EPR Chapter-2
Introduction to Python Programming

- Python is popular language.
- First version is released in 1991.
- Many popular companies like Google, amazon, Quora, Instagram , Youtube are using python.

Que: Why to choose python?
- Python is simple because it is having simple syntax similar to English language.
- Python is free. It does not require any license. (open source license)
- It is free to install, use, and distribute, even for commercial purposes.
- Python is platform independent. It can work on any OS like windows, Linux or MAC.
- Python is portable because code or program written on one platform can work on other platform too.
- Python is interpreted.
- Python is popular.

You can download Python 2 or Python 3.
Nowadays people are using Python 3.

Install Anaconda Navigator.(editor)
Spyder
Jupyter notebook

OpenCV

Python Numbers:

In python there are three main numeric type.

1. Integer number
2. Floating number
3. Complex number

Integer	**Float**	**Complex**
6 10 3	1.5 0.99 3.33333333333333	2 + 3j 50 + 8j 3 + 5j

1. Integer Numbers

Integer number can be positive or negative.

```
# Following numbers are integers
x = 10
y = -10
z = 123456789
```

In Python 3, there is no limit to how long an integer value can be. It can grow to have as many digits as your computer's memory space allows.

X=999999999999999999999999999999999999999999999999999999999999999999999999999

We normally write numbers in decimal but python allows to write and perform operations in **binary (base 2) / octal (base 8) / hexadecimal (base 16)** too.

Prefix	Interpretation	Base
"0b" or "0B"	Binary	2
"0o" or "0O"	Octal	8
"0x" or "0X"	Hexadecimal	16

```
# Integers in binary, octal and hexadecimal formats

# binary
print(0b10111011)
# Prints 187
#10111011=187 in decimal
```

128	64	32	16	8	4	2	1
1	0	1	1	1	0	1	1

```
=128 + 32 + 16 + 8 + 2 + 1
=187


# octal
print(0o10)
# Prints 8
#10= 8 in decimal

# hex
print(0xFF)
# Prints 255
```

Octal: 0,1,2,3,4,5,6,7, 10,11,12,13,14,15,16,17,20,21,22,23,24,25,26,27,30….

Decimal: 0,1,2,3,4,5,6,7,8,9, 10,11……..19, 20

Boolean is subtype of integer.

```
# Boolean in python
x = True
x = False
```

2. Floating Point Numbers:

Floating point or float number is negative or positive with fractional part.

```
# Following numbers are floats
x = 10.1
y = -10.5
z = 1.123456
```

To represent the numbers in scientific notation you have to use "e" or "E" in numbers.

```
# Scientific notation

print(42e3)
```
#42 x 10^3=42000

```
# Prints 42000.0

print(4.2e-3)
```
#4.2 x 10^{-3}=4.2/10^{3}=4.2/1000=42/10000=0.0042
```
# Prints 0.0042
```
Represent 0.335 in scientific notation.
0.335=335/1000=335 /10^{3}=335 x 10^{-3} = (335/10) x 10^{-3} x 10)=
(3.35 x 10^{-2} = scientific notation)

3. Complex Numbers:

- A complex number contains real and imaginary part.
- (real part + imaginary part)
- Imaginary part can be represented with j or J.

```
# following numbers are complex numbers
x = 2j
y = 3+4j
```

To print real or imaginary part from number Y, use (y.real) pr (y.imag)
```
x = 3+4j

# real part
print(x.real)
# Prints 3.0

# imaginary part
print(x.imag)
# Prints 4.0
```

Que: Explain Python Operators.

Operators are used for performing various operations on numbers and variables. Python offers following seven categories of operators.

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators

- Membership operators
- Bitwise operators

<u>Arithmetic Operators</u>

Arithmetic operators are used to perform simple mathematical operations on numeric values (except complex).

Operator	Meaning	Example
+	Addition	x + y
–	Subtraction	x – y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

<u>Example:</u>

```
x = 6
y = 2

# addition
print(x + y)          # 8

# subtraction
print(x - y)          # 4

# multiplication
print(x * y)          # 12

# division
print(x / y)          # 3

# modulus
print(x % y)          # 0

# exponentiation
print(x ** y)         # 36

# floor division
print(x // y)         # 3
```

Assignment Operators (=)

Assignment operators are used to assign new values to variables.

Operator	Meaning	Example	Equivatent to
=	Assignment	x = 3	x = 3
+=	Addition assignment	x += 3	x = x + 3
-=	Subtraction assignment	x -= 3	x = x – 3
*=	Multiplication assignment	x *= 3	x = x * 3
/=	Division assignment	x /= 3	x = x / 3
%=	Modulus assignment	x %= 3	x = x % 3
//=	Floor division assignment	x //= 3	x = x // 3
**=	Exponentiation assignment	x **= 3	x = x ** 3
&=	Bitwise AND assignment	x &= 3	x = x & 3
|=	Bitwise OR assignment	x |= 3	x = x | 3
^=	Bitwise XOR assignment	x ^= 3	x = x ^ 3
>>=	Bitwise right shift assignment	x >>= 3	x = x >> 3
<<=	Bitwise left shift assignment	x <<= 3	x = x << 3

Comparison Operators

Comparison operators are used to compare two values.

Operator	Meaning	Example
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Example:

```
x = 6
y = 2

# equal to
print(x == y)          # False

# not equal to
print(x != y)          # True
```

```
# greater than
print(x > y)          # True

# less than
print(x < y)          # False

# greater than or equal to
print(x >= y)         # True

# less than or equal to
print(x <= y)         # False
```

Logical Operators

Logical operators are used to join two or more conditions.

Operator	Description	Example
and	Returns True if both statements are true	x > 0 and y < 0
or	Returns True if one of the statements is true	x > 0 or y < 0
not	Reverse the result, returns False if the result is true	not(x > 0 and y < 0)

Example:

```
x = 2
y = -2

# and
print(x > 0 and y < 0)        # True

# or
print(x > 0 or y < 0)         # True

# not
print(not(x > 0 and y < 0))     # False
```

Identity Operators

Identity operators are used to check if two objects point to the same object, with the same memory location.

