A short of code can reduce
lots of human work.

FIVE RUDIMENTS OF PYTHON

String,List,Tuple,Set,
Dictionary

Ravishankar Chavare

Deepak Chauhan

Dedicated To

Python World Community Developers. I have tried to summarise the knowledge in easy langugae which is already available in the Internet world.

About Us

Ravishankar Chavare(Author)
LinkedIN | **Github**

Completed bachelor of engineering(Computer science and engineering) in MIT Aurangabad in the year of 2016. Automated different workflows with the help of python programming.Mentored “CINFO” project in kharagpur winter of code conducted by IIT Kharagpur.

Deepak Chauhan(Editor)
LinkedIn | **Github**

Experienced Coordinator with a demonstrated history of working in the computer software industry.Skilled in Python, C++, C, Microsoft Excel, Microsoft Word, Public Speaking, Microsoft Office, and Management. Strong professional with a Master of Computer Applications MCA focused in Computer Programming, Specific Applications from National Institute of Technology Jamshedpur.

Prerequisite

- Use a python version greater than 3.6 . All examples used in books are written on python version 3.6 .
- Download python from here https://www.python.org/downloads/

Index

Chapter 1. String

Chapter 2.List

Chapter 3. Tuple

Chapter 4. Set

Chapter 5. Dictionary 126

Chapter - 01
STRINGS

1.Strings

1.1 Introduction

A string is the collection of alphabets, numbers, and special characters. Strings play a vital role in any programming language. Python doesn't support **char** data type. Python uses a string as a data type for storing single character, word, a paragraph in the string data type.

1.1.1 Definition

Strings are created with single inverted commas or double inverted commas. Alphanumeric and special characters enclosed within single or double inverted commas are called Strings.

Syntax

```
Variable_name  =  "Value"
```

1.1.2 Single Line string

A bunch of alphanumeric or special characters within single or double quotes are defined on a single line.

1.2.1 Single quotation mark.

```
Variable_name =  'string Value'
```

1.2.2 Double quotation mark.

```
Variable_name =  "string Value"
```

1.2.3 Parentheses

```
Variable_name =  ("string Value")
```

1.1.3 MultiLine String

A bunch of lines text can be defined with the help of a combination of a single quote (3 Times) Before and after of stings.

1.3.1 Single Quotation mark

```
Variable_name = '''First Line
Second Line
Third Line'''
```

1.3.2 Double Quotation mark

```
Variable_name = """First Line
Second Line
Third Line"""
```

1.1.4 Unicode Format

1.4.1.1 Unicode Example

All Unicode characters from other than English considered strings in python. Unicode in python is defined as follows.

Example

```
unicode_example = "\U00008000"
print(f"Unicode Example:{unicode_example}")

marathi_example = "अ"
print(f"Marathi Example:{marathi_example}")

bengali_example = "ঀ"
print(f"Bengali Example:{bengali_example}")
```

Output

```
Unicode Example:耀
Marathi Example:अ
Bengali Example:ঀ
```

1.1.5 Other Format

1. Contains ‘

```
Variable_name = "it's"
```

2. Contains “

```
Variable_name = 'it"s'
```

1.2. Escape Special Characters

In some situations, we need to print a string containing special characters such as ` **or ".** We use a special symbol named backslash (\) before the special symbol to remove the special meaning of single apostrophe (‘) and double apostrophe (“).

Syntax

```
use / (backslash) before the ' or " to remove the special meaning
```

The wrong way to define a string

```
Text = 'it's cold'
# it will produce a SyntaxError: invalid syntax
```

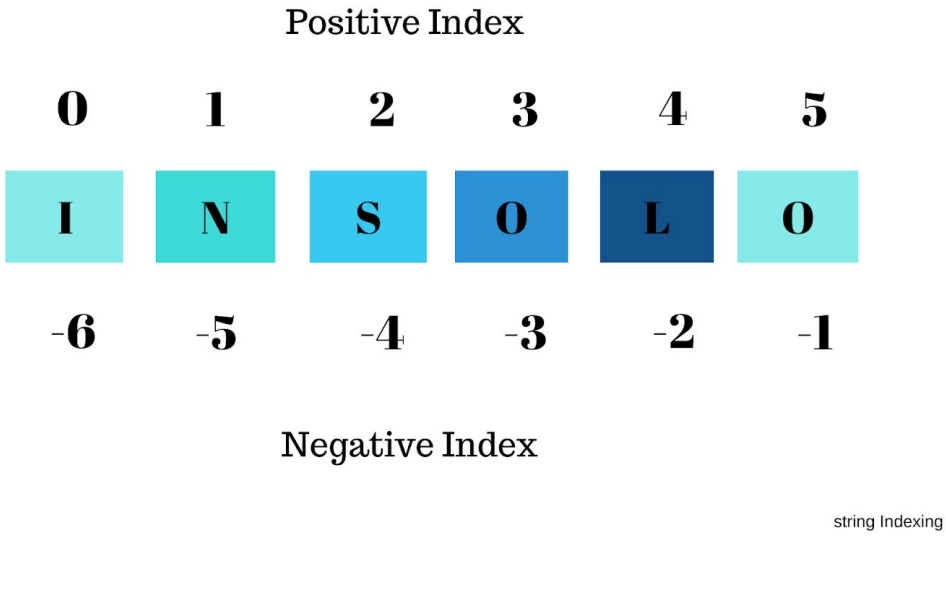
In the above program, the string is enclosed within a single quote('). The final string should be considered until the second single quote string after second quotes has some mismatch no of quotes so it will generate an exception.

The correct way to define the above string

```
Text1 = "it's cold"
#or Another Way
Text2 = 'it\'s cold'
```

1.3. Access String

Every character from a string can be accessed using positive and negative indices. Elements or characters are accessed from an input string with the help of integer indexes where the positive index starts from 0 from the beginning of the string and the negative index starts from -1 from the end of the string.



(image 1.1)

Example

```
# simple string example
text  = "Insolo"

#print string
print(text)
```

Output

```
Insolo
```

1.3.1 Positive Indices

Strings first element accessed with 0th indexes second element accessed with index 1 and third element accessed with index 2. To access strings elements write a positive index in square brackets after string name. If given an index which is larger than the element in string then python gives an error string index out of range.

Syntax

```
StringName[positive index]
```

Where

String Name: String Variable Name or a String object.

Index: Positive Number.

Example

```
text  = "Insolo"

‘’’ I:0  n:1 s:2 o:3 l:4 o:5 ‘’’
# Access First Element(I)
print(text[0])
```

Output

```
I
```

If Index Out of range

```
text  = "Insolo"
print(text[100])
```

It will Produce an Exception

```
in <module>
    print(text[100])
IndexError: string index out of range
```

1.3.2 Negative Indices

Python supports negative indexing to access elements from a string. The last element from the string is positioned at -1 index. The second last element is positioned at -2. A larger index than the length of the string will create an exception as an index out of range.

Syntax

```
String_name[negative index]
```

Where

StringName : String Variable Name or String object.
Index : Negative Number.

Example

```
text  = "Insolo"

# Last Element using Negative Index
print("index(last element) -1 :",text[-1])

# First Element using Negative Index
print("index(first element) -6 :",text[-6])
```

Output

```
index(last element) -1 : o
index(first element) -6 : I
```

The following example will create an exception

```
text  = "Insolo"
print("index -1 :",text[-80])
```

Error

```
IndexError: string index out of range
```

1.3.3 Slicing String

To access a single element from a string we can pass a single positive or negative index after the string name in square braces. But when we want to access multiple elements from a string slicing is used. Slicing is a technique to cut a string into slices(pieces). With the help of slicing, we can create a substring. Slicing doesn't give any error when you provide out of range positive or negative indexes for extracting substring.

