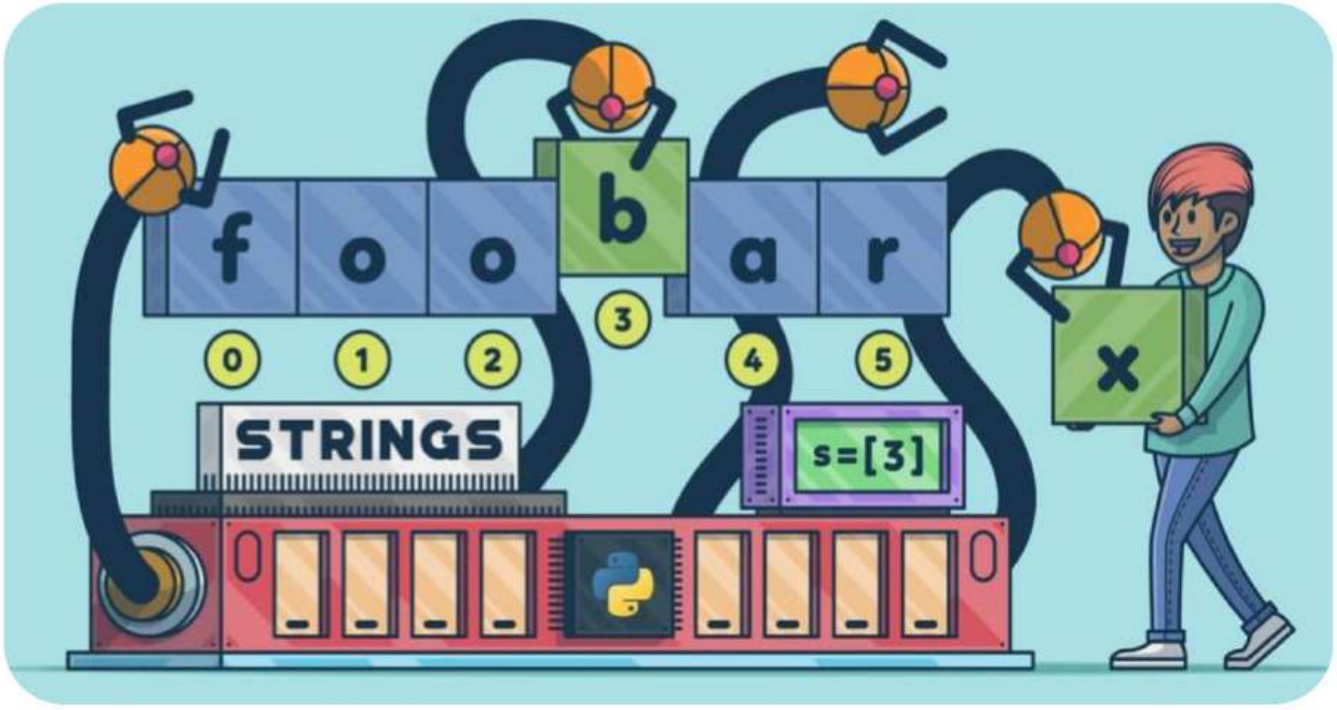


STRING

A string is an immutable sequence of characters, such as letters, digits, spaces, and special characters.

- A string is an instance of the `str` class.
- It is a sequence of characters, where each character is a single Unicode point.
- Strings are enclosed in quotes (either single `'` or double quotes `" "`).
- Strings can contain:
  - Letters (a-z, A-Z)
  - Digits (0-9)
  - Spaces
  - Special Characters (!, @, #, $, etc.)

Index	0	1	2	3	4	5	6	7	8	9	10	11
Character	H	e	l	l	o		W	o	r	l	d	!
Reverse Index	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
msg = "Hello"

print(msg[0])       # prints H
print(msg[4])       # prints o
print(msg[-0])      # prints H, -0 is same as 0
print(msg[-1])      # prints o
print(msg[-2])      # prints l
print(msg[-5])      # prints H
print(msg[-6])      # IndexError: string index out of range
```

```
A sub-string can be sliced out of a string

s[start : end] - extract from start to end-1
print(msg[0:5])

s[start :] - extract from start to end
print(msg[1:])

s[: end] - extract from start to end-1
print(msg[0:100])

s[-start :] - extract from -start(included) to end
print(msg[-5:])

s[: -end] - extract from beginning to -end-1
print(msg[:-2])
```

String Properties

```
As we know Python Strings are Immutable - they cannot be changed

message="Hello"
message[0]="M"      # rejected attempt to mutate(change) string

message="Hii.."
print(message)      # msg is a variable it can change



Strings can be concatenated by using +

name="Arjun"
sur_name="Diwedi"

print(name + sur_name)



Strings can be replicated during printing

print("-" * 50)     # prints --------------------------------------------------



Whether one string is part of another can be found out using (in)

print("e" in "Hello")   # prints True
print("z" in "Hello")   # prints False
print("lo" in "Hello")  # prints True
```

ASCII Table

```
Dec  = Decimal Value
Char = Character

'5' has the int value 53
if we write '5'-'0' it evaluates to 53-48, or the int 5
if we write char c = 'B'+32; then c stores 'b'
```

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Built-in Function

len(): Returns the length of the string, which is the number of characters in the string.

```
my_string = "Hello, World!"
print(len(my_string))
```

OUTPUT: 13

max(): Returns the character with the highest Unicode value int the string.

```
my_string = "Hello, World!"
print(max(my_string))
```

OUTPUT: r (Here 'r' is having the Highest Unicode point value in the string)

min(): Returns the character with the lowest Unicode value int the string.

```
my_string = "Hello, World!"
print(min(my_string))
```

OUTPUT: ' ' (Here it returns the character with the lowest Unicode point value in the string "Hello, World!", which is a space(' ')).

String Methods

- **Content test Function**

isalpha(): Returns True if all characters int the string are alphabetic i.e (a-z, A-Z), and the string is not empty

```
str1 = "Hello"
str2 = "Hello123"

print(str1.isalpha())   # OUTPUT: True
print(str2.isalpha())   # OUTPUT: False
```

isdigit(): Returns True if all characters int the string are digits i.e. (0-9), and the string is not empty

```
str1 = "123"
str2 = "123abc"

print(str1.isdigit())   # OUTPUT: True
print(str2.isdigit())   # OUTPUT: False
```

isalnum(): Returns True if all characters int the string are alphanumeric i.e (a-z, A-Z, 0-9), and the string is not empty

```
str1 = "Hello123"
str2 = "Hello@123"

print(str1.isalnum())   # OUTPUT: True
print(str2.isalnum())   # OUTPUT: False  (Special Character and Spaces are not allowed)

age = "20"
name = "Spiderman"
```

```
print(age.isalnum())    # OUTPUT: True
```

Explanation: The string consists of only numeric character digits and no alphabetic character. isalnum() method checks for alphanumeric characters (a-z, A-Z, 0-9) in string. If the string contains only numeric characters or a combination of both alphabet and numeric digits, it will return True. Because numeric character digits comes under (a-z, A-Z, 0-9).

```
print(name.isalnum())   # OUTPUT: True
```

Explanation: The string consists of only alphabetic character and no numeric character digits. isalnum() method checks for alphanumeric characters (a-z, A-Z, 0-9) in string. If the string contains only alphabetic characters or a combination of both alphabet and numeric digits, it will return True. Because alphabetic character comes under (a-z, A-Z, 0-9).

Note: The Method returns False only if the string is empty or contains non-alphanumeric characters. Such as !, @, #, $, %, ^, &, *, ().

isupper(): Returns True if all alphabetic characters in the string are uppercase and there is at least one character

```
str1 = "HELLO"
str2 = "Hello"


print(str1.isupper())   # OUTPUT: True
print(str2.isupper())   # OUTPUT: False
```

islower(): Returns True if all alphabetic characters in the string are lowercase and there is at least one character

```
str1 = "hello"
str2 = "Hello"

print(str1.islower())   # OUTPUT: True
print(str2.islower())   # OUTPUT: False
```

startswith(): Retrurns True if the string starts with the specified prefix.

```
my_string = "Hello, World!"

print(my_string.startswith("Hello"))    # OUTPUT: True
print(my_string.startswith("World"))    # OUTPUT: False
```

endswith(): Retrurns True if the string ends with the specified suffix

```
my_string = "Hello, World!"

print(my_string.endswith("World!"))    # OUTPUT: True
print(my_string.endswith("Hello"))     # OUTPUT: False
```

- **Search and replace**

find(): This method is used to find the first occurrence of a substring within a string. It returns the index of the first occurrence of the substring, or -1 if the substring is not found.

