**UNIT 2**

**Built-in Data Types**

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes. In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

|  |  |
| --- | --- |
| Text Type: | Str |
| Numeric Types: | int, float, complex |
| Sequence Types: | list, tuple, range |
| Mapping Type: | Dict |
| Set Types: | set, |
| Boolean Type: | Bool |
| Binary Types: | bytes, bytearray, memoryview |



**Variable**

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

a = 5

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type. Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Operator:

Operators are the constructs which can manipulate the value of operands. Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operators:

Python language supports the following types of operators.

* Arithmetic Operators
* Comparison (Relational) Operators
* Assignment Operators
* Logical Operators
* Bitwise Operators
* Membership Operators
* Identity Operators

Let us have a look on all operators one by one.

## Python Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + Addition | Adds values on either side of the operator. | a + b = 30 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -10 |
| \* Multiplication | Multiplies values on either side of the operator | a \* b = 200 |
| / Division | Divides left hand operand by right hand operand | b / a = 2 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 0 |
| \*\* Exponent | Performs exponential (power) calculation on operators | a\*\*b =10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) − | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0 |

## Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | If the values of two operands are equal, then the condition becomes true. | (a == b) is not true. |
| != | If values of two operands are not equal, then condition becomes true. | (a != b) is true. |
| <> | If values of two operands are not equal, then condition becomes true. | (a <> b) is true. This is similar to != operator. |
| > | If the value of left operand is greater than the value of right operand, then condition becomes true. | (a > b) is not true. |
| < | If the value of left operand is less than the value of right operand, then condition becomes true. | (a < b) is true. |
| >= | If the value of left operand is greater than or equal to the value of right operand, then condition becomes true. | (a >= b) is not true. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

## Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

[ [Show Example](https://www.tutorialspoint.com/python/assignment_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assigns values from right side operands to left side operand | c = a + b assigns value of a + b into c |
| += Add AND | It adds right operand to the left operand and assign the result to left operand | c += a is equivalent to c = c + a |
| -= Subtract AND | It subtracts right operand from the left operand and assign the result to left operand | c -= a is equivalent to c = c - a |
| \*= Multiply AND | It multiplies right operand with the left operand and assign the result to left operand | c \*= a is equivalent to c = c \* a |
| /= Divide AND | It divides left operand with the right operand and assign the result to left operand | c /= a is equivalent to c = c / a |
| %= Modulus AND | It takes modulus using two operands and assign the result to left operand | c %= a is equivalent to c = c % a |
| \*\*= Exponent AND | Performs exponential (power) calculation on operators and assign value to the left operand | c \*\*= a is equivalent to c = c \*\* a |
| //= Floor Division | It performs floor division on operators and assign value to the left operand | c //= a is equivalent to c = c // a |

## Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands −

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

There are following Bitwise operators supported by Python language

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & Binary AND | Operator copies a bit to the result if it exists in both operands | (a & b) (means 0000 1100) |
| | Binary OR | It copies a bit if it exists in either operand. | (a | b) = 61 (means 0011 1101) |
| ^ Binary XOR | It copies the bit if it is set in one operand but not both. | (a ^ b) = 49 (means 0011 0001) |
| ~ Binary Ones Complement | It is unary and has the effect of 'flipping' bits. | (~a ) = -61 (means 1100 0011 in 2's complement form due to a signed binary number. |
| << Binary Left Shift | The left operands value is moved left by the number of bits specified by the right operand. | a << 2 = 240 (means 1111 0000) |
| >> Binary Right Shift | The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 = 15 (means 0000 1111) |

## Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

[ [Show Example](https://www.tutorialspoint.com/python/logical_operators_example.htm) ]

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and Logical AND | If both the operands are true then condition becomes true. | (a and b) is true. |
| or Logical OR | If any of the two operands are non-zero then condition becomes true. | (a or b) is true. |
| not Logical NOT | Used to reverse the logical state of its operand. | Not(a and b) is false. |

## Python Membership Operators

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| In | Evaluates to true if it finds a variable in the specified sequence and false otherwise. | x in y, here in results in a 1 if x is a member of sequence y. |
| not in | Evaluates to true if it does not finds a variable in the specified sequence and false otherwise. | x not in y, here not in results in a 1 if x is not a member of sequence y. |

## Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| Is | Evaluates to true if the variables on either side of the operator point to the same object and false otherwise. | x is y, here **is** results in 1 if id(x) equals id(y). |
| is not | Evaluates to false if the variables on either side of the operator point to the same object and true otherwise. | x is not y, here **is not** results in 1 if id(x) is not equal to id(y). |

## Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

[ [Show Example](https://www.tutorialspoint.com/python/operators_precedence_example.htm) ]

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | **\*\***  Exponentiation (raise to the power) |
| 2 | **~ + -**  Complement, unary plus and minus (method names for the last two are +@ and -@) |
| 3 | **\* / % //**  Multiply, divide, modulo and floor division |
| 4 | **+ -**  Addition and subtraction |
| 5 | **>> <<**  Right and left bitwise shift |
| 6 | **&**  Bitwise 'AND' |
| 7 | **^ |**  Bitwise exclusive `OR' and regular `OR' |
| 8 | **<= < > >=**  Comparison operators |
| 9 | **<> == !=**  Equality operators |
| 10 | **= %= /= //= -= += \*= \*\*=**  Assignment operators |
| 11 | **is is not**  Identity operators |
| 12 | **in not in**  Membership operators |
| 13 | **not or and**  Logical operators |

An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The *else* statement is an optional statement and there could be at most only one **else** statement following **if**.

Syntax

The syntax of the *if...else* statement is −

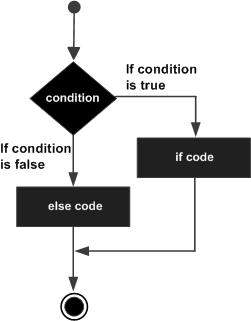
if expression:

statement(s)

else:

statement(s)

Flow Diagram



What are Mutable Data Types?

Anything is said to be mutable when anything can be modified or changed. The term "mutable" in Python refers to an object's capacity to modify its values. These are frequently the things that hold a data collection.

What are Immutable Data Types?

Immutable refers to a state in which no change can occur over time. A Python object is referred to as immutable if we cannot change its value over time. The value of these Python objects is fixed once they are made.

List of Mutable and Immutable objects

**Python mutable data types:**

* Lists
* Dictionaries
* Sets
* User-Defined Classes (It depends on the user to define the characteristics of the classes)

**Python immutable data types:**

* Numbers (Integer, Float, Complex, Decimal, Rational & Booleans)
* Tuples
* Strings

The Python id() Function

When you define a Python object, the program sets a memory section aside. Every Python object has a distinct address that informs the application where the item is located in memory. Every object in Python has a distinct ID connected to the object's memory address. Use the built-in Python id() function to read the special ID.

Let's read the position of a sample string object in memory, for instance:

**Code**

1. # Python program to show how to use the id function
2. # Initializing a string object
3. string = "string"
4. # Printing the id of the string object
5. **print**(id(string))

**Output:**

140452604995952

Immutable in Python

As we have already told, an immutable Python object cannot be altered.

In Python, the integer data type is a prime case of an immutable object or data type. Let's conduct an easy experiment to understand this. Let's first make two integer variables, x and y. The variable y is a reference to the variable x:

Now, x and y both point towards the same location in memory. In other words, both integer variables should have the same ID returned by the id() function. Let's confirm that this is true:

**Code**

1. # Python program to create two variables having the same memory reference
2. # Initializing a variable
3. x = 3
4. # Storing it in another variable
5. y = x
6. # Checking if two have the same id reference
7. **print**("x and y have same id: ", id(x) == id(y))

**Output:**

x and y have same id: True

As a result, the variables x and y points to a single integer object. In other words, even though there is just one integer variable, two variables refer to it.

Let us change x's value now. Next, let's compare the reference ids of x and y once more:

**Code**

1. # Python program to check if x and y have the same ids after changing the value
2. x = 3
3. y = x
4. # Changing the value of x
5. x = 13
6. # Checking if x and y still point to the same memory location
7. **print**("x and y have the same ids: ", id(x) == id(y))

**Output:**

x and y have the same ids: False

This is because x now points to a distinct integer object. Therefore, the integer variable 3 itself remained constant. However, the variable x that previously referred to it now directs users to the new integer entity 13.

Therefore, even though it appears like we modified the original integer variable, we didn't. In Python, the integer object is an immutable data type, meaning that an integer object cannot be changed once created.

Let's conduct a similar test once more using a mutable object.