Syntax:

```
string_name[start:end:step]
```

where

start

- The start is the starting index of the string and must be numeric.
- start can be empty If you are extracting substring from 0th indexes.
- Elements positioned at the start index are included in the final substring.
- Start can be a positive index or negative index.
- Example :0,1,5,-1,-2

end

- The end is the ending index of a string that must be numeric.
- Elements positioned at the end index are excluded from the final substring.
- the end can be empty If you are extracting substring till the last element.
- The end can be a positive index or negative index.
- Example :0,1,5,-1,-2.

step

- Numbers.
- The step is an optional parameter.
- The step is the difference or stride.
- Example :0,1,5,-1,-2.

Example

```
text = "Hello Python"

# for substring Hello H is positioned at 0th index and o is positioned
at 4 index
start = 0
end = 4

substring = text[start:end]
print(substring)
```

Output

```
Hell
```

but we need to print **Hello** where end **o** is positioned at 4th index and end element in slicing is excluded so we get output Hell. To get a full substring Hello we just need to increment the end index by 1 (4+1) now the end is 5 and the start is 0.

Example

```
text = "Hello Python"
print(text[0:5])
print(text[:5])
```

Output

```
Hello
```

Other Some Examples

```
text = "Hello Python"
text[6:8] : Py
text[6:12]: Python
text[6:]  : Python
```

1.3.3.1 Positive step parameter

When we want to Extract substring from a full normal string use start and end parameter as empty and pass only step(difference parameter). When the step parameter used in slicing is also called extended slicing.

Example

```
text = "Hello Python"

#Extract substring : HloPto
substring=text[::2]
print(substring)
```

Output

```
HloPto
```

When you provide a start and end indexes as greater than the length of string slicing did not create any exception.

Two points to Remembers

1) When the start value is greater than the negative length of the string the string extractions start from the 0^{th} index.

Example

```
text = "Hi"
print(text[-5:])
```

Output

```
Hi
```

The length of the text string is 2 and the start element is given as -5 which is greater than (-2). String extracted starts from the first element.

2) When the end value is greater than the negative length of the string then the final string should be empty.

Example

```
text = "Hi"
print(text[:-5])
```

Output

```
""
```

Here the position of every Element

Element	Positive Index	Negative Index
	4	-5
	3	-4
	2	-3
H	1	-2
i	0	-1

text[:-5] where start=0 and end=-5 so the final output of string is empty.

Example

```
text = "Hello Python"
first = text[0:1000]
print(first)

negative = text[-100:100]
print(first)
```

Output

```
Hello Python
Hello Python
```

1.3.3.2 Negative step parameter

For slicing a string we can also use negative indexing. The negative step parameter reads a given string from the right side to the left side (End to Beginning).

Syntax

```
string_variable[start:end:<Negative Number>]
```

Where

Start: integer number, starting index of string, included in the result

End: integer number, ending index of string, excluded from the result.

Step: Negative Number, difference, or step.

Example

```
text = "Hello Python"

#To reverse a complete string
reverse=text[::-1]
print(reverse)
```

Output

```
nohtyP olleH
```

Above Example start to read a string from ending and read until the first character.

Get every Second Element From the last element to the first element

```
text = "Hello Python"
reverse=text[::-2]
print(reverse)
```

Output

```
nhy le
```

In the above example, the python interpreter reads the string from the end and returns every 2nd element until the first element.

1.4 Strings Methods

Python has provided a set of built-in methods to manipulate strings. After applying a specific method to string it will return a new string it will not affect the original string because the string object is immutable. To get a list of all available methods in string use dir(str).

Example

```
methods = [val for val in dir(str) if val[0]!='_']
print(methods)
```

Output

```
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

1.4.1 Basic Methods

1.4.1.1 capitalize

capitalize() is a python build in method to return a capitalized string. This method creates the first element of string capitalized.

Syntax

```
string_variable.capitalize()
```

Example

```
text = "python"
result = text.capitalize()
print(result)
```

Output

```
Python
```

1.4.1.2 upper

upper() is a python string object built-in method used to return a copy of the uppercase string.

Syntax

```
string_variable.upper()
```

Example

```
text = "python"
result = text.upper()
print(result)
```

Output

```
PYTHON
```

1.4.1.3 lower

lower() built-in method string object returns a copy of the lowercase string.

Syntax

```
string_variable.lower()
```

Example

```
text = "PYTHON"
result = text.lower()
print(result)
```

Output

```
python
```

1.4.1.4 title

title() method returns a string where every first character of that string is capitalized.

Syntax

```
string_variable.title()
```

Example

```
text = "hello python"
result = text.title()
print(result)
```

Output

```
Hello Python
```

1.4.1.5 swapcase

swapcase() method converts lowercase characters to the uppercase character and uppercase character to lowercase characters.

Syntax

```
string_variable.swapcase()
```

Example

```
text = "HELLO python"
result = text.swapcase()
print(result)
```

Output

```
hello PYTHON
```

1.4.1.6 index

index() method returns the first index in the string if the string is found. Raise an exception if the substring is not found in the input string. If you want to find substrings in the user-specified area(index) just pass the start and end index area where you want to find the substring.

Syntax

```
string_variable.index(<Substring>,start,end)
```

Example

```
text = "Python"
result = text.index('n')
print(result)
```

Output

```
5
```

An Exception is raised when a string is not found

```
text = "Python"
text.index('N') # ValueError: substring not found
```

Output

```
ValueError: substring not found
```

Search substring Index at a specified position (search **o** after Hello)

```
text = "Hello Python"
index=text.index('o',6,12)
print(index)
```

Output

```
10
```

1.4.1.7 find

find() method returns the first index if substring found in string otherwise returns -1.

Syntax

```
string_variable.find(<Substring>,start,end)
```

Example : Find l in string

```
text = "Hello Python"
index=text.find('l')
print(index)
```

Output

```
2
```

In the above example, l is placed at 2 and 3 indexes but text.find('l') returns 2 its lowest index (first occurrence).

Example : Find '**l**' in 6 to 12 index

```
text = "Hello Python"
index=text.find('l',6,12)
print(index)
```

Output

```
-1
```

In the above example, l is not placed between 6 and 12 so the output is -1.

1.4.2 Boolean Methods

1.4.2.1 isalpha

isalpha() method returns

True : All characters of a string are alphabets

False : If any other non-alphabetic characters.

Syntax

```
string.isalpha()
```

Example

```
text="Python"
res=text.isalpha()
print(res)
```

Output

```
True
```

Other Examples

```
"Python1"  :    False
"Python$"  :    False
"Hello"    :    True
"1234"     :    False
"$#"       :    False
```

1.4.2.2 isdigit

isdigit() method returns

True : If all characters in strings are digit.

False : If any non-digit character.

Syntax

```
string.isdigit()
```

Example

```
text="123"
res=text.isdigit()
print(res)
```

Output

```
True
```

Some Other Examples

```
"123.0"    :    False
"123four"  :    False
""         :    False
"467" :    True
```

1.4.2.3 isdecimal

isdecimal() metod returns

True : If all characters are decimal.

False : If Any non-decimal character.

Syntax

