**Python important Inbuilt functions and uses:**

**List:**

Pop() method :

Definition and Usage

The pop() method removes the element at the specified position.

Syntax

*list*.pop(*pos*)

Parameter Values

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| *pos* | Optional. A number specifying the position of the element you want to remove, default value is -1, which returns the last item |

Remove the second element of the fruit list:

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

**Note:** The pop() method returns removed value.

# Python List remove()

The remove() method removes the first matching element (which is passed as an argument) from the list.

### Example

# create a list

prime\_numbers = [2, 3, 5, 7, 9, 11]

# remove 9 from the list

prime\_numbers.remove(9)

# Updated prime\_numbers List

print('Updated List: ', prime\_numbers)

# Output: Updated List: [2, 3, 5, 7, 11]

**EXamle 2:**

# animals list

animals = ['cat', 'dog', 'dog', 'guinea pig', 'dog']

# 'dog' is removed

animals.remove('dog')

# Updated animals list

print('Updated animals list: ', animals)

**Output : only one dog is deleted.**

Updated animals list: ['cat', 'dog', 'guinea pig', 'dog']

**Example 3:**

# animals list

animals = ['cat', 'dog', 'rabbit', 'guinea pig']

# Deleting 'fish' element

animals.remove('fish')

# Updated animals List

print('Updated animals list: ', animals)

**Output**

Traceback (most recent call last):

File ".. .. ..", line 5, in <module>

animal.remove('fish')

ValueError: list.remove(x): x not in list

**Python Double Colon (**::**) Syntax**

Syntax :

collection[start:stop:step]

In the syntax above:

* **collection**denotes the data collection (list, string, array, and so on).
* **start** denotes index where the slicing operation should start from.
* **stop**denotes index where the operation should stop {not includes index}.
* **step**denotes the sequence of iterating through the elements.

### Python Double Colon Example #1

In this example, we'll focus on the start parameter:

number\_list = [2,4,6,8,10,12]

print(number\_list[2:])

# [6, 8, 10, 12]

### Python Double Colon Example #2

stop parameter comes after the first colon and before the second colon. Unlike the start parameter, the specified index will not be included.

Here's an example:

number\_list = [2,4,6,8,10,12]

print(number\_list[:2])

# [2, 4]

### Python Double Colon Example #3

### Without specifying values for the start and stop parameters, you have access the whole list. No element gets sliced off.

number\_list = [2,4,6,8,10,12]

print(number\_list[::2])

# [2, 6, 10]

# **Python List sort() Method**

### **Example**

Sort the list alphabetically:

cars = ['Ford', 'BMW', 'Volvo']  
  
cars.sort()

Definition and Usage

The sort() method sorts the list ascending by default.

You can also make a function to decide the sorting criteria(s).

Syntax

*list*.sort(reverse=True|False, key=myFunc)

### **Example 1**

Sort the list by the length of the values:

# A function that returns the length of the value:  
def myFunc(e):  
  return len(e)  
  
cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']  
cars.sort(key=myFunc)

### **Example 2**

def myFunc(e):  
  return len(e)

# Reverse Order  
cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']  
cars.sort(reverse=True, key=myFunc)

List Initialization :

List Declarations:

lst = list(range(1,101))

print(lst)

|  |
| --- |
| import numpy **as** np  lst = np.arange(1,101).tolist()  print(lst) |

Output:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

Most Usable Way:

lst = [i **for** i **in** range(1,101)]

print(lst)

Output:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

important Syntax:

List\_name = [val]\*req\_len

Dp = [-1]\*100

#this will create a list of length 100 containing -1 value.

# Enumerate()

Python eases the programmers’ task by providing a built-in function enumerate() for this task. Enumerate() method adds a counter to an iterable and returns it in a form of enumerating object.

**Syntax:**

enumerate(iterable, start=0)

# Python program to illustrate

# enumerate function

l1 **=** ["eat", "sleep", "repeat"]

s1 **=** "geek"

# creating enumerate objects

obj1 **=** enumerate(l1)

obj2 **=** enumerate(s1)

print ("Return type:", type(obj1))

**print** (list(enumerate(l1)))

# changing start index to 2 from 0

print (list(enumerate(s1, 2)))

**Output:**

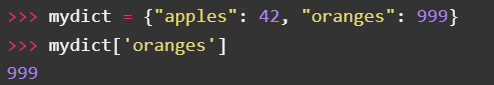
Return type:

[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]