Create a list, for instance, and put it in a second variable. Then, compare the reference IDs of the two lists. Let's then make some changes to the list we created. Then, let's see if the reference IDs of both lists still coincide:

**Code**

1. # Python program to check if the lists pointing to the same memory location will have the same reference ids after modifying one of the lists
2. # Creating a Python list object
3. num = [2, 4, 6, 8]
4. # Storing the list in another variable
5. l = num
6. # Checking if the two variables point to the same memory location
7. **print**("Both list variables have the same ids: ", id(num) == id(l))
8. # Modifying the num list
9. num.append(10)
10. # Checking if, after modifying num, both the lists point to the same memory location
11. **print**("Both list variables have the same ids: ", id(num) == id(l))

**Output:**

Both list variables have the same ids: True

Both list variables have the same ids: True

Furthermore, their reference IDs match. As a result, the list objects num and l point to the memory location. This demonstrates that we could explicitly change the list object by adding one more element. The list data structures must therefore be mutable. And Python lists operate in this way.

Example of Mutable Objects in Python

List

As a result of their mutable nature, lists can change their contents by incorporating the assignment operators or the indexing operators.

Let's look at an example of that.

**Code**

1. # Python program to show that a list is a mutable data type
2. # Creating a list
3. list1 = ['Python', 'Java', 23, False, 5.3]
4. **print**("The original list: ", list1)
5. # Changing the value at index 2 of the list
6. list1[2]='changed'
7. **print**("The modified list: ", list1)

**Output:**

The original list: ['Python', 'Java', 23, False, 5.3]

The modified list: ['Python', 'Java', 'changed', False, 5.3]

Dictionary

Due to the mutability of dictionaries, we can modify them by implementing a built-in function update or using keys as an index.

Let's look at an illustration of that.

**Code**

1. # Python program to show that a dictionary is a mutable data type
2. # Creating a dictionary
3. dict\_ = {1: "a", 2: "b", 3: "c"}
4. **print**("The original dictionary: ", dict\_)
5. # Changing the value of one of the keys of the dictionary
6. dict\_[2]= 'changed'
7. **print**("The modified dictionary: ", dict\_)

**Output:**

The original dictionary: {1: 'a', 2: 'b', 3: 'c'}

The modified dictionary: {1: 'a', 2: 'changed', 3: 'c'}

Set

Due to the mutability of sets, we can modify them using a built-in function (update).

**Code**

1. # Python program to show that a set is a mutable data type
2. # Creating a set
3. set\_ = {1, 2, 3, 4}
4. **print**("The original set: ", set\_)
5. # Updating the set using the update function
6. update\_set = {'a', 'b', 'c'}
7. set\_.update(update\_set)
8. **print**("The modified set: ", set\_)

**Output:**

The original set: {1, 2, 3, 4}

The modified set: {1, 2, 3, 4, 'b', 'a', 'c'}

Example of Immutable Python Objects

int

Since int in Python is an immutable data type, we cannot change or update it.

As we previously learned, immutable objects shift their memory address whenever they are updated.

Here is an illustration of that:

**Code**

1. # Python program to show that int is an immutable data type
2. int\_ = 25
3. **print**('The memory address of int before updating: ', id(int\_))
4. # Modifying an int object by giving a new value to it
5. int\_ = 35
6. **print**('The memory address of int after updating: ', id(int\_))

**Output:**

The memory address of int before updating: 11531680

The memory address of int after updating: 11532000

float

Since the float object in Python is an immutable data type, we cannot alter or update it. As we previously learned, immutable objects shift their memory address whenever they are updated.

Here is an illustration of that:

**Code**

1. # Python program to show that float is an immutable data type
3. float\_ = float(34.5)
4. **print**('The memory address of float before updating: ', id(float\_))
6. # Modifying the float object by giving a new value to it
7. float\_ = float(32.5)
8. **print**('The memory address of float after updating: ', id(float\_))

**Output:**

The memory address of float before updating: 139992739659504

The memory address of float after updating: 139992740128048

String

Since strings in Python are immutable data structures, we cannot add or edit any data. We encountered warnings stating that strings are not changeable when modifying any section of the string.

Here is an illustration of that:

**Code**

1. # Python program to show that a string is an immutable data type
2. # Creating a string object
3. string = 'hello peeps'
4. # Trying to modify the string object
5. string[0] = 'X'
6. **print**(string)

**Output:**

TypeError Traceback (most recent call last)

<ipython-input-3-4e0fff91061f> in <module>

3 string = 'hello peeps'

4

----> 5 string[0] = 'X'

6

7 print(string)

TypeError: 'str' object does not support item assignment

Tuple

Because tuples in Python are immutable by nature, we are unable to add or modify any of their contents. Here is an illustration of that:

**Code**

1. # Python program to show that a tuple is an immutable data type
2. # Creating a tuple object
3. tuple\_ = (2, 3, 4, 5)
4. # Trying to modify the tuple object
5. tuple\_[0] = 'X'
6. **print**(tulple\_)

**Output:**

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-5-e011ebc4971e> in <module>

5

6 # Trying to modify the tuple object

----> 7 tuple\_[0] = 'X'

8 print(tulple\_)

9

TypeError: 'tuple' object does not support item assignment

**LIST**

**List Declaration**

**Code**

# a simple list

list1 = [1, 2, "Python", "Program", 15.9]

list2 = ["Amy", "Ryan", "Henry", "Emma"]

# printing the list

**print**(list1)

**print**(list2)

# printing the type of list

**print**(type(list1))

**print**(type(list2))

**Output:**

[1, 2, 'Python', 'Program', 15.9]

['Amy', 'Ryan', 'Henry', 'Emma']

< class ' list ' >

< class ' list ' >

Characteristics of Lists

The characteristics of the List are as follows:

* The lists are in order.
* The list element can be accessed via the index.
* The mutable type of List is
* The number of various elements can be stored in a list.

**Ordered List Checking**

**Code**

1. # example
2. a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6 ]
3. b = [ 1, 2, 5, "Ram", 3.50, "Rahul", 6 ]
4. print (a == b)

**Output:**

False

The indistinguishable components were remembered for the two records; however, the subsequent rundown changed the file position of the fifth component, which is against the rundowns' planned request. False is returned when the two lists are compared.

**Code**

1. # example
2. a = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6]
3. b = [ 1, 2, "Ram", 3.50, "Rahul", 5, 6]
4. a == b

**Output:**

True

Records forever protect the component's structure. Because of this, it is an arranged collection of things.

Let's take a closer look at the list example.

**Code**

# list example in detail

emp = [ "John", 102, "USA"]

Dep1 = [ "CS",10]

Dep2 = [ "IT",11]

HOD\_CS = [ 10,"Mr. Holding"]

HOD\_IT = [11, "Mr. Bewon"]

**print**("printing employee data ...")

**print**(" Name : %s, ID: %d, Country: %s" %(emp[0], emp[1], emp[2]))

**print**("printing departments ...")

**print**("Department 1:\nName: %s, ID: %d\n Department 2:\n Name: %s, ID: %s"%( Dep1[0], Dep2[1], Dep2[0], Dep2[1]))

**print**("HOD Details ....")

**print**("CS HOD Name: %s, Id: %d" %(HOD\_CS[1], HOD\_CS[0]))

**print**("IT HOD Name: %s, Id: %d" %(HOD\_IT[1], HOD\_IT[0]))

**print**(type(emp), type(Dep1), type(Dep2), type(HOD\_CS), type(HOD\_IT))

**Output:**

printing employee data...

Name : John, ID: 102, Country: USA

printing departments...

Department 1:

Name: CS, ID: 11

Department 2:

Name: IT, ID: 11

HOD Details ....

CS HOD Name: Mr. Holding, Id: 10

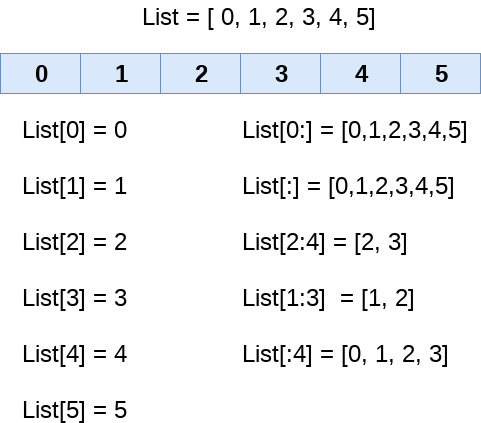
IT HOD Name: Mr. Bewon, Id: 11

<class ' list '> <class ' list '> <class ' list '> <class ' list '> <class ' list '>

**List Indexing and Splitting**

The indexing procedure is carried out similarly to string processing. The slice operator [] can be used to get to the List's components.