```
string.isdecimal()
```

Example

```
text = "12"
res=text.isdecimal()
print(res)
```

Output

```
True
```

Other some examples

```
"10.0" :   False
"A"    :   False
".3"   :   False
```

1.4.2.3 islower

islower() method returns

True : If all characters from a string are lowercase.

False : If any character is uppercase.

Syntax

```
string.islower()
```

Example

```
text  = "hello"
res = text.islower()
print(res)
```

Output

```
True
```

Other Example

```
"&h"    :  True
"Hello" : False
```

1.4.2.4 isupper

isupper() method returns

True : If all characters from a string are Uppercase.

False : If any lower character is in the string.

Syntax

```
string.isupper()
```

Example

```
text  = "HELLO"
res = text.isupper()
print(res)
```

Output

```
True
```

Other Example

```
"Hello"  : False
"Hello"  : False
"PYTHON" : True
```

1.4.2.5 istitle

istitle() method returns

True : If the string is title cased.

False : If the string is not title cased.

Syntax

```
string.istitle()
```

Example

```
text  = "Hello"
res = text.istitle()
print(res)
```

Output

```
True
```

Other Examples

```
"hello"    :    False
```

1.4.3 Other Methods

1.4.3.1 split

The **split()** method returns a list of words in strings separated by a given delimiter. This method started to split string from starting of the string.

Syntax

```
text.split(sep=None, maxsplit=-1)
```

Where

Sep : Separator which wants to be used as a Delimiter. If the separator is not provided then strings are split using whitespace.

Maxsplit : Default a is -1, No of times to split(if 2 then string split in 3 str).

Example

```
text = "Red,Green,Blue,Yellow"

splited_text = text.split(",")
print(splited_text)
```

Output

```
['Red', 'Green', 'Blue', 'Yellow']
```

Using maxsplit parameter

```
text = "Red,Green,Blue,Yellow"

splited_text = text.split(",",2)
print(splited_text)
```

Output

```
['Red', 'Green', 'Blue,Yellow']
```

Other Example

```
text.split(",",1) : ['Red', 'Green,Blue,Yellow']
```

1.4.3.2 rsplit

rsplit() method splits the string from the right side and returns a list containing the words. If a separator is not provided then whitespace is considered as Separator.

Syntax

```
text.rsplit(sep=None, maxsplit=-1)
```

Example

```
text = "Red,Green,Blue,Yellow"

splited_text =text.rsplit(",",2)
print(splited_text)
```

Output

```
['Red,Green', 'Blue', 'Yellow']
```

Here we have provided comma as separator and the max split is 2 so rsplit() method split given string from the right side and split into 3 part (only 2 commas used to split)

Some more Examples

```
text = "Red,Green,Blue,Yellow"
text.rsplit()        :    ['Red,Green,Blue,Yellow']
text.rsplit(",")     :    ['Red', 'Green', 'Blue', 'Yellow']
text.rsplit(',',2)   :    ['Red,Green', 'Blue', 'Yellow']
text.rsplit(',',4)   :    ['Red', 'Green', 'Blue', 'Yellow']
```

1.4.3.3 join

join() methods return a concatenated string from an iterable object with given input as a delimiter.

Syntax

```
"<separator>".join(iterable)
```

where

Separator : Anything which you want to use as a separator after joining the string.

Iterable : Any Iterable object such as list, tuple, set which contains string elements.

Join a list with comma(,)

```
lst = ['Red', 'Green', 'Blue', 'Yellow']
result_string = ",".join(lst)
print(result_string)
```

Output

```
Red,Green,Blue,Yellow
```

all colors from the list joined together to form a final string and add a comma(,) as a separator.

Following Example Generate an error

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

```
result_string = ",".join(lst)
print(result_string)
```

Error

```
result_string = ",".join(lst)
TypeError: sequence item 0: expected str instance, int found
```

You can convert list element to string and solve above program

```
lst = [1,2,3,4]
result_string = ",".join(map(str,lst))
print(result_string)
```

Output

```
1,2,3,4
```

Here we have converted an integer element list into string data type by using map(str,lst) and joining that list element with a comma(,).

Some Other Examples

```
#String
text = "Hello"
print(",".join(text))
# Output : H,e,l,l,o

#List
lst = ['a','b','c']
print("".join(lst))
# Output : abc

#Tuple
tple = ('b','c','d',)
print("-".join(tple))
# Output : b-c-d

#Set
st = {'p','y','t','h','o','n'}
print("".join(st))
# Output : otpnhy
```

Output

```
H,e,l,l,o
abc
b-c-d
ytnoph
```

1.4.3.4 strip

strip() method removes leading and trailing characters from a given string. if characters are given then strip() removes leading and trailing characters from a given string.

Syntax

```
string.strip([char])
```

where

char : any character which needs to remove from start and end of the string, if char is not provided then default value spaces will be removed from front and end of the strings.

Example

```
text = "  Python  "
stripped_text = text.strip()
print(stripped_text)
```

Output

```
Python
```

Remove $ from end and start of string

```
text = "$$$Python$$$"
stripped_text = text.strip("$")
print(stripped_text)
```

Output

```
Python
```

1.4.3.5 lstrip

lstrip() method removes leading space from the input string and returns that new string.

Syntax

```
string.lstrip([char])
```

Example

```
text = "    Python   "
striped_text = text.rstrip()
print(striped_text)
```

Output

```
“Python   ”
```

1.4.3.6 rstrip

rstrip() method removes Trailing space from the input string and returns that new string.

Syntax

```
string.rstrip([char])
```

Example

```
text = "    Python   "
striped_text = text.rstrip()
print(striped_text)
```

Output

```
“     Python”
```

1.4.3.7 index

index() method returns a first integer index if the substring is found in input string if the substring is not found it raises an error. index() method starts searching given substring from the beginning of the string.

Syntax

```
string_variable.index(<Substring>,start,end)
```

Start and *end* parameters are optional if these parameters are not given then the substring is searched in the whole input string.

Example

```
text = "Python"

#Search Index of t
t_index = text.index("t")
print(t_index)
```

Output

```
2
```

Raise an Exception when substring not foundException

```
ValueError: substring not found
```

Some other Example

```
text = "Python"
text.index("y") : 1
text.index("n") : 5
text.index("h") : 3
```

Find **o** after index 6

```
text = "Hello Python"
# Find o after index 6
index = text.index('o',6,)
print(index)
```

Output

```
10
```

1.4.3.8 rindex

rindex() method is similar to the *index()* method but the difference is that rindex() finds out the highest index of the substring in the input *string.rindex*() search element from ending(right side to left side) of the string.

Syntax

```
string_variable.index(<Substring>,start,end)
```

Where

Substring : Substring which you want to find.
Start : Starting position index.
End : Ending position index .

Example

```
text = "Hello Python"

#Find out first and last index of o
first_index = text.index('o')
last_index = text.rindex('o')

print(f'Given string           : {text}')
print(f'o found at first index : {first_index}')
print(f'o found at last index  : {last_index}')
```

Output

```
Given string           : Hello Python
o found at first index : 4
o found at last index  : 10
```

1.4.3.9 find

find() method is similar to the index method but the only difference is that **find()** returns -1 when the substring is not found in the given string where **index()** method raises an exception. *find()* returns the lowest index of substring.find() method starts searching given substring from the beginning(Left side to the right) and if substring found return index otherwise return -1 if the substring is not found.

Syntax