```
sentence = "Hello, how are you?"

index = sentence.find("how")

print(index)
```

OUTPUT: 7

replace(): This method is used to replace all occurrences of a specified substring within a string with another substring. It returns a new string with the replacement applied.

```
sentence = "Hello, how are you?"

index1 = id(sentence)
print(index1)

sentence = sentence.replace("how", "where")
index2 = id(sentence)
print(index2)

print(sentence)
```

OUTPUT: Hello, where are you?

In the above example it seems like string is been Mutated, but it is not so memory address before and after are different. If the String is Immutable. Both the memory address has to be same after mutating.

- **Trims Whitespaces**

lstrip(): This method is used to remove leading(left) whitespaces characters from a string including \t. It returns a new string with the leading whitespace characters removed.

```
text = "\tHello, how are you?"

print(text)     # prints (        Hello, how are you?)

print(text.lstrip())     # prints (Hello, how are you?)
```

rstrip(): This method is used to remove trailing(right) whitespaces characters from a string including \t. It returns a new string with the trailing whitespace characters removed.

```
text = "Hello, how are you?\t"

print(text)     # prints (Hello, how are you?       )

print(text.rstrip())     # prints (Hello, how are you?)
```

strip(): This method is used to remove both leading an trailing whitespace characters from a string including \t.

```
text = "\tHello, how are you?\t"

print(text)     # prints (        Hello, how are you?        )

print(text.strip())     # prints (Hello, how are you?)
```

- **Split, Partition and Join**

split(): This method is used to split a string into a list of substrings based on a specified delimiter.

Example 1:

```
sentence = "Hello, how are you?"

words = sentence.split(" ")     # Here (" ") is a specified delimiter

print(words)
print(type(words))
```

OUTPUT:

```
['Hello,', 'how', 'are', 'you?']
<class 'list'>
```

Output is generated into the List and 'Hello,' 'how' 'are' 'you?' are the substring

Example 2:

```
Text="apple,banana,orange"

Fruits=Text.split(",")      # Here (",") is a specified delimiter

print(Fruits)
print(type(Fruits))
```

OUTPUT:

```
['apple', 'banana', 'orange']
<class 'list'>
```

Example 3:

```
Text="apple,banana,orange"  # Normal String

ffruits,sfruits,tfruits= Text.split(",")

print(ffruits,sfruits,tfruits)

print(type(ffruits))
print(type(sfruits))
print(type(tfruits))
```

OUTPUT: apple banana orange

Example 4:

```
name="Arjun Diwedi"
first, last = name.split(" ")
print(f"Hello, {first}")
print(f"{last} is your Surname")
```

OUTPUT:

```
Hello, Arjun
Diwedi is your Surname
```

Note: You have to Enter only 2 Words with Spaces, no single word

String Conversions

```
.upper() Converts all characters in a string to uppercase
Text="hello world"

Text=Text.upper()

print(Text)

OUTPUT: HELLO WORLD
```

```
.lower(): Converts all characters in a string to lowercase.
Text="HELLO WORLD"

Text=Text.lower()

print(Text)

OUTPUT: hello world
```

```
.capitalize() --> Capitalize user's name (Just the very First Letter)
name="arjun"

name=name.capitalize()

print(f"Hello, {name}")

OUTPUT:
Hello, Arjun
```

```
.title() --> Capitalize the 1st Letter of each word
name=input("What's your name? ")

name=name.title()

print(f"Hello, {name}")

OUTPUT:
What's your name? dhananjay dubey
Hello, Dhananjay Dubey
```

```
.swapcase(): This method is used to convert the uppercase character in a string to
lowercase and vice versa. It returns a new string with the case of each letter swapped.
org_str = "Hello, World!"
new_str = org_str.swapcase()

print(new_str)

OUTPUT: hELLO, wORLD!
```

String Comparison

```
s1 = "Bombay"
s2 = "bombay"
s3 = "Nagpur"
s4 = "Bombaywala"
s5 = "Bombay"

print(s1 == s2)     # prints False
print(s1 == s5)     # prints True
print(s1 != s3)     # prints True
print(s1 > s5)      # prints False
print(s1 < s2)      # prints True
print(s1 <= s4)     # prints True
```

String comparison is done in lexicological order (alphabetical) character by character.
Capitals are considered to be less than their lowercase counterparts.
Result of comparison operation is either True or False

Note: If there are two different string variable and their object i.e. value are same then they both contain the same address

Let's see an Example:

```
string1 = "INDIA"
index1 = id(string1)
print(index1)

string2 = "INDIA"
index2 = id(string2)
print(index2)

if index1 == index2:
    print("Both the Memory Address are same")

else:
    print("Both the Memory Address are Different")
```

OUTPUT:

```
2600608861280
2600608861280

Both the Memory Address are same
```

LIST

- List is an ordered sequence of items. It is one of the most used datatypes in Python and is very flexible. List can contain heterogeneous values such as integers, floats, strings, tuples, lists and dictionaries but they are commonly used to store collections of homogeneous objects. The list data type in Python programming is just like an array that can store a group of elements and we can refer to these elements using a single name.
- Declaring a list is pretty straight forward. Items separated by commas (,) are enclosed within brackets [].
- Lists are mutable which means that value of elements of a list can be altered by using index.
- A string is a sequence of characters and list is a sequence of values, but a list of characters is not same as string. We can convert string to a list of characters.

List Index

lst = [321, "Hello", True, 88.99, "World!"]

→ We can use the index operator [] to acess an item in a list.
Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

length = 5

	321	"Hello"	TRUE	88.99	World!
index	0	1	2	3	4
negative inde:	-5	-4	-3	-2	-1

Container is an entity which contains multiple data items. It is also known as a collection.

Python has following container data types:
Lists Tuples Sets Dictionaries

A list can grow or shrink during execution of the program. Hence it is also known as a Dynamic Array.
A list is defined by writing comma-separated elements within [].

Example:
```
num = [10, 20, 30, 40, 50]
name = ["Sanjay", "Anil", "Radha", "Suparna"]
```

List can contain dissimilar types, though usually they are a collection of similar types.

Example:
```
animal = ["Zebra", 155.5, 110]
```

Items in list can be repeated
```
ages = [25, 26, 25, 27, 26]       # duplicates allowed
num = [10] * 5                    # stores [10, 10, 10, 10, 10]
lst = []                          # empty list, valid
```

Accessing List Elements

Example 1: Entire List can be printed by just using the name of the list
```
l = ["Hello", 20, True, 78.23]
print(l)
```

Example 2: Like strings, individual elements in a list can be accessed using indices.
```
animal = ["Zebra", 155.5, 110]
ages = [25, 26, 25, 27, 26]

print(animal[1], ages[3])
```
OUTPUT: 155.5 27

Example 3: Like strings, lists can be sliced
```
animal = ["Zebra", 155.5, 110]
ages = [25, 26, 25, 27, 26]

print(animal[1:3])      # prints [155.5, 110]
print(ages[3:])         # prints [27, 26]
```

Looping in List

If we wish to process each item in the list, we should be able to iterate through the list. This can done using a while or for loop.

Example 1:

```
animals = ["Zebra", "Tiger", "Lion", "Jackal", "Kangaroo"]

using while loop
i = 0

while i < len(animals):
    print(animals[i])
    i+=1
```

Example 2:
Using more convenient for loop

```
for a in animals:
    print(a)
```

OUTPUT:

```
Zebra
Tiger
Lion
Jackal
Kangaroo
```

Iterating through a list using for loop. If we wish to keep track of index of the elements that a is referring to, we can do so using the built-in enumerate() function.

Example 3:

```
animals = ["Zebra", "Tiger", "Lion", "Jackal", "Kangaroo"]

for index,a in enumerate(animals):
    print(index, a)
```

OUTPUT:

```
0 Zebra
1 Tiger
2 Lion
3 Jackal
4 Kangaroo
```

Basic List Operations

Mutability: Unlike strings, lists are mutable (changeable). So lists can be updated as shown below:

```
animals = ["Zebra", "Tiger", "Lion", "Jackal", "Kangaroo"]
ages = [25, 26, 25, 27, 26, 28, 25]

animals[2] = "Rhinoceros"

ages[5] = 31

ages[2:5] = [24, 25, 32]    # sets items 2 to 5 with values 24, 25, 32
ages[2:5] = []              # delete items 2 to 4
```

Concatenation: One list can be concatenated(appended) at the end of another as shown below:

```
lst = [3, 4, 5, 6]
lst = lst + [1, 2, 3]

print(lst)      # prints [3, 4, 5, 6, 1, 2, 3]
```

Merging: Two lists can be merged to create a new list

```
a = [10, 20, 30]
b = [100, 200, 300]

c = a + b

print(c)        # prints [10, 20, 30, 100, 200, 300]
```

Conversion: A string/tuple/set can be converted into a list using the list() conversion function.

