Introduction to Programming Using Python

Microsoft Python Certification | **Exam 98-381**

Lecture 2:

Data Structures

Lecturer: **James W. Jiang**, Ph.D. | Summer, 2018

Microsoft™

SAWY PRO

PROFESSIONAL EDUCATION TRAINING

Lists:

About Lists

How to create a list?

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.).

```
# empty list
my_list = []


# list of integers
my_list = [1, 2, 3]


# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

Elements in a List

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items.

```
number_list = [10, 20, 30, 40]
```

```
number_list [1]
```

```
In [38]: number_list = [10, 20, 30, 40]
    ...: >>> number_list [1]
    ...:
Out[38]: 20
```

Elements in a List

```
fruit_list = ['banana', 'pear', 'strawberry',
'strawberry ice cream']

fruit_list [3]
```

```
In [40]: fruit_list = ['banana', 'pear', 'strawberry', 'strawberry ice cream']
    ...: >>> fruit_list [3]
    ...:
Out[40]: 'strawberry ice cream'
```

Elements in a List

```
number_list = [10, 20, 30, 40]

fruit_list = ['banana', 'pear', 'strawberry',
'strawberry ice cream']

mixed_list = [10, 38.88, 'pineapple',['banana', 20]]


mixed_list[3]

['banana', 20]


mixed_list[2]

'pineapple'
```

Negative Index

```
guests = ['Christopher','Susan','Bill','Satya']

#print the last entry in the list
print(guests[-1])

#print the second last entry in the list
print(guests[-2])
```

Lists:
Basic List Operation

List Operations

Python Expression	Results	Description
`len([1, 2, 3])`	3	Length
`[1, 2, 3] + [4, 5, 6]`	[1, 2, 3, 4, 5, 6]	Concatenation
`['Hi!'] * 4`	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
`3 in [1, 2, 3]`	True	Membership
`for x in [1, 2, 3]: print x`	1 2 3	Iteration

List Operation

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
c
```

```
In [1]: a = [1, 2, 3]
   ...: b = [4, 5, 6]
   ...: c = a + b
   ...: c
   ...:
Out[1]: [1, 2, 3, 4, 5, 6]
```

List Operation

```
[0] * 4

[0, 0, 0, 0]

[1, 2, 3] * 3

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

index()

The index() function will search the list and return the index of the position where the value was found

```
guests = ['Christopher','Susan','Bill','Satya']

#this will return the index in the list
#where the name Bill is found
print(guests.index('Bill'))
aList = [123, 'xyz', 'zara', 'abc'];
print ("Index for xyz : ", aList.index( 'xyz' ))
print ("Index for zara : ", aList.index( 'zara' ))
```

```
In [4]: guests = ['Christopher','Susan','Bill','Satya']
   ...:
   ...: #this will return the index in the list
   ...: #where the name Bill is found
   ...: print(guests.index('Bill'))
   ...: aList = [123, 'xyz', 'zara', 'abc'];
   ...: print ("Index for xyz : ", aList.index( 'xyz' ))
   ...: print ("Index for zara : ", aList.index( 'zara' ))
   ...:
2
```

Count()

This method returns `count` of how many times obj occurs in list.

```
aList = [123, 'xyz', 'zara', 'abc', 123];
print ("Count for 123 : ", aList.count(123))
print ("Count for zara : ", aList.count('zara'))
```

```
In [5]: aList = [123, 'xyz', 'zara', 'abc', 123];
   ...: print ("Count for 123 : ", aList.count(123))
   ...: print ("Count for zara : ", aList.count('zara'))
   ...:
Count for 123 :  2
Count for zara :  1
```

min()

The method `min()` returns the elements from the list with minimum value.

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700,
200]
print ("min value element : ", min(list1))# error
print ("min value element : ", min(list2))
```

max()

The method `max` returns the elements from the list with maximum value.

```
list1, list2 = [123, 'xyz', 'zara', 'abc'], [456, 700,
200]
print ("Max value element : ", max(list1))
print ("Max value element : ", max(list2))
```

Lists:

List Slices

List Slices

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
t[1:3]
['b', 'c']
t[:4]
['a', 'b', 'c', 'd']
t[3:]
['d', 'e', 'f']
```

List Slices

```
t[:]
['a', 'b', 'c', 'd', 'e', 'f']


t = ['a', 'b', 'c', 'd', 'e', 'f']
t[1:3] = ['x', 'y']
t
['a', 'x', 'y', 'd', 'e', 'f']
```

Nested List

Also, a list can even have another list as an item. This is called nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

Nested List

```
# Nested List
n_list = ["Happy", [2,0,1,5]]

# Nested indexing

# Output: a
print(n_list[0][1])

# Output: 5
print(n_list[1][3])
```

Lists:
Update Lists

Lists Are Mutable

Unlike strings, lists are **mutable**. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned:

```
mixed_list = [10, 38.88, 'pineapple',['banana', 20]]
mixed_list [2] = 100
mixed_list
[10, 38.88, 100, ['banana', 20]]
mixed_list [1] = 'apple'
mixed_list
[10, 'apple', 100, ['banana', 20]]
```

Update a List

```
guests = ['Christopher','Susan','Bill','Satya']
print("first value is " + guests[0])

#change the first value in the list to Steve
guests[0] = 'Steve'
print("first value is now " + guests[0])
```

append()

```
guests = ['Christopher','Susan','Bill','Satya']

#add a new value to the end of the list
guests.append('Steve')

#display the last value in the list
print(guests[-1])
```

Lists:

Delete List Elements

Delete Elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

```
t = ['a', 'b', 'c']
x = t.pop(1)
t
['a', 'c']
x
'b'
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

Delete an Element

If you don't need the removed value, you can use the del operator:

```
t = ['a', 'b', 'c']
del t[1]
t
['a', 'c']
```

Delete an Element

If you know the element you want to remove (but not the index), you can use remove:

```
t = ['a', 'b', 'c']
t.remove('b')
t
['a', 'c']
```

The return value from remove is None.

Remove Elements

To remove more than one element, you can use del with a slice index:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
del t[1:5]
t
['a', 'f']
```

Remove Elements

```
guests = ['Christopher','Susan','Bill','Satya']

#add a new value to the end of the list
guests.remove('Christopher')

#display the last value in the list
print(guests[0])
```

Lists:
Sort Lists

sort()

```
guests = []
name = "  "
while name !="DONE" :
    name = input("Enter guest name (enter DONE if no mor
e names) : ")
    if name.upper() != "DONE" :
        guests.append(name)
guests.sort()
for guest in guests :
     print(guest)
```

```
In [53]: guests = []
    ...: name = "  "
    ...: while name !="DONE" :
    ...:
ame = input("Enter guest name (enter DONE if no more names) : ")
    ...:     if name.upper() != "DONE" :
    ...:         guests.append(name)
    ...: guests.sort()
    ...: for guest in guests :
    ...:      print(guest)
    ...:

Enter guest name (enter DONE if no more names) : James

Enter guest name (enter DONE if no more names) : Jason

Enter guest name (enter DONE if no more names) : Jerry

Enter guest name (enter DONE if no more names) : John

Enter guest name (enter DONE if no more names) : DONE
James
Jason
Jerry
John
```

sort()

This method does not return any value but it changes from the original list.