```
string_variable.find(<Substring>,start,end)
```

1.4.3.10 rfind

rfind() method starts searching elements from ending (right side to left). it returns -1 if substring not found otherwise return the index of the found substring. It returns the highest index of the substring.

Syntax

```
string_variable.rfind(<Substring>,start,end)
```

Example

```
text = "Hello Python"

#Find out first and last index of o
first_index = text.find('o')
last_index = text.rfind('o')
not_index = text.find('m')
print(f'Given string           : {text}')
print(f'o found at first index : {first_index}')
print(f'o found at last index  : {last_index}')
print(f'If Substring is not found : {not_index}')
```

Output

```
Given string           : Hello Python
o found at first index : 4
o found at last index  : 10
If Substring is not found : -1
```

1.4.3.11 startswith

startswith() method **True** if the string starts with given prefix otherwise return **False**. You can also pass the start and end parameter to check the given prefix in the range of *start* and *end*. *startswith()* compared with case sensitive.

Syntax

```
string_variable.startswith(<prefix>,start,end)
```

Where

prefix : Any character or symbol. It can be tuple if multiple elements you want to try.
start : Integer, *start* index of specified string
end : Integer, *end* index of the specified string

Example

```
text = "Hello Python"

# Check String starts with H or not
result = text.startswith("H")
print(f'String Starts With H: {result}')

start_withh = text.startswith('h')
print(f'String Starts With h: {start_withh}')

start_withk = text.startswith('k')
print(f'String Starts With k: {start_withk}')
```

Output

```
String Starts With H: True
String Starts With h: False
String Starts With k: False
```

Example With Multiple prefix

```
text = "Hello Python"
print(f"Given Input String :{text}")

res = text.startswith(('h','H'))
print(f"Starts with h or H :{res}")

res = text.startswith(('P','H'))
print(f"Starts with P or H :{res}")

res = text.startswith(('K','h'))
print(f"Starts with K or h :{res}")
```

Output

```
Given Input String :Hello Python
Starts with h or H :True
Starts with P or H :True
Starts with K or h :False
```

Some Other Example

```
text = "Hello Python"
text.startswith("P",6)  : True
text.startswith('n')    : False
text.startswith("e",1,4): True
text.startswith(('b','k')):False
text.startswith(('n','t'),8) :True
text.startswith(" ",5) : True
```

1.4.3.12 endswith

endswith() method returns **True** if input string ends with given suffix otherwise returns **False.**

Syntax

```
string_variable.endswith(<suffix>,start,end)
```

Where
suffix: any character or symbol. It can be tuple if there are multiple elements you want to try.
start : integer, *start* index of the specified string.
end : integer, *end* index of the specified string.

Example

```
text = "Hello Python"

result = text.endswith("n")
print(f"text ends with n :{result}")

result = text.endswith("o")
print(f"text ends with o :{result}")

result = text.endswith("o",0,5)
print(f"Hello ends with o :{result}")

result = text.endswith(("k","o"),0,5)
print(f"Hello ends with o or k :{result}")

result = text.endswith(("n","N"))
print(f"Hello ends with n or N :{result}")
```

Output

```
text ends with n :True
text ends with o :False
Hello ends with o :True
Hello ends with o or k :True
Hello ends with n or N :True
```

1.4.3.13 zfill

zfill() method is used to add leading zeros(0) to given string or character.

zfill() never truncates an original element. If *zfill()* input width is greater than or equal to string then zero are not added. Leading Zeros are only added when the length of the string is less than the given width to the *zfill()* method.

Syntax

```
string_variable.zfill(<width>)
```

original(t)	After Zfill(t.zfill(3))
0	000
2	002
3	003
10	010
50	050
100	100

Example

```
text = '1'

result = text.zfill(4)
print(result)
```

Output

```
0001
```

Example of Zfill using For Loop

```
for i in range(11):
  print(str(i).zfill(3))
```

Output

```
000
001
002
003
004
005
006
007
008
009
010
```

String Example

```
text = "Hello"

#Apply zfill width 10
result = text.zfill(10)

#print the result
print(result)
```

Output

```
00000Hello
```

In the above example, the length of the text is 5, and width 10 is given. So 5 leading zero are added to the final string.

```
Result = 00000 + "Hello"
Result = 00000Hello
```

1.5. String Formatting

- C style
- Using Built-in Method
- F string

When we want to combine different elements with each other string formatting can help us. A Simple example of String concatenation.

Example

```
name = "Python"
age = 25

result = "name is "+ name +" and age is "+str(age)
print(result)
```

Output

```
name is Python and age is 25
```

String Formatting Helps us to reduce the above code and add some simplicity to write in a proper manner.

1.5.1 C Style String Formatting

Python supports a C style string formatting. It can be achieved with the help of % Special Symbol.

Syntax

```
"%<Types>" % (<variables>)
```

Where Types from

```
%s    : for String
%d    : for integer
%f    : for float
%.<precision> : Floating Point Numbers
```

Variables by comma separator in the tuple

```
name = "Python"
age = 25

result = "name is %s and age is %d"%(name,age)
print(result)
```

Output

```
name is Python and age is 25
```

1.5.2 Format Built-in Method

String Formatting is one good way to join and combines two or more strings with different data. **format()** method returns a formatted string.

Syntax

```
string.format(*args,**kwargs)
```

Where the **format**() method accepts any n number of arguments and these arguments substituted in the string with the help of **{** and **}** in the string. We can pass position, index, or keyword inside {} to access elements.

Example

```
name = "Python"
age = 25

result = "name is {} and age is {}".format(name,age)
print(result)
```

Output

```
name is Python and age is 25
```

With the help of List

```
lst = ["Python","1989","100"]

result = " First Element:{} \n second Element:{} \n Third
Element:{}".format(*lst)
print(result)
```

Output

```
First Element:Python
second Element:1989
Third Element:100
```

Some Other Example

```
a,b,c = 4,5,6

#Using Position
result = "a :{} b :{} c:{}".format(a,b,c)
print(result)

#Using Index
result = "a :{1} b :{0} c:{2}".format(b,a,c)
print(result)

#Using Index
result = "a :{a} b :{b} c:{c}".format(b=b,a=a,c=c)
print(result)
```

Output

```
a :4 b :5 c:6
a :4 b :5 c:6
a :4 b :5 c:6
```

1.5.3 F String

F string is a new and improved way to format a string .*F string* is introduced in python 3.6. *F string* is more readable, faster, and easy to debug as compared to % formatting and *format()* method. Get value from the list and dictionary object which is passed to as input.

Syntax

```
f"{variable1} {variable2} ...."
```

Example

```
a,b,c = 4,5,6

result = f"a:{a} b:{b} c:{c}"
print(result)
```

Output

```
a:4 b:5 c:6
```

Fstring with dictionary objects

```
dct = {'name':'Python','age':20}

result = f"name is {dct['name']}"
print(result)
```

Output

```
name is Python
```

Example of List

```
lst = [1,2,3,4,5]
result = f"list object is {lst}"
print(result)
```

Output

```
list object is [1, 2, 3, 4, 5]
```

Perform normal mathematical Calculation

```
print(f"{5*5}")
```

Output

```
25
```

Use of *F string* To call a Function

```
def Add(a,b):
  return a+b

res = f"Addition of 5 and 6 is :{Add(5,6)}"
print(res)
```

Output

```
Addition of 5 and 6 is :11
```

Some Disadvantages of f strings are

1. Cannot include a backslash

   ```
   f"{'\Hello'}"
   ```

2. Cannot include # (Inline comment)

   ```
   f"{'Hello' #inlineComment}"
   ```

1.6. Solved string examples

1.6.1 Reverse a String

Example: Using [::-1]