The index ranges from 0 to length -1. The 0th index is where the List's first element is stored; the 1st index is where the second element is stored, and so on.



We can get the sub-list of the list using the following syntax.

1. list\_varible(start:stop:step)

* The beginning indicates the beginning record position of the rundown.
* The stop signifies the last record position of the rundown.
* Within a start, the step is used to skip the nth element: stop.

The start parameter is the initial index, the step is the ending index, and the value of the end parameter is the number of elements that are "stepped" through. The default value for the step is one without a specific value. Inside the resultant Sub List, the same with record start would be available, yet the one with the file finish will not. The first element in a list appears to have an index of zero.

Consider the following example:

**Code**

list = [1,2,3,4,5,6,7]

**print**(list[0])

**print**(list[1])

**print**(list[2])

**print**(list[3])

# Slicing the elements

**print**(list[0:6])

# By default, the index value is 0 so its starts from the 0th element and go for index -1.

**print**(list[:])

**print**(list[2:5])

**print**(list[1:6:2])

**Output:**

1

2

3

4

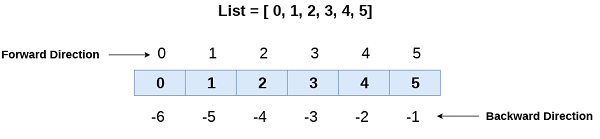
[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 7]

[3, 4, 5]

[2, 4, 6]

In contrast to other programming languages, Python lets you use negative indexing as well. The negative indices are counted from the right. The index -1 represents the final element on the List's right side, followed by the index -2 for the next member on the left, and so on, until the last element on the left is reached.



Let's have a look at the following example where we will use negative indexing to access the elements of the list.

**Code**

# negative indexing example

list = [1,2,3,4,5]

**print**(list[-1])

**print**(list[-3:])

**print**(list[:-1])

**print**(list[-3:-1])

**Output:**

5

[3, 4, 5]

[1, 2, 3, 4]

[3, 4]

Negative indexing allows us to obtain an element, as previously mentioned. The rightmost item in the List was returned by the first print statement in the code above. The second print statement returned the sub-list, and so on.

**Updating List Values**

Due to their mutability and the slice and assignment operator's ability to update their values, lists are Python's most adaptable data structure. Python's append () and insert () methods can also add values to a list.

Consider the following example to update the values inside the List.

**Code**

# updating list values

list = [1, 2, 3, 4, 5, 6]

**print**(list)

# It will assign value to the value to the second index

list[2] = 10

**print**(list)

# Adding multiple-element

list[1:3] = [89, 78]

**print**(list)

# It will add value at the end of the list

list[-1] = 25

**print**(list)

**Output:**

[1, 2, 3, 4, 5, 6]

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

[1, 89, 78, 4, 5, 25]

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove ()** method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

**Code**

list = [1, 2, 3, 4, 5, 6]

**print**(list)

# It will assign value to the value to second index

list[2] = 10

**print**(list)

# Adding multiple element

list[1:3] = [89, 78]

**print**(list)

# It will add value at the end of the list

list[-1] = 25

**print**(list)

**Output:**

[1, 2, 3, 4, 5, 6]

[1, 2, 10, 4, 5, 6]

[1, 89, 78, 4, 5, 6]

[1, 89, 78, 4, 5, 25]

**Python List Operations**

The concatenation (+) and repetition (\*) operators work in the same way as they were working with the strings. The different operations of list are

1. Repetition
2. Concatenation
3. Length
4. Iteration
5. Membership

We check how the list responds to various operators.

1. Repetition

The redundancy administrator empowers the rundown components to be rehashed on different occasions.

**Code**

# repetition of list

# declaring the list

list1 = [12, 14, 16, 18, 20]

# repetition operator \*

l = list1 \* 2

**print**(l)

**Output:**

[12, 14, 16, 18, 20, 12, 14, 16, 18, 20]

2. Concatenation

It concatenates the list mentioned on either side of the operator.

**Code**

# concatenation of two lists

# declaring the lists

list1 = [12, 14, 16, 18, 20]

list2 = [9, 10, 32, 54, 86]

# concatenation operator +

l = list1 + list2

**print**(l)

**Output:**

[12, 14, 16, 18, 20, 9, 10, 32, 54, 86]

3. Length

It is used to get the length of the list

**Code**

# size of the list

# declaring the list

list1 = [12, 14, 16, 18, 20, 23, 27, 39, 40]

# finding length of the list

len(list1)

**Output:**

9

4. Iteration

The for loop is used to iterate over the list elements.

**Code**

# iteration of the list

# declaring the list

list1 = [12, 14, 16, 39, 40]

# iterating

**for** i **in** list1:

**print**(i)

**Output:**

12

14

16

39

40

5. Membership

It returns true if a particular item exists in a particular list otherwise false.

**Code**

# membership of the list

# declaring the list

list1 = [100, 200, 300, 400, 500]

# true will be printed if value exists

# and false if not

**print**(600 **in** list1)

**print**(700 **in** list1)

**print**(1040 **in** list1)

**print**(300 **in** list1)

**print**(100 **in** list1)

**print**(500 **in** list1)

**Output:**

False

False

False

True

True

True

**Iterating a List**

A list can be iterated by using a for - in loop. A simple list containing four strings, which can be iterated as follows.

**Code**

# iterating a list

list = ["John", "David", "James", "Jonathan"]

**for** i **in** list:

    # The i variable will iterate over the elements of the List and contains each element in each iteration.

**print**(i)

**Output:**

John

David

James

Jonathan

**Adding Elements to the List**

The append() function in Python can add a new item to the List. In any case, the annex() capability can enhance the finish of the rundown.

Consider the accompanying model, where we take the components of the rundown from the client and print the rundown on the control center.

**Code**

#Declaring the empty list

l =[]

#Number of elements will be entered by the user

n = int(input("Enter the number of elements in the list:"))

# for loop to take the input

**for** i **in** range(0,n):

  # The input is taken from the user and added to the list as the item

  l.append(input("Enter the item:"))

**print**("printing the list items..")

# traversal loop to print the list items

**for** i **in** l:

**print**(i, end = "  ")

**Output:**

Enter the number of elements in the list:10

Enter the item:32

Enter the item:56

Enter the item:81

Enter the item:2

Enter the item:34

Enter the item:65

Enter the item:09

Enter the item:66

Enter the item:12

Enter the item:18

printing the list items..

32 56 81 2 34 65 09 66 12 18

**Removing Elements from the List**

The remove() function in Python can remove an element from the List. To comprehend this idea, look at the example that follows.

**Example -**

**Code**

list = [0,1,2,3,4]

**print**("printing original list: ");

**for** i **in** list:

**print**(i,end=" ")

list.remove(2)

**print**("\nprinting the list after the removal of first element...")

**for** i **in** list:

**print**(i,end=" ")

**Output:**

printing original list:

0 1 2 3 4

printing the list after the removal of first element...

0 1 3 4

**Python List Built-in Functions**

Python provides the following built-in functions, which can be used with the lists.

1. len()
2. max()
3. min()

**len( )**

**It is used to calculate the length of the list.**

**Code**

# size of the list

# declaring the list

list1 = [12, 16, 18, 20, 39, 40]

# finding length of the list

len(list1)

**Output:**

6

**Max( )**

**It returns the maximum element of the list**

**Code**

# maximum of the list

list1 = [103, 675, 321, 782, 200]

# large element in the list

**print**(max(list1))

**Output:**

782

**Min( )**

**It returns the minimum element of the list**

**Code**

# minimum of the list

list1 = [103, 675, 321, 782, 200]

# smallest element in the list

**print**(min(list1))

**Output:**

103

Let's have a look at the few list examples.

**Example: 1-** Create a program to eliminate the List's duplicate items.

**Code**

list1 = [1,2,2,3,55,98,65,65,13,29]

# Declare an empty list that will store unique values

list2 = []

**for** i **in** list1:

**if** i **not** **in** list2:

        list2.append(i)

**print**(list2)

**Output:**

[1, 2, 3, 55, 98, 65, 13, 29]

**Example:2-** Compose a program to track down the amount of the component in the rundown.

**Code**

1. list1 = [3,4,5,9,10,12,24]
2. sum = 0
3. **for** i **in** list1:
4. sum = sum+i
5. **print**("The sum is:",sum)

**Output:**

The sum is: 67

In [8]:

**Example: 3-** Compose the program to find the rundowns comprise of somewhere around one normal component.

**Code**

1. list1 = [1,2,3,4,5,6]
2. list2 = [7,8,9,2,10]
3. **for** x **in** list1:
4. **for** y **in** list2:
5. **if** x == y:
6. **print**("The common element is:",x)

**Output:**

The common element is: 2

# Set

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation. we cannot directly access any element of the set by the index

Creating a set:

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

Example 1: Using curly braces

Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}

**print**(Days)

**print**(type (Days))

**print**("looping through the set elements ... ")

**for** i **in** Days:

**print**(i)

**Output:**

{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}

<class 'set'>

looping through the set elements ...