```
aList = ['xyz', 'zara', 'abc', 'xyz'];
aList.sort();
print ("List : ", aList)
```

```
In [13]: aList = ['xyz', 'zara', 'abc', 'xyz'];
    ...: aList.sort();
    ...: print ("List : ", aList)
    ...:
List :  ['abc', 'xyz', 'xyz', 'zara']
```

reverse()

This method does not return any value but reverse the given object from the list.

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.reverse();
print ("List : ", aList)
```

```
In [14]: aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
    ...: aList.reverse();
    ...: print ("List : ", aList)
    ...:
List :  ['xyz', 'abc', 'zara', 'xyz', 123]
```

Lists:

Add List Elements

extend

extend takes a list as an argument and appends all of the elements:

```
t1 = ['a', 'b', 'c']

t2 = ['d', 'e']

t1.extend(t2)

t1

['a', 'b', 'c', 'd', 'e’]
```

This example leaves t2 unmodified.

join

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
t = ['bananas', 'are', 'good', 'for', 'you']

delimiter = ' '

s = delimiter.join(t)

s

'bananas are good for you'
```

insert

This method does not return any value but it inserts the given element at the given index.

```
aList = [123, 'xyz', 'zara', 'abc']

aList.insert( 3, 2009)

print ("Final List : ", aList)
```

```
In [15]: aList = [123, 'xyz', 'zara', 'abc']
    ...: aList.insert( 3, 2009)
    ...: print ("Final List : ", aList)
    ...:
Final List :  [123, 'xyz', 'zara', 2009, 'abc']
```

Lists:
Conversion

Convert String to List

```
s = 'banana'
t = list(s)
t
['b', 'a', 'n', 'a', 'n', 'a']
```

split

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

```
s = 'Eat more bananas, will u?'

t = s.split()

t

['Eat', 'more', 'bananas,', 'will', 'u?']
```

split

An optional argument called a delimiter specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
s = 'bananas-are-good-for-you'
delimiter = '-'
t = s.split(delimiter)
t
['bananas', 'are', 'good', 'for', 'you']
```

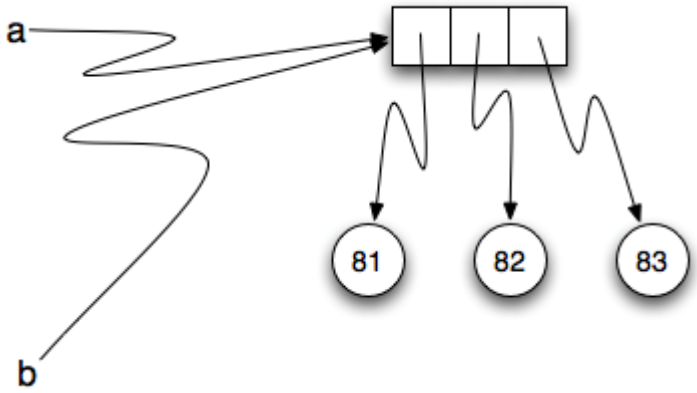
Lists:

List Aliasing vs Cloning

Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

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

b = a

b is a

True
```



Aliasing

The association of a variable with an object is called a reference. In this example, there are two references to the same object.

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

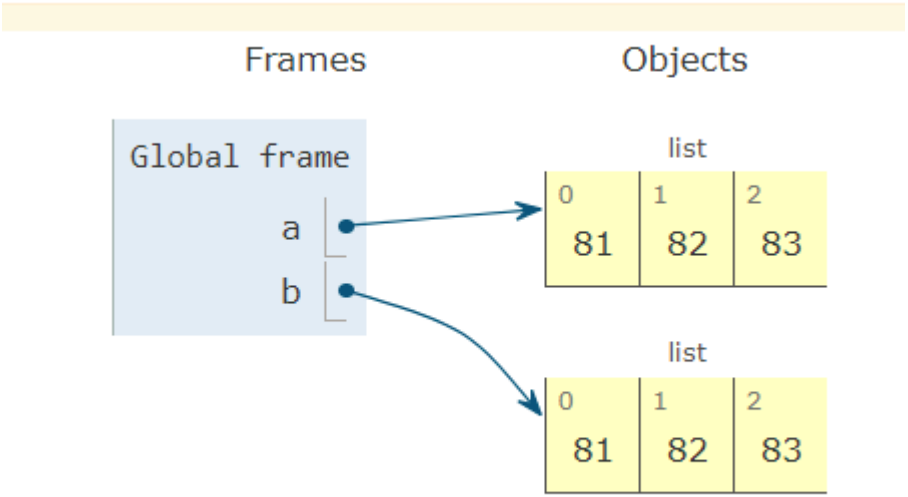
If the aliased object is mutable, changes made with one alias affect the other:

```
b[0] = 42

a

[42, 2, 3]
```

Aliasing

```
a = [81, 82, 83]
b = [81, 82, 83]
print(a == b)
print(a is b)
```

```
True
False
```



Pros and Cons of Aliasing

Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects.

Of course, for immutable objects, there's no problem. That's why Python is free to alias strings and integers when it sees an opportunity to economize.

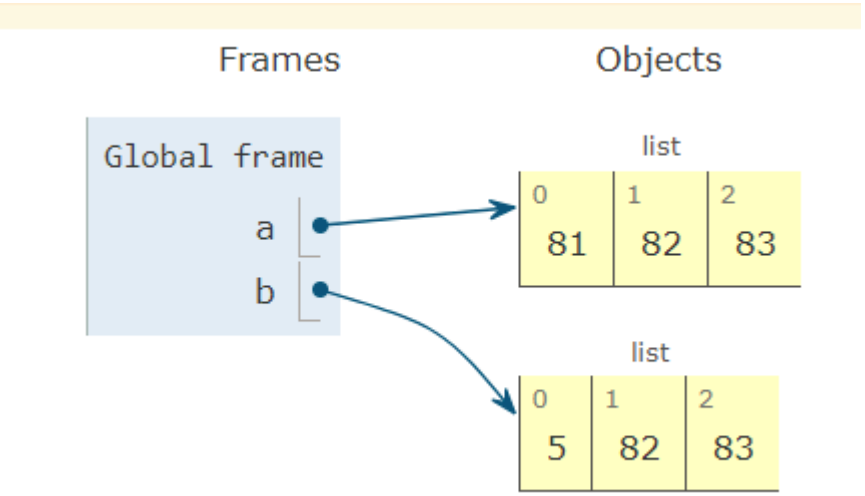
How to Avoid Aliasing

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator.

Cloning

```
a = [81, 82, 83]
b = a[:]          # make a clone using slice
print(a == b)
print(a is b)
True
False
b[0] = 5
print(a)
print(b)
[81, 82, 83]
[5, 82, 83]
```



Lists:
Display All Values

range

```
guests = ['Christopher','Susan','Bill','Satya']

#Create a loop that executes four times
#Since we have four values
for steps in range(4) : # range(4): 0, 1, 2, 3

    #Remember the value of steps goes up by one
    #Each time the loop executes
    print(guests[steps])
```

len

Use the `len()` function to find out how many entries are in your list