```
def Reverse(text):
 return text[::-1]

text = "Python"
res = Reverse(text)
print(res)
```

Output

```
nohtyP
```

Example : Using while Loop

```
def Reverse(text):
  i = len(text)
  rev = ""
  while i!=0:
     rev+=text[i-1]
     i-=1
  return rev

text = "Python"
res = Reverse(text)
print(res) #nohtyP
```

Output

```
nohtyP
```

1.6.2 Palindrome

Create a function to check whether the given string is palindrome or not.

Some Example Use Case of Palindrome

Text	Reversed Text	Palindrome(True/False)
MADAM	MADAM	True
PYTHON	NOHTYP	False
MAM	MAM	True

Example: using [::-1]

```
def Palindrome(text):
  flag  = True

  if text != text[::-1]:
    flag = False
  return flag


text = "MADAM"
res = Palindrome(text)
print(res)
```

To achieve the same result You can also use the reversed**()** or **reverse()** method instead of using **text[::-1] .**

Output

```
True
```

Example 2 :

Text = “MADAM”
Length(l) = 5

index(i)	i	0	1	2	3	4
Text	t	M	A	D	A	M
Formula	l-(i+1)	5-(0+1)	5-(1+1)	5-(2+1)	5-(3+1)	5-(4+1)
Compare index	c	4	3	2	1	0
Compare now	t[i] == t[c]	True	True	True	True	True

All comparisons are true so the input string is a palindrome. If any of the comparisons become *false* that string is not Palindrome.

Example : Using For Loop

```
def Palindrome(text):
  flag  = True
  length = len(text)
  for i,val in enumerate(text):
      if val != text[length-(i+1)]:
            flag = False
            break
  return flag

text = "MADAM"
res = Palindrome(text)
print(res)
```

enumerate() is useful for obtaining an index of given list,string or iterable objects.

Output

```
True
```

Example : Using While Loop

```
def Palindrome(text):
  flag  = True
  length = len(text)
  i = 0
  while i!=length:
    if text[i] != text[length-(i+1)]:
      flag = False
      break
    i+=1
  return flag
text = "MADAM"
res = Palindrome(text)
print(res)
```

Output

```
True
```

Using For Loop

```
def Palindrome(text):
  flag  = False
  length = len(text)

  # Create Reverse String
  rev = ""
  for i in range(length,0,-1):
    rev = rev + text[i-1]

  # Compare Reversed String with Original String
  if text == rev:
    flag = True

  return flag

# Driver Code To Call Palindrome Function
text = "MADAM"
res = Palindrome(text)
print(res)
```

Output

```
True
```

Chapter - 02

LIST

2.List

2.1. Introduction

A *List* is a collection of different data elements that are changeable in the future whenever needed. The *list* is similar to an array but the only difference is that array store *homogeneous*(same data type elements) elements where the *list* can store *heterogeneous*(elements are mixtures with different data type) elements.

When you create a simple *list*, a python interpreter creates an array-like data structure in memory to hold your heterogeneous data. *List* indexing starts from 0 , where the first element placed at o index and last index placed at -1 index position. A *list* is a mutable object which means *List* allows modification after object creation. Where mutable objects have setter and getter properties but immutable objects have only getter properties they don't allow to modify the object after its creation.

Syntax

Blank list

```
blank_list = list()
# OR
blank_list = []
```

List with elements

```
list(iterable)
# OR
[item1,item2,item3......]
```

Where

iterable : Any Iterable object such as string, tuple, set, etc..

Example : Empty List

```
empty_list = list()
# or
blank_list = []

print(empty_list,blank_list)
```

Output

```
[]  []
```

Example : List Of Numbers

```
#Create Numbers List
number_list = [1,2,3,4,5,6,7]

print(number_list)
```

Output

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

Example : List Of String Elements

```
#Create String Elements List
string_list = ['H','E','L','L','O']

print(string_list)
```

Output

```
['H', 'E', 'L', 'L', 'O']
```

If you check the type of every element in the list it will be a string

Example: String to List

```
#Create String Elements List
strings = "HELLO"

string_list = list(strings)
```

```
print(string_list)
```

Output

```
['H', 'E', 'L', 'L', 'O']
```

Example : Mixed Types Data

```
#Create String Elements List
mixed_list = [1,'H',10.5,(1,2)]

print(mixed_list)
```

Output

```
[1, 'H', 10.5, (1, 2)]
```

mixed_list contains Heterogeneous elements

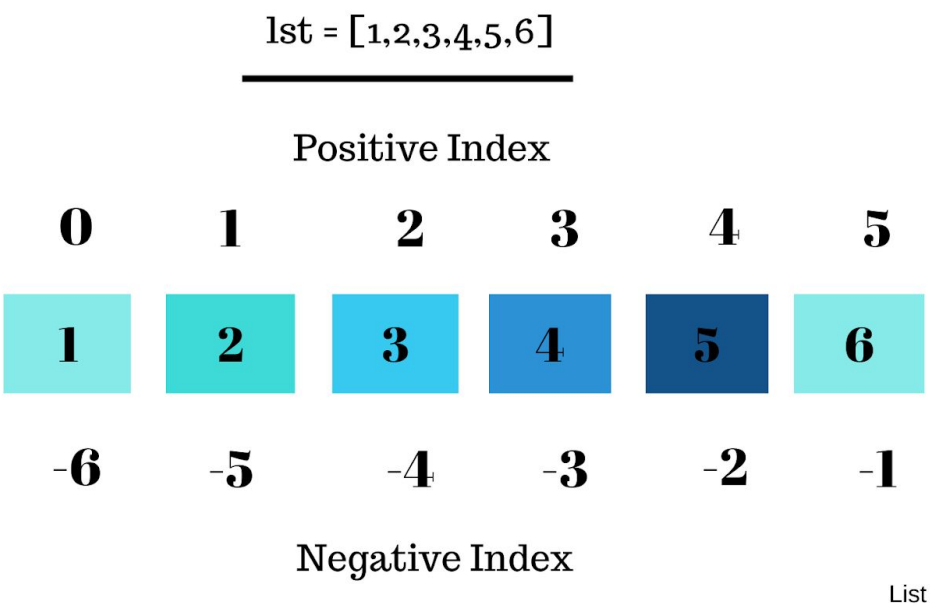
1 : int

H : str

10.5 :float

(1,2) : Tuple

2.2 Access List



List Indexing

Image (2.1)

2.2.1 Single Element

Consider an **N** element in the list, the list element starts from the 0^{th} index, and the *end* index is placed at (N-1).To access single elements from the list we can use positive or negative indexing. where the first element is placed at the 0th index and the last element is placed at -1 index. If out of range index is given while accessing elements it will raise an exception.

Syntax

```
list[index]
```

Where

Index : positive or negative integer.

Consider the following list

```
colors = ["Red","Blue","Green","Yellow"]
```

Elements	Positive Index	NegativeIndex
Red	0	-4
Blue	1	-3
Green	2	-2
Yellow	3	-1

In the above example colors is a python list containing 4 elements so the length of the colors list is 4. The red element is placed at the 0^{th} index. Blue elements are placed at 1 index in this way all elements are positioned one after one. The last element Yellow is positioned at the 3rd index. Similarly, the list element can be accessed using negative indexes. The last element is positioned at -1, and the second last element is placed at -2.

Example: Get the first Element