[(2, 'g'), (3, 'e'), (4, 'e'), (5, 'k')]

**Dictionaries:**

Dictionaries – like lists – are collections of objects. Unlike lists, dictionaries are unordered collections. They are not indexed by sequential numbers, but by keys:

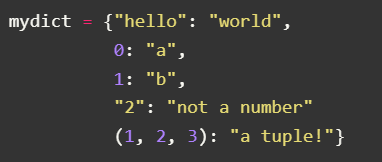
  
An empty dictionary can be initialized by either using the dict() constructor function, or simply with a pair of curly braces:



Accessing a dictionary's values by index (i.e. its keys) uses the same square bracket notation as other sequence-type objects:

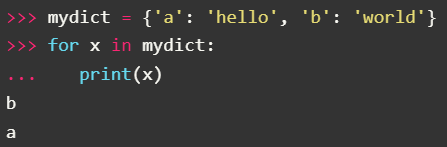
Valid Keys:

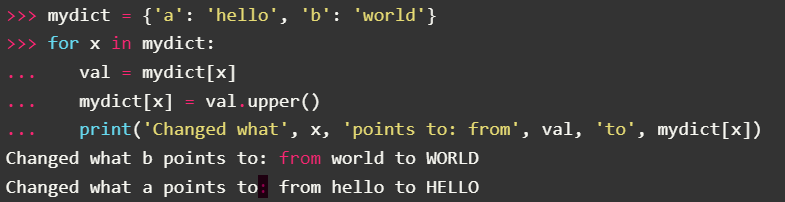
The **keys** of a dictionary can be any kind of **immutable** type, which includes: strings, numbers, and tuples:



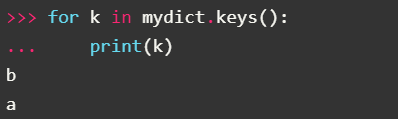
**Iterating through the keys**

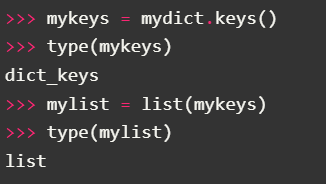
If we pass a dict object into a for-loop, by default, only the **key** will be yielded:





Using keys() method:





**Iterating through the values**

If we want to iterate only through a dictionary's values, or to get a list of its values, we can call its values() method:

>>> mydict = {'a': 'hello', 'b': 'world'}

>>> for v in mydict.values():

... print(v)

world

hello

>>> myvals = mydict.values()

>>> type(myvals)

dict\_values

>>> mylist = list(myvals)

>>> type(mylist)

list

## Iterating through key-value pairs with items()

Oftentimes, we'd like to have access to both the key and the value for every key-value pair in a dictionary. We could use the keys() method and then derive each key's value inside the loop:

>>> mydict = {'a': 'hello', 'b': 'world'}

>>> for key in mydict.keys():

>>> val = mydict[key]

>>> print("Key", key, 'points to', val)

Key a points to hello

Key b points to world

## get()

Similar to trying to access a list by too big of an index value, accessing a non-existent key of a dictionary will raise a KeyError:

>>> mydict = {'z': 999}

>>> x = mydict['a']

KeyError: 'a'

The get(k) method provides a safe way to test for a key, k. If the dictionary has key k, the get(k) method will return the value. If not, then a NoneType object is returned:

>>> mydict = {'z': 999}

>>> mydict.get('z')

999

>>> mydict.get('oooga boooga!!!')

>>> type(mydict.get('heysadf'))

NoneType

This is especially useful when looping through a list of dictionaries, in which not all the dictionaries have all of the same keys:

names = []

names.append({'first': 'Dan', 'last': 'Nguyen', 'suffix': 'III'})

names.append({'first': 'Jane'})

for name in names:

x = name.get('first')

y = name.get('last')

z = name.get('suffix')

print(x, y, z)

The output:

Dan Nguyen III

Jane None None

## Use update() to set several key-value pairs at once

>>> a = {'first': 'Dan', 'last': 'Nguyen'}

>>> b = {'last': 'Smith', 'suffix': 'Jr.'}

>>> a.update(b)

>>> print(a)

{'first': 'Dan', 'last': 'Smith', 'suffix': 'Jr.'}

## Insertion of elements

In a **list**, members are added into memory sequentially:

>>> mylist = []

>>> mylist.append('a')

>>> mylist.append('b')

>>> mylist.append('c')

Python only allows us to (i.e. change) the value at an existing index:

>>> mylist[0] = 'A'

set

>>> mydict = {}

>>> mydict[99999999] = "hello"

### The consequences of being unordered

A **list** is considered ordered because its members are arranged in the same order that they were inserted into the list. Every time we iterate through a list, sequentially, we can assume that its members will always be accessible in the same order that they were inserted.