Operator	Description	Example
is	Returns true if both variables are the same object	x is y
is not	Returns true if both variables are not the same object	x is not y

List in Python:

- A list is a sequence of values.
- It is similar to array.
- Values in the list are known as items / elements.
  L=[1,2,3]
- List is an ordered thing.
- It remembers the order of items inserted.
- Items in the list are accessed by an index.
- List can have any type of data like numbers and string.
- List can be nested.
- List is changeable means user can add, remove or replace any items / values.

- **There are many methods to create the list.**
  1. Enclose the values of list inside the square bracket. It will create the list.

```
# A list of integers
L = [1, 2, 3]
```

```
# A list of strings
L = ['red', 'green', 'blue']
```

- List items need not to be of same data type. It include multiple data types.

```
# A list of mixed datatypes
L = [ 1, 'abc', 1.23, (3+4j), True]
```

- A list can have zero items in it. Such type of list is known as an empty list.
- It can be create by simply typing square brackets without any values.

```
L=[]
```

- User can create list using **List()** constructor.
- List() constructor is used to convert other data type into list.

```
# Convert a string to a list
L = list('abc')
print(L)
# Prints ['a', 'b', 'c']
```

- To create a list **“list comprehension”** method can be used.

How to create nested List???

A list can contain sublists, which in turn can contain sublists themselves, and so on. This is known as nested list.

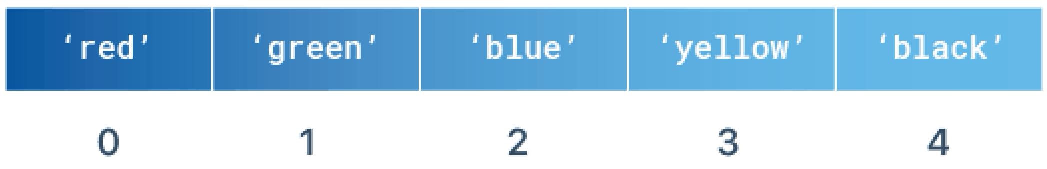
L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']

🟥	List 3 items
🟩	List 2 items
🟨	List 1 items

How to access the items from the list?? (Positive indexing)

Index is used to access items from the list. There is a mapping between values and index. Each index can have fix value from the list.

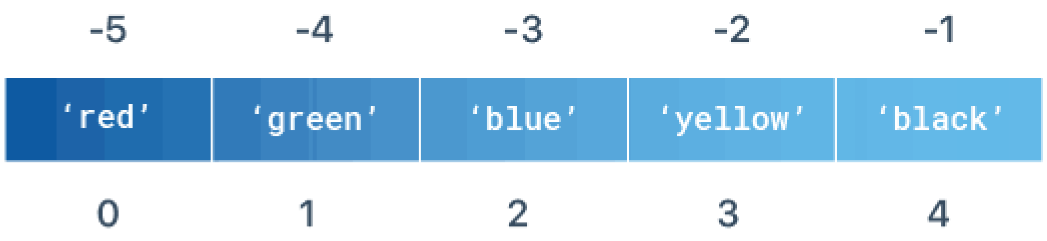
```
L = ['red', 'green', 'blue', 'yellow', 'black']
```



Index always start with '0'. So if you want to print all items in the list by index then use following code.

Print(L[0])→red
Print(L[1])→green
Print(L[2])→blue
Print(L[3])→yellow
Print(L[4])→black
Print(L[5])→it displays index error

<u>Negative List Indexing:</u>



Positive indexing starts from left to right and it includes ‘0’.
Negative indexing starts from right to left and do not include ‘0’.

Negative indexing counts from backward or end of the list.

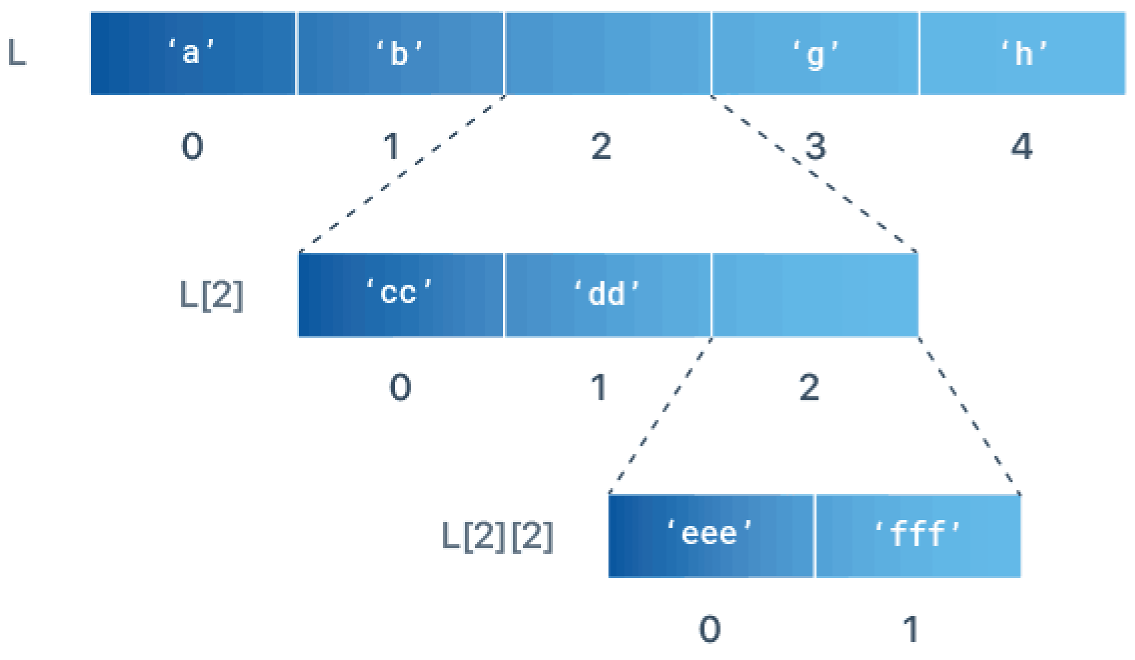
Print(L[-1])→Black
Print(L[-2])→yellow
Print(L[-3])→blue
Print(L[-4])→green
Print(L[-5])→red

L[0] = L[-5] = Red
L[1] = L[-4] = Green
L[2] = L[-3] = Blue
L[3] = L[-2] = Yellow
L[4] = L[-1] = Black

Que:
Give the differences between positive list indexing and negative list indexing.