```
colors = ["Red","Blue","Green","Yellow"]

first_positive = colors[0]  # List Start at 0th index
first_negative = colors[-4] # (-Length) of list

print(f'Original List : {colors}')
print(f'Elements at 0th index  : {first_positive}')
print(f'Elements at -4 index : {first_negative}')
```

output

```
Original List : ['Red', 'Blue', 'Green', 'Yellow']
Elements at 0th index  : Red
Elements at -4 index : Red
```

Example : Get Last Element

```
colors = ["Red","Blue","Green","Yellow"]

last_positive = colors[3]  # (length-1)
last_negative = colors[-1] # Last Element at -1 index

print(f'Original List : {colors}')
print(f'Elements at 3 index  : {last_positive}')
print(f'Elements at -1 index : {last_negative}')
```

Output

```
Original List : ['Red', 'Blue', 'Green', 'Yellow']
Elements at 3 index  : Yellow
Elements at -1 index : Yellow
```

Example : Get third Element

```
colors = ["Red","Blue","Green","Yellow"]

third_positive = colors[2]  # Third Element placed at index 2
third_negative = colors[-2]

print(f'Original List : {colors}')
print(f'Elements at 3 index  : {third_positive}')
print(f'Elements at -1 index : {third_negative}')
```

Output

```
Original List : ['Red', 'Blue', 'Green', 'Yellow']
Elements at 3 index  : Green
Elements at -1 index : Green
```

Example :Exception Out of Range

```
colors = ["Red","Blue","Green","Yellow"]
print(colors[6])
```

Above program raise an exception `IndexError: list index out of range`

2.2.2 Slicing

Slicing helps to fetch multiple elements from a list on a given range basis. slicing can work with positive and negative indexes. When out of range slicing is given it doesn't create any exception in such a scenario it will return a blank list.

Syntax

```
list[start:end]
```

Where

start : positive or negative integer number which is starting index of the list,

the start index is included in the result list.

end : positive or negative integer number which is ending index of the list,

the end index is excluded from the result list.

Example : Fetch first 2 elements

```
colors = ["Red","Blue","Green","Yellow"]

# Fetch Red and Blue
start = 0
end = 2

first_two = colors[start:end]
print(first_two)
```

Output

```
['Red', 'Blue']
```

We have provided start as 0 and end index as 2. The element at 0^{th} index is **Red** is included in the result set . element at 1^{st} index Blue is included in the final result set. Elements are included whose index is in [0,1] . Element at index 2 is excluded from the result .

Example : Fetch ['Red', 'Blue']

```
colors = ["Red","Blue","Green","Yellow"]


first2_pos = colors[:2]   # Positive Index
first2_neg = colors[-4:-2] # Negative Index

print(f'Original List : {colors}')
print(f'colors[:2]    : {first2_pos}')
print(f'colors[-4:-2] : {first2_neg}')
```

Output

```
Original List : ['Red', 'Blue', 'Green', 'Yellow']
colors[:2]    : ['Red', 'Blue']
colors[-4:-2] : ['Red', 'Blue']
```

Example : Fetch last 2 elements

```
colors = ["Red","Blue","Green","Yellow"]


last2_pos = colors[2:4]   # Positive Index
last2_neg = colors[-2:] # Negative Index

print(f'Original List : {colors}')
print(f'colors[2:4]   : {last2_pos}')
print(f'colors[-2:]   : {last2_neg}')
```

Output

```
Original List : ['Red', 'Blue', 'Green', 'Yellow']
colors[2:4]   : ['Green', 'Yellow']
colors[-2:]   : ['Green', 'Yellow']
```

When you want to fetch elements till the end of the list then you can put the end element as empty . Python interpreter automatically stops for fetching element after last element.

Example : Fetch ['Blue', 'Green', 'Yellow']

```
colors = ["Red","Blue","Green","Yellow"]

# Fetch Blue, Green, and Yellow

three_pos = colors[1:] # using Positive Index
three_neg = colors[-3:] # Negative Index
print(f'Original List : {colors}')
print(f'colors[1:]    : {three_pos}')
print(f'colors[-3:]   : {three_neg}')
```

Output

```
Original List : ['Red', 'Blue', 'Green', 'Yellow']
colors[1:]    : ['Blue', 'Green', 'Yellow']
colors[-3:]   : ['Blue', 'Green', 'Yellow']
```

Example : Out of start and end values

```
colors = ["Red","Blue","Green","Yellow"]

#Examples
elements1 = colors[10:20]
elements2 = colors[2:10]
elements3 = colors[-10:2]

#Display the Result
print(f'colors[10:20] : {elements1}')
print(f'colors[2:10]  : {elements2}')
print(f'colors[-10:2] : {elements3}')
```

Output

```
colors[10:20] : []
colors[2:10]  : ['Green', 'Yellow']
colors[-10:2] : ['Red', 'Blue']
```

2.2.3 Extended Slicing

Extended slicing is one improved way to fetch multiple elements from a list or any iterable objects such as string, tuple, string etc. Extended slicing is similar to normal slicing but it takes a third parameter as difference or step to iterate that object after every step. Step parameters should be a positive number if you want to read the list from begining to end, and negative number if you want to read list from ending to beginning .

Syntax

```
list[start:end:step]
```

Where

start : Integer Number, included in the final result.

end : Integer Number, excluded from the final result.

steps : steps you want to iterate over an iterable object.

All parameters are optional according to the scenario. If any parameter is not given then all elements fetched from a list .

Example : Fetch all elements

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

result = lst[::]
print(result)
```

Output

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

Example : Fetch every 2nd element

```
lst = [1,2,3,4,5,6,7,8,9,10]
result = lst[::2]
print(result)
```

Output

```
[1, 3, 5, 7, 9]
```

Example : Fetch every 2nd element in first half list

```
lst = [1,2,3,4,5,6,7,8,9,10]
start = 0
end = len(lst)//2     # Find out Mid of List
# Fetch every 2nd element from list index 0 and 5
result = lst[start:end:2]
print(result)
```

Output

```
[1, 3, 5]
```

Example : Fetch every 3rd element

```
lst = [1,2,3,4,5,6,7,8,9,10]
result = lst[::3]
print(result)
```

Output

```
[1, 4, 7, 10]
```

Example : Fetch every 2nd element from end

```
lst = [1,2,3,4,5,6,7,8,9,10]
result = lst[::-2]
print(result)
```

Output

```
[10, 8, 6, 4, 2]
```

2.3 List Operations

- Insert
- Delete
- Update

2.3.1 Insert

Python lists support three different methods to insert new elements into a given input list.

2.3.1.1 append

append() method helps to append a single object to the end of the list. The element added using the *append()* method will always be positioned at **-1** or **(length-1)** position in the list. *append()* method is only used when we want to add an element at the last index.