```
guests = ['Christopher','Susan','Bill','Satya']

#Find out how many entries are in the list
nbrEntries = len(guests)

#Create a loop that executes once for each entry
for steps in range(nbrEntries) :
    print(guests[steps])
```

Elegant Way to Create New Lists

List comprehension is an elegant and concise way to create new list from an existing list in Python. List comprehension consists of an expression followed by for statement inside square brackets. Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]

# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

print(pow2)
```

Elegant Way to Create New Lists

```
pow2 = [2 ** x for x in range(10) if x > 5]

pow2

[64, 128, 256, 512]

odd = [x for x in range(20) if x % 2 == 1]

odd

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

[x+y for x in ['Python ','C '] for y in
['Language','Programming']]

['Python Language', 'Python Programming', 'C Language',
'C Programming']
```

Lists:

List Comprehension

Iterating through a string Using for Loop

```
h_letters = []


for letter in 'human':
    h_letters.append(letter)


print(h_letters)
```

```
In [1]: h_letters = []
   ...:
   ...: for letter in 'human':
   ...:     h_letters.append(letter)
   ...:
   ...: print(h_letters)
   ...:
['h', 'u', 'm', 'a', 'n']
```

Iterating through a string Using List Comprehension

```
h_letters = [ letter for letter in 'human' ]
print( h_letters)
```

```
In [2]: h_letters = [ letter for letter in 'human' ]
   ...: print( h_letters)
   ...:
['h', 'u', 'm', 'a', 'n']
```

Using Lambda functions inside List

```
h_letters = list(map(lambda x: x, 'human'))


h_letters
```

```
In [3]: h_letters = list(map(lambda x: x, 'human'))
   ...:
   ...: h_letters
   ...:
Out[3]: ['h', 'u', 'm', 'a', 'n']
```

Using if with List Comprehension

```
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list)
```

```
In [4]: number_list = [ x for x in range(20) if x % 2 == 0]
   ...: print(number_list)
   ...:
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Nested IF with List Comprehension

```
num_list = [y for y in range(100) if y % 2 == 0 if y % 5
== 0]

print(num_list)
```

```
In [5]: num_list = [y for y in range(100) if y % 2 == 0 if
y % 5 == 0]
   ...: print(num_list)
   ...:
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

if...else With List Comprehension

```
obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
print(obj)
```

```
In [6]: obj = ["Even" if i%2==0 else "Odd" for i in
range(10)]
   ...: print(obj)
   ...:
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',
'Odd', 'Even', 'Odd']
```

Transpose of a Matrix using List Comprehension

```
matrix = [[1, 2],[3,4],[5,6],[7,8]]

print (matrix)

transpose = [[row[i] for row in matrix] for i in
range(2)]

print (transpose)
```

```
In [8]: matrix = [[1, 2],[3,4],[5,6],[7,8]]
   ...: print (matrix)
   ...: transpose = [[row[i] for row in matrix] for i in
range(2)]
   ...: print (transpose)
   ...:
[[1, 2], [3, 4], [5, 6], [7, 8]]
[[1, 3, 5, 7], [2, 4, 6, 8]]
```

Dictionaries:
Dictionary

What Is a Dictionary

A Dictionary Is a Mapping

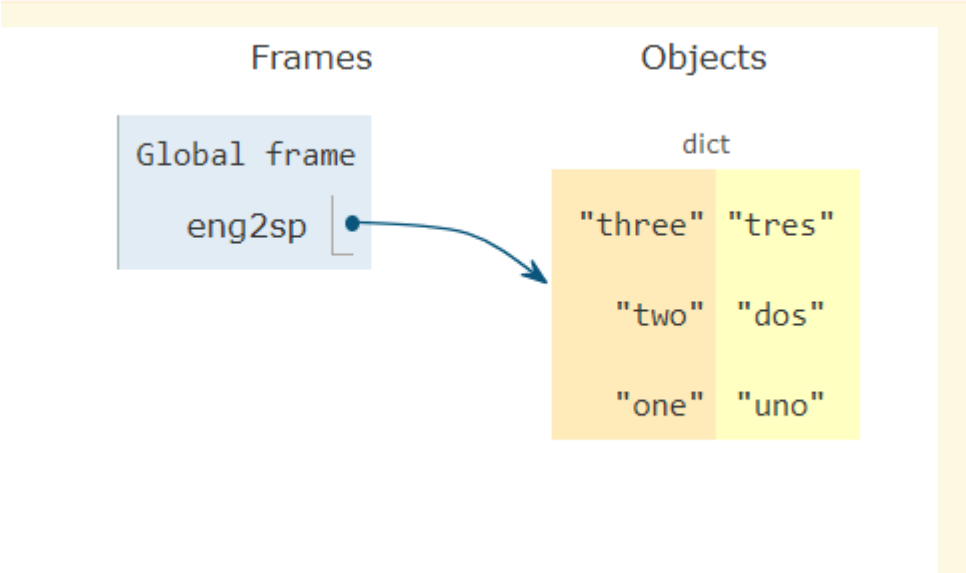
A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called keys, and a collection of values. Each key is associated with a single value. The association of a key and a value is called a key-value pair or sometimes an item.

Key-Value Pairs

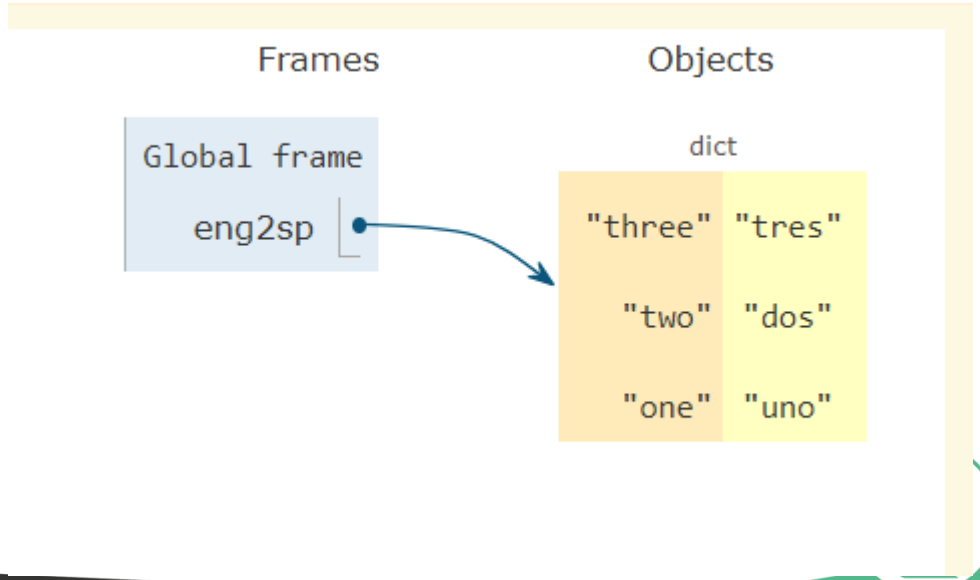
One way to create a dictionary is to start with the empty dictionary and add key-value pairs. The empty dictionary is denoted {}

```
eng2sp = {}
eng2sp['one'] = 'uno'
eng2sp['two'] = 'dos'
eng2sp['three'] = 'tres'
```



Key-Value Pairs

The first assignment creates an empty dictionary named eng2sp. The other assignments add new key-value pairs to the dictionary. The left hand side gives the dictionary and the key being associated. The right hand side gives the value being associated with that key. We can print the current value of the dictionary in the usual way. The key-value pairs of the dictionary are separated by commas. Each pair contains a key and a value separated by a colon.