How to Access List Items in Nested List

```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
```



How to print ‘a’→print(L[0])
How to print ‘b’→print(L[1])
How to print ‘g’→print(L[3])
How to print ‘h’→print(L[4])

How to print ‘cc’→print(L[2][0])
How to print ‘dd’→print(L[2][1])

How to print ‘eee’→print(L[2][2][0])
How to print ‘fff’→print(L[2][2][1])

Slicing in the List:

- A segment of a list is known as slice.
- Slice is also a list.
- The slice operator [n:m] returns the part of the list from the “nth ” item to the “mth” item, including the first(n) but excluding the last(m).

L = ['a', 'b', 'c', 'd', 'e', 'f']

-6	-5	-4	-3	-2	-1
a	b	c	d	e	f
0	1	2	3	4	5

```
print(L[2:5])
# Prints ['c', 'd', 'e']

print(L[0:2])
# Prints ['a', 'b']

print(L[3:-1])
# Prints ['d', 'e']
```

Change items / values in the List:

You can replace an existing element with a new value by assigning the new value to the index.

```
L = ['red', 'green', 'blue']
L[0] = 'orange'
print(L)
# it prints L = ['orange', 'green', 'blue']
```

```
L[-1] = 'violet'
print(L)
# it prints L = ['orange', 'green', 'violet']
```

Add items / values to the list:

We have to use append() and insert() methods to add the items in the list.

append() method is used to add the items in the list at the end only.

```
L = ['red', 'green', 'yellow']
L.append('blue')
print(L)
#it prints L=['red', 'green', 'yellow',’blue’]
```

If you want to add the item at specific location then use insert() method.

```
L = ['red', 'green', 'yellow']
L.insert(1, 'blue')
print(L)
#it prints L=['red', ’blue’, 'green', 'yellow']

L.insert(-2, ‘orange’)
Print(L)
#it prints L=['red', ’blue’, ‘orange’, 'green', 'yellow']
```

Combining the List:

extend() method is used to combine two lit into one. It actually merges one list into another.

```
L = ['red', 'green', 'yellow']
L.extend([1,2,3])
print(L)
# Prints ['red', 'green', 'yellow', 1, 2, 3]


# concatenation operator(+)
L = ['red', 'green', 'blue']
```

```
L = L + [1,2,3]
print(L)
# Prints ['red', 'green', 'blue', 1, 2, 3]
```

L += [1,2,3] is having the same function as L = L + [1,2,3]

Remove single item from List:

Remove method is used to remove particular item from list.

```
L = ['red', 'green', 'blue']
# Remove 'green'
L.remove('green')
print(L)
# Prints ['red', 'blue']
```

L. remove(1)→it will not remove item by index.

The remove() method removes item based on specified value and not by index.

If you want to delete list items based on the index, use pop() method or del keyword.

Remove List items by index:

Pop method is use to remove the items from the list by it’s index.

```
L = ['red', 'green', 'blue']
# Remove 2nd list item
L.pop(1)
print(L)
# Prints ['red', 'blue']
```

User can remove the items by negative index also.

```
L = ['red', 'green', 'blue']
# Remove 2nd list item
X = L.pop(-2)
print(L)
print(X)
# Prints ['red', 'blue']
#print ‘green’
```

If index is not specified while removing the items using pop() method then by default it will remove the item with index -1.

```
L = ['red', 'green', 'blue']
L.pop()
print(L)
# Prints ['red', 'green']
```

Clearing the List Item:

Use clear() method to remove the items from the list.

```
L = ['red', 'green', 'blue']
L.clear()
print(L)
# Prints []
```

Copy the list:

How to copy in C using assignment operator(=).

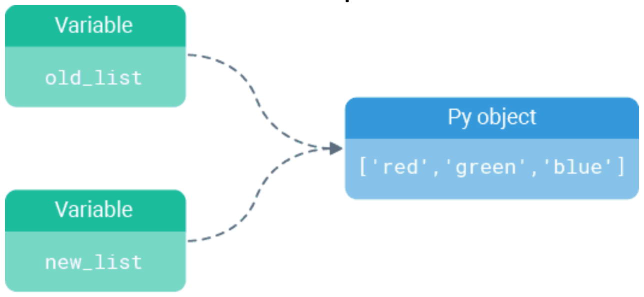
```
A=15;
B=A;
```

A	15
B	15

Copy in Python Using Assignment operator:

```
old_List = ['red', 'green', 'blue']
new_List = old_List
```

When you execute `new_List = old_List`, you don't actually have two lists. The assignment just makes the two variables point to the one list in memory.



```
old_List = ['red', 'green', 'blue']
new_List = old_List

new_List[0] = 'xx'
print(old_List)
# Prints ['xx', 'green', 'blue']

print(new_List)
# Prints ['xx', 'green', 'blue']
```

When assignment operator is used then two varibles are created which point out same List space. So if you modify the list using new variable then it modifie the list with old variable also.

Copy in Python using Copy() method:

```
old_List = ['red', 'green', 'blue']
new_List = old_List.copy()

new_List[0] = 'xx'
print(old_List)
# Prints ['red', 'green', 'blue']

print(new_List)
# Prints ['xx', 'green', 'blue']
```

Another method to copy:

```
L = ['red', 'green', 'blue']
X = L[:]
print(X)
# Prints ['red', 'green', 'blue']

L = ['red', 'green', 'blue']
X = L[0:1]
print(X)
# Prints ['red', 'green']
```

Finding List Length:

```
L=[‘red’,’green’,blue’]
X=Len(L)
#len() function is used to find the length
Print(x)
#print(len(L)) can also be used
```

Check if particular item exists in List??

To determine whether a particular item exist in the list, use if statement.

```
# Check for presence
L = ['red', 'green', 'blue']
if 'red' in L:
   print('yes')
```

```
# Check for absence
L = ['red', 'green', 'blue']
if 'yellow' not in L:
   print('yes')
```

How to iterate through list:

To print all values from the list as per specifications, user needs to use for loop.

```
L = ['red', 'green', 'blue']
for item in L:
   print(item)
```

```
# Prints red
# Prints green
# Prints blue
```

This method works well if you only need to read the items of the list. But if you want to update them, you need the indexes.

