although this behaviour can be useful, it is error-prone.
In general, it is safer to avoid aliasing when you are working with mutable objects.

→ So, for lists we have to copy the list. We can do it in following ways:

```
l = [1, 2, 3]
l1 = l.copy()      # 1st way
l2 = list(l)       # 2nd way
l3 = l[:]          # 3rd way
```

→ Now, if we make any change in l1, l2, l3, it won't

affect the list 'l'.

Comparing Two Lists

```
l1 = [1, 2, 3]
l2 = [1, 2, 3]
l3 = [1, 3, 2]
l4 = [4]
print ( l1 == l2)      # True
print ( l2 == l3)      # False
print ( l2 < l3)       # True
print ( l4 > l1 )      # True

print ( [1,2] < [2,1] )     # True
print ( [1] < [1,2,3] )     # True
print ( [2,3] < [3] )       # True
print ( [] < [1])           # True
```

Passing lists as parameter in Functions

```
def add (x) :
    x.append (1)
    return x

x = [5]
print ( add(x))
print (x)
```

OUTPUT

```
[5, 1]
[5, 1]
```

→ This is known as call by reference (unlike what we know in int or strings (call by value)).

TUPLES

Notation : ( )

Creation : tuple( ) or t = (10,)

Mutability : Immutable

Types of Elements which can be stored : Any

Order of Elements : Ordered

Duplicate Elements : Allowed

Operations : Indexing, Slicing, Iteration

Sorting : Not possible

Tuple Methods : count( ), index( )

More About Tuples

→ A tuple is a sequence of values much like a list.

→ The values stored in a tuple can be of any type, and they are indexed by integers.

→ The important difference is that tupples are immutable.

→ Tuples are comparable and hashable.

(hashable means its value does not change during its lifetime)

→ Syntactically, a tuple is a comma separated list of values:

```
t = 1, 2, 3, 4
```

→ Although it is not necessary, but it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at code:

```
t = (1, 2, 3, 4)
```

→ To create a tuple with a single element, you have to include the final comma:

```
t1 = ('a',)
t2 = ('a')
print(type(t1), type(t2))
```

Output: <class 'tuple'> <class 'str'>

→ If the argument is a sequence (string, list, tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
t = tuple('hello')
print(t)
```

OUTPUT: ('h', 'e', 'l', 'l', 'o')

→ **Indexing** :

```
t = ('a', 'b', 'c')
print(t[0])
```

Output: a

→ **Slicing** :

```
t = (1, 2, 3, 4)
print(t[1:3])
```

Output: (2, 3)

→ We cannot modify the elements of tuple.

CODE :

```
t[0] = 'A'
```

OUTPUT: TypeError : object doesn't support item assignment

→ **CHANGING A TUPLE**

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable datatype like a list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

CODE :

```
#changing tuple values
my_tuple = (4, 2, 3, [6, 5])

#TypeError: 'tuple' object does not support item assignment
# my_tuple[1] = 9
```

```
#However, item of mutable element can be changed
my_tuple[3][0] = 9    # output = (4,2,3,[9,5])
print('my_tuple')

#Tuples can be reassigned
my_tuple = ('g','a','g','n','e','e','t')
print('my_tuple')
```

OUTPUT : (4, 2, 3, [9, 5])
('g','a','g','n','e','e','t')

→ We can use + operator to combine two tuples. This is called concatenation.
→ We can also repeat the elements in a tuple for a given no. of times using the * operator

```
CODE: print((1,2,3) + (4,5,6))
      print((1,2) * 3)
```

OUTPUT : (1, 2, 3, 4, 5, 6)
(1,2,1,2,1,2)

→ TUPLE METHODS

```
CODE: t1 = ('a','p','p','l','e')
      print(t1.count('p'))    # output = 2
      print(t1.index('l'))    # output = 3
```

Advantages Of Tuple Over List

Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing tuple over a list.

(1) We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.

(2) Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.

(3) Tuples that contain immutable elements can be used as a key for a dictionary. With lists, it is not possible.

(4) If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Date June 10, 2021

classmate Date Page

SETS

Notation : { }

Creation : set()

Mutability : Mutable

Type of Elements which can be stored : Hashable

Order of Elements : Unordered

Duplicate Elements : Not allowed

Operations : Add, Delete

Sorting : Not possible

More About Sets

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (can't be changed).

However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symetric difference etc.

Creating Python Sets

→ A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using built-in set() function.

→ It can have any no. of items and they may be of different types (integer, float, tuple, string etc). But a set cannot have mutable elements like lists, sets or dictionaries as elements.

CODE1:

```
#Different types of set in Python
#set of integers

my_set = {1, 2, 3}
print (my_set)

#set of mixed datatypes

my_set = {1.0, "Hello", (1,2,3)}
print (my_set)
```

OUTPUT:

```
{1, 2, 3}
{1.0, "Hello", (1,2,3)}
```

CODE 2:

```
# set cannot have duplicates
my_set = {1, 2, 3, 4, 3, 2}
print (my_set)
```

OUTPUT:

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

CODE 3:
```
# We can make set from a list
my_set = set([1,2,3,2])
print(my_set)
```

OUTPUT: {1,2,3}

CODE 4:
```
# set cannot have mutable items
# here [3,4] is a mutable list
# this will cause an error

my_set = {1,2,[3,4]}
```

OUTPUT: TypeError: unhashable type: 'list'

→ Creating an empty set is a bit tricky.

→ Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

CODE 5:
```
# Distinguish set & dictionary while creating empty set

# initialize a with {}
a = {}

# check data type of a
print(type(a))

# initialize a with set()
```

```
a = set()

# Check datatype of a
print(type(a))
```

OUTPUT :
```
<class 'dict'>
<class 'set'>
```

Modifying a Set in Python

→ Sets are mutable. However, since they are unordered, indexing has no meaning.

→ We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

→ We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

CODE :
```
my_set = {1, 3}
print(my_set)

my_set.add(2)
print(my_set)

my_set.update([2, 3, 4])
```

```
print (my_set)

my_set.update ([4,5], {1, 6, 8})
print (my_set)
```

OUTPUT:

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

Removing Elements from A set

A particular item can be removed from a set using the methods discard() and remove().

The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set)

CODE:

```
#Difference bw discard() & remove()

mySet = {1, 3, 4, 5, 6}
mySet.discard(4)
print (mySet)

mySet.remove(6)
print (mySet)
```

```
# discard an element not present in mySet
mySet.discard(2)
print(mySet)

# remove an element not present in myset
# you will get an error → KeyError
mySet.remove(2)
```

OUTPUT:

```
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
Traceback (most recent call last):
  File "<string>", line , in <module>
KeyError: 2
```

→ Similarly, we can remove and return an item using the pop() method.

→ Since set is an unordered datatype, there is no way of determining which item will be popped. It is completely arbitrary.

→ We can also remove all items from a set using the clear() method.

Python Set Operations

(1) Set Union (|)

Union of A & B is a set of all elements from both sets.

Operator : | or union()

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

print (A|B)

C = {8, 9}
D = B.union(C)
print(D)
```

output : {1, 2, 3, 4, 6}
{1, 4, 6, 8, 9}

(2) Set Intersection (&)

It returns the element that is common in both the sets.

Operator : & method : intersection()

CODE 2:
```
A = {1, 2, 3}
B = {1, 4, 6}
C = {8, 9}

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

output :

{1}
{}

(3) Set Difference (-)

Difference of the set B from set A (A - B) is the set of elements that are only in A but not in B.

Similarly, (B-A) is a set of elements in B but not in A.

Operator: - method: difference()

CODES:
```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

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

output:
```
{1, 2, 3}
{6, 7, 8}
```

Other Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member (Do nothing if the element is not in set)

Method	Description
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and other.
isdisjoint()	Returns True if two sets have null intersection
issubset()	Returns True if other set contains this set.
issuperset()	Returns True if this set contains another set
symmetric_difference()	Returns the symmetric diff. of two sets as new set
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself & others

Date
June 11, 2021

classmate
Date
Page

DICTIONARY

Notation : {'Key' : 'Value'}

Creation : d = {} or dict()

Mutability : Mutable

Type of Elements which can be stored : Keys : Hashable
Values : Any

Order of Elements : Unordered

Duplicate Elements : Keys → not allowed
Values → allowed

Operations : Keys → Add, Delete
Values → Add, Update, Delete

Sorting : Possible

More About Dictionaries

Python dictionary is an unordered collection of items. Each item of a dictionary has a key:value pair.

Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.