Friday

Tuesday

Monday

Saturday

Thursday

Sunday

Wednesday

Example 2: Using set() method

Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])

**print**(Days)

**print**(type(Days))

**print**("looping through the set elements ... ")

**for** i **in** Days:

**print**(i)

**Output:**

{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}

<class 'set'>

looping through the set elements ...

Friday

Wednesday

Thursday

Saturday

Monday

Tuesday

Sunday

It can contain any type of element such as integer, float, tuple etc. But mutable elements (list, dictionary, set) can't be a member of set. Consider the following example.

# Creating a set which have immutable elements

set1 = {1,2,3, "JavaTpoint", 20.5, 14}

**print**(type(set1))

#Creating a set which have mutable element

set2 = {1,2,3,["Javatpoint",4]}

**print**(type(set2))

**Output:**

<class 'set'>

Traceback (most recent call last)

<ipython-input-5-9605bb6fbc68> in <module>

4

5 #Creating a set which holds mutable elements

----> 6 set2 = {1,2,3,["Javatpoint",4]}

7 print(type(set2))

TypeError: unhashable type: 'list'

In the above code, we have created two sets, the set **set1** have immutable elements and set2 have one mutable element as a list. While checking the type of set2, it raised an error, which means set can contain only immutable elements.

Creating an empty set is a bit different because empty curly {} braces are also used to create a dictionary as well. So Python provides the set() method used without an argument to create an empty set.

# Empty curly braces will create dictionary

set3 = {}

**print**(type(set3))

# Empty set using set() function

set4 = set()

**print**(type(set4))

**Output:**

<class 'dict'>

<class 'set'>

Let's see what happened if we provide the duplicate element to the set.

set5 = {1,2,4,4,5,8,9,9,10}

**print**("Return set with unique elements:",set5)

**Output:**

Return set with unique elements: {1, 2, 4, 5, 8, 9, 10}

In the above code, we can see that **set5** consisted of multiple duplicate elements when we printed it remove the duplicity from the set.

Adding items to the set

Python provides the **add()** method and **update()** method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set. Consider the following example.

Example: 1 - Using add() method

Months = set(["January","February", "March", "April", "May", "June"])

**print**("\nprinting the original set ... ")

**print**(months)

**print**("\nAdding other months to the set...")

Months.add("July")

Months.add ("August")

**print**("\nPrinting the modified set...")

**print**(Months)

**print**("\nlooping through the set elements ... ")

**for** i **in** Months:

**print**(i)

**Output:**

printing the original set ...

{'February', 'May', 'April', 'March', 'June', 'January'}

Adding other months to the set...

Printing the modified set...

{'February', 'July', 'May', 'April', 'March', 'August', 'June', 'January'}

looping through the set elements ...

February

July

May

April

March

August

June

January

To add more than one item in the set, Python provides the **update()** method. It accepts iterable as an argument.

Consider the following example.

AD

Example - 2 Using update() function

Months = ({"January","February", "March", "April", "May", "June"})

**print**("\nprinting the original set ... ")

**print**(Months)

**print**("\nupdating the original set ... ")

Months.update({"July","August","September","October"});

**print**("\nprinting the modified set ... ")

**print**(Months);

**Output:**

printing the original set ...

{'January', 'February', 'April', 'May', 'June', 'March'}

updating the original set ...

printing the modified set ...

{'January', 'February', 'April', 'August', 'October', 'May', 'June', 'July', 'September', 'March'}

Removing items from the set

Python provides the **discard()** method and **remove()** method which can be used to remove the items from the set. The difference between these function, using discard() function if the item does not exist in the set then the set remain unchanged whereas remove() method will through an error.

Consider the following example.

Example-1 Using discard() method

months = set(["January","February", "March", "April", "May", "June"])

**print**("\nprinting the original set ... ")

**print**(months)

**print**("\nRemoving some months from the set...");

months.discard("January");

months.discard("May");

**print**("\nPrinting the modified set...");

**print**(months)

**print**("\nlooping through the set elements ... ")

**for** i **in** months:

**print**(i)

**Output:**

printing the original set ...

{'February', 'January', 'March', 'April', 'June', 'May'}

Removing some months from the set...

Printing the modified set...

{'February', 'March', 'April', 'June'}

looping through the set elements ...

February

March

April

June

Python provides also the **remove()** method to remove the item from the set. Consider the following example to remove the items using **remove()** method.

AD

Example-2 Using remove() function

months = set(["January","February", "March", "April", "May", "June"])

**print**("\nprinting the original set ... ")

**print**(months)

**print**("\nRemoving some months from the set...");

months.remove("January");

months.remove("May");

**print**("\nPrinting the modified set...");

**print**(months)

**Output:**

printing the original set ...

{'February', 'June', 'April', 'May', 'January', 'March'}

Removing some months from the set...

Printing the modified set...

{'February', 'June', 'April', 'March'}

We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped from set.

Consider the following example to remove the item from the set using pop() method.

Months = set(["January","February", "March", "April", "May", "June"])

**print**("\nprinting the original set ... ")

**print**(Months)

**print**("\nRemoving some months from the set...");

Months.pop();

Months.pop();

**print**("\nPrinting the modified set...");

**print**(Months)

**Output:**

printing the original set ...

{'June', 'January', 'May', 'April', 'February', 'March'}

Removing some months from the set...

Printing the modified set...

{'May', 'April', 'February', 'March'}

In the above code, the last element of the **Month** set is **March** but the pop() method removed the **June and January** because the set is unordered and the pop() method could not determine the last element of the set.

Python provides the clear() method to remove all the items from the set.

Consider the following example.

Months = set(["January","February", "March", "April", "May", "June"])

**print**("\nprinting the original set ... ")

**print**(Months)

**print**("\nRemoving all the items from the set...");

Months.clear()

**print**("\nPrinting the modified set...")

**print**(Months)

**Output:**

printing the original set ...

{'January', 'May', 'June', 'April', 'March', 'February'}

Removing all the items from the set...

Printing the modified set...

set()

AD

Difference between discard() and remove()

Despite the fact that **discard()** and **remove()** method both perform the same task, There is one main difference between discard() and remove().

If the key to be deleted from the set using discard() doesn't exist in the set, the Python will not give the error. The program maintains its control flow.

On the other hand, if the item to be deleted from the set using remove() doesn't exist in the set, the Python will raise an error.

Consider the following example.

Example-

Months = set(["January","February", "March", "April", "May", "June"])

**print**("\nprinting the original set ... ")

**print**(Months)

**print**("\nRemoving items through discard() method...");

Months.discard("Feb"); #will not give an error although the key feb is not available in the set

**print**("\nprinting the modified set...")

**print**(Months)

**print**("\nRemoving items through remove() method...");

Months.remove("Jan") #will give an error as the key jan is not available in the set.

**print**("\nPrinting the modified set...")

**print**(Months)

**Output:**

printing the original set ...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through discard() method...

printing the modified set...

{'March', 'January', 'April', 'June', 'February', 'May'}

Removing items through remove() method...

Traceback (most recent call last):

File "set.py", line 9, in

Months.remove("Jan")

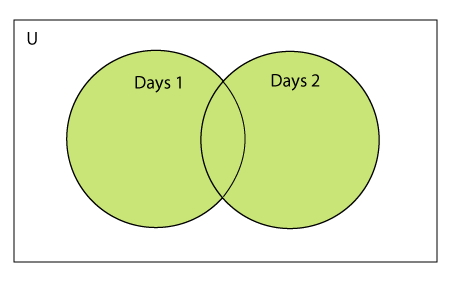
KeyError: 'Jan'

Python Set Operations

Set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.

Union of two Sets

To combine two or more sets into one set in Python, use the union() function. All of the distinctive characteristics from each combined set are present in the final set. As parameters, one or more sets may be passed to the union() function. The function returns a copy of the set supplied as the lone parameter if there is just one set. The method returns a new set containing all the different items from all the arguments if more than one set is supplied as an argument.



Consider the following example to calculate the union of two sets.