In contrast, the members of **dictionary** are not stored in any particular order. No matter what order you add key-value pairs into a dictionary, we have no idea what order they'll come out as when we iterate through the dictionary:

>>> mydict = {}

>>> mydict['a'] = 0

>>> mydict['b'] = 1

>>> mydict['c'] = 2

>>> for k in mydict:

... print(k)

b

c

a

Satck:

A **stack** is a linear data structure that stores items in a [**Last-In/First-Out (LIFO)**](https://www.geeksforgeeks.org/lifo-last-in-first-out-approach-in-programming/)

**The functions associated with stack are:**

* **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
* **size()** – Returns the size of the stack – Time Complexity: O(1)
* **top() / peek()**– Returns a reference to the topmost element of the stack – Time Complexity: O(1)
* **push(a)** – Inserts the element ‘a’ at the top of the stack – Time Complexity: O(1)
* **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)

### Implementation:

* list
* Collections.deque
* queue.LifoQueue

|  |
| --- |
| stack **=** []    # append() function to push  # element in the stack  stack.append('a')  stack.append('b')  stack.append('c')    **print**('Initial stack')  **print**(stack)    # pop() function to pop  # element from stack in  # LIFO order  print('\nElements popped from stack:')  print(stack.pop())  print(stack.pop())  **print**(stack.pop())    print('\nStack after elements are popped:')  print(stack)    # uncommenting print(stack.pop())  # will cause an IndexError  # as the stack is now empty |

**Output**

Initial stack

['a', 'b', 'c']

Elements popped from stack:

c

b

a

Stack after elements are popped:

[]

|  |
| --- |
| **from** collections **import** deque    stack **=** deque()    # append() function to push  # element in the stack  stack.append('a')  stack.append('b')  stack.append('c')    **print**('Initial stack:')  **print**(stack)    # pop() function to pop  # element from stack in  # LIFO order  print('\nElements popped from stack:')  **print**(stack.pop())  print(stack.pop())  print(stack.pop())    **print**('\nStack after elements are popped:')  print(stack)    # uncommenting print(stack.pop())  # will cause an IndexError  # as the stack is now empty |

**Output**

Initial stack:

deque(['a', 'b', 'c'])

Elements popped from stack:

c

b

a

Stack after elements are popped:

deque([])

|  |
| --- |
| **from** queue **import** LifoQueue    # Initializing a stack  stack **=** LifoQueue(maxsize**=**3)    # qsize() show the number of elements  # in the stack  print(stack.qsize())    # put() function to push  # element in the stack  stack.put('a')  stack.put('b')  stack.put('c')    **print**("Full: ", stack.full())  **print**("Size: ", stack.qsize())    # get() function to pop  # element from stack in  # LIFO order  print('\nElements popped from the stack')  print(stack.get())  print(stack.get())  print(stack.get())    print("\nEmpty: ", stack.empty()) |

**Output**

0

Full: True

Size: 3

Elements popped from the stack

c

b

a

Empty: True

Queue:

Like stack, queue is a linear data structure that stores items in First In First Out (FIFO) manner.

Operations associated with queue are: 

* **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity : O(1)
* **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity : O(1)
* **Front:** Get the front item from queue – Time Complexity : O(1)
* **Rear:** Get the last item from queue – Time Complexity : O(1)

## Implementation

* list
* collections.deque
* queue.Queue

List implementation :

|  |
| --- |
| queue **=** []    # Adding elements to the queue  queue.append('a')  queue.append('b')  queue.append('c')    **print**("Initial queue")  **print**(queue)    # Removing elements from the queue  print("\nElements dequeued from queue")  print(queue.pop(0))  **print**(queue.pop(0))  **print**(queue.pop(0))    **print**("\nQueue after removing elements")  print(queue)    # Uncommenting print(queue.pop(0))  # will raise and IndexError  # as the queue is now empty |

**Output:** 

Initial queue

['a', 'b', 'c']

Elements dequeued from queue

a

b

c

Queue after removing elements

[]

#### Implementation using collections.deque

**from** collections **import** deque

# Initializing a queue

q **=** deque()

# Adding elements to a queue

q.append('a')

q.append('b')

q.append('c')