```
string:
l = list("Africa")
print(l)       # prints ['A', 'f', 'r', 'i', 'c', 'a']

tuple:
t = (10, "Hello", True, 23.12)
l = list(t)
print(l)        # prints [10, 'Hello', True, 23.12]

set
s = {45, "Apple", True, 23.12}
l = list(s)
print(l)        # prints [True, 23.12, 45, 'Apple']
```

```
dict
d = {

    "Age": 10,
    "Name": "Arjun",
    "isAdult": True,
    "Per": 78.23
}
l = list(d)
print(l)    # prints ['Age', 'Name', 'isAdult', 'Per']
```

```
Aliasing: On assigning one list to another, both refer to the same list. Changing one
changes the other. This assignment is often known as shallow copy or aliasing.

lst1 = [10, 20, 30, 40, 50]

lst2 = lst1     # doesn't copy the content of lst1 to lst2, lst2 refers or represent the
lst1(we can say that it's an another name for the lst1, we can see below both the lists
i.e. lst1 and lst2 are having same memory address)

index1 = id(lst1)
index2 = id(lst2)

if index1 == index2:
    print("Both the List have same Memory Address")

else:
    print("Both the List have different Memory Address")

OUTPUT:
Both the List have same Memory Address

print(lst1)     # prints [10, 20, 30, 40, 50]
print(lst2)     # prints [10, 20, 30, 40, 50]

lst1[0] = 100

print(lst1)     # prints [100, 20, 30, 40, 50]
print(lst2)     # prints [100, 20, 30, 40, 50]
```

```
Cloning: This involves coping contents of one list into another. After copying both
refer to different lists, though both contain same values. Changing one list, doesn't
change another.

lst1 = [10, 20, 30, 40, 50]
lst2 = []

lst2 = lst2 + lst1
```

```
index1 = id(lst1)
index2 = id(lst2)

if index1 == index2:
    print("Both the List have same Memory Address")

else:
    print("Both the List have different Memory Address")

OUTPUT:
Both the List have different Memory Address

print(lst1)     # prints [10, 20, 30, 40, 50]
print(lst2)     # prints [10, 20, 30, 40, 50]

lst1[0] = 100

print(lst1)     # prints [100, 20, 30, 40, 50]
print(lst2)     # prints [10, 20, 30, 40, 50]
```

```
There is an alternative we can use copy()
copy(): Creates a shallow copy of the list

original_list = [1, 2, 3]

new_list = original_list.copy()

print(new_list)

Output: [1, 2, 3]
```

```
Identity: Whether the two variables are referring to the same list can be checked using
the is identity operator as shown below.

lst1 = [10, 20, 30, 40, 50]
lst2 = [10, 20, 30, 40, 50]

lst3 = lst1

print(lst1 is lst2)         # prints False
print(lst1 is lst3)         # prints True
print(lst1 is not lst2)     # prints True

Identity Difference between int and str

num1 = 10 ; s1 ="Hii"
num2 = 10 ; s2 ="Hii"


print(num1 is num2)     # prints True
print(s1 is s2)         # prints True
```

Searching: An element can be searched in a list using the in membership operator as shown below.

```
lst = ['a', 'e', 'i', 'o', 'u']

print('a' in lst)       # prints True since 'a' is present in list
print('b' in lst)       # prints False since 'b' is absent in list
print('z' not in lst)   # prints True since 'z' is absent in list
```

Comparison: It is possible to compare contents of two lists.
Comparison is done item by item till there is a mismatch.

```
lst1 = [1, 2, 3, 4, 5]
lst2 = [1, 2, 5]

print(lst1 < lst2)     # prints True
```

Emptiness: We can check is a list is empty using not operator

```
lst = []

if not lst:
    print("Empty List")

else:
    print("Not an Empty List")
```

OUTPUT: Empty List

Built-in Functions on List

```
len(): Returns number of items in the list
num = [10, 10, 10, 10, 10, 10, 10]

length = len(num)

print(length)       # prints 7
```

```
max(): Returns maximum element in the list

Example 1:
num = [741, 852, 963, 789, 456, 123, 456]

maximum1 = max(num)

print(maximum1)      # prints 963


Example 2:
lst = ["Hello,", 78.13, 53, True, "world!"]

maximum2 = max(lst)

print(maximum2)

OUTPUT: error (max() does not supports if there is a str in list)


Example 3:
lst = ["Hello",  "world!"]

maximum3 = max(lst)

print(maximum3)

OUTPUT: world!
```

```
min(): Returns minimum element in the list

Example 1:

num = [741, 852, 963, 789, 456, 123, 456]

minimum1 = min(num)

print(minimum1)      # prints 123
```

```
Example 2:
lst = ["Hello,", 78.13, 53, True, "world!"]

minimum2 = min(lst)

print(minimum2)

OUTPUT: error (min() does not supports if their is a str in list)


Example 3:

lst = ["Hello",  "world!"]

minimum3 = min(lst)

print(minimum3)

OUTPUT: Hello
```

```
sum(): return sum of all elements in the list

Example 1:
num = num = [741, 852, 963, 789, 456, 123, 456]

print(sum(num))     # prints 4380


Example 2:
lst = [78.13, 53, True]

print(sum(lst))         # prints 132.13
```

```
Instead of using the 'and' and 'or' logical operator, we can use the built in functions
all() and any() to get the same effect.

a, b, c = 10, 20, 30

# t = (a>5, b>20, c>15)
# print(t)          # prints (True, False, True)

result1 = all((a>5, b>20, c>15))
print(result1)          # False, as second condition is False
print(type(result1))    # prints <class 'bool'>


result2 = any((a>5, b>20, c>15))
print(result2)           # True, as one of the condition is True
print(type(result2))     # prints <class 'bool'>
```

all(): Function returns True if all elements of its parameter are True.

Example: Check is all elements in the list are greater than 0

```
my_list = [1, 3, 4, 5, 6, 7]

print(all(x>0 for x in my_list))
        # 1>0 for x in my_list      # True
        # 3>0 for x in my_list      # True
        # 4>0 for x in my_list      # True
        # 5>0 for x in my_list      # True
        # 6>0 for x in my_list      # True
        # 7>0 for x in my_list      # True
```

As we can see all the conditions are True so the OUTPUT is True, because all() checks that every condition is True

any(): Function returns True is at least one elements of its parameter are True.

```
my_list = [1, 3, 5, 6, 7]

print(any(x%2 == 0 for x in my_list))
```

OUTPUT: True (6%2 == 0)

del(): deletes element or slice or entire list

```
lst = [10, 20, 30, 40, 50]

del(lst[3])
print(lst)      # prints [10, 20, 30, 50]

del(lst[2:5])
print(lst)      # prints [10, 20]

del(lst[:])     # delete entire list
print(lst)

lst = []        # another way to delete an entire list
```

clear(): Removes all elements from the list

```
my_list = [1, 2, 3]
print(id(my_list))

my_list.clear()
print(my_list)
print(id(my_list))
```

OUTPUT: []

```
sort(): Returns sorted list, list remains unchanged

lst = [10, 2, 0, 50, 4]
index1 = id(lst)

lst.sort()
index2 = id(lst)
print(lst)      # prints [0, 2, 4, 10, 50]


if index1 == index2:
        print("Memory Address is same")
else:
        print("Memory Address is different")
```

```
sorted(): Returns sorted list, Another list has to be created, and takes original list
as a parameter
lst1 = [10, 2, 0, 50, 4]
index1 = id(lst1)

lst2 = sorted(lst1)
index2 = id(lst2)
print(lst2)      # prints [0, 2, 4, 10, 50]


if index1 == index2:
        print("Memory Address is same")
else:
        print("Memory Address is different")
```

```
reverse(): Returns Reversed list, list remains unchanged
lst = [10, 2, 0, 50, 4]

lst.reverse()
print(lst)      # prints [4, 50, 0, 2, 10]
```

```
reversed(): Returns Reversed list, Another list has to be created, and takes original
list as a parameter
lst1 = [10, 2, 0, 50, 4]
index1 = id(lst1)

lst2 = reversed(lst1)
index2 = id(lst2)
print(lst2)      # prints [0, 2, 4, 10, 50]


if index1 == index2:
        print("Memory Address is same")
else:
        print("Memory Address is different")
```

List Methods

append(): Adds an element to the end of the list

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

my_list.append(4)

print(my_list)

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

extend(one value): Extends the list by adding all elements from another iterable (i.e. list)