Examples of Dictionaries

```
# empty dictionary
my_dict = {}
type(my_dict)
# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}
type(my_dict)
# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}
type(my_dict)
```

dict()

```
# using dict()
my_dict = dict({1:'apple', 2:'ball'})
type(my_dict)
# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
type(my_dict)
```

Access Values

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

```
In [14]: dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
    ...: print ("dict['Name']: ", dict['Name'])
    ...:
dict['Name']:  Zara
```

```
In [15]: print ("dict['Age']: ", dict['Age'])
dict['Age']:  7
```

Dictionaries:
Add New Entry

Add New Entry

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print (dict)
dict['School'] = "DPS School"; # Add new entry
print (dict)
print ("dict['School']: ", dict['School'])
```

```
In [17]: dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
    ...: print (dict)
    ...:
{'Name': 'Zara', 'Age': 7, 'Class': 'First'}
```

```
In [18]: dict['School'] = "DPS School"; # Add new entry
    ...: print (dict)
    ...:
{'Name': 'Zara', 'Age': 7, 'Class': 'First', 'School': 'DPS School'}
```

Generate a Dictionary

```
squares = {x: x*x for x in range(6)}


# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
print(squares)
```

Dictionaries:
Update Values

Modify Key-Value Pair

Dictionaries are also mutable. As we've seen before with lists, this means that the dictionary can be modified by referencing an association on the left hand side of the assignment statement. In the previous example, instead of deleting the entry for pears, we could have set the inventory to 0.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

inventory['pears'] = 0

{'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 0}
```

Update Values

```
inventory = {'apples': 430, 'bananas': 312, 'oranges': 525, 'pears': 217}

inventory['bananas'] = inventory['bananas'] + 200

inventory

{'apples': 430, 'bananas': 512, 'oranges': 525, 'pears': 217}

numItems = len(inventory)

numItems

4
```

Update Values

```
my_dict = {'name':'Jack', 'age': 26}
my_dict['age'] = 27 # update value
print(my_dict) #Output: {'age': 27, 'name': 'Jack'}
my_dict['address'] = 'Downtown'  # add item
print(my_dict)
# Output: {'address': 'Downtown', 'age': 27, 'name':
'Jack'}
```

Dictionaries:
Remove Elements

clear

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
print(dict)
dict.clear();     # remove all entries in dict
print(dict)
del dict ;        # delete entire dictionary
```

```
In [19]: dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
    ...: del dict['Name']; # remove entry with key 'Name'
    ...: print(dict)
    ...:
{'Age': 7, 'Class': 'First'}
```

```
In [21]: dict.clear();     # remove all entries in dict
    ...: print(dict)
    ...:
{}
```

Remove Key-Value Pair

The del statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock. If someone buys all of the pears, we can remove the entry from the dictionary.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges':
525, 'pears': 217}

del inventory['pears']

{'apples': 430, 'bananas': 312, 'oranges': 525}
```

Dictionaries:

Dictionary Methods

List of Methods

Method	Parameters	Description
keys	none	Returns a view of the keys in the dictionary
values	none	Returns a view of the values in the dictionary
items	none	Returns a view of the key-value pairs in the dictionary
get	key	Returns the value associated with key; None otherwise
get	key,alt	Returns the value associated with key; alt otherwise

Underlying Keys

The keys method returns what Python 3 calls a view of its underlying keys. We can iterate over the view or turn the view into a list by using the list conversion function.

```
inventory = {'apples': 430, 'bananas': 312, 'oranges':
525, 'pears': 217}

for akey in inventory.keys():

# the order in which we get the keys is not defined

   print("Got key", akey, "which maps to value",
inventory[akey])

   ks = list(inventory.keys())

print(ks)
```

Underlying Keys

```
In [11]: inventory = {'apples': 430, 'bananas': 312,
'oranges': 525, 'pears': 217}
    ...: for akey in inventory.keys():
    ...: # the order in which we get the keys is not
defined
    ...:     print("Got key", akey, "which maps to value",
inventory[akey])
    ...:     ks = list(inventory.keys())
    ...: print(ks)
    ...:
Got key apples which maps to value 430
Got key bananas which maps to value 312
Got key oranges which maps to value 525
Got key pears which maps to value 217
['apples', 'bananas', 'oranges', 'pears']
```

Keys in a Dictionary

It is so common to iterate over the keys in a dictionary that you can omit the keys method call in the for loop — iterating over a dictionary implicitly iterates over its keys.

```
for k in inventory:
    print("Got key", k)
```

```
Got key apples
Got key bananas
Got key oranges
Got key pears
```

Iteration

```
for k in inventory:
    print("Got key", k, "which maps to value", inventory[k])
```

```
Got key apples which maps to value 430
Got key bananas which maps to value 312
Got key oranges which maps to value 525
Got key pears which maps to value 217
```

Values and Items Methods

The values and items methods are similar to keys. They return view objects which can be turned into lists or iterated over directly. Note that the items are shown as tuples containing the key and the associated value.

```
print(list(inventory.values()))
print(list(inventory.items()))
for (k,v) in inventory.items():
    print("Got", k, "that maps to", v)
```

Values and Items Methods

```
In [12]: print(list(inventory.values()))
    ...: print(list(inventory.items()))
    ...: for (k,v) in inventory.items():
    ...:     print("Got", k, "that maps to", v)
    ...:
[430, 312, 525, 217]
[('apples', 430), ('bananas', 312), ('oranges', 525),
('pears', 217)]
Got apples that maps to 430
Got bananas that maps to 312
Got oranges that maps to 525
Got pears that maps to 217
```

Verify Keys in a Dictionary

The `in` and `not in` operators can test if a key is in the dictionary:

```
print('apples' in inventory)
print('cherries' in inventory)
entry = 'pineapple'
if entry in inventory:
    print(inventory[entry])
else:
    print("We have no", entry)
```

Verify Keys in a Dictionary

```
In [13]: print('apples' in inventory)
    ...: print('cherries' in inventory)
    ...: entry = 'pineapple'
    ...: if entry in inventory:
    ...:     print(inventory[entry])
    ...: else:
    ...:     print("We have no", entry)
    ...:
True
False
We have no pineapple
```

get

The get method allows us to access the value associated with a key, similar to the [ ] operator. The important difference is that get will not cause a runtime error if the key is not present. It will instead return None. There exists a variation of get that allows a second parameter that serves as an alternative return value in the case where the key is not present. This can be seen in the final example below. In this case, since “cherries” is not a key, return 0 (instead of None).

get

```
print(inventory.get("apples"))
print(inventory.get("cherries"))
print(inventory.get("cherries", 0))
```

```
430
None
0
```