If you want to access the item by index then you will need to combine range() and len() functions.

```
# Loop through the list and double each item
L = [1, 2, 3, 4]

for i in range(len(L)):
   L[i] = L[i] * 2
   print(L)
#len(L) will return 4
#range(4)
# Prints [2, 4, 6, 8]
```

L(0)=1→L(0)*2→1*2→2
L(1)=2→L(1)*2→2*2→4
L(2)=3→L(2)*2→3*2→6
L(3)=4→L(3)*2→4*2→8

How to reverse the List:

To reverse the list use reverse() method.

```
L = ['red', 'green', 'blue']
L.reverse()
print(L)
# Prints ['blue', 'green', 'red']

L = [1, 2, 3, 4, 5]
L.reverse()
print(L)
# Prints [5, 4, 3, 2, 1]
```

Python Tuple:

- A tuple is an ordered collection of values.
- A tuple is similar to List in many ways.
- Items in a tuple can be accessed by an index.
- Tuple contains any sort of objects like numbers, character, Boolean, list and even tuple.
- **Tuples are immutable means it cannot be editable**. User can not add, remove or modify the values.

<u>How to create Tuple?</u>

- It is created by placing comma separated value in parentheses ().

```
# A tuple of integers
T = (1, 2, 3)
```

```
# A tuple of strings
T = ('red', 'green', 'blue')
```

```
# A tuple with mixed datatypes
T = (1, 'abc', 1.23, True)
```

```
# An empty tuple
T = ()
```

A tuple containing zero items is known as empty tuple.

```
# A tuple without parentheses
T = 1, 'abc', 1.23, True
```

Tuple doesn’t require parentheses. It is just a list separated by comma.

What do you mean by singleton Tuple?

A Tuple which is having only one value followed by comma is known as singleton tuple.

T=4

T=4,

T=(4,)

T=(4)→it is not a Tuple

<u>The Tuple() Constructor:</u>

User can convert other data type into tuple using tuple() constructor.

```
# Convert a list to a tuple
T = tuple([1, 2, 3])
print(T)
# Prints (1, 2, 3)
```

```
# Convert a string to a tuple
T = tuple('abc')
print(T)
```

# Prints ('a', 'b', 'c')

Nested Tuple:

A tuple inside other tuple (sub tuple) is known as nested tuple.

Example:

T = ('red', ('green', 'blue'), 'yellow')

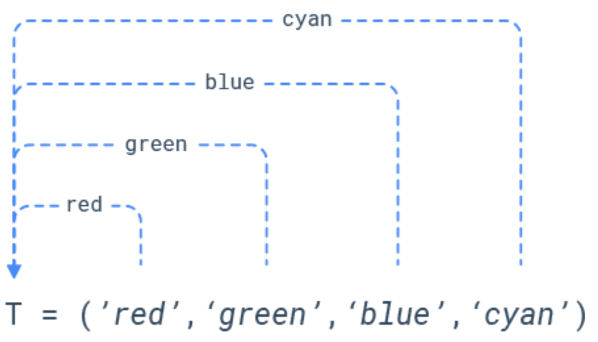
Explain Tuple packing and Unpacking:

Tuple packing:

When a tuple is created, the items in the tuple are packed together into the object.

```
T = ('red', 'green', 'blue', 'cyan')
print(T)
# Prints ('red', 'green', 'blue', 'cyan')
```

In above example, the values ‘red’, ‘green’, ‘blue’ and ‘cyan’ are packed together in a tuple.



Tuple Unpacking:

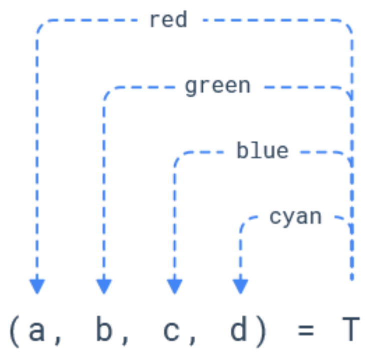
When a packed tuple is assigned to a new tuple, the individual items are unpacked (assigned to the items of a new tuple).

```
 T = ('red', 'green', 'blue', 'cyan')
(a, b, c, d) = T

print(a)
# Prints red

print(b)
# Prints green
```

print(c)
# Prints blue

print(d)
# Prints cyan

In above example T is unpacked to 4 variables a, b, c, d.



When unpacking, the number of variables on the left must match the number of items in the tuple.

```
# Common errors in tuple unpacking
T = ('red', 'green', 'blue', 'cyan')
(a, b) = T
```

```
T = ('red', 'green', 'blue')
(a, b, c, d) = T
```

<u>Use of packing / unpacking concept:</u>
When you want to swap two variables without use of temporary variable then this concept is used.

<u>Que</u>: Write python program for swapping of two numbers using Tuple.

```
# Swap values of 'a' and 'b'
a = 1
b = 99

a, b = b, a→packing / unpacking
```

print(a)
# Prints 99

print(b)
# Prints 1

Que: Give the comparison between List and Tuple.

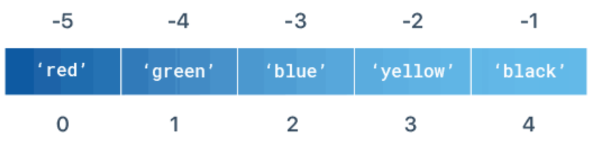
	List	**Tuple**
1	List items can be changed.	Tuple items cannot be changed.
2	List is created by []. (square bracket)	Tuple is created by (). (round bracket)
3	List items can be accessed by index.	Tuple items can be accessed by index.
4.	Nested list is possible.	Nested Tuple is also possible.
5	L=[1,2,3,4]	T=(1,2,3,4)

Que: Split an email address into user name and domain using Tuple.