Syntax

```
list.append(<obj>)
```

Where

obj : Any object such as string, list, tuple or any python object

Example : append 5 to list

```
lst = [1,2,3,4]
#Print Original 1st
print(f'Original List(lst)       : {lst}')

#Append 5 to 1st
lst.append(5)

print(f'List after lst.append(5) : {lst}')
```

Output

```
Original List(lst)       : [1, 2, 3, 4]
List after lst.append(5) : [1, 2, 3, 4, 5]
```

Example : Different Object append

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

#Append Number
lst.append(5)

#Append string
lst.append("Hello")

#Append list
lst.append([5,6])

#Append Tuple
lst.append((7,8,))


print(f"List after element append :\n{lst}")
```

Output

```
List after element append :
[1, 2, 3, 4, 5, 'Hello', [5, 6], (7, 8)]
```

2.3.1.2 insert

insert() method insert an element at the specified position where the *append()* method appends at the last index only. insert accepts two parameters, first one is position, and the second parameter is the input element which you want to add into the list. if the element is already present at a given position then that element is shifted right by one step.

Syntax

```
list.insert(pos,<obj>)
```

Where

Pos:

integer positive or negative numbers.

out of range positive numbers are given then the element is inserted at last index.
out of range negative number is given then the element is inserted at 0^{th} index.

obj :

Any python object,

Example : insert 5 at 2^{nd} index

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

# insert 5 at 2nd index
lst.insert(2,5)
print(lst)
```

Output

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

Example : insert object at -1 location

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

# insert 5 at -1 index
lst.insert(-1,5)
print(lst)
```

Output

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

In the above example 4 already exists on - 1 location . When we insert 5 at -1 index then 4 elements are shifted right side by one step.

Example : Out of range positive position

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

# insert 5 at 20th index
lst.insert(20,5)
print(lst)
```

Output

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

In the above example the length of the lst is 4 and we have inserted 5 elements at the 20th index which is not available means out of the index. so 5 elements added at the last position and can be accessed as follows.

```
lst[4]
#Or
lst[-1]
```

Example : Out of range Negative position

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

# insert 5 at -20th index
lst.insert(-20,5)
print(lst)
```

Output

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

Here first element 1 is accessed with the help of -4 but we are passing position is -20 which is out of range . so any position index after -4 are inserted at first position in the list.

5 can be accessed

```
lst[0]
#or
lst[-5]
```

2.3.1.3 extend

extend() method appends the elements from iterable to given list. *insert()* and *append()* method used to add a single element at a time .But *extend()* method helps to append multiple elements at a time.

Syntax

```
list.extend(<iterable>)
```

Where

iterable : any iterable object such as string, list, tuple, set

Example : Extend list

```
lst = [1,2,3,4]
print(f'Original List           : {lst}')

# Add 5,6 to lst
lst.extend([5,6])

print(f'After lst.extend([5,6]) : {lst}')
```

Output

```
Original List           : [1, 2, 3, 4]
After lst.extend([5,6]) : [1, 2, 3, 4, 5, 6]
```

Example : Extend String to list

```
lst = ['A']
print(f'Original List             : {lst}')

# Add [H,E,L,L,O] to lst
lst.extend("HELLO")

print(f'After lst.extend("HELLO") : {lst}')
```

Output

```
Original List             : ['A']
After lst.extend("HELLO") : ['A', 'H', 'E', 'L', 'L', 'O']
```

2.3.2 Delete

To remove element from an python list there are two methods

- pop
- remove

2.3.2.1 pop

pop() method removes an element at a given index and returns that element. If index is not provided the *pop()* method removes the last element from the list. *pop()* method gives an exception if the given index is not in the list.

Remove last element

```
list.pop()
```

Remove With Index

```
list.pop(<index>)
```

Where

index : Positive or negative integer position.

Example : Remove last element from list

```
lst = [1,2,3,4,5]
print(f'Original List      : {lst}')
# Remove last element from list
lst.pop()

print(f'List after lst.pop(): {lst}')
```

Output

```
Original List       : [1, 2, 3, 4, 5]
List after lst.pop(): [1, 2, 3, 4]
```

Example : remove 3rd index value

```
lst = ["Red","Blue","Green"]
print(f'Original List        : {lst}')

# Remove Blue
lst.pop(1)

print(f'List after lst.pop(1): {lst}')
```

Output

```
Original List       : ['Red', 'Blue', 'Green']
List after lst.pop(): ['Red', 'Green']
```

Example : Out of range input

```
lst = ["Red","Blue","Green"]
print(f'Original List     : {lst}')

# Remove 5 index value
lst.pop(5)
print(lst)
```

Output Exception

```
Original List     : ['Red', 'Blue', 'Green']
Traceback (most recent call last):
    lst.pop(5)
IndexError: pop index out of range
```

When we pass a position equal to length of the list or larger than length at that time *pop()* method raises an exception and returns that exception instead of removing an element from that given index.

2.3.2.2 remove

remove() method takes input as an element and removes that element from the list. It raises an exception if that element does not exist in the list.

Syntax

```
list.remove(<element>)
```

Where

element : Any element

Example : Remove 4

```
lst = [1,2,3,4,5]
print(f'Original List: {lst}')
# Remove 4
lst.remove(4)

print(f'List after lst.remove(4): {lst}')
```

Output

```
Original List: [1, 2, 3, 4, 5]
List after lst.remove(4): [1, 2, 3, 5]
```

Example : remove Blue

```
lst = ["Red","Blue","Green"]
print(f'Original List      : {lst}')

# Remove Blue
lst.remove("Blue")

print(f'List after lst.remove("Blue"): {lst}')
```

Output

```
Original List         : ['Red', 'Blue', 'Green']
List after lst.remove("Blue"): ['Red', 'Green']
```

Example : If Element is Not Exist

```
lst = ["Red","Blue","Green"]
print(f'Original List     : {lst}')

# Remove CYAN
lst.remove("CYAN")
```

Output Exception

```
Original List     : ['Red', 'Blue', 'Green']
Traceback (most recent call last):
    lst.remove("CYAN")
ValueError: list.remove(x): x not in list
```

2.3.3 Update

Python list is a mutable object so after the creation of the list object, we can modify the list element as we require. To update a list element we need an index of that object which we want to update and assign a new value with the help of the index.

Syntax

Single Element Update

```
list[<index>] = <new_value>
```

Single Element on Nested List

```
list[<rowindex>][<columnindex>] = <new_value>
```

Multiple Element Update

```
list[startindex:endindex] = <new_value>
```

Example : Update Blue with Green

```
lst = ["Red","Blue","Cyan","Pink"]
print(f'Original List     : {lst}')

# Update Blue with Green
# Blue index is 1
lst[1] = "Green"

print(f'List After Modification   : {lst}')
```

Output

```
Original List     : ['Red', 'Blue', 'Cyan', 'Pink']
List After Modification   : ['Red', 'Green', 'Cyan', 'Pink']
```

Example : Update last 3 values with only one Red value

```
lst = ["Red","Blue","Cyan","Pink"]
print(f'Original List     : {lst}')

# Update last three index with Red
lst[1:] = ["Red"]

print(f'List After Modification   : {lst}')
```

Output

```
Original List     : ['Red', 'Blue', 'Cyan', 'Pink']
List After Modification   : ['Red', 'Red']
```

Example : Update Red:R ,Green:G,Blue:B

```
lst = ["Red","Green","Blue"]
print(f'Original List     : {lst}')

lst[:] ="RGB"
print(f'List After Modification   : {lst}')
```

Output

```
Original List     : ['Red', 'Green', 'Blue']
List After Modification   : ['R', 'G', 'B']
```

Example : Update Nested list

```
lst = ["Red","Green","Blue",[5,6]]
print(f'Original List     : {lst}')

# Update 6 with 7
lst[3][1] =7

print(f'List After Modification   : {lst}')
```

Output

```
Original List     : ['Red', 'Green', 'Blue', [5, 6]]
List After Modification   : ['Red', 'Green', 'Blue', [5, 7]]
```

Example : Exception When given index is larger than length of list

```
lst = ["Red","Green","Blue",[5,6]]
print(f'Original List     : {lst}')

lst[9] =7

print(f'List After Modification   : {lst}')
```

Output

```
Original List     : ['Red', 'Green', 'Blue', [5, 6]]
Traceback (most recent call last):
    lst[9] =7
IndexError: list assignment index out of range
```

2.4 Methods

List Have some default method for manipulation with list elements .

Example : Fetch all List Methods

```
list_methods = [method for method in dir(list) if not
(method.startswith("_"))]
print(list_methods)
```

Output

```
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

Categories of List Methods

Insertion Methods

- append
- insert
- extend

Deletion Methods

- pop
- remove

Other Methods

- clear
- copy
- count
- index
- reverse
- sort

We have already discussed the Deletion and insertion methods. We are going to learn the remaining other list method with syntax and their different examples.