.values()

This method returns a list of all the values available in a given dictionary.

```
dict = {'Name': 'Zara', 'Age': 7}

print ("Value : %s" %  dict.values())

print ("Value : %s" %  dict)

dict.values()
```

```
In [17]: dict = {'Name': 'Zara', 'Age': 7}
    ...: print ("Value : %s" %  dict.values())
    ...: print ("Value : %s" %  dict)
    ...: dict.values()
    ...:
Value : dict_values(['Zara', 7])
Value : {'Name': 'Zara', 'Age': 7}
Out[17]: dict_values(['Zara', 7])
```

.update()

The method update() adds dictionary dict2's key-values pairs in to dict. This function does not return anything.

```
dict = {'Name': 'Zara', 'Age': 7}

dict2 = {'Sex': 'female' }

dict.update(dict2)

print ("Value : %s" %  dict)
```

```
In [25]: dict = {'Name': 'Zara', 'Age': 7}
    ...: dict2 = {'Sex': 'female' }
    ...: dict.update(dict2)
    ...: print ("Value : %s" %  dict)
    ...:
Value : {'Name': 'Zara', 'Age': 7, 'Sex': 'female'}
```

.setdefault()

This method returns the key value available in the dictionary and if given key is not available then it will return provided default value.

```
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.setdefault('Age', None))
print ("Value : %s" %  dict.setdefault('Sex', None))
```

```
In [26]: dict = {'Name': 'Zara', 'Age': 7}
    ...: print ("Value : %s" %  dict.setdefault('Age', None))
    ...:
Value : 7
```

```
In [27]: print ("Value : %s" %  dict.setdefault('Sex', None))
Value : None
```

.keys()

This method returns a list of all the available keys in the dictionary.

```
dict = {'Name': 'Zara', 'Age': 7}

print ("Value : %s" %  dict.keys())
```

```
In [29]: dict = {'Name': 'Zara', 'Age': 7}
    ...: print ("Value : %s" %  dict.keys())
    ...:
Value : dict_keys(['Name', 'Age'])
```

.items()

The method items() returns a list of dict's (key, value) tuple pairs

```
dict = {'Name': 'Zara', 'Age': 7}
print ("Value : %s" %  dict.items())
```

```
In [30]: dict = {'Name': 'Zara', 'Age': 7}
    ...: print ("Value : %s" %  dict.items())
    ...:
Value : dict_items([('Name', 'Zara'), ('Age', 7)])
```

.copy()

This method returns a shallow copy of the dictionary.

```
dict1 = {'Name': 'Zara', 'Age': 7};

dict2 = dict1.copy()

print("New Dictionary : %s" %  str(dict2))
```

Dictionaries:

Other Dictionary Operations

Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that membership test is for keys only, not for values.

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True

print(1 in squares)

# Output: True

print(2 not in squares)

# membership tests for key only not value

# Output: False

print(49 in squares)
```

Iterating Through a Dictionary

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
for i in squares:
    print(squares[i])
```

```
In [18]: squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
    ...: for i in squares:
    ...:     print(squares[i])
    ...:
1
9
25
49
81
```

Dictionaries:
Nested Dictionary

Nested Dictionary

In Python, a nested dictionary is a dictionary inside a dictionary. It's a collection of dictionaries into one single dictionary.

```
nested_dict = { 'dictA': {'key_1': 'value_1'},
                'dictB': {'key_2': 'value_2'}}
```

Here, the nested_dict is a nested dictionary with the dictionary dictA and dictB. They are two dictionary each having own key and value.

Nested Dictionary

We're going to create dictionary of people within a dictionary.

```
people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex':
'Female'}}


print(people)
```

```
In [19]: people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
    ...:           2: {'name': 'Marie', 'age': '22', 'sex':
'Female'}}
    ...:
    ...: print(people)
    ...:
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2:
{'name': 'Marie', 'age': '22', 'sex': 'Female'}}
```

Access elements of a Nested Dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex':
'Female'}}

print(people[1]['name'])

print(people[1]['age'])

print(people[1]['sex'])
```

```
In [20]: people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
    ...:           2: {'name': 'Marie', 'age': '22', 'sex':
'Female'}}
    ...: print(people[1]['name'])
    ...: print(people[1]['age'])
    ...: print(people[1]['sex'])
    ...:
John
27
Male
```

Add element to a Nested Dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex':
'Female'}}

people[3] = {}

people[3]['name'] = 'Luna'

people[3]['age'] = '24'

people[3]['sex'] = 'Female'

people[3]['married'] = 'No'

print(people[3])
```

Add element to a Nested Dictionary

```
In [21]: people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
    ...:           2: {'name': 'Marie', 'age': '22', 'sex':
'Female'}}
    ...: people[3] = {}
    ...: people[3]['name'] = 'Luna'
    ...: people[3]['age'] = '24'
    ...: people[3]['sex'] = 'Female'
    ...: people[3]['married'] = 'No'
    ...: print(people[3])
    ...:
{'name': 'Luna', 'age': '24', 'sex': 'Female', 'married':
'No'}
```

Add a Nested Dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex':
'Female'},
          3: {'name': 'Luna', 'age': '24', 'sex':
'Female', 'married': 'No'}}


people[4] = {'name': 'Peter', 'age': '29', 'sex':
'Male', 'married': 'Yes'}

print(people[4])
```

Add a Nested Dictionary

```
In [22]: people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
    ...:           2: {'name': 'Marie', 'age': '22', 'sex':
'Female'},
    ...:           3: {'name': 'Luna', 'age': '24', 'sex':
'Female', 'married': 'No'}}
    ...:
    ...: people[4] = {'name': 'Peter', 'age': '29', 'sex':
'Male', 'married': 'Yes'}
    ...: print(people[4])
    ...:
{'name': 'Peter', 'age': '29', 'sex': 'Male', 'married':
'Yes'}
```

Delete elements from a Nested Dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},

          2: {'name': 'Marie', 'age': '22', 'sex':
'Female'},

          3: {'name': 'Luna', 'age': '24', 'sex':
'Female', 'married': 'No'},

          4: {'name': 'Peter', 'age': '29', 'sex':
'Male', 'married': 'Yes'}}

del people[3]['married']

del people[4]['married']

print(people[3])

print(people[4])
```

Delete elements from a Nested Dictionary

```
In [23]: people = {1: {'name': 'John', 'age': '27', 'sex':
'Male'},
    ...:           2: {'name': 'Marie', 'age': '22', 'sex':
'Female'},
    ...:           3: {'name': 'Luna', 'age': '24', 'sex':
'Female', 'married': 'No'},
    ...:           4: {'name': 'Peter', 'age': '29', 'sex':
'Male', 'married': 'Yes'}}
    ...: del people[3]['married']
    ...: del people[4]['married']
    ...: print(people[3])
    ...: print(people[4])
    ...:
{'name': 'Luna', 'age': '24', 'sex': 'Female'}
{'name': 'Peter', 'age': '29', 'sex': 'Male'}
```