Address = xyz@python.org
(user, domain) = address.split('@')
Print(user)
#it prints user
Print(domain)
#it prints python.org

Access Tuple Items:
You can access tuple items by writing index in square bracket.

```
T = ('red', 'green', 'blue', 'yellow', 'black')
print(T[0])
# Prints red
print(T[2])
# Prints blue
```



Start with '0' index from left side and increase index as you move towards right side. (Positive indexing)

Start with ‘-1’ index from right side and decrease index as you move towards left side. (Negative indexing)

```
T = ('red', 'green', 'blue', 'yellow', 'black')
print(T[-1])
# Prints black
print(T[-2])
# Prints yellow
print(T[-5])
# Prints red
```

Tuple Slicing:

It is similar to list slicing. Tuple slicing is accessing range or specific items from the tuple.

T[a:b]= it starts to print items from a and prints items upto (b-1).
T[1:5]= it starts to print items from index 1 and prints items upto index 4.

```
T = ('a', 'b', 'c', 'd', 'e', 'f')
```

```
print(T[2:5])
# Prints ('c', 'd', 'e')
```

```
print(T[0:2])
# Prints ('a', 'b')
```

```
print(T[3:-1])
# Prints ('d', 'e')
```

Change the items in Tuple:

Tuples are immutable (unchangeable). Once a tuple is created, it cannot be modified.

```
T = ('red', 'green', 'blue')
T[0] = 'black'
# Triggers TypeError: 'tuple' object does not support item assignment
```

```
T = (1, [2, 3], 4)

T[1][0] = 'xyz'
Print(T)
#it will print  (1, [xyz, 3], 4)
```

Tuple is immutable only at top level. A list inside a tuple can be modified as usual.

<u>Delete a Tuple:</u>

Items or values inside a tuple cannot be modified or deleted but tuple itself can be deleted. Del keyword is used to delete the tuple.

```
T = ('red', 'green', 'blue')
del T
```

<u>Tuple Concatenation & Repetition:</u>

Two operators (+) and (*) are used to concatenate and replicate the tuple.

```
# Concatenate
T = ('red', 'green', 'blue') + (1, 2, 3)
print(T)
# Prints ('red', 'green', 'blue', 1, 2, 3)

# Replicate
T = ('red',) * 3
print(T)
# Prints ('red', 'red', 'red')
```

<u>Finding a length of Tuple:</u>
len() function is used to find the length of tuple.

```
T=('red','green','blue')
Print(len())

#it will print 3
```

Check the items inside the tuple:

To check the values inside the tuple ‘in’ and ‘not in’ operators are used.

```
# Check for presence
T = ('red', 'green', 'blue')
if 'red' in T:
   print('yes')
#it prints yes
```

```
# Check for absence
T = ('red', 'green', 'blue')
if 'yellow' not in T:
   print('yes')
#It prints yes
```

Iterate using for loop:

```
T = ('red', 'green', 'blue')
for item in T:
   print(item)
# Prints red green blue
```

Tuple sorting:

There are two methods to sort the values inside the tuple.

Method-1:

Use the inbuilt function sorted() to sort the items.

```
T = ('cc', 'aa', 'dd', 'bb')
print(tuple(sorted(T)))
# Prints ('aa', 'bb', 'cc', 'dd')
```

- First sort the tuple
- Using tuple constructor create new sorted tuple
- Print the new tuple.

Sort in reverse order

```
T = ('cc', 'aa', 'dd', 'bb')
print(tuple(sorted(T,reverse=True)))
```

# Prints ('dd', 'cc', 'bb', 'aa')

Method-2:
Use the sort() method with list to sort the data.

Steps:
1. First convert tuple into list with list constructor.
2. Sort that list with sort() method.
3. Convert that list into Tuple.
4. Display the Tuple

```
T = ('cc', 'aa', 'dd', 'bb')
tmp = list(T)
# convert tuple to list
tmp.sort()
# sort list
T = tuple(tmp)
# convert list to tuple
print(T)
# Prints ('aa', 'bb', 'cc', 'dd')
```

Count the number of values from the Tuple:
Use count() method with tuple name.

```
# Count the number of occurrences of ‘red’
T = ('red', 'green', 'blue')
print(T.count('red'))
# Prints 1
```

```
# Count the number of occurrences of number ‘9’
T = (1, 9, 7, 3, 9, 1, 9, 2)
print(T.count(9))
# Prints 3
```

Finding the index from Tuple:

```
# Find index of 'green' in a tuple
T = ('red', 'green', 'blue', 'yellow')
print(T.index('green'))
```

# Prints 1

Max() function:

X=max(10,20,30,40)
Print(X)
#ptint x=40

x=max('red', 'green', 'blue')
print(x)
# if argument with max() function are strings then string with highest value in alphabetical order will be printed.

Blue, green, red

#it prints red

Min() function:

X=min(10,20,30,40)
Print(X)
#print x=10

x=min('red', 'green', 'blue')
print(x)
#it prints blue

Python Set:

What is set??
Set is collection of meaningful data.

Whether set is ordered or unordered??

Set is unordered means we can arrange data in any order.
A={1,2,3,4,5}
A={5,4,3,2,1}
A={3,2,1,5,4}

To create the set we are using curly brackets { }.

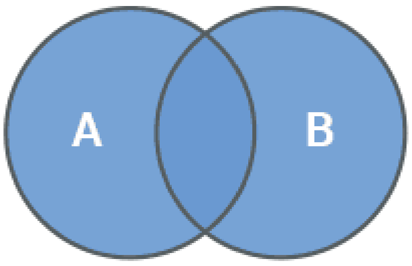
List: []→square bracket
Tuple: ()→round brackets
Set: {}→ curly brackets

A={1,1,1,2,3}
It is not a set because it contains duplicate items.

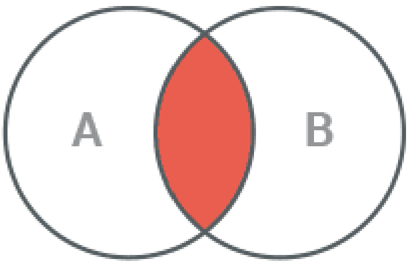
Set doesn’t have any duplicate data.

Python set is unordered collection of unique items.

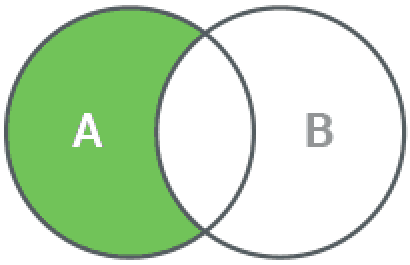
Whenever you want to perform any mathematical operations such as union , intersection, difference, symmetric difference etc. then you must require to use python set.