**print**("Initial queue")

print(q)

# Removing elements from a queue

**print**("\nElements dequeued from the queue")

**print**(q.popleft())

print(q.popleft())

print(q.popleft())

**print**("\nQueue after removing elements")

print(q)

Initial queue

deque(['a', 'b', 'c'])

Elements dequeued from the queue

a

b

c

Queue after removing elements

deque([])

#### Implementation using queue.Queue

**from** queue **import** Queue

# Initializing a queue

q **=** Queue(maxsize **=** 3)

# qsize() give the maxsize

# of the Queue

print(q.qsize())

# Adding of element to queue

q.put('a')

q.put('b')

q.put('c')

# Return Boolean for Full

# Queue

**print**("\nFull: ", q.full())

# Removing element from queue

**print**("\nElements dequeued from the queue")

print(q.get())

print(q.get())

**print**(q.get())

# Return Boolean for Empty

# Queue

print("\nEmpty: ", q.empty())

q.put(1)

**print**("\nEmpty: ", q.empty())

print("Full: ", q.full())

**Output:** 

0

Full: True

Elements dequeued from the queue

a

b

c

Empty: True

Empty: False

Full: False

Important Algorithms :

### Bubble Sort

*Sequence: 2, 23, 10, 1*

***First Iteration***

*(****2, 23****, 10, 1) –> (****2, 23****, 10, 1),*

*(2,****23, 10****, 1) –> (2,****10, 23****, 1)*

*(2, 10,****23, 1****) –> (2, 10,****1, 23****)*

***Second Iteration***

*(****2, 10****, 1, 23) –> (****2, 10,****1, 23)*

*(2,****10, 1****, 23) –> (2,****1, 10****, 23)*

***Third Iteration***

*(****2, 1****, 10, 23) –> (****1, 2****, 10, 23)*

# Python3 program for Bubble Sort Algorithm Implementation

**def** bubbleSort(arr):

    n **=** len(arr)

    # For loop to traverse through all

    # element in an array

**for** i **in** range(n):

**for** j **in** range(0, n **-** i **-** 1):

            # Range of the array is from 0 to n-i-1

            # Swap the elements if the element found

            #is greater than the adjacent element

**if** arr[j] > arr[j **+** 1]:

                arr[j], arr[j **+** 1] **=** arr[j **+** 1], arr[j]

# Driver code

# Example to test the above code

arr **=** [ 2, 1, 10, 23 ]

bubbleSort(arr)

print("Sorted array is:")

**for** i **in** range(len(arr)):

    print("%d" **%** arr[i])

**Time : Complexity:**O(n2)  
**Auxiliary Space:**O(1)

### Selection Sort

*Sequence: 7, 2, 1, 6*

*(7, 2****, 1****, 6) –> (****1****, 7, 2, 6), In the first traverse it finds the minimum element(i.e., 1) and it is placed at 1st position.*

*(1, 7,****2****, 6) –> (1,****2****, 7, 6), In the second traverse it finds the 2nd minimum element(i.e., 2) and it is placed at 2nd position.*

*(1, 2****, 7, 6****) –> (1, 2,****6, 7****), In the third traverse it finds the next minimum element(i.e., 6) and it is placed at 3rd position.*

*After the above iterations, the final array is in sorted order, i.e., 1, 2, 6, 7.*

# Selection Sort algorithm in Python

**def** selectionSort(array, size):

**for** s **in** range(size):

        min\_idx **=** s

**for** i **in** range(s **+** 1, size):

            # For sorting in descending order

            # for minimum element in each loop

**if** array[i] < array[min\_idx]:

                min\_idx **=** i

        # Arranging min at the correct position

        (array[s], array[min\_idx]) **=** (array[min\_idx], array[s])

# Driver code

data **=** [ 7, 2, 1, 6 ]

size **=** len(data)

selectionSort(data, size)

print('Sorted Array in Ascending Order is :')

print(data)

Sorted Array in Ascending Order is :

[1, 2, 6, 7]

**Time Complexity:** O(n2)  
**Auxiliary Space:** O(1)

### Insertion Sort

*equence: 7, 2, 1, 6*

*(****7, 2****, 1, 6) –> (****2, 7****, 1, 6), In the first iteration, the first 2 elements are compared, here 2 is less than 7 so insert 2 before 7.*

*(2,****7, 1****, 6) –> (2,****1, 7****, 6), In the second iteration the 2nd and 3rd elements are compared, here 1 is less than 7 so insert 1 before 7.*