**Example 1: using union | operator**

Days1 = {"Monday","Tuesday","Wednesday","Thursday", "Sunday"}

Days2 = {"Friday","Saturday","Sunday"}

**print**(Days1|Days2) #printing the union of the sets

**Output:**

{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday', 'Thursday'}

Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.

**Example 2: using union() method**

Days1 = {"Monday","Tuesday","Wednesday","Thursday"}

Days2 = {"Friday","Saturday","Sunday"}

**print**(Days1.union(Days2)) #printing the union of the sets

**Output:**

{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday', 'Saturday'}

Now, we can also make the union of more than two sets using the union() function, for example:

**Program:**

# Create three sets

set1 = {1, 2, 3}

set2 = {2, 3, 4}

set3 = {3, 4, 5}

# Find the common elements between the three sets

common\_elements = set1.union(set2, set3)

# Print the common elements

**print**(common\_elements)

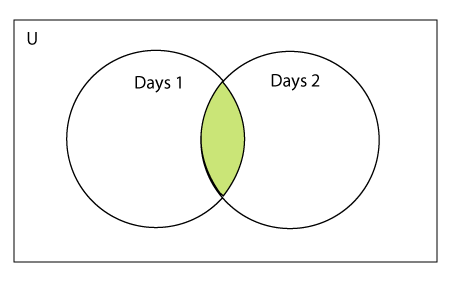
**Output:**

{1, 2, 3, 4, 5}

### The intersection of two sets

To discover what is common between two or more sets in Python, apply the intersection() function. Only the items in all sets being compared are included in the final set. One or more sets can also be used as the intersection() function parameters. The function returns a copy of the set supplied as the lone parameter if there is just one set. The method returns a new set that only contains the elements in all the compared sets if multiple sets are supplied as arguments.

The intersection of two sets can be performed by the **and &** operator or the **intersection() function**. The intersection of the two sets is given as the set of the elements that common in both sets.



Consider the following example.

**Example 1: Using & operator**

Days1 = {"Monday","Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday","Tuesday","Sunday", "Friday"}

**print**(Days1&Days2) #prints the intersection of the two sets

**Output:**

{'Monday', 'Tuesday'}

**Example 2: Using intersection() method**

set1 = {"Devansh","John", "David", "Martin"}

set2 = {"Steve", "Milan", "David", "Martin"}

**print**(set1.intersection(set2)) #prints the intersection of the two sets

**Output:**

{'Martin', 'David'}

**Example 3:**

set1 = {1,2,3,4,5,6,7}

set2 = {1,2,20,32,5,9}

set3 ={1,4,20,10}

set4 =set1.intersection(set2,set3)

**print**(set4)

**Output:**

{1,2,5}

Similarly, as the same as union function, we can perform the intersection of more than two sets at a time,

For Example:

**Program**

# Create three sets

set1 = {1, 2, 3}

set2 = {2, 3, 4}

set3 = {3, 4, 5}

# Find the common elements between the three sets

common\_elements = set1.intersection(set2, set3)

# Print the common elements

**print**(common\_elements)

**Output:**

{3}

## The intersection\_update() method

The **intersection\_update()** method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).

The **intersection\_update()** method is different from the intersection() method since it modifies the original set by removing the unwanted items, on the other hand, the intersection() method returns a new set.

Consider the following example.

a = {"Devansh", "bob", "castle"}

b = {"castle", "dude", "emyway"}

c = {"fuson", "gaurav", "castle"}

a.intersection\_update(b, c)

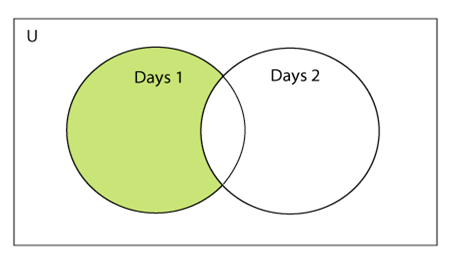
**print**(a)

**Output:**

{'castle'}

## Difference between the two sets

The difference of two sets can be calculated by using the subtraction (-) operator or **intersection()** method. Suppose there are two sets A and B, and the difference is A-B that denotes the resulting set will be obtained that element of A, which is not present in the set B.



Consider the following example.

**Example 1 : Using subtraction ( - ) operator**

Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday", "Tuesday", "Sunday"}

**print**(Days1-Days2) #{"Wednesday", "Thursday" will be printed}

**Output:**

{'Thursday', 'Wednesday'}

**Example 2 : Using difference() method**

Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday", "Tuesday", "Sunday"}

**print**(Days1.difference(Days2)) # prints the difference of the two sets Days1 and Days2

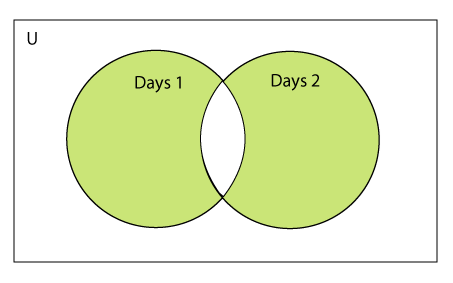
**Output:**

{'Thursday', 'Wednesday'}

## Symmetric Difference of two sets

In Python, the symmetric Difference between set1 and set2 is the set of elements present in one set or the other but not in both sets. In other words, the set of elements is in set1 or set2 but not in their intersection.

The Symmetric Difference of two sets can be computed using Python's symmetric\_difference() method. This method returns a new set containing all the elements in either but not in both. Consider the following example:



**Example - 1: Using ^ operator**

a = {1,2,3,4,5,6}

b = {1,2,9,8,10}

c = a^b

**print**(c)

**Output:**

{3, 4, 5, 6, 8, 9, 10}

**Example - 2: Using symmetric\_difference() method**

a = {1,2,3,4,5,6}

b = {1,2,9,8,10}

c = a.symmetric\_difference(b)

**print**(c)

**Output:**

{3, 4, 5, 6, 8, 9, 10}

## Set comparisons

In Python, you can compare sets to check if they are equal, if one set is a subset or superset of another, or if two sets have elements in common.

Here are the set comparison operators available in Python:

* ==: checks if two sets have the same elements, regardless of their order.
* !=: checks if two sets are not equal.
* <: checks if the left set is a proper subset of the right set (i.e., all elements in the left set are also in the right set, but the right set has additional elements).
* <=: checks if the left set is a subset of the right set (i.e., all elements in the left set are also in the right set).
* >: checks if the left set is a proper superset of the right set (i.e., all elements in the right set are also in the left set, but the left set has additional elements).
* >=: checks if the left set is a superset of the right set (i.e., all elements in the right set are also in the left).

**Consider the following example.**

Days1 = {"Monday",  "Tuesday", "Wednesday", "Thursday"}

Days2 = {"Monday", "Tuesday"}

Days3 = {"Monday", "Tuesday", "Friday"}

#Days1 is the superset of Days2 hence it will print true.

**print** (Days1>Days2)

#prints false since Days1 is not the subset of Days2

**print** (Days1<Days2)

#prints false since Days2 and Days3 are not equivalent

**print** (Days2 == Days3)

**Output:**

True

False

False

### Set Programming Example

**Example - 1:** Write a program to remove the given number from the set.

my\_set = {1,2,3,4,5,6,12,24}

n = int(input("Enter the number you want to remove"))

my\_set.discard(n)

**print**("After Removing:",my\_set)

**Output:**

Enter the number you want to remove:12

After Removing: {1, 2, 3, 4, 5, 6, 24}

**Example - 2:** Write a program to add multiple elements to the set.

set1 = set([1,2,4,"John","CS"])

set1.update(["Apple","Mango","Grapes"])

**print**(set1)

**Output:**

{1, 2, 4, 'Apple', 'John', 'CS', 'Mango', 'Grapes'}

**Example - 3:** Write a program to find the union between two set.

set1 = set(["Peter","Joseph", 65,59,96])

set2  = set(["Peter",1,2,"Joseph"])

set3 = set1.union(set2)

**print**(set3)

**Output:**

{96, 65, 2, 'Joseph', 1, 'Peter', 59}

**Example- 4:** Write a program to find the intersection between two sets.

set1 = {23,44,56,67,90,45,"Javatpoint"}

set2 = {13,23,56,76,"Sachin"}

set3 = set1.intersection(set2)

**print**(set3)

**Output:**

{56, 23}

**Example - 5:** Write the program to add element to the frozenset.

set1 = {23,44,56,67,90,45,"Javatpoint"}

set2 = {13,23,56,76,"Sachin"}

set3 = set1.intersection(set2)

**print**(set3)

**Output:**

TypeError: 'frozenset' object does not support item assignment

Above code raised an error because frozensets are immutable and can't be changed after creation.