```
list1 = [1, 2, 3]

list2 = [4, 5]

list1.extend(list2)

print(list1)

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

insert(index, value): Inserts an element at a specified position in the list

```
my_list = [85, 65, 97]

my_list.insert(1, 4) # here 1 is the Index and 4 is the Value to be inserted on the index 1

print(my_list)
OUTPUT: [85, 4, 65, 97]
```

remove(value): Removes the first occurrence of a value from the list

```
Example 1:
my_list = [1, 2, 3]

my_list.remove(2) # here 2 is the Value from list

print(my_list)
Output: [1, 3]

Example 2:
my_list = [1, 2, 3]

my_list.remove(30) # here 30 is value that is not present in the list which is out of range so error is given

print(my_list)     # Output: error
```

pop(index): Removes and returns the element at a specified index in the list

Example 1:

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

popped_element = my_list.pop(1) # here 1 is the index Value

print(popped_element)
print(my_list)
OUTPUT: 2
       [1, 3]
```

Example 2:

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

popped_element = my_list.pop() # here index is not mentioned in () so pop() removes last item in list

print(popped_element)
print(my_list)
OUTPUT: 3
        [1, 2]
```

Example 3:

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

popped_element = my_list.pop(30) # here 30 is index which is out of range so error is given

print(popped_element)
print(my_list)
OUTPUT: error
```

count(value): Returns the number of occurrences of a value in the list

```
my_list1 = [1, 2, 2, 3, 2]
my_list2 = ["Yash", "Swaraj", "Yash"]

count1 = my_list1.count(2)
count2 = my_list2.count("Yash")

print(count1) # OUTPUT: 3
print(count2) # OUTPUT: 2
```

index(value): Returns the index of the first occurrence of a value in the list

Example 1:

```
my_list = [1, 2, 3, 4]

print(my_list.index(3)) # here 3 is the Value
OUTPUT: 2
```

```
Example 2:
my_list = [1, 2, 3, 4]

print(my_list.index(45)) # here 45 is the Value which is not present in the List

OUTPUT: error
```

```
copy(): Creates a shallow copy of the list

original_list = [1, 2, 3]

new_list = original_list.copy()

print(new_list)
Output: [1, 2, 3]
```

```
clear(): Removes all elements from the list

my_list = [1, 2, 3]

my_list.clear()

print(my_list)

OUTPUT: []
```

List Varieties

It is possible to create a nested list

```
a = [1, 3, 5, 7, 9]
b = [2, 4, 6, 8, 10]

c = [a, b]
print(c)

print(c[0][0] , c[1][2]) # 0th Element of 0th list, 2nd element of 1st list
```

A list may be embedded in another list

```
x = [1, 2, 3, 4];
y = [10, 20, x, 30];

print(y)        # prints [10, 20, [1, 2, 3, 4], 30]
```

It is possible to unpack a string or list within a list using the * operator

```
s = "Hello"
l = [*s]
print(l)    # prints ['H', 'e', 'l', 'l', 'o']

x = [1, 2, 3, 4]

y = [10, 20, 30];
y = [10, 20, *x, 30];

print(y)    # prints [10, 20, 1, 2, 3, 4, 30]
```

TUPLE

- Tuple is an ordered sequences of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and that are usually faster than list as it cannot change dynamically.
- It is defined within parentheses ( ) where items are separated by comma (,). A tuple data type in Python programming is similar to a list data type, which also contains heterogeneous items/elements.
- A tuple is also a linear data structure. A Python tuple is a sequence of data values called as items or elements. A tuple is a collection of items which is ordered and unchangeable.
- A tuple is a heterogeneous data structure and used for grouping data. Each element or value that is inside of a tuple is called an item.
- A tuple is an immutable data type. An immutable data type means that we cannot add or remove items from the tuple data structure.
- Values in tuples can also be accessed by their index values, which are integers starting from 0.
- In short, tuples are just like lists, but you cannot change their values.

```
The tuple data is enclosed within () as shown below.

a = ()      # empty list
b = (10,)   # tuple with one item. ',' after 10 is neccesary
c = ("Sanjay", 25, 34555.50)    # tuple with dissimilar items
d = (10, 20, 30, 40)    #tuple with similar items


While initializing a tuple, we may drop '()'

c = "Sanjay", 25, True, 45.23
print(type(c))      # prints <class 'tuple'>
```

```
Tuple items repeatation

tpl1 = (10,) * 5  # stores (10, 10, 10, 10, 10)

tpl2 = (10) * 5   # Basically it is not a tuple. Beacuse tuple with one item. ',' after
10 is neccesary
print(tpl2)       # stores 50
print(type(tpl2)) # <class 'int'>
```

```
Accessing Tuple Elements

Example 1: Entire Tuple can be printed by just using the name of the tuple

tpl = ('Sanjay', 25, 34555.50)
print(tpl)      # prints ('Sanjay', 25, 34555.5)


Example 2: Like strings and list, individual elements in a tuple can be accessed using
indices starting with 0.


Tuple is an ordered collection. So order of insertion of elements in a tuple is same as
the order of access.


msg = ('Handle', 'Exceptions', 'Like', 'a', 'boss')
print(msg[1], msg[3])

OUTPUT: Exceptions a

Example 3: Like strings and list, tuples can be sliced to yield smaller tuples.

# tuple_object[start:stop:step]
# start : The index where the slice begins.
# stop : The index where the slice ends (exclusive) means -1.
# step : The step or stride between elements.


animal = ("Zebra", 155.5, 110)
ages = (25, 26, 25, 27, 26)

print(animal[1:3])      # prints [155.5, 110]
print(ages[3:])         # prints [27, 26]




my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

# Slicing from index 2 to index 7 (exclusive) with a step of 2
sliced_tuple = my_tuple[2:7:2]

print(sliced_tuple) # prints (3, 5, 7)
```

Looping in Tuples

If we wish to process each item in a tuple, we should be able to iterate through it. This can be done using a while loop or for loop.

Example 1:
using while loop

```
tpl = (10, 20, 30, 40, 50)
i = 0

while i < len(tpl):
    print(tpl[i])
    i += 1
```

OUTPUT:
10
20
30
40
50

Example 2:
using for loop

```
for n in tpl:
    print(n)
```

OUTPUT:
10
20
30
40
50

Example 3:
While iterating through a tuple using a for loop, if we wish to keep track of index of the element that is being currently processed, we can do so using the built-in enumerate() function.

```
tpl = (10, 20, 30, 40, 50)

for index, n in enumerate(tpl):
    print(index, n)
```

OUTPUT:
0 10
1 20
2 30
3 40
4 50

Tuple Operations

Mutability - Unlike a list, a tuple is immutable, i.e. it cannot be modified.

```
msg = ("Fall", "In", "Line")

msg[0] = "FALL"     # error
msg[1:3] = ("Above", "Mark")    # error
```

Since a tuple is immutable operations like .append(), .extend(), .insert(), .remove(), and .clear() don't have these methods.

Additionally, tuples don't support the del statement on items.

```
point = (7, 14, 21)
del point[2]    # error
```

Though a tuple itself is immutable, it can contain mutable objects like lists.
Mutable lists, immutable string—all can belong to tuple

```
s = ([1, 2, 3, 4], [4, 5], 'Ocelot')
```

If a tuple contains a list, the list can be modified since list is a mutable object.

```
s = ([1, 2, 3, 4], [10, 20], "Onida")

s[1][1] = 45   # changes first item of list at index 1, i.e 20

print(s)    ([1, 2, 3, 4], [10, 45], 'Onida')
```

one more way to change first item of first list

```
p = s[1]    # s[1] is [10, 20] so now p is another name of the list situated at s[1]

print(p)

p[1] = 100
print(s) # here due to aliasing list is mutable so list in tuple s is also modified
```

OUTPUT: ([1, 2, 3, 4], [10, 100], 'Onida')

Concatenation: One tuple can be concatenated at the end of another as shown below:
In this after concatenating another tuple object has been created

```
tpl = (3, 4, 5, 6)
print(id(tpl))      # prints 3183380673328

tpl += (1, 2, 3)

print(tpl)      # prints (3, 4, 5, 6, 1, 2, 3)
print(id(tpl))      # prints 3183375938848
```

Since tuple is immutable after concatenating another tuple object has been created and another memory address is assigned

Merging: Two tuple can be merged to create a new list

```
a = (10, 20, 30)
b = (100, 200, 300)

c = a + b

print(c)        # prints (10, 20, 30, 100, 200, 300)
```

Conversion: A string/list/set can be converted into a tuple using the tuple() conversion function.

string:

```
l = tuple("Africa")
print(l)       # prints ('A', 'f', 'r', 'i', 'c', 'a')
```

list:

```
l = [10, "Hello", True, 23.12]
t = tuple(l)
print(t)        # prints (10, 'Hello', True, 23.12)
```

set

```
s = {45, "Apple", True, 23.12}
t = tuple(s)
print(t)        # prints (True, 'Apple', 45, 23.12)
```

dict

```
d = {
    "Age": 10,
    "Name": "Arjun",
    "isAdult": True,
    "Per": 78.23
}
t = tuple(d)
print(t)    # prints ('Age', 'Name', 'isAdult', 'Per')
```

Identity: Whether the two variables are referring to the same tuple can be checked using the is identity operator as shown below.