Iterate from a nested dictionary

```
people = {1: {'Name': 'John', 'Age': '27', 'Sex':
'Male'},
          2: {'Name': 'Marie', 'Age': '22', 'Sex':
'Female'}}


for p_id, p_info in people.items():
    print("\nPerson ID:", p_id)

    for key in p_info:
        print(key + ':', p_info[key])
```

Iterate from a nested dictionary

```
In [24]: people = {1: {'Name': 'John', 'Age': '27', 'Sex':
'Male'},
    ...:           2: {'Name': 'Marie', 'Age': '22', 'Sex':
'Female'}}
    ...:
    ...: for p_id, p_info in people.items():
    ...:     print("\nPerson ID:", p_id)
    ...:
    ...:     for key in p_info:
    ...:         print(key + ':', p_info[key])
    ...:

Person ID: 1
Name: John
Age: 27
Sex: Male

Person ID: 2
Name: Marie
Age: 22
Sex: Female
```

Other Data Types:
Tuples

tuple

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example

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

Empty Tuple

The empty tuple is written as two parentheses containing nothing

```
tup1 = ();
```

Tuple with a Single Value

To write a tuple containing a single value you have to include a comma, even though there is only one value：

```
tup1 = (50,);
# only parentheses is not enough
my_tuple = ("hello")
print(type(my_tuple)) # Output: <class 'str'>
```

Tuple with a Single Value

```
# need a comma at the end
# Output: <class 'tuple'>
my_tuple = ("hello",)
print(type(my_tuple))

# parentheses is optional
# Output: <class 'tuple'>
my_tuple = "hello",
print(type(my_tuple))
```

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For 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]);
```

Updating Tuples

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
# Following action is not valid for tuples
# tup1[0] = 100; # generate an error
# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print (tup3);
```

Updating Tuples

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

Delete Tuple Entirely

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded. To explicitly remove an entire tuple, just use the del statement. For example

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

print tup;

del tup;

print "After deleting tup : ";

print tup;
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string. In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in Tuple Functions

Sr.No.	Function with Description
cmp(tuple1, tuple2)	Compares elements of both tuples.
len(tuple)	Gives the total length of the tuple.
max(tuple)	Returns item from the tuple with max value.
min(tuple)	Returns item from the tuple with min value.
tuple(seq)	Converts a list into tuple.

Nested Tuple

Nested tuple are accessed using nested indexing, as shown in the example below.

```
# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(n_tuple[0][3]) # Output: 's'
# nested index
print(n_tuple[1][1]) # Output: 4
```

```
In [27]: # nested tuple
    ...: n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
    ...: # nested index
    ...: print(n_tuple[0][3]) # Output: 's'
    ...: # nested index
    ...: print(n_tuple[1][1]) # Output: 4
    ...:
s
4
```

Python Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Method	Description
count(x)	Return the number of items that is equal to x
index(x)	Return index of first item that is equal to x

.count() and .index()

```
my_tuple = ('a','p','p','l','e',)


# Count
print(my_tuple.count('p')) # Output: 2


# Index
print(my_tuple.index('l')) # Output: 3
```

Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
my_tuple = ('a','p','p','l','e',)

# In operation

print('a' in my_tuple) # Output: True

print('b' in my_tuple) # Output: False

# Not in operation

print('g' not in my_tuple)
```

Iterating Through a Tuple

Using a for loop we can iterate though each item in a tuple.

```
# Output:
# Hello John
# Hello Kate
for name in ('John','Kate'):
    print("Hello",name)
```

Built-in Functions with Tuple

Function	Description
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.

Built-in Functions with Tuple

Function	Description
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Retrun the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

Advantages of Tuple over List

Since, tuples are quite similar to lists, both of them are used in similar situations as well. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Other Data Types:
Sets

What Is Set?

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

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

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

create a set

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

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

Create Sets

```
# set of integers
my_set = {1, 2, 3}
print(my_set)
# set of mixed datatypes
my_mixed_set = {1.0, "Hello", (1, 2, 3)}
print(my_mixed_set)
weird_set = set("A weird set")
print(weird_set)
```

Every element is unique

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


my_set = set([1,2,3,2])
print(my_set)
```

```
In [28]: my_set = {1,2,3,4,3,2}
    ...: print(my_set)
    ...:
    ...: my_set = set([1,2,3,2])
    ...: print(my_set)
    ...:
{1, 2, 3, 4}
{1, 2, 3}
```

Creating an empty set

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.

```
# initialize a with {}
a = {}
# check data type of a
# Output: <class 'dict'>
print(type(a))
```

Creating an empty set

```
# initialize a with set()
a = set()
# check data type of a
# Output: <class 'set'>
print(type(a))
```

change a set in Python

Sets are mutable. But since they are unordered, indexing have no meaning. We cannot access or change an element of set using indexing or slicing. Set does not support it. We can add 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.

change a set in Python

```
# initialize my_set
my_set = {1,3}
print(my_set)
# if you uncomment line 9,
my_set[0] # you will get an error, no index for set
# add an element
my_set.add(2) # Output: {1, 2, 3}
print(my_set)
```

change a set in Python

```
# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2,3,4])
print(my_set)


# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4,5], {1,6,8})
print(my_set)
```

change a set in Python

```
# add another set as an element to make a nested set
my_set.add(frozenset([2,3,4]))
print(my_set)
```

```
In [33]: my_set.add(frozenset([2,3,4]))
    ...: print(my_set)
    ...:
{1, 2, 3, 4, frozenset({2, 3, 4})}
```

remove elements from a set

A particular item can be removed from set using methods, `discard()` and remove(). The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But `remove()` will raise an error in such condition.

.discard()

```
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)
```

.remove()

```
# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)


# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)
```

.pop()

Similarly, we can remove and return an item using the pop() method. Set being **unordered**, 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 **clear()**.