**Example - 6:** Write the program to find the issuperset, issubset and superset.

set1 = set(["Peter","James","Camroon","Ricky","Donald"])

set2 = set(["Camroon","Washington","Peter"])

set3 = set(["Peter"])

issubset = set1 >= set2

**print**(issubset)

issuperset = set1 <= set2

**print**(issuperset)

issubset = set3 <= set2

**print**(issubset)

issuperset = set2 >= set3

**print**(issuperset)

**Output:**

False

False

True

True

## Python Built-in set methods

Python contains the following methods to be used with the sets.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | [add(item)](https://www.javatpoint.com/python-set-add-method) | It adds an item to the set. It has no effect if the item is already present in the set. |
| 2 | clear() | It deletes all the items from the set. |
| 3 | copy() | It returns a shallow copy of the set. |
| 4 | difference\_update(....) | It modifies this set by removing all the items that are also present in the specified sets. |
| 5 | [discard(item)](https://www.javatpoint.com/python-set-discard-method) | It removes the specified item from the set. |
| 6 | intersection() | It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified). |
| 7 | intersection\_update(....) | It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified). |
| 8 | Isdisjoint(....) | Return True if two sets have a null intersection. |
| 9 | Issubset(....) | Report whether another set contains this set. |
| 10 | Issuperset(....) | Report whether this set contains another set. |
| 11 | [pop()](https://www.javatpoint.com/python-set-pop-method) | Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty. |
| 12 | [remove(item)](https://www.javatpoint.com/python-set-remove-method) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 13 | symmetric\_difference(....) | Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError. |
| 14 | symmetric\_difference\_update(....) | Update a set with the symmetric difference of itself and another. |
| 15 | union(....) | Return the union of sets as a new set. (i.e. all elements that are in either set.) |
| 16 | update() | Update a set with the union of itself and others. |

# **Python Tuples**

A comma-separated group of items is called a Python triple. The ordering, settled items, and reiterations of a tuple are to some degree like those of a rundown, but in contrast to a rundown, a tuple is unchanging.

The main difference between the two is that we cannot alter the components of a tuple once they have been assigned. On the other hand, we can edit the contents of a list.

**Example**

1. ("Suzuki", "Audi", "BMW"," Skoda ") is a tuple.

### Features of Python Tuple

* Tuples are an immutable data type, meaning their elements cannot be changed after they are generated.
* Each element in a tuple has a specific order that will never change because tuples are ordered sequences.

### Forming a Tuple:

All the objects-also known as "elements"-must be separated by a comma, enclosed in parenthesis (). Although parentheses are not required, they are recommended.

Any number of items, including those with various data types (dictionary, string, float, list, etc.), can be contained in a tuple.

**Code**

1. # Python program to show how to create a tuple
2. # Creating an empty tuple
3. empty\_tuple = ()
4. print("Empty tuple: ", empty\_tuple)
5. # Creating tuple having integers
6. int\_tuple = (4, 6, 8, 10, 12, 14)
7. print("Tuple with integers: ", int\_tuple)
8. # Creating a tuple having objects of different data types
9. mixed\_tuple = (4, "Python", 9.3)
10. print("Tuple with different data types: ", mixed\_tuple)
11. # Creating a nested tuple
12. nested\_tuple = ("Python", {4: 5, 6: 2, 8:2}, (5, 3, 5, 6))
13. print("A nested tuple: ", nested\_tuple)

**Output:**

Empty tuple: ()

Tuple with integers: (4, 6, 8, 10, 12, 14)

Tuple with different data types: (4, 'Python', 9.3)

A nested tuple: ('Python', {4: 5, 6: 2, 8: 2}, (5, 3, 5, 6))

Parentheses are not necessary for the construction of multiples. This is known as triple pressing.

**Code**

1. # Python program to create a tuple without using parentheses
2. # Creating a tuple
3. tuple\_ = 4, 5.7, "Tuples", ["Python", "Tuples"]
4. # Displaying the tuple created
5. print(tuple\_)
6. # Checking the data type of object tuple\_
7. print(type(tuple\_) )
8. # Trying to modify tuple\_
9. **try**:
10. tuple\_[1] = 4.2
11. except:
12. print(TypeError )

**Output:**

(4, 5.7, 'Tuples', ['Python', 'Tuples'])

<class 'tuple'>

<class 'TypeError'>

The development of a tuple from a solitary part may be complex.

Essentially adding a bracket around the component is lacking. A comma must separate the element to be recognized as a tuple.

**Code**

1. # Python program to show how to create a tuple having a single element
2. single\_tuple = ("Tuple")
3. print( type(single\_tuple) )
4. # Creating a tuple that has only one element
5. single\_tuple = ("Tuple",)
6. print( type(single\_tuple) )
7. # Creating tuple without parentheses
8. single\_tuple = "Tuple",
9. print( type(single\_tuple) )

**Output:**

<class 'str'>

<class 'tuple'>

<class 'tuple'>

## Accessing Tuple Elements

A tuple's objects can be accessed in a variety of ways.

**Indexing**

Indexing We can use the index operator [] to access an object in a tuple, where the index starts at 0. The indices of a tuple with five items will range from 0 to 4. An Index Error will be raised assuming we attempt to get to a list from the Tuple that is outside the scope of the tuple record. An index above four will be out of range in this scenario.

Because the index in Python must be an integer, we cannot provide an index of a floating data type or any other type. If we provide a floating index, the result will be TypeError.

The method by which elements can be accessed through nested tuples can be seen in the example below.

**Code**

1. # Python program to show how to access tuple elements
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Collection")
4. print(tuple\_[0])
5. print(tuple\_[1])
6. # trying to access element index more than the length of a tuple
7. **try**:
8. print(tuple\_[5])
9. except Exception as e:
10. print(e)
11. # trying to access elements through the index of floating data type
12. **try**:
13. print(tuple\_[1.0])
14. except Exception as e:
15. print(e)
16. # Creating a nested tuple
17. nested\_tuple = ("Tuple", [4, 6, 2, 6], (6, 2, 6, 7))
18. # Accessing the index of a nested tuple
19. print(nested\_tuple[0][3])
20. print(nested\_tuple[1][1])

**Output:**

AD

Python

Tuple

tuple index out of range

tuple indices must be integers or slices, not float

l

6

**Negative Indexing**: Python's sequence objects support negative indexing.

The last thing of the assortment is addressed by - 1, the second last thing by - 2, etc.

**Code**

1. # Python program to show how negative indexing works in Python tuples
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Collection")
4. # Printing elements using negative indices
5. print("Element at -1 index: ", tuple\_[-1])
6. print("Elements between -4 and -1 are: ", tuple\_[-4:-1])

**Output:**

Element at -1 index: Collection

Elements between -4 and -1 are: ('Python', 'Tuple', 'Ordered')

## Slicing

Tuple slicing is a common practice in Python and the most common way for programmers to deal with practical issues. Look at a tuple in Python. Slice a tuple to access a variety of its elements. Using the colon as a straightforward slicing operator (:) is one strategy.

To gain access to various tuple elements, we can use the slicing operator colon (:).

**Code**

1. # Python program to show how slicing works in Python tuples
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
4. # Using slicing to access elements of the tuple
5. print("Elements between indices 1 and 3: ", tuple\_[1:3])
6. # Using negative indexing in slicing
7. print("Elements between indices 0 and -4: ", tuple\_[:-4])
8. # Printing the entire tuple by using the **default** start and end values.
9. print("Entire tuple: ", tuple\_[:])

**Output:**

Elements between indices 1 and 3: ('Tuple', 'Ordered')

Elements between indices 0 and -4: ('Python', 'Tuple')

Entire tuple: ('Python', 'Tuple', 'Ordered', 'Immutable', 'Collection', 'Objects')

## Deleting a Tuple

A tuple's parts can't be modified, as was recently said. We are unable to eliminate or remove tuple components as a result.

However, the keyword del can completely delete a tuple.

**Code**

1. # Python program to show how to delete elements of a Python tuple
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Objects")
4. # Deleting a particular element of the tuple
5. **try**:
6. del tuple\_[3]
7. print(tuple\_)
8. except Exception as e:
9. print(e)
10. # Deleting the variable from the global space of the program
11. del tuple\_
12. # Trying accessing the tuple after deleting it
13. **try**:
14. print(tuple\_)
15. except Exception as e:
16. print(e)

**Output:**

'tuple' object does not support item deletion

name 'tuple\_' is not defined

### Repetition Tuples in Python

**Code**

1. # Python program to show repetition in tuples
2. tuple\_ = ('Python',"Tuples")
3. print("Original tuple is: ", tuple\_)
4. # Repeting the tuple elements
5. tuple\_ = tuple\_ \* 3
6. print("New tuple is: ", tuple\_)

**Output:**

Original tuple is: ('Python', 'Tuples')

New tuple is: ('Python', 'Tuples', 'Python', 'Tuples', 'Python', 'Tuples')

### Tuple Methods

Like the list, Python Tuples is a collection of immutable objects. There are a few ways to work with tuples in Python. With some examples, this essay will go over these two approaches in detail.