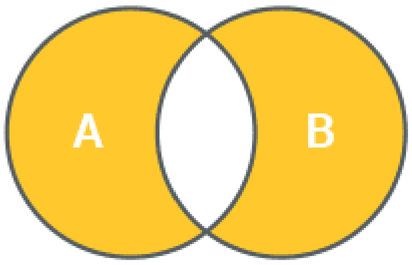
Union



Intersection



Difference



Symmetric Difference

First figure indicates union operation where all data from set A and B are merged together.

Second figure indicates intersection operation where common data between set A and set B is highlighted.

Third figure difference operation is shown where items of B from A are excluded or removed. (Remove the part of B from A)

A – B = A – (common elements between A and B)

Fourth figure indicates symmetric difference operation where common elements between two sets are remove from their union.

Properties of Python Sets:

1. Python sets are unordered. (It is not necessary to store items in particular order).
2. Python Sets items are unique. (Duplicate items are not allowed.)
3. Python sets are unindexed. (items cannot be accessed by particular index.)
4. Python sets are changeable (mutable). (items from the sets can be removed, added or changed)

Creating set in python:

One can create the python sets by placing items separated by comma in curly braces { }.

```
# A set of strings
S = {'red', 'green', 'blue'}

# A set of mixed datatypes
S = {1, 'abc', 1.23, (3+4j), True}
```

Set doesn't allow duplicate items so if you add the duplicate items in a set then it will be removed automatically.

```
S = {'red', 'green', 'blue', 'red'}
print(S)
# Prints {'blue', 'green', 'red'}
```

Python set is changeable (mutable) but it cannot contains items which are changeable (mutable).

List → changeable (mutable)
Tuple → not changeable (Not mutable)

Que: A Python set contain List as an item.
1. True
2. False

Que: A Python set cannot contain List as an item.
1. True
2. False

Que: A Python set contain Tuple as an item.
1. True
2. False

Que: A Python set cannot contain Tuple as an item.
1. True
2. False

Set Constructor:

A python set can also be created using set() constructor.

```
S = set('abc')
print(S)
# Prints {'a', 'b', 'c'}
```

```
# Convert list into set
S = set([1, 2, 3])
print(S)
# Prints {1, 2, 3}
```

range() is used to display the items in specified range.
Range(0,4)→ it displays value 0,1,2,3 (4 is not displayed)

To store successive numbers in set use following method.

```
# Set of successive integers
S = set(range(0, 4))
print(S)
# Prints {0, 1, 2, 3}
```

```
S = set(range(1, 10))
print(S)
# Prints {1,2,3,4,5,6,7,8,9}
```

Set constructor

You can also create a set using a type constructor called set().

```
# Set of items in an iterable
S = set('abc')
print(S)
# Prints {'a', 'b', 'c'}

# Set of successive integers
S = set(range(0, 4))
print(S)
# Prints {0, 1, 2, 3}

# Convert list into set
S = set([1, 2, 3])
print(S)
# Prints {1, 2, 3}

#add single value
S = {'red', 'green', 'blue'}
S.add('yellow')
print(S)
# Prints {'blue', 'green', 'yellow', 'red'}
# To add singl value in a set.

#add multiple values
S = {'red', 'green', 'blue'}
S.update(['yellow', 'orange'])
print(S)
# Prints {'blue', 'orange', 'green', 'yellow', 'red'}
```

Remove Items from a Set

To remove a single item from a set, use remove() or discard() method.

```
# with remove() method
S = {'red', 'green', 'blue'}
S.remove('red')
print(S)
# Prints {'blue', 'green'}

# with discard() method
S = {'red', 'green', 'blue'}
```

```
S.discard('red')
print(S)
# Prints {'blue', 'green'}
```

remove() vs discard()

Both methods work exactly the same. The only difference is that If specified item is not present in a set:

- remove() method raises KeyError
- discard() method does nothing

The pop() method removes random item from a set and returns it.

```
S = {'red', 'green', 'blue'}
x = S.pop()
print(S)
# Prints {'green', 'red'}

# removed item
print(x)
# Prints blue
```

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

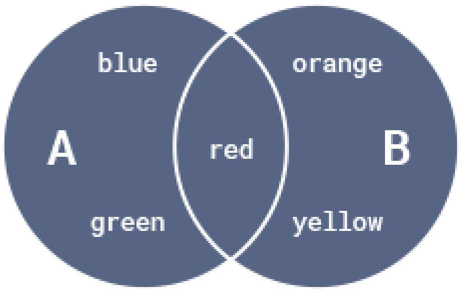
```
S = {'red', 'green', 'blue'}
S.clear()
print(S)
# Prints set()

# Check for presence
S = {'red', 'green', 'blue'}
if 'red' in S:
   print('yes')

# Check for absence
S = {'red', 'green', 'blue'}
if 'yellow' not in S:
   print('yes')
```

Set Operations:

1. Union Operation:

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
```

A U B = {red, green, blue, yellow, orange}



```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
```

```
# by operator
print(A | B)
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```

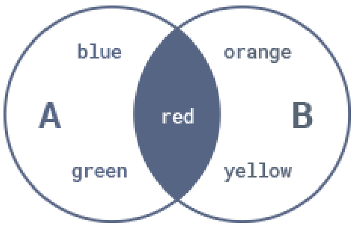
```
# by method
print(A.union(B))
# Prints {'blue', 'green', 'yellow', 'orange', 'red'}
```

Intersection Operation:

A = {'red', 'green', 'blue'}

B = {'yellow', 'red', 'orange'}

A (intersection) B = {red}



```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
```

```
# by operator
print(A & B)
# Prints {'red'}
```

```
# by method
print(A.intersection(B))
# Prints {'red'}
```

Difference Operation:

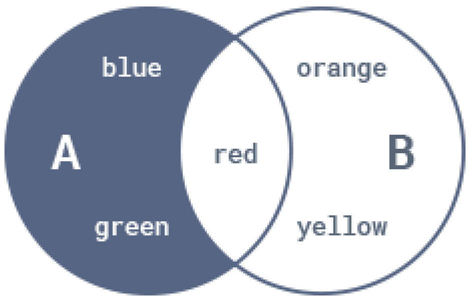
Set difference operation can be performed by subtracting members of one set from others.