*(****2, 1****, 7, 6) –> (****1, 2****, 7, 6), After the second iteration (1, 7) elements are not in ascending order so first these two elements are arranged. So, insert 1 before 2.*

*(1, 2,****7, 6****) –> (1, 2,****6, 7****), During this iteration the last 2 elements are compared and swapped after all the previous elements are swapped.*

# Creating a function for insertion sort algorithm

**def** insertion\_sort(list1):

        # Outer loop to traverse on len(list1)

**for** i **in** range(1, len(list1)):

            a **=** list1[i]

            # Move elements of list1[0 to i-1],

            # which are greater to one position

            # ahead of their current position

            j **=** i **-** 1

**while** j >**=** 0 **and** a < list1[j]:

                list1[j **+** 1] **=** list1[j]

                j **-=** 1

            list1[j **+** 1] **=** a

**return** list1

# Driver code

list1 **=** [ 7, 2, 1, 6 ]

print("The unsorted list is:", list1)

**print**("The sorted new list is:", insertion\_sort(list1))

The unsorted list is: [7, 2, 1, 6]

The sorted new list is: [1, 2, 6, 7]

**Time Complexity:**O(n2)  
**Auxiliary Space:**O(1)

# Merge Sort

Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves.

**def** merge(arr, l, m, r):

    n1 **=** m **-** l **+** 1

    n2 **=** r **-** m

    L **=** [0] **\*** (n1)

    R **=** [0] **\*** (n2)

**for** i **in** range(0, n1): L[i] **=** arr[l **+** i]

**for** j **in** range(0, n2): R[j] **=** arr[m **+** 1 **+** j]

i, j, k = 0, 0, l

**while** i < n1 **and** j < n2:

**if** L[i] <**=** R[j]:

            arr[k] **=** L[i]

            i **+=** 1

**else**:

            arr[k] **=** R[j]

            j **+=** 1

        k **+=** 1

**while** i < n1:

        arr[k] **=** L[i]

        i **+=** 1

        k **+=** 1

**while** j < n2:

        arr[k] **=** R[j]

        j **+=** 1

        k **+=** 1

**def** mergeSort(arr, l, r):

**if** l < r:

        m **=** l**+**(r**-**l)**//**2

        mergeSort(arr, l, m)

        mergeSort(arr, m**+**1, r)

        merge(arr, l, m, r)

arr **=** [12, 11, 13, 5, 6, 7]

n **=** len(arr)

print("Given array is")

**for** i **in** range(n):

**print**("%d" **%** arr[i],end**=**" ")

mergeSort(arr, 0, n**-**1)

print("\n\nSorted array is")

**for** i **in** range(n):

    print("%d" **%** arr[i],end**=**" "

**Output**

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**Time Complexity:** O(n\*log(n))

**Auxiliary Space:** O(n)

# QuickSort

# Function to find the partition position

**def** partition(array, low, high):

    # choose the rightmost element as pivot

    pivot **=** array[high]

    # pointer for greater element

    i **=** low

    # traverse through all elements

    # compare each element with pivot

**for** j **in** range(low, high):

**if** array[j] <**=** pivot:

            # If element smaller than pivot is found

            # swap it with the greater element pointed by i

            # Swapping element at i with element at j

            (array[i], array[j]) **=** (array[j], array[i])

i=i+1

    # Swap the pivot element with the greater element specified by i

    (array[i **+** 1], array[high]) **=** (array[high], array[i **+** 1])

    # Return the position from where partition is done

**return** i **+** 1

# function to perform quicksort

**def** quickSort(array, low, high):

**if** low < high:

        # Find pivot element such that

        # element smaller than pivot are on the left

        # element greater than pivot are on the right

        pi **=** partition(array, low, high)

        # Recursive call on the left of pivot

        quickSort(array, low, pi **-** 1)

        # Recursive call on the right of pivot

        quickSort(array, pi **+** 1, high)

data **=** [1, 7, 4, 1, 10, 9, **-**2]

**print**("Unsorted Array")

**print**(data)

size **=** len(data)

quickSort(data, 0, size **-** 1)

**print**('Sorted Array in Ascending Order:')

print(data)

**Output**

Unsorted Array

[1, 7, 4, 1, 10, 9, -2]

Sorted Array in Ascending Order:

[-2, 1, 1, 4, 7, 9, 10]

**Time Complexity:**Worst case time complexity is O(N2) and average case time complexity is O(N log N)  
**Auxiliary Space:**O(1)