The following are some examples of these methods.

**Count () Method :**

The times the predetermined component happens in the Tuple is returned by the count () capability of the Tuple.

**Code**

1. # Creating tuples
2. T1 = (0, 1, 5, 6, 7, 2, 2, 4, 2, 3, 2, 3, 1, 3, 2)
3. T2 = ('python', 'java', 'python', 'Tpoint', 'python', 'java')
4. # counting the appearance of 3
5. res = T1.count(2)
6. print('Count of 2 in T1 is:', res)
7. # counting the appearance of java
8. res = T2.count('java')
9. print('Count of Java in T2 is:', res)

**Output:**

Count of 2 in T1 is: 5

Count of java in T2 is: 2

**Index() Method:**

The Index() function returns the first instance of the requested element from the Tuple.

**Parameters:**

* The thing that must be looked for.
* Start: (Optional) the index that is used to begin the final (optional) search: The most recent index from which the search is carried out
* Index Method

**Code**

1. # Creating tuples
2. Tuple\_data = (0, 1, 2, 3, 2, 3, 1, 3, 2)
3. # getting the index of 3
4. res = Tuple\_data.index(3)
5. print('First occurrence of 1 is', res)
6. # getting the index of 3 after 4th
7. # index
8. res = Tuple\_data.index(3, 4)
9. print('First occurrence of 1 after 4th index is:', res)

**Output:**

First occurrence of 1 is 2

First occurrence of 1 after 4th index is: 6

### ****Tuple Membership Test :****

Utilizing the watchword, we can decide whether a thing is available in the given Tuple.

**Code**

1. # Python program to show how to perform membership test **for** tuples
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", "Collection", "Ordered")
4. # In operator
5. print('Tuple' in tuple\_)
6. print('Items' in tuple\_)
7. # Not in operator
8. print('Immutable' not in tuple\_)
9. print('Items' not in tuple\_)

**Output:**

True

False

False

True

### Iterating Through a Tuple

A for loop can be used to iterate through each tuple element.

**Code**

1. # Python program to show how to iterate over tuple elements
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable")
4. # Iterating over tuple elements using a **for** loop
5. **for** item in tuple\_:
6. print(item)

**Output:**

Python

Tuple

Ordered

Immutable

### Changing a Tuple

Tuples, instead of records, are permanent articles. This suggests that once the elements of a tuple have been defined, we cannot change them. However, the nested elements can be altered if the element itself is a changeable data type like a list. Multiple values can be assigned to a tuple through reassignment.

**Code**

1. # Python program to show that Python tuples are immutable objects
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable", [1,2,3,4])
4. # Trying to change the element at index 2
5. **try**:
6. tuple\_[2] = "Items"
7. print(tuple\_)
8. except Exception as e:
9. print( e )
10. # But inside a tuple, we can change elements of a mutable object
11. tuple\_[-1][2] = 10
12. print(tuple\_)
13. # Changing the whole tuple
14. tuple\_ = ("Python", "Items")
15. print(tuple\_)

**Output:**

'tuple' object does not support item assignment

('Python', 'Tuple', 'Ordered', 'Immutable', [1, 2, 10, 4])

('Python', 'Items')

The + operator can be used to combine multiple tuples into one. This phenomenon is known as concatenation. We can also repeat the elements of a tuple a predetermined number of times by using the \* operator. This is already demonstrated above. The aftereffects of the tasks + and \* are new tuples.

**Code**

1. # Python program to show how to concatenate tuples
2. # Creating a tuple
3. tuple\_ = ("Python", "Tuple", "Ordered", "Immutable")
4. # Adding a tuple to the tuple\_
5. print(tuple\_ + (4, 5, 6))

**Output:**

('Python', 'Tuple', 'Ordered', 'Immutable', 4, 5, 6)

### Tuples have the following advantages over lists:

* Triples take less time than lists do.
* Due to tuples, the code is protected from accidental modifications. It is desirable to store non-changing information in "tuples" instead of "records" if a program expects it.
* A tuple can be used as a dictionary key if it contains immutable values like strings, numbers, or another tuple. "Lists" cannot be utilized as dictionary keys because they are mutable.

# **Python Dictionary**

Dictionaries are a useful data structure for storing data in Python because they are capable of imitating real-world data arrangements where a certain value exists for a given key.

The data is stored as key-value pairs using a Python dictionary.

* This data structure is mutable
* The components of dictionary were made using keys and values.
* Keys must only have one component.
* Values can be of any type, including integer, list, and tuple.

A dictionary is, in other words, a group of key-value pairs, where the values can be any Python object. The keys, in contrast, are immutable Python objects, such as strings, tuples, or numbers. Dictionary entries are ordered as of Python version 3.7. In Python 3.6 and before, dictionaries are generally unordered.

**Creating the Dictionary**

Curly brackets are the simplest way to generate a Python dictionary, although there are other approaches as well. With many key-value pairs surrounded in curly brackets and a colon separating each key from its value, the dictionary can be built. (:). The following provides the syntax for defining the dictionary.

**Syntax:**

1. Dict = {"Name": "Gayle", "Age": 25}

In the above dictionary **Dict**, The keys **Name** and **Age** are the strings which comes under the category of an immutable object.

Let's see an example to create a dictionary and print its content.

**Code**

1. Employee = {"Name": "Johnny", "Age": 32, "salary":26000,"Company":"^TCS"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)

**Output**

<class 'dict'>

printing Employee data ....

{'Name': 'Johnny', 'Age': 32, 'salary': 26000, 'Company': TCS}

Python provides the built-in function **dict()** method which is also used to create the dictionary.

The empty curly braces {} is used to create empty dictionary.

**Code**

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")
4. **print**(Dict)
5. # Creating a Dictionary
6. # with dict() method
7. Dict = dict({1: 'Hcl', 2: 'WIPRO', 3:'Facebook'})
8. **print**("\nCreate Dictionary by using  dict(): ")
9. **print**(Dict)
10. # Creating a Dictionary
11. # with each item as a Pair
12. Dict = dict([(4, 'Rinku'), (2, Singh)])
13. **print**("\nDictionary with each item as a pair: ")
14. **print**(Dict)

**Output**

Empty Dictionary:

{}

Create Dictionary by using dict():

{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}

Dictionary with each item as a pair:

{4: 'Rinku', 2: 'Singh'}

## Accessing the dictionary values

To access data contained in lists and tuples, indexing has been studied. The keys of the dictionary can be used to obtain the values because they are unique from one another. The following method can be used to access dictionary values.

**Code**

1. Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**("Name : %s" %Employee["Name"])
5. **print**("Age : %d" %Employee["Age"])
6. **print**("Salary : %d" %Employee["salary"])
7. **print**("Company : %s" %Employee["Company"])

**Output**

ee["Company"])

Output

<class 'dict'>

printing Employee data ....

Name : Dev

Age : 20

Salary : 45000

Company : WIPRO

Python provides us with an alternative to use the get() method to access the dictionary values. It would give the same result as given by the indexing.

AD

## Adding Dictionary Values

The dictionary is a mutable data type, and utilising the right keys allows you to change its values. Dict[key] = value and the value can both be modified. An existing value can also be updated using the update() method.

#### **Note: The value is updated if the key-value pair is already present in the dictionary. Otherwise, the dictionary's newly added keys.**

Let's see an example to update the dictionary values.