An item has a key and a corresponding value that is expressed as a pair (key: value)

While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

CODE:

```
d = {}                          #empty dictionary
d = {1: 'apple', 2: 'ball'}     # dictionary with integer keys
d = {'name': 'John', 1: [2,3]}  # dictionary with mixed keys
d = dict({1: 'A', 2: 'B'})      # using dict()
d = dict([(1,'A'), (2,'B')])    # from sequence having each
                                # item as a pair
```

Accessing Elements From Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with get() method.

If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

CODE:

```
# get vs [] for retreiving elements
d = {'name': 'Jack', 'age': 26}

print(d['name'])          # output → Jack
print(d.get('age'))       # output → 26

#Trying to access keys which doesn't exist

print(d.get('address'))   #output → None
print(d['address'])       # output → KeyError
```

OUTPUT:

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 11, in <module>
    print(d['address'])
KeyError: 'address'
```

Changing & Adding Dictionary Elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

CODE:

```
#Changing and adding Dictionary elements
d = {'name': 'John', 'age': 26}

#update value
d['age'] = 27

# add item
d['address'] = 'Downtown'

print(d)
```

OUTPUT: {'name': 'John', 'age': 27, 'address': 'Downtown'}

Removing Elements from Dictionary

We can remove a particular item in a dictionary by using the pop() method. This method removed an item with the provided key and returns the value.

The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. All the items can be removed at once, using the clear() method

We can also use the del keyword to remove individual items or entire dictionary itself.

CODE :

```
#Removing Elements from a dictionary

squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# remove a particular item, Returns its value
print(squares.pop(4))

print(squares)

# remove an arbitrary item, return (key, value)
print(squares.popitem())

print(squares)

# removes all items
squares.clear()
print(squares)

# delete the dictionary itself
del squares

print(squares)                    # Error
```

OUTPUT :

```
16
{1:1, 2:4, 3:9, 5:25}
(5,25)
{1:1, 2:4, 3:9}
{}
NameError: name 'squares' is not defined
```

Dictionary Methods

clear()	items()	setdefault()
copy()	keys()	update()
fromkeys()	pop()	values()
get()	popitem()	

Some of the methods we have seen earlier, let us see a few more by the following code:

CODE:
```
# Dictionary Methods

marks = {}.fromkeys(['Math', 'English', 'science'], 0)

print(marks)

for item in marks.items():
    print(item)

print(list(sorted(marks.keys())))
```

OUTPUT:
```
{'Math': 0, 'English': 0, 'Science': 0}
('Math', 0)
('English', 0)
('Science', 0)
['English', 'Math', 'Science']
```

NOTE: sorted() is a dictionary function which return a new sorted list of keys in dictionary.

Some Dictionary Operations

(1) Membership Test

We can test if a key is in a dictionary or not using the keyword in.
Membership test is only for the keys & not for the values.

CODE:
```
squares = {1:1, 2:4, 4:16}

print(1 in squares)
print(3 in squares)
print(16 in squares)
print(5 not in squares)
```

OUTPUT:
```
True
False
False
True
```

(2) Iterating through dictionary

We can iterate through each key in a dictionary using a for loop.

CODE:
```
squares = {1:1, 3:9, 5:25, 7:49}
for i in squares:
    print(squares[i])
```

OUTPUT:
```
1
9
25
49
```

Property	List	Tuple	Dictionary	Set
Notation	[ ]	( )	{'Key': 'Value'}	{ }
Creation	list()	tuple()	dict()	set()
Mutability	Mutable	Immutable	Mutable	Mutable
Type of elements which can be stored	Any	Any	Keys: Hashable Values: Any	Hashable
Order of elements	Ordered	Ordered	Unordered*	Unordered
Duplicate elements	Allowed	Allowed	Keys: Not allowed Values: Allowed	Not allowed
Operations	Add, Update, Delete	None	Keys: Add, Delete Values: Add, Update, Delete	Add, Delete
Operations	Indexing, Slicing, Iteration	Indexing, Slicing, Iteration	Iteration	Iteration
Sorting	Possible	Not possible	Possible	Not possible

*Python 3.6 and earlier. Dictionaries are ordered as per Python 3.7 and above.

List methods	Tuple methods	Dictionary methods	Set methods
append() clear() copy() count() extend() index() insert() pop() remove() reverse() sort()	count() index()	clear() copy() fromkeys() get() items() keys() pop() popitem() setdefault() update() values()	add() clear() copy() difference() difference_update() discard() intersection() intersection_update() isdisjoint() issubset() issuperset() pop() remove() symmetric_difference() symmetric_difference_update() union() update()