A – B → It means remove members of B from A

A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

A – B = {green, blue} = A – (A intersection B)
B – A = {yellow, orange} = B – (A intersection B)

It can be performed by "-" operator or difference method.



```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by operator
print(A - B)
# Prints {'blue', 'green'}

# by method
print(A.difference(B))
# Prints {'blue', 'green'}


A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

# by operator
print(B - A)
```

```
# Prints {'yellow', 'orange'}
```

```
# by method
print(B.difference(A))
# Prints {'yellow', 'orange'}
```

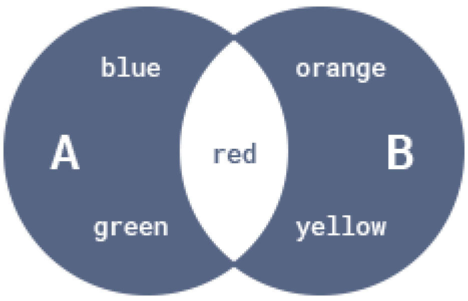
4. Symmetric Difference:

Set A and Set B

The set of all elements which belongs to either A or B but doesn’t belongs to both.

It can be performed by ‘^’ operator or symmetric_difference() method.

A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}



Dark portion in above figure represents symmetric difference output.

A^B = {blue, green, orange, yellow}

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}
```

```
# by operator
print(A ^ B)
# Prints {'orange', 'blue', 'green', 'yellow'}
```

```
# by method
print(A.symmetric_difference(B))
# Prints {'orange', 'blue', 'green', 'yellow'}
```

<u>Python frozen set:</u>

Python offers another inbuilt type which is frozen set. It is similar to set except it is not changeable (unchangeable) or immutable.

frozenset() function is use to create the frozenset.

On frozenset, we can perform the non-modifying operations like finding length, performing union etc.

On frozenset, we can't perform the operations like addition or deletion of data.

<u>Python code for frozenset:</u>

```
S = frozenset({'red', 'green', 'blue'})
print(S)
# Prints frozenset({'green', 'red', 'blue'})

print(len(S))
# Prints 3

# performing union
print(S | {'yellow'})
# Prints frozenset({'blue', 'green', 'yellow', 'red'})

# removing an item
S.pop()
# Triggers AttributeError: 'frozenset' object has no attribute 'pop'

# adding an item
S.add('yellow')
# Triggers AttributeError: 'frozenset' object has no attribute 'add'
```

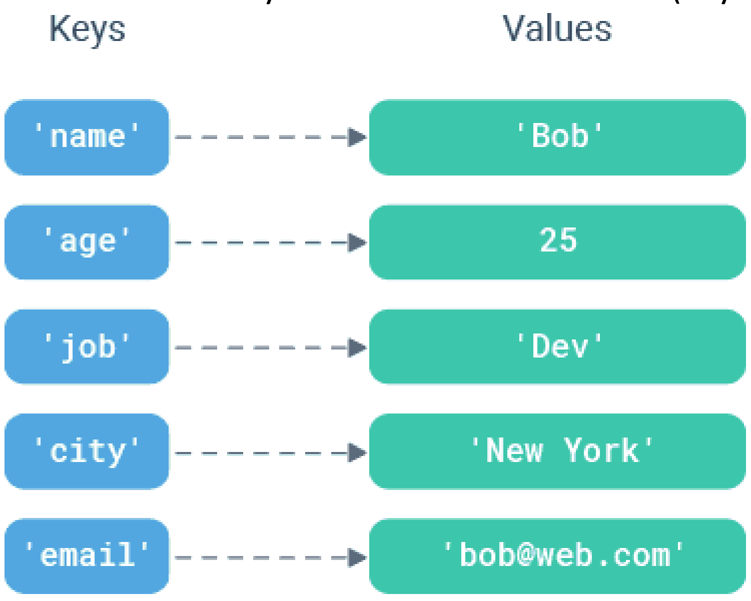
<u>Python Dictionary:</u>

Dictionaries are python implementation of data structure. It is also known as associative array.

Dictionary can be considered as set of indexes and set values.

Each key is mapped with specific values.

Association of key and its value is known as (key:value) pair.



How to create dictionary?

```
# Create a dictionary to store employee record
D = {'name': 'Bob',
    'age': 25,
    'job': 'Dev',
    'city': 'New York',
    'email': 'bob@web.com'}
```

Step-1: start writing in set { }
Step-2: write down the key within single quote(' ')
Step-3: separate key and its value by colon (:)
Step-4: type the value after column within single quote(' ')
Step-5: Insert the new records separated by coma. (,)

Using dict() constructor we can create the dictionary.

16-2-2022

You can convert two-value sequences into a dictionary with Python's dict() constructor. The first item in each sequence is used as the key and the second as the value.

```
# Create a dictionary with a list of two-item tuples
L = [('name', 'Bob'),
    ('age', 25),
    ('job', 'Dev')]
```

```
D = dict(L)
print(D)
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

```
# Create a dictionary with a tuple of two-item lists
T = (['name', 'Bob'],
    ['age', 25],
    ['job', 'Dev'])

D = dict(T)
print(D)
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

When the keys are simple strings, it is sometimes easier to specify key:value pairs using keyword arguments.

```
D = dict(name = 'Bob',
       age = 25,
       job = 'Dev')

print(D)
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

Other Ways to Create Dictionaries

There are lots of other ways to create a dictionary.
You can use dict() function along with the zip() function, to combine separate lists of keys and values obtained dynamically at runtime.

```
# Create a dictionary with list of zipped keys/values
keys = ['name', 'age', 'job']
values = ['Bob', 25, 'Dev']

D = dict(zip(keys, values))

print(D)
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev'}
```

You'll often want to create a dictionary with default values for each key. The fromkeys() method offers a way to do this.

```
# Initialize dictionary with default value '0' for each key
keys = ['a', 'b', 'c','d']
defaultValue = 5

D = dict.fromkeys(keys,defaultValue)

print(D)
# Prints {'a': 0, 'b': 0, 'c': 0}
```

Important Properties of a Dictionary

Dictionaries are pretty straightforward, but here are a few points you should be aware of when using them.