### Example - 1:

**Code**

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")
4. **print**(Dict)
5. # Adding elements to dictionary one at a time
6. Dict[0] = 'Peter'
7. Dict[2] = 'Joseph'
8. Dict[3] = 'Ricky'
9. **print**("\nDictionary after adding 3 elements: ")
10. **print**(Dict)
11. # Adding set of values
12. # with a single Key
13. # The Emp\_ages doesn't exist to dictionary
14. Dict['Emp\_ages'] = 20, 33, 24
15. **print**("\nDictionary after adding 3 elements: ")
16. **print**(Dict)
17. # Updating existing Key's Value
18. Dict[3] = 'JavaTpoint'
19. **print**("\nUpdated key value: ")
20. **print**(Dict)

**Output**

Empty Dictionary:

{}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky'}

Dictionary after adding 3 elements:

{0: 'Peter', 2: 'Joseph', 3: 'Ricky', 'Emp\_ages': (20, 33, 24)}

Updated key value:

{0: 'Peter', 2: 'Joseph', 3: 'JavaTpoint', 'Emp\_ages': (20, 33, 24)}

### Example - 2:

**Code**

AD

1. Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Enter the details of the new employee....");
6. Employee["Name"] = input("Name: ");
7. Employee["Age"] = int(input("Age: "));
8. Employee["salary"] = int(input("Salary: "));
9. Employee["Company"] = input("Company:");
10. **print**("printing the new data");
11. **print**(Employee)

**Output**

<class 'dict'>

printing Employee data ....

Employee = {"Name": "Dev", "Age": 20, "salary":45000,"Company":"WIPRO"} Enter the details of the new employee....

Name: Sunny

Age: 38

Salary: 39000

Company:Hcl

printing the new data

{'Name': 'Sunny', 'Age': 38, 'salary': 39000, 'Company': 'Hcl'}

AD

## Deleting Elements using del Keyword

The items of the dictionary can be deleted by using the **del** keyword as given below.

**Code**

1. Employee = {"Name": "David", "Age": 30, "salary":55000,"Company":"WIPRO"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)
5. **print**("Deleting some of the employee data")
6. **del** Employee["Name"]
7. **del** Employee["Company"]
8. **print**("printing the modified information ")
9. **print**(Employee)
10. **print**("Deleting the dictionary: Employee");
11. **del** Employee
12. **print**("Lets try to print it again ");
13. **print**(Employee)

**Output**

<class 'dict'>

printing Employee data ....

{'Name': 'David', 'Age': 30, 'salary': 55000, 'Company': 'WIPRO'}

Deleting some of the employee data

printing the modified information

{'Age': 30, 'salary': 55000}

Deleting the dictionary: Employee

Lets try to print it again

NameError: name 'Employee' is not defined.

The last print statement in the above code, it raised an error because we tried to print the Employee dictionary that already deleted.

## Deleting Elements using pop() Method :

A dictionary is a group of key-value pairs in Python. You can retrieve, insert, and remove items using this unordered, mutable data type by using their keys. The pop() method is one of the ways to get rid of elements from a dictionary. In this post, we'll talk about how to remove items from a Python dictionary using the pop() method.

The value connected to a specific key in a dictionary is removed using the pop() method, which then returns the value. The key of the element to be removed is the only argument needed. The pop() method can be used in the following ways:

**Code**

1. # Creating a Dictionary
2. Dict1 = {1: 'JavaTpoint', 2: 'Educational', 3: 'Website'}
3. # Deleting a key
4. # using pop() method
5. pop\_key = Dict1.pop(2)
6. **print**(Dict1)

**Output**

{1: 'JavaTpoint', 3: 'Website'}

Additionally, Python offers built-in functions popitem() and clear() for removing dictionary items. In contrast to the clear() method, which removes all of the elements from the entire dictionary, popitem() removes any element from a dictionary.

## Iterating Dictionary

A dictionary can be iterated using for loop as given below.

### Example 1

**Code**

1. # for loop to print all the keys of a dictionary
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
3. **for** x **in** Employee:
4. **print**(x)

**Output**

Name

Age

salary

Company

### Example 2

**Code**

1. #for loop to print all the values of the dictionary
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"} **for** x **in** Employee:
3. **print**(Employee[x])

**Output**

John

29

25000

WIPRO

### Example - 3

**Code**

1. #for loop to print the values of the dictionary by using values() method.
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
3. **for** x **in** Employee.values():
4. **print**(x)

**Output**

John

29

25000

WIPRO

### Example 4

**Code**

1. #for loop to print the items of the dictionary by using items() method
2. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"WIPRO"}
3. **for** x **in** Employee.items():
4. **print**(x)

**Output**

('Name', 'John')

('Age', 29)

('salary', 25000)

('Company', 'WIPRO')

## Properties of Dictionary Keys

1. In the dictionary, we cannot store multiple values for the same keys. If we pass more than one value for a single key, then the value which is last assigned is considered as the value of the key.

Consider the following example.

**Code**

1. Employee={"Name":"John","Age":29,"Salary":25000,"Company":"WIPRO","Name":
2. "John"}
3. **for** x,y **in** Employee.items():
4. **print**(x,y)

**Output**

Name John

Age 29

Salary 25000

Company WIPRO

2. The key cannot belong to any mutable object in Python. Numbers, strings, or tuples can be used as the key, however mutable objects like lists cannot be used as the key in a dictionary.

Consider the following example.

**Code**

1. Employee = {"Name": "John", "Age": 29, "salary":26000,"Company":"WIPRO",[100,201,301]:"Department ID"}
2. **for** x,y **in** Employee.items():
3. **print**(x,y)

**Output**

Traceback (most recent call last):

File "dictionary.py", line 1, in

Employee = {"Name": "John", "Age": 29, "salary":26000,"Company":"WIPRO",[100,201,301]:"Department ID"}

TypeError: unhashable type: 'list'

## Built-in Dictionary Functions

A function is a method that can be used on a construct to yield a value. Additionally, the construct is unaltered. A few of the Python methods can be combined with a Python dictionary.

The built-in Python dictionary methods are listed below, along with a brief description.

* **len()**

AD

The dictionary's length is returned via the len() function in Python. The string is lengthened by one for each key-value pair.

**Code**

1. dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
2. len(dict)

**Output**

4

* **any()**

Like how it does with lists and tuples, the any() method returns True indeed if one dictionary key does have a Boolean expression that evaluates to True.

**Code**

1. dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
2. any({'':'','':'','3':''})

**Output**

True

* **all()**

Unlike in any() method, all() only returns True if each of the dictionary's keys contain a True Boolean value.

**Code**

1. dict = {1: "Ayan", 2: "Bunny", 3: "Ram", 4: "Bheem"}
2. all({1:'',2:'','':''})

**Output**

False

* **sorted()**

Like it does with lists and tuples, the sorted() method returns an ordered series of the dictionary's keys. The ascending sorting has no effect on the original Python dictionary.

**Code**

1. dict = {7: "Ayan", 5: "Bunny", 8: "Ram", 1: "Bheem"}
2. sorted(dict)

**Output**

[ 1, 5, 7, 8]

## Built-in Dictionary methods

The built-in python dictionary methods along with the description and Code are given below.

* **clear()**

It is mainly used to delete all the items of the dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # clear() method
4. dict.clear()
5. **print**(dict)

**Output**

{ }

* **copy()**

It returns a shallow copy of the dictionary which is created.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # copy() method
4. dict\_demo = dict.copy()
5. **print**(dict\_demo)

**Output**

{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}

* **pop()**

It mainly eliminates the element using the defined key.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # pop() method
4. dict\_demo = dict.copy()
5. x = dict\_demo.pop(1)
6. **print**(x)

**Output**

{2: 'WIPRO', 3: 'Facebook', 4: 'Amazon', 5: 'Flipkart'}

**popitem()**

removes the most recent key-value pair entered

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # popitem() method
4. dict\_demo.popitem()
5. **print**(dict\_demo)

**Output**

{1: 'Hcl', 2: 'WIPRO', 3: 'Facebook'}

* **keys()**

It returns all the keys of the dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # keys() method
4. **print**(dict\_demo.keys())

**Output**

dict\_keys([1, 2, 3, 4, 5])

* **items()**

It returns all the key-value pairs as a tuple.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # items() method
4. **print**(dict\_demo.items())

**Output**

dict\_items([(1, 'Hcl'), (2, 'WIPRO'), (3, 'Facebook'), (4, 'Amazon'), (5, 'Flipkart')])

* **get()**

It is used to get the value specified for the passed key.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # get() method
4. **print**(dict\_demo.get(3))

**Output**

Facebook

* **update()**

It mainly updates all the dictionary by adding the key-value pair of dict2 to this dictionary.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # update() method
4. dict\_demo.update({3: "TCS"})
5. **print**(dict\_demo)

**Output**

{1: 'Hcl', 2: 'WIPRO', 3: 'TCS'}

**values()**

It returns all the values of the dictionary with respect to given input.

**Code**

1. # dictionary methods
2. dict = {1: "Hcl", 2: "WIPRO", 3: "Facebook", 4: "Amazon", 5: "Flipkart"}
3. # values() method
4. **print**(dict\_demo.values())

**Output**

dict\_values(['Hcl', 'WIPRO', 'TCS'])