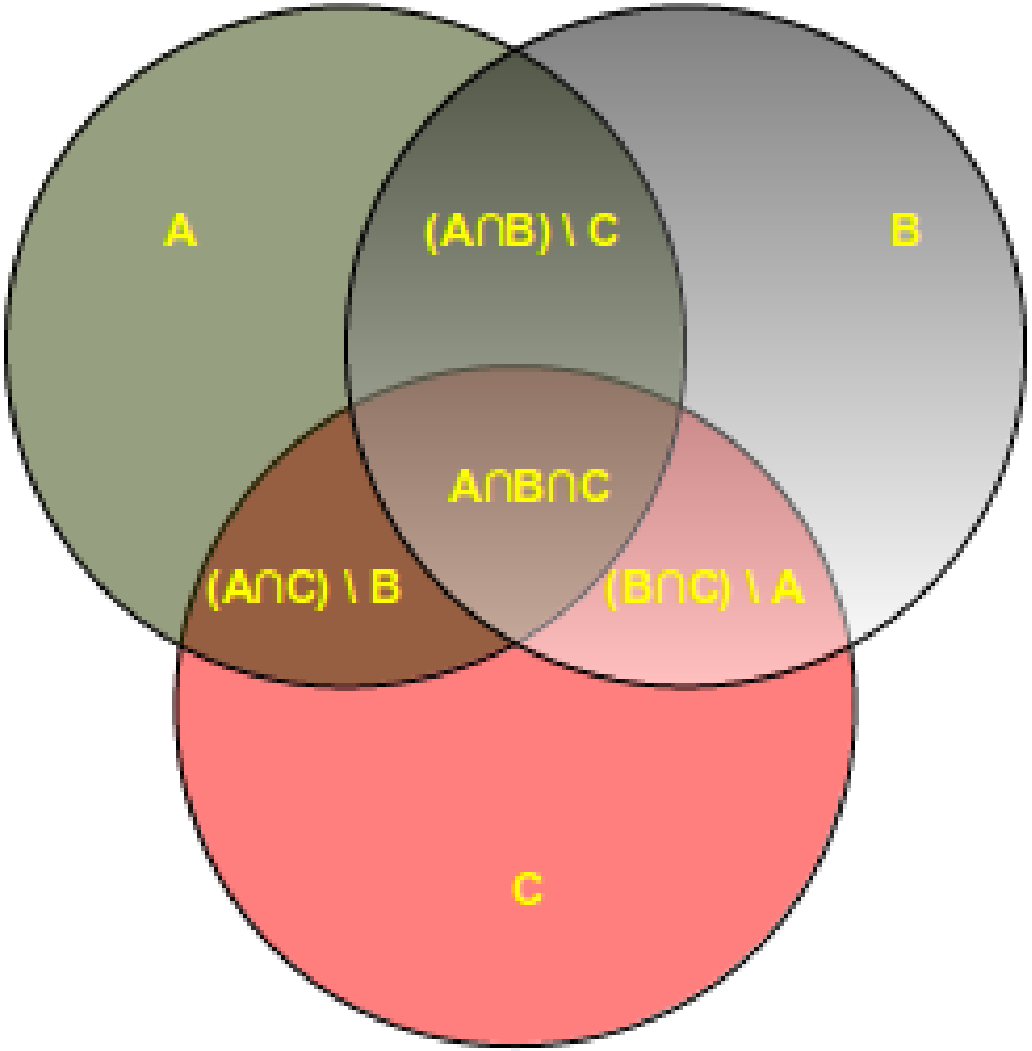
```
# initialize my_set

# Output: set of unique elements

my_set = set("HelloWorld")

print(my_set)

# pop an element randomly

# Output: random element

print(my_set.pop())
```

.pop()

```
# pop another element
# Output: random element
my_set.pop()
print(my_set)

# clear my_set
#Output: set()
my_set.clear()
print(my_set)
```

Other Data Types:
Set Operations

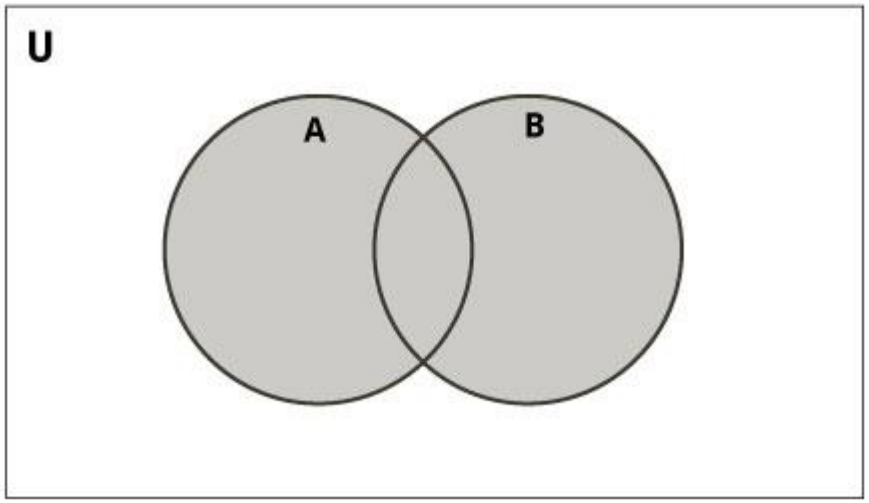
Mathematical Operations of Three Sets



Set Union

Union of A and B is a set of all elements from both sets. Union is performed using | operator. Same can be accomplished using the method union().

```
# initialize A and B

A = {1, 2, 3, 4, 5}; B = {4, 5, 6, 7, 8}

# use | operator; Output: {1, 2, 3, 4, 5, 6, 7, 8}

print(A | B)
```



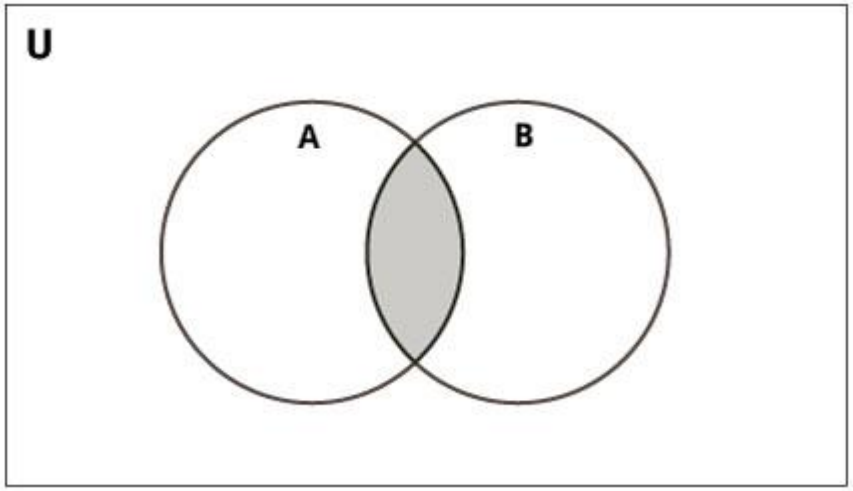
Set Union

```
# use union function
>>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}

# use union function on B
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection

Intersection of A and B is a set of elements that are common in both sets. Intersection is performed using & operator. Same can be accomplished using the method intersection().

```
# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}


# use & operator

# Output: {4, 5}

print(A & B)
```



Set Intersection

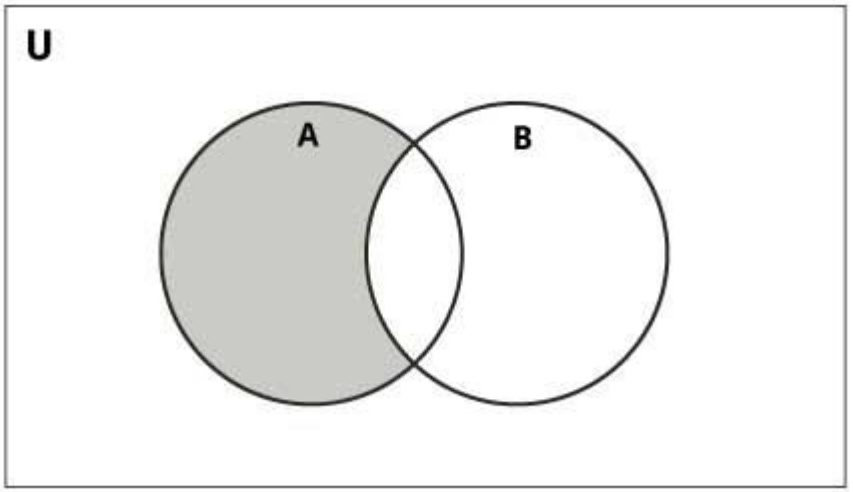
Try the following examples on Python shell.