```
tpl1 = (10, 20, 30, 40, 50)
tpl2 = (10, 20, 30, 40, 50)

tpl3 = tpl1

print(tpl1 is tpl2)         # prints False
print(tpl1 is tpl3)         # prints True
print(tpl1 is not tpl2)     # prints True
```

Searching: An element can be searched in a tuple using the in membership operator as shown below.

```
tpl = ('a', 'e', 'i', 'o', 'u')

print('a' in tpl)       # prints True since 'a' is present in tuple
print('b' in tpl)       # prints False since 'b' is absent in tuple
print('z' not in tpl)   # prints True since 'z' is absent in tuple
```

Comparison: It is possible to compare contents of two tuples.
Comparison is on done item by item till there is a mismatch.

```
tpl1 = (1, 2, 3, 4, 5)
tpl2 = (1, 2, 5)

print(tpl1 < tpl2)     # prints True
```

Emptiness: We can check is a tuple is empty using not operator

```
tpl = ()

if not tpl:
    print("Empty tuple")

else:
    print("Not an Empty tuple")
```

OUTPUT: Empty tuple

Built-in Functions on Tuple

len(): Returns number of items in the tuple

```
num = (10, 10, 10, 10, 10, 10, 10)
length = len(num)
print(length)       # prints 7
```

max(): Returns maximum element in the tuple

Example 1:

```
num = (741, 852, 963, 789, 456, 123, 456)
maximum1 = max(num)
print(maximum1)      # prints 963
```

Example 2:

```
tpl = ("Hello,", 78.13, 53, True, "world!")
maximum2 = max(tpl)
print(maximum2)
OUTPUT: error (max() does not supports if there is a str in tuple)
```

Example 3:

```
tpl = ("Hello",  "world!")
maximum3 = max(tpl)
print(maximum3)
OUTPUT: world!
```

min(): Returns minimum element in the tuple

Example 1:

```
num = (741, 852, 963, 789, 456, 123, 456)
minimum1 = min(num)
print(minimum1)      # prints 123
```

Example 2:

```
tpl = ("Hello,", 78.13, 53, True, "world!")
minimum2 = min(tpl)
print(minimum2)
OUTPUT: error (min() does not supports if their is a str in list)
```

Example 3:

```
tpl = ("Hello",  "world!")
minimum3 = min(tpl)
print(minimum3)
OUTPUT: Hello
```

sum(): return sum of all elements in the tuple

Example 1:

```
num = (741, 852, 963, 789, 456, 123, 456)
print(sum(num))     # prints 4380
```

Example 2:

```
tpl = (78.13, 53, True)
print(sum(tpl))         # prints 132.13
```

Instead of using the 'and' and 'or' logical operator, we can use the built in functions all() and any() to get the same effect.

```
a, b, c = 10, 20, 30

# t = (a>5, b>20, c>15)
# print(t)          # prints (True, False, True)

result1 = all((a>5, b>20, c>15))
print(result1)          # False, as second condition is False
print(type(result1))    # prints <class 'bool'>


result2 = any((a>5, b>20, c>15))
print(result2)           # True, as one of the condition is True
print(type(result2))     # prints <class 'bool'>
```

all(): Function returns True if all elements of its parameter are True.

Example: Check is all elements in the tuple are greater than 0

```
my_tuple = (1, 3, 4, 5, 6, 7)
print(all(x>0 for x in my_tuple))
```

As we can see all the conditions are True so the OUTPUT is True, because all() checks that every condition is True

any(): Function returns True is at least one elements of its parameter are True.

```
my_tuple = (1, 3, 5, 6, 7)
print(any(x%2 == 0 for x in my_tuple))
OUTPUT: True (6%2 == 0)
```

Aliasing:

```
student_info = ("Linda", 18, ["Math", "Physics", "History"])
student_profile = student_info

id(student_info) == id(student_profile)         # True
id(student_info[0]) == id(student_profile[0])   # True
id(student_info[1]) == id(student_profile[1])   # True
id(student_info[2]) == id(student_profile[2])   # True
                    OR
student_info = ("Linda", 18, ["Math", "Physics", "History"])
student_profile = copy(student_info)

id(student_info) == id(student_profile)         # True
id(student_info[0]) == id(student_profile[0])   # True
id(student_info[1]) == id(student_profile[1])   # True
id(student_info[2]) == id(student_profile[2])   # True

student_profile[2][2] = "Computer science"
student_profile         # ('Linda', 18, ['Math', 'Physics', 'Computer science'])
student_info            # ('Linda', 18, ['Math', 'Physics', 'Computer science'])
```

```
deepcopy()

student_info = ("Linda", 18, ["Math", "Physics", "History"])

student_profile = deepcopy(student_info)

id(student_info) == id(student_profile)    # False
id(student_info[0]) == id(student_profile[0])   # True
id(student_info[1]) == id(student_profile[1])   # True
id(student_info[2]) == id(student_profile[2])   # False


student_profile[2][2] = "Computer science"
print(student_profile)  # ('Linda', 18, ['Math', 'Physics', 'Computer science'])
print(student_info)     # ('Linda', 18, ['Math', 'Physics', 'History'])
```

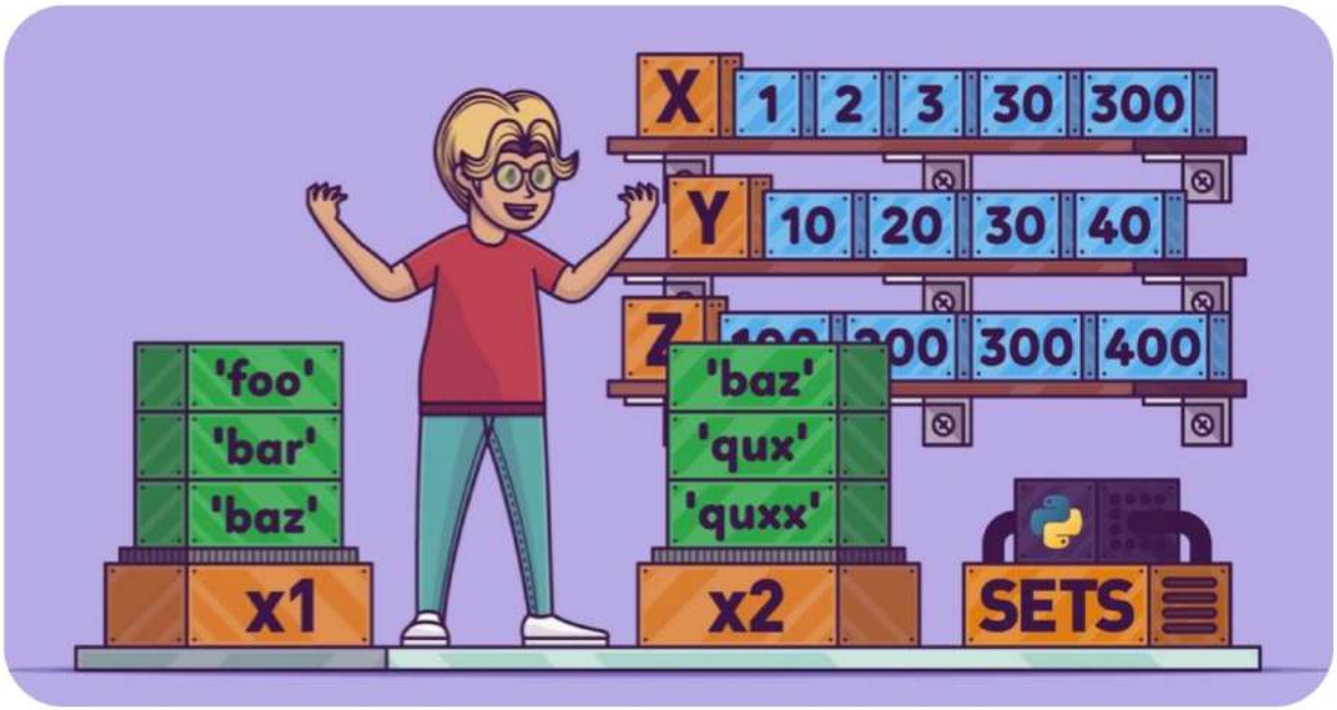
```
sorted(): Sorts the elements of the tuple and returns a new sorted list, takes original
tuple as a parameter


tpl1 = (10, 2, 0, 50, 4)
index1 = id(tpl1)

tpl2 = sorted(tpl1)
index2 = id(tpl2)
print(tpl2)      # prints [0, 2, 4, 10, 50]
print(tuple(tpl2))      # prints (0, 2, 4, 10, 50)

if index1 == index2:
        print("Memory Address is same")
else:
        print("Memory Address is different")
```