1. Keys must be unique:

A key can appear in a dictionary only once.
Even if you specify a key more than once during the creation of a dictionary, the last value for that key becomes the associated value.

```
D = {'name': 'Bob',
    'age': 25,
    'name': 'Jane'}
print(D)
# Prints {'name': 'Jane', 'age': 25}
```

Notice that the first occurrence of 'name' is replaced by the second one.

2. Key must be immutable type:

You can use any object of immutable type as dictionary keys – such as numbers, strings, booleans or tuples.

```
D = {(2,2): 25,
    True: 'a',
    'name': 'Bob'}
```

An exception is raised when mutable object is used as a key.

```
# TypeError: unhashable type: 'list'
D = {[2,2]: 25,
    'name': 'Bob'}
```

3. Value can be of any type:

There are no restrictions on dictionary values. A dictionary value can be any type of object and can appear in a dictionary multiple times.

```
# values of different datatypes
D = {'a':[1,2,3],
    'b':{1,2,3}}

# duplicate values
D = {'a':[1,2],
    'b':[1,2],
    'c':[1,2]}
```

Access Dictionary Items

The order of key:value pairs is not always the same. In fact, if you write the same example on another PC, you may get a different result. In general, the order of items in a dictionary is unpredictable.

But this is not a problem because the items of a dictionary are not indexed with integer indices. Instead, you use the keys to access the corresponding values. You can fetch a value from a dictionary by referring to its key in square brackets [].

```
D = {'name': 'Bob',
    'age': 25,
    'job': 'Dev'}

print(D['name'])
# Prints Bob
```

If you refer to a key that is not in the dictionary, you'll get an exception.

```
print(D['salary'])
# Triggers KeyError: 'salary'
```

To avoid such exception, you can use the special dictionary get() method. This method returns the value for key if key is in the dictionary, else None, so that this method never raises a KeyError.

```
# When key is present
print(D.get('name'))
# Prints Bob

# When key is absent
print(D.get('salary'))
# Prints None
```

Add or Update Dictionary Items

Adding or updating dictionary items is easy. Just refer to the item by its key and assign a value. If the key is already present in the dictionary, its value is replaced by the new one.

```
D = {'name': 'Bob',
    'age': 25,
    'job': 'Dev'}

D['name'] = 'Sam'
print(D)
# Prints {'name': 'Sam', 'age': 25, 'job': 'Dev'}
```

If the key is new, it is added to the dictionary with its value.

```
D = {'name': 'Bob',
    'age': 25,
    'job': 'Dev'}

D['city'] = 'New York'
print(D)
# Prints {'name': 'Bob', 'age': 25, 'job': 'Dev', 'city': 'New York'}
```

Merge Two Dictionaries

Use the built-in update() method to merge the keys and values of one dictionary into another. Note that this method blindly overwrites values of the same key if there’s a clash.

```
D1 = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}
```

```
D2 = {'age': 30,
      'city': 'New York',
      'email': 'bob@web.com'}

D1.update(D2)
print(D1)
# Prints {'name': 'Bob', 'age': 30, 'job': 'Dev',
#        'city': 'New York', 'email': 'bob@web.com'}
```

Remove Dictionary Items
There are several ways to remove items from a dictionary.

Remove an Item by Key
If you know the key of the item you want, you can use pop() method. It removes the key and returns its value.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

x = D.pop('age')
print(D)
# Prints {'name': 'Bob', 'job': 'Dev'}
# get removed value
print(x)
# Prints 25
```

If you don’t need the removed value, use the del statement.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

del D['age']
print(D)
# Prints {'name': 'Bob', 'job': 'Dev'}
```

Remove Last Inserted Item
The popitem() method removes and returns the last inserted item.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}
```

```
x = D.popitem()
print(D)
# Prints {'name': 'Bob', 'age': 25}

# get removed pair
print(x)
# Prints ('job', 'Dev')
```

In versions before 3.7, popitem() would remove a random item.

Remove all Items

To delete all keys and values from a dictionary, use clear() method.

```
D = {'name': 'Bob',
    'age': 25,
    'job': 'Dev'}

D.clear()
print(D)
# Prints {}
```

Get All Keys, Values and Key:Value Pairs

There are three dictionary methods that return all of the dictionary’s keys, values and key-value pairs: keys(), values(), and items(). These methods are useful in loops that need to step through dictionary entries one by one.
All the three methods return iterable object. If you want a true list from these methods, wrap them in a list() function.

```
D = {'name': 'Bob',
    'age': 25,
    'job': 'Dev'}

# get all keys
print(list(D.keys()))
# Prints ['name', 'age', 'job']

# get all values
print(list(D.values()))
# Prints ['Bob', 25, 'Dev']

# get all pairs
```

```
print(list(D.items()))
# Prints [('name', 'Bob'), ('age', 25), ('job', 'Dev')]
```

Iterate Through a Dictionary

If you use a dictionary in a for loop, it traverses the keys of the dictionary by default.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

for x in D:
    print(x)
# Prints name age job
```

To iterate over the values of a dictionary, index from key to value inside the for loop.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

for x in D:
    print(D[x])
# Prints Bob 25 Dev
```

Check if a Key or Value Exists

If you want to know whether a key exists in a dictionary, use in and not in operators with if statement.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

print('name' in D)
# Prints True
print('salary' in D)
# Prints False
```

To check if a certain value exists in a dictionary, you can use method values(), which returns the values as a list, and then use the in operator.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

print('Bob' in D.values())
# Prints True
```

```
print('Sam' in D.values())
# Prints False
```

in Operator on List vs Dictionary

The in operator uses different algorithms for lists and dictionaries. For lists, it uses a search algorithm. As the list gets longer, the search time gets longer. For dictionaries, Python uses a different algorithm called Hash Table, which has a remarkable property: the operator takes the same amount of time, regardless of how many items are in the dictionary.

Find Dictionary Length

To find how many key:value pairs a dictionary has, use len() method.

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

print(len(D))
# Prints 3
```