```
# use intersection function on A
>>> A.intersection(B)
{4, 5}

# use intersection function on B
>>> B.intersection(A)
{4, 5}
```

Set Difference

Difference of A and B (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of element in B but not in A. Difference is performed using - operator. Same can be accomplished using the method difference().

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}


# use - operator on A
# Output: {1, 2, 3}
print(A - B)
```

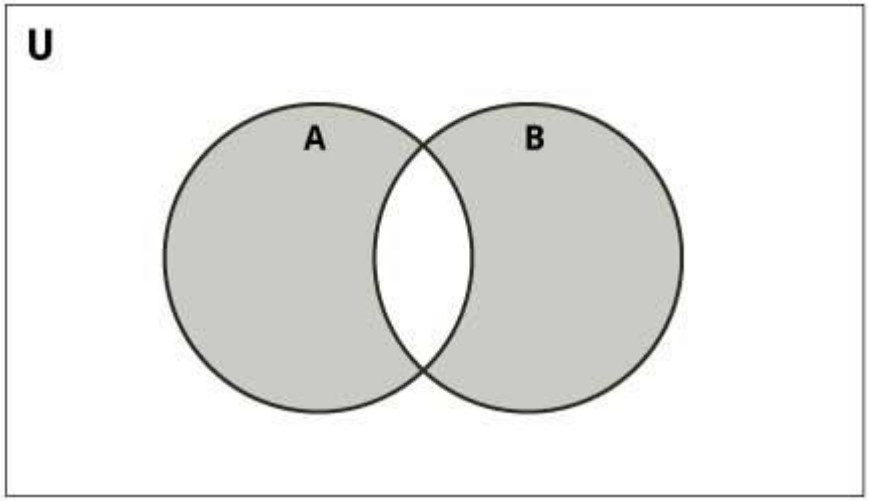


Set Difference

```
# use difference function on A
>>> A.difference(B) # A - B
{1, 2, 3}

# use - operator on B
>>> B - A
{8, 6, 7}

# use difference function on B
>>> B.difference(A) # B - A
{8, 6, 7}
```

Set Symmetric Difference

Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both. Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric_difference().




```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use ^ operator
# Output: {1, 2, 3, 6, 7, 8}
print(A ^ B)
```

Set Symmetric Difference

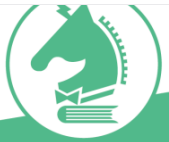
```
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}

# use symmetric_difference function on B
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Other Data Types:

Python Set Methods

Different Python Set Methods

Method	Description
add()	Add an element to a set
clear()	Remove all elements form a set
copy()	Return a shallow copy of a set
difference()	Return the difference of two or more sets as a new set
difference_update()	Remove all elements of another set from this set
discard()	Remove an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Return the intersection of two sets as a new set
intersection_update()	Update the set with the intersection of itself and another
isdisjoint()	Return True if two sets have a null intersection
issubset()	Return True if another set contains this set



Different Python Set Methods

Method	Description
issuperset()	Return True if this set contains another set
pop()	Remove and return an arbitary set element. Raise KeyErrorif the set is empty
remove()	Remove an element from a set. If the element is not a member, raise a KeyError
symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets in a new set
update()	Update a set with the union of itself and others

Set Membership Test

```
# initialize my_set
my_set = set("apple")


# check if 'a' is present
# Output: True
print('a' in my_set)


# check if 'p' is present
# Output: False
print('p' not in my_set)
```

```
In [34]: my_set = set("apple")

In [35]: my_set
Out[35]: {'a', 'e', 'l', 'p'}
```

Iterating Through a Set

Using a for loop, we can iterate though each item in a set.

```
for letter in set("apple"):
...     print(letter)
```

Set Membership Test

```
# initialize my_set
my_set = set(["apple"])
print(my_set)
# check if 'a' is present
# Output: False
print('a' in my_set)
```

```
In [45]: # initialize my_set
    ...: my_set = set(["apple"])
    ...: print(my_set)
    ...: # check if 'a' is present
    ...: # Output: False
    ...: print('a' in my_set)
    ...:
{'apple'}
False
```

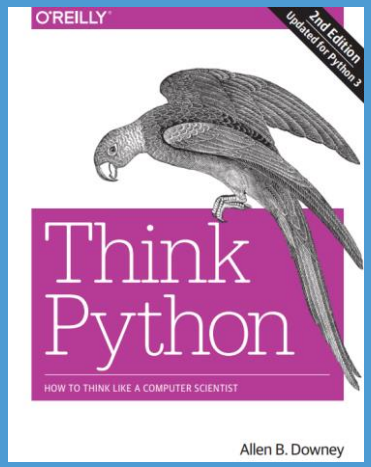
Python Frozenset

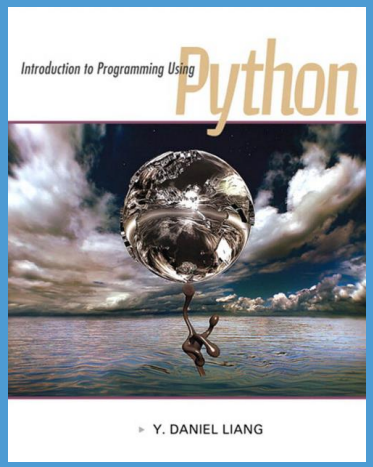
Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets. Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

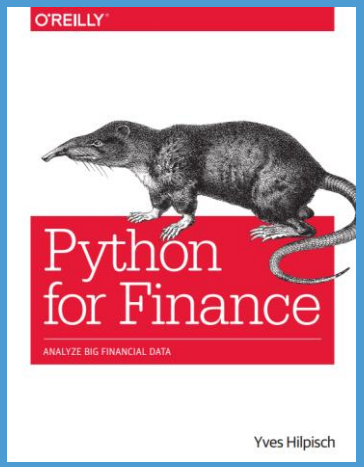
Frozensets can be created using the function frozenset().

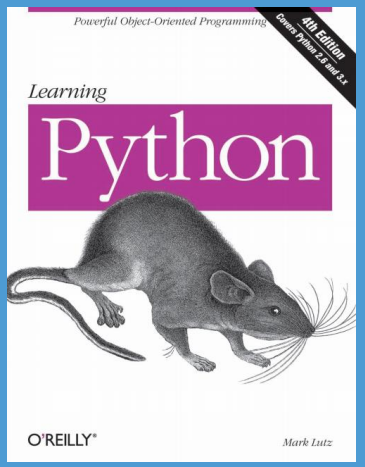
```
# initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])
```

References


O'REILLY
2nd Edition
Updated for Python 3
Think Python
HOW TO THINK LIKE A COMPUTER SCIENTIST
Allen B. Downey


Introduction to Programming Using Python
Y. DANIEL LIANG


O'REILLY
Python for Finance
ANALYZE BIG FINANCIAL DATA
Yves Hilpisch


Powerful Object-Oriented Programming
4th Edition
Learning
Python
O'REILLY
Mark Lutz