```
reversed(): Returns Reversed tuple in hexadecimal, Another tuple has to be created, and
takes original tuple as a parameter

tpl1 = (10, 2, 0, 50, 4)
index1 = id(tpl1)

reversed(tpl1)
index2 = id(tpl1)
print(tuple(reversed(tpl1)))      # prints (4, 50, 0, 2, 10)

if index1 == index2:
        print("Memory Address is same") # True
else:
        print("Memory Address is different")

print(tpl1)       # prints (10, 2, 0, 50, 4)
Original tuple is unchanged
```

```
tpl = (12, 15, 13, 23, 22)

count(value):  return no. of times i.e. value appears in tuple
print(tpl.count(23))

index(value): return index of item
print(tpl.index(22)) # prints 4
print(tpl.index(50)) # reports valueError as 50 is absent in lst
```

```
It is possible to create a tuple of tuples.
Example 1:
a = (1, 3, 5, 7, 9)
b = (2, 4, 6, 8, 10)

c = (a, b)
print(c)    # prints ((1, 3, 5, 7, 9), (2, 4, 6, 8, 10))

print(c[0][0], c[1][2]) # 0th element of 0th tuple, 2nd element of 1st tuple
OUTPUT: 1 6

Example 2:
records = (
            ('Priyanka', 24, 3455.50), ('Shailesh', 25, 4555.50),
            ('Subhash', 25, 4505.50), ('Sugandh', 27, 4455.55)
          )

print(records[0][0], records[0][1], records[0][2])      # prints Priyanka 24 3455.5
print(records[1][0], records[1][1], records[1][2])      # prints Shailesh 25 4555.5

for n, a, s in records:
    print(n,a,s)
OUTPUT:
Priyanka 24 3455.5
Shailesh 25 4555.5
Subhash 25 4505.5
Sugandh 27 4455.55

A tuple may be embedded in another tuple.
x = (1, 2, 3, 4)
y = (10, 20, x, 30)
print(y)    # outputs (10, 20, (1, 2, 3, 4), 30)
```

```
It is possible to unpack a tuple within a tuple using the * operator.
x = (1, 2, 3, 4)
y = (10, 20, *x, 30)
print(y) # outputs (10, 20, 1, 2, 3, 4, 30)

Tuple Unpacking:
my_tuple = (1, 2, 3)
a, b, c = my_tuple
print(a)     # Output: 1
print(b)     # Output: 2
print(c)     # Output: 3
```



SETS

- The set data structure in Python programming is implemented to support mathematical set operations. Set is an unordered collection of unique items. Items stored in a set are not kept in any particular order.
- A set data structure in Python programming includes an unordered collection of items without duplicates. Sets are un-indexed that means we cannot access set items by referring to an index.
- Sets are changeable (mutable) i.e., we can change or update a set as and when required. Type of elements in a set need to be the same; various mixed data type value can also be passed to the set.
- The sets in Python are typically used for mathematical operations like union, intersection, difference and complements etc.

```
a = {} # This way of representing empty set is wrong
a = set()   # empty set, use () instead of {}

b = {20}    # set with one item
c = {"Sanjay", 25, 34555.505}    # set with multiple items

d = {10, 10, 10, 10, 10}
print(d)    # prints {10}
```

Note: While storing element in a set, its hash value is computed using a hashing technique to determine where it should be stored in the set.

What is Hashing:

Hashing is a fundamental data structure that efficiently stores and retrieves in a way that allows for quick access. It involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key. This method is commonly used in databases, caching systems, and various programming applications to optimize search and retrieval operations.

```
s = {12, 23, 45, 16, 52}
t = {16, 52, 12, 23, 45}
u = {52, 12, 16, 45, 23}

print(s)    # prints {16, 52, 23, 12, 45}
print(t)    # prints {16, 52, 23, 12, 45}
print(u)    # prints {16, 52, 23, 12, 45}
```

It is possible to create a set of strings and tuples, but not a set of lists

```
s1 = {"Morning", "Evening"}     # works because strings are immutable
s2 = {(12, 23), (15, 25), (17, 34)}     # works because tuples are immutable
s3 = {[12, 23], [15, 25], [17, 34]} # error because list are mutable
```

Since strings and tuples are immutable, their hash value remains same at all times. Hence a set of strings or tuples is permitted. However, a list may change, so its hash value may change, hence a set of lists is not permitted.

Accessing Set Elements

Entire set can be printed by just using the name of the set. Set is an unordered collection. Hence order of insertion is not same as the order of access.

```
s = {15, 25, 35, 45, 55}
print(s)    # prints {35, 45, 15, 55, 25}
```

Being an unordered collection, items in a set cannot be accessed using indices. Sets cannot be sliced using [ ].

Looping in Sets

Like strings, lists and tuples, sets too can be iterated over using a for loop.

```
s = {12, 15, 13, 23, 22, 16, 17}

for ele in s :
    print(ele)
```

OUTPUT:

```
16
17
22
23
12
13
15
```

Note that unlike a string, list or tuple, a while loop should not be used to access the set elements. This is because we cannot access a set element using an index, as in s[i]. Built-in function enumerate( ) can be used with a set. The enumeration is done on access order, not insertion order.

Basic Set Operations

```
Like lists Sets are mutable too. Their contents can be changed.
s = {'gate', 'fate', 'late'}
s.add('rate') # adds one more element to set s

If we want an immutable set, we should use a frozenset.
s = frozenset({'gate', 'fate', 'late'})
s.add('rate') # error


# Concatenation - doesn't work
# Merging - doesn't work

# While converting a set using set( ), repetitions are eliminated.
lst = [10, 20, 10, 30, 40, 50, 30]
s = set(lst) # will create set containing 10, 20, 30, 40, 50
```

Built-in Functions on Sets

```
Many built-in functions can be used with sets.

s = {10, 20, 30, 40, 50}
len(s)      # return number of items in set s
max(s)      # return maximum element in sets
min(s)      # return minimum element in set s
sorted(s)   # return sorted list (not sorted set)
sum(s)      # return sum of all elements in set s
any(s)      # return True if any element of s is True
all(s)      # return True if all elements of s are True
```

Set Methods

```
Any set is an object of type set. Its methods can be accessed using the syntax
s.method(). Usage of commonly used set methods is shown below:

s = {12, 15, 13, 23, 22, 16, 17}
t = {'A', 'B', 'C'}

u = set ( )        # empty set

s.add('Hello')     # adds 'Hello' to s
s.update(t)        # adds elements of t to s

u = s.copy( )      # performs deep copy (cloning)

s.remove(15)       # deletes 15 from s
s.remove(101)      # would raise error, as 101 is not a member of s
s.discard(12)      # removes 12 from s
s.discard(101)     # won't raise an error, though 101 is not in s
s.clear( )         # removes all elements
```

Following methods can be used on 2 sets to check the relationship between them:

```
s = {12, 15, 13, 23, 22, 16, 17}
t = {13, 15, 22}

print(s.issuperset(t)) # prints True
print(s.issubset(t))   # prints False
print(s.isdisjoint(t)) # prints False
```

Mathematical Set Operations

Following union, intersection and difference operations can be carried out on sets:

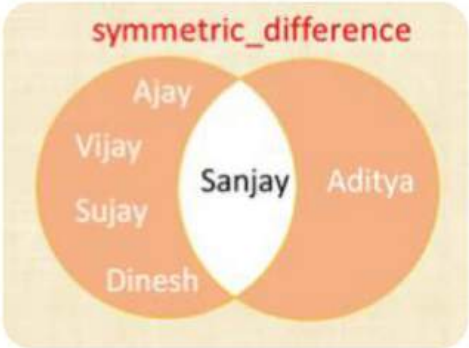
```
engineers = {'Vijay', 'Sanjay', 'Ajay', 'Sujay', 'Dinesh'}
managers = {'Aditya', 'Sanjay'}
```

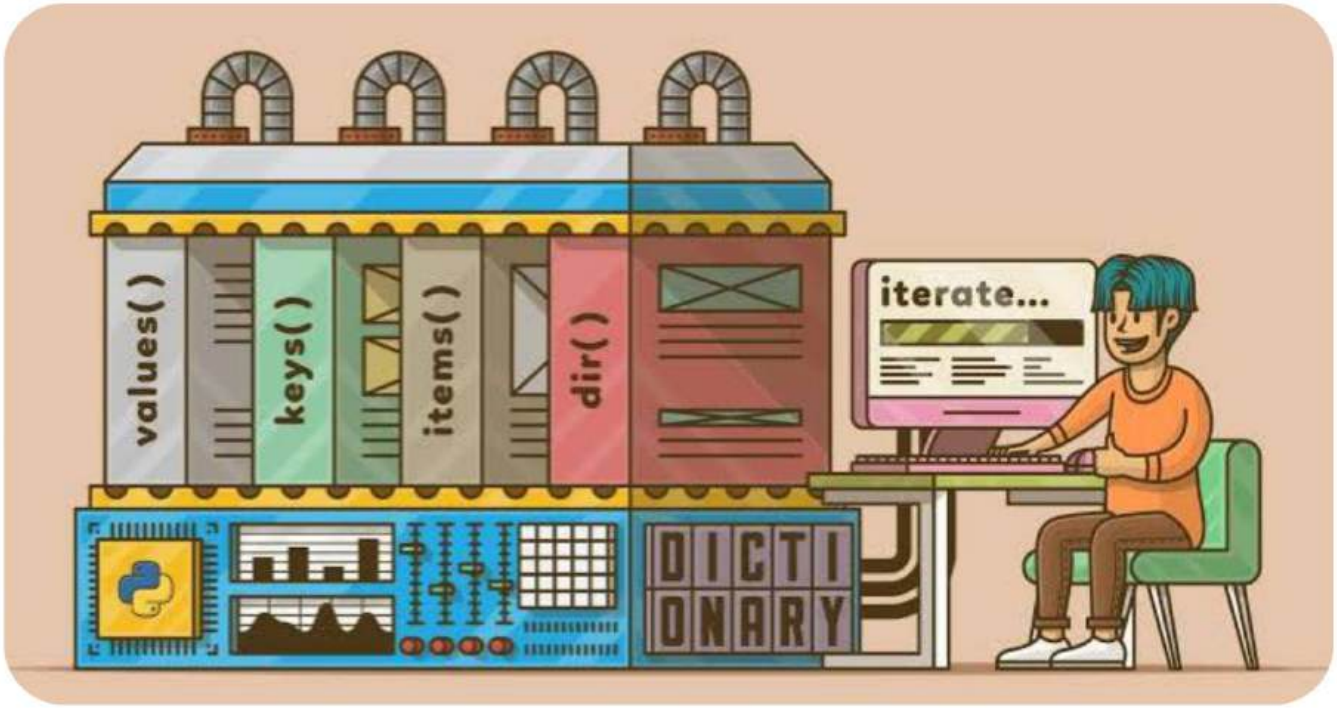

```
union - all people in both categories
print(engineers | managers)
OUTPUT:
{'Sujay', 'Ajay', 'Dinesh', 'Vijay', 'Aditya', 'Sanjay'}
```


```
intersection - who are engineers and managers
print(engineers & managers)
{'Sanjay'}
```

```
difference - engineers who are not managers
print(engineers - managers)
OUTPUT: {'Sujay', 'Ajay', 'Vijay', 'Dinesh'}

difference - managers who are not engineers
print(managers - engineers)
OUTPUT: {'Aditya'}
```


```
symmetric difference -
managers who are not engineers &
engineers who are not managers

print(managers ^ engineers)
OUTPUT: {'Sujay', 'Ajay', 'Dinesh', 'Vijay', 'Aditya'}
```





DICTIONARIES

- The dictionary data structure is used to store key value pairs indexed by keys. A dictionary is an associative data structure, means that elements/items are stored in a non-linear fashion.
- Python dictionary is an unordered collection of items or elements. Items stored in a dictionary are not kept in any particular order. The Python dictionary is a sequence of data values called as items or elements.
- While other compound data types have only value as an element, a dictionary has a `key:value` pair. Each value is associated with a key.
- Dictionaries are optimized to retrieve values when the key is known. A key and its value are separated by a colon (:) between them.
- The items or elements in a dictionary are separated by commas and all the elements must be enclosed in curly braces.

  Syntax: to define a dictionary in Python programming is as follows:

  <dictionary_name> = {**key1:**value1, **key2:**value2, **key3:**value3}

Example:
```
Emp = {"ID": 20, "Name": "Vijay", "Gender": "Male", "SalaryPerHr": 40}
```

- A pair of curly braces with no values in between is known as `empty dictionary`. Dictionary items are accessed by keys, not by their position (index).
- The values in a dictionary can be duplicated, but the keys in the dictionary are unique. Dictionaries are changeable (mutable). We can change or update the items in dictionary as and when required.
- The key can be looked up in much the same way that we can look up a word in a paper-based dictionary to access its definition i.e., the word is the `'key'` and the definition is its corresponding `'value'`.
- Dictionaries can be nested i.e. a dictionary can contain another dictionary.

Creating Dictionary

```
d1 = { }     # empty dictionary

d2 = {'A101' : 'Amol', 'A102' : 'Anil', 'B103' : 'Ravi'}
Here, 'A101', 'A102', 'B103' are keys, whereas, 'Amol', 'Anil', 'Ravi' are values.

Different keys may have same values.
d = {10 : 'A', 20 : 'A', 30 : 'Z'}    # ok


Keys must be unique. If keys are same, but values are different, latest key value pair
get stored.
d = {10 : 'A', 20 : 'B', 10 : 'Z'}    # will store {10 : 'Z', 20 : 'B'}

If key value pairs are repeated, then only one pair gets stored.
d = {10 : 'A', 20 : 'B', 10 : 'A'}    # will store {10 : 'A', 20 : 'B'}
```

```
Creating dictionary using dict()

d1 = dict({1:"Orange", 2:"Mango", 3:"Banana"})
print(d1)   # prints {1: 'Orange', 2: 'Mango', 3: 'Banana'}

d2 = dict([(1,"Red"), (2,"Yellow"), (3,"Green")])
print(d2)   # prints {1: 'Red', 2: 'Yellow', 3: 'Green'}

d3 = dict(one=1, two=2, three=3)
print(d3)   # prints {'one': 1, 'two': 2, 'three': 3}
```

```
Constructing a Dictionary
We can create an empty dictionary as follows:
d = {} or d = dict()

d = {}
print(type(d))      # prints <class 'dict'>

d = dict()
print(type(d))      # prints <class 'dict'>
```

```
data = {1:"Pune", 2:"Noida", 3:"Delhi"}

print(data[3])      # prints Delhi
print(data[1])      # prints Pune
```

```
Unlike sets, dictionaries preserve insertion order. However, elements are not accessed
using the position, but using the key.

d = {'A101' : 'Dinesh', 'A102' : 'Shrikant', 'B103' : 'Sudhir'}
print(d['A102'])   # prints value for key 'A102' i.e. Shrikant

Thus, elements are not position indexed, but key indexed.
Dictionaries cannot be sliced using [ ].
```

```
info1 = {
    "Name": "Mani",
    "Occupation": "Doctor",
    "Year": 1998
}
print(info1)


We can also create dictionary like this
info2 = {"Name": "Mani", "Occupation": "Doctor","Year":1998}

print(info2)
```

```
We can add entries as follows
info = {
    "Name": "Mani",
    "Occupation": "Doctor",
    "Year": 1998
}

info["Place"] = "Pune"
print(info)
OUTPUT: {'Name': 'Mani', 'Occupation': 'Doctor', 'Year': 1998, 'Place': 'Pune'}
```

```
It's important to note that dictionaries are very flexible in the data types they can
hold. For example:
my_dict = {"key1": 123, "key2": [11, 12, 13], "key3": ["d", "a", "t", "a"]}

print(my_dict)   # prints {'key1': 123, 'key2': [11, 12, 13], 'key3': ('d','a','t','a')}

Let's call items from the dictionary
print(my_dict["key3"])      # prints ('d', 'a', 't', 'a')

Can call an index on that value
print(my_dict["key2"][1])   # prints 12

Can even call methods on that value
print(my_dict["key3"][1].upper())   # prints A
```

We can affect the values of a key as well. For instance:

```
my_dict["key1"] = my_dict["key1"] - 120
print(my_dict["key1"])  # prints 3

print(my_dict)
OUTPUT: {'key1': 3, 'key2': [11, 12, 13], 'key3': ['d', 'a', 't', 'a']}
```

Looping in Dictionaries

Like strings, lists, tuples and sets, dictionaries too can be iterated over using a for loop. There are three ways to do so:

```
courses = {'DAA' : 'CS', 'AOA' : 'ME', 'SVY' : 'CE' }


iterate over key-value pairs
for k, v in courses.items( ) :
    print(k, v)



iterate over keys
for k in courses.keys( ) :
    print(k)




iterate over keys - shorter way
for k in courses :
    print(k)



iterate over values
for v in courses.values( ) :
    print(v)
```

While iterating through a dictionary using a for loop, if we wish to keep track of index of the key-value pairs that is being referred to, we can do so using the built-in enumerate( ) function.

```
courses = {'DAA' : 'CS', 'AOA' : 'ME', 'SVY' : 'CE' }

for i, (k, v) in enumerate(courses.items( )) :
    print(i,k,v)
```

Note that ( ) around k, v are necessary.

Dictionary Operations

Dictionaries are mutable. So we can perform add/delete/modify operations on a dictionary.

We can also create keys by assignment. For instance, if we started with an empty dictionary. We could continually add to it:

```
Creating a new dictionary
d = {}

Creating a new key through assignment
d["Animal"] = "Dog"
print(d)        # prints {'Animal': 'Dog'}


d["Number"] = 42
print(d)        # prints {'Animal': 'Dog', 'Number': 42}


Update value at key
d["Animal"] = "Cat"
print(d)        # prints {'Animal': 'Cat', 'Number': 42}
```

If the specified key is not available then we will get KeyError

```
d = {100:"Puneet", 200:"Rohit", 300:"Aman"}
print(d[500])

OUTPUT :
KeyError
```

We can avoid error using in operator

```
if 500 in d:
    print(d[500])
else:
    print("Key Not Present!")

OUTPUT: Key Not Present!
```

```
Courses = {101:'CPP', 102:'DS', 201:'OOP', 226:'DAA', 601:'Crypt', 442:'Web'}

add, modify, delete
courses[444] = 'Web Services'   # add new key-value pair
courses[201] = 'OOP Using java' # modify value for a key
del(courses[102]) # delete a key-value pair
del(courses)      # delete dictionary object
```

Note that any new addition will take place at the end of the existing dictionary, since dictionary preserves the insertion order.
Dictionary keys cannot be changed in place.

Given below are the operations that work on lists and tuples.

Identity: You can check if two dictionaries are the same object using the is operator. The is operator checks if two variables point to the same object in memory.

```
dict1 = {'a': 1, 'b': 2}
dict2 = {'a': 1, 'b': 2}
print(dict1 is dict2) # prints False
```

Emptiness:
The not operator is used to check if a dictionary is empty.

```
dict3 = {}
if not dict3:
    print("Dictionary is empty")
```

OUTPUT:
Dictionary is empty

Conversion

```
dict4 = {'a': 1, 'b': 2}
list_of_tuples = list(dict4.items())
print(list_of_tuples)
```

OUTPUT:

```
[('a', 1), ('b', 2)]
```

Aliasing: You can create an alias for a dictionary by assigning it to another variable. Assigning one dictionary to another variable creates an alias, meaning both variables point to the same object in memory.

```
dict5 = {'a': 1, 'b': 2}
dict6 = dict5
dict6['c'] = 3
print(dict5)      # prints {'a': 1, 'b': 2, 'c': 3}
```

Searching: You can check if a key exists in a dictionary using the in operator.

```
dict7 = {'a': 1, 'b': 2}
if 'a' in dict7:
print("Key 'a' exists in the dictionary")
```

Cloning: Certainly! In Python, there are a few ways to clone or create a copy of a dictionary. Here are some methods along with examples and explanations:

1. Using the copy() method:
- The copy() method creates a shallow copy of the dictionary.
- Example:

```
dict1 = {'a': 1, 'b': 2}
dict2 = dict1.copy()
dict2['c'] = 3
print(dict1) # {'a': 1, 'b': 2}
print(dict2) # {'a': 1, 'b': 2, 'c': 3}
```

Explanation: In this example, dict2 is a shallow copy of dict1. Modifying dict2 does not affect dict1.

2. Using the dict() constructor:
- The dict() constructor can be used to create a new dictionary from an existing dictio.
- Example:

```
dict3 = {'a': 1, 'b': 2}
dict4 = dict(dict3)
dict4['c'] = 3
print(dict3) # {'a': 1, 'b': 2}
print(dict4) # {'a': 1, 'b': 2, 'c': 3}
```

Explanation: In this example, dict4 is created using the dict() constructor with dict3 as an argument. Modifying dict4 does not affect dict3.

3. Using dictionary unpacking:
- Dictionary unpacking can be used to create a copy of a dictionary.
- Example:

```
dict5 = {'a': 1, 'b': 2}
dict6 = {**dict5}
dict6['c'] = 3
print(dict5) # {'a': 1, 'b': 2}
print(dict6) # {'a': 1, 'b': 2, 'c': 3}
```

Explanation: In this example, dict6 is created by unpacking dict5 using the ** operator. Modifying dict6 does not affect dict5.

4. Using the copy module:
- The copy module provides the deepcopy() function to create a deep copy of a dictionar.
- Example:

```
dict7 = {'a': [1, 2], 'b': [3, 4]}
dict8 = copy.deepcopy(dict7)
dict8['a'].append(5)
print(dict7) # {'a': [1, 2], 'b': [3, 4]}
print(dict8) # {'a': [1, 2, 5], 'b': [3, 4]}
```

Explanation: In this example, dict8 is a deep copy of dict7. Modifying the nested list in dict8 does not affect dict7.

Comparison - doesn't work: Two dictionary objects cannot be compared using <, >.

Concatenation - doesn't work Merging - doesn't work: Two dictionaries cannot be concatenated using +. Two dictionaries cannot be merged using the form z = s + t.

Built-in Functions on Dictionaries

Many built-in functions can be used with dictionaries.

```
d = {101 : 'CPP', 102 : 'DS', 201 : 'OOP'}

len(d)      # return number of key-value pairs
max(d)      # return maximum key in dictionary d
min(d)      # return minimum key in dictionary d
sorted(d)   # return sorted list of keys
sum(d)      # return sum of all keys if keys are numbers
any(d)      # return True if any key of dictionary d is True
all(d)      # return True if all keys of dictionary d are True


reversed(d)  # can be used for reversing dict/keys/values

courses = {101 : 'CPP', 102 : 'DS', 201 : 'OOP'}

for k, v in reversed(courses.items( )) :
    print(k, v)

OUTPUT:
201 OOP
102 DS
101 CPP
```

Dictionary Methods

There are many dictionary methods. Many of the operations performed by them can also be performed using built-in functions. The useful dictionary methods are shown below:

```
c = { 'CS101' : 'CPP', 'CS102' : 'DS', 'CS201' : 'OOP'}
d = { 'ME126' : 'HPE', 'ME102' : 'TOM', 'ME234' : 'AEM'}

print(c.get('CS102', 'Absent')) # prints DS
print(c.get('EE102', 'Absent')) # prints Absent
print(c['EE102'])    # raises keyerror
c.update(d)          # updates c with items in d
print(c.popitem( ))  # removes and returnsitem in LIFO order
print(c.pop('CS102'))# removes key and returns its value

c.clear( ) # clears all dictionary entries

animals = {'Tiger' : 141, 'Lion' : 152, 'Leopard' : 110}
birds = {'Eagle' : 38, 'Crow': 3, 'Parrot' : 2}

combined = {** animals, ** birds }

lst = [12, 13, 14, 15, 16]
d = dict.fromkeys(lst, 25) # keys - list items, all values set to 25

print(d)  # prints {12: 25, 13: 25, 14: 25, 15: 25, 16: 25}
```

```python
contacts = [
    {"name": "Prashant", "number": "1234567890"},
    {"name": "Nikhil", "number": "1112131415"},
    {"name": "Swaraj", "number": "1617181920"}
]

name = input("Enter Name: ")

for person in contacts:
    if person["name"] == name:
        number = person["number"]
        print(f"Found {number}")
```

Property	String	List	Tuple	Set	Dictionary
Object	✓	✓	✓	✓	✓
Collection	✓	✓	✓	✓	✓
Mutable		✓		✓	✓
Ordered	✓	✓	✓		
Indexed by position	✓	✓	✓		
Indexed by key					✓
Iterable	✓	✓	✓	✓	✓
Slicing allowed	✓	✓	✓		
Nesting allowed	✓	✓	✓		
Heterogeneous elements		✓	✓	✓	✓