2.4.1 clear

clear() is a list built-in method used to remove all the elements from the input list.

Syntax

```
list.clear()
```

Example

```
lst = [1,2,3]
print(f'List Elements          :  {lst}')

# Remove all elements from list
lst.clear()
print(f'List after lst.clear() :  {lst}')
```

Output

```
List Elements          :  [1, 2, 3]
List after lst.clear() :  []
```

2.4.2 count

count() methods returns the number of elements(frequency count) with specified value. Any type of object can be used as a parameter to count method. *count()* method returns an integer number that shows frequency of given elements if element does not exist in list it will return 0.

Syntax

```
list.count(<Search_elemment>)
```

Where

Search_elemment : Any Python object such as tuple, list string etc..

Example : Count Number in list

```
lst = [1,2,3,2,3,1,2,3,5,1,7,7,2,1,2]

# Find count of 1
result = lst.count(1)
print(f' 1 found in lst {result} times')
```

Output

```
1 found in lst 4 times
```

Example : String Count

```
lst = ["Red","Blue","Green","Red"]

# Find count of Red
result = lst.count("Red")
print(f'{result} times found Red  in lst')
```

Output

```
2 times found Red  in lst
```

Example : Count of List in Nested List

```
lst = [1,2,3,2,3,[1,2],3,5,[1,2],7,7,2,[1,2]]


# Find count of [1,2]
result = lst.count([1,2])
print(f'{result} times found [1,2]  in lst')
```

Output

```
3 times found [1,2]  in lst
```

2.4.3 index

index() method returns an first(lowest) index of searched element otherwise return Exception named ValueError if value is not present in the list.

Syntax

```
list.index(<element>)
```

where

element : Any element whose index you want to find.

Example

```
colors = ["Red","Green","Blue"]

# Find index of Green
idx = colors.index("Green")# 1
print(f"Green found at {idx} index")
```

Output

```
Green found at 1 index
```

Example : element does not exists in list

```
colors = ["Red","Green","Blue"]

# Find index of "Black"
idx = colors.index("Black")
print(f"Black found at {idx} index")
```

Output Error

```
 idx = colors.index("Black")
ValueError: 'Black' is not in list
```

2.4.4 reverse

reverse() method returns an updated copy of the reversed list of input list.

Syntax

```
list.reverse()
```

Example

```
lst = [1,3,6,7,8]
print(f"Main List   : {lst}")

# Reverse the lst
lst.reverse()
print(f"List after lst.reverse()   : {lst}")
```

Output

```
Main List   : [1, 3, 6, 7, 8]
List after lst.reverse()   : [8, 7, 6, 3, 1]
```

2.4.4.1 reversed

reverse() method applies reverse on the main list and modify the indexes but when we need a temporary reverse of list use **reversed()** python built in function and return an object .

Syntax

```
reversed(<iterator object>)
```

Example

```
lst = [1,3,6,7,8]
print(f"Main List   : {lst}")

# Reverse the lst
final = reversed(lst)
print(f"Final  List reversed(lst)   : {list(final)}")
print(f"Main List After reversed(lst)   : {lst}")
```

Output

```
Main List   : [1, 3, 6, 7, 8]
Final  List reversed(lst)   : [8, 7, 6, 3, 1]
Main List After reversed(lst)   : [1, 3, 6, 7, 8]
```

2.4.5 sort

sort() method will return an ascending order sorted list . when reverse parameter is given as True return descending order list.

Syntax

```
list.sort(key=None, reverse=False)
```

Where

reverse :

True : Sort list in descending order

False : Default, Sort list in ascending order

key:

The key is any function that applies before sorting.
Default key is none
key takes a function and transform every element before applying sort

Example : Ascending Sort

```
lst = [1,4,2,8,7,6]
print(f'List Before Sort      : {lst}')

# Apply Sort method on list
lst.sort()  # or lst.sort(reverse=False)

print(f'List After lst.sort() : {lst}')
```

Output

```
List Before Sort      : [1, 4, 2, 8, 7, 6]
List After lst.sort() : [1, 2, 4, 6, 7, 8]
```

Example : Descending Sort

```
lst = [1,4,2,8,7,6]
print(f'List Before Sort      : {lst}')

# Apply Sort method on the list
lst.sort(reverse=True)

print(f'After lst.sort(reverse=True) : {lst}')
```

Output

```
List Before Sort      : [1, 4, 2, 8, 7, 6]
After lst.sort(reverse=True) : [8, 7, 6, 4, 2, 1]
```

Example : Sort Ascending list Using length of elements

```
lst = ["Abc","Python","insolo","Test"]
print(f'List Before Sort      : {lst}')
v=6

# Apply With len Function
lst.sort(key=lambda a:len(a))


print(f'After lst.sort(key=lambda a:len(a)) : {lst}')
```

Output

```
List Before Sort      : ['Abc', 'Python', 'insolo', 'Test']
After lst.sort(key=lambda a:len(a)) : ['Abc', 'Test', 'Python',
'insolo']
```

You can Also Pass length Function as lst.sort(key=len) . Similarly, the *reverse()* method *sort()* method also returns an updated list of original list so we can not get original list elements after applying the *sort()* method. To overcome this problem we have a python built in Function Called sorted as solution.

2.4.5.1 sorted

sorted() returns a list which consists of Ascending sorted elements.

Syntax

```
sorted(<iterable>,/,*,key=None,reverse=False)
```

Where

reverse :

True : Returns Descending order list.
False : Return Ascending Sorted list.

key

A customized function
The default value is None

Example

```
lst = [1,4,2,8,7,6]
print(f'List Before Sort: {lst}')


sorted_list = sorted(lst)

print(f'Sorted list   : {sorted_list}')
print(f'Main list     : {lst}')
```

Output

```
List Before Sort: [1, 4, 2, 8, 7, 6]
Sorted list   : [1, 2, 4, 6, 7, 8]
Main list     : [1, 4, 2, 8, 7, 6]
```

sorted() function doesn't affect the original array. It will create a copy of the original array and apply sorted and return that sorted list.

2.5 List Comprehension

List comprehension is generally more compact than a normal function and loop for recreating a list. List comprehension is used for creating new lists from any other iterable object. List comprehension also helps us to reduce line of code. A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal.

Syntax

```
[<item> for <item> in <listIterable>]
```

Example : Create a list which contains cube of 1 to 10

Normal for loop

```
result = []
for i in range(1,11):
  val = i**3
  result.append(val)
print(result)
```

Using list comprehension

```
result = [i**3 for i in range(1,11)]
print(result)
```

Above two programs gives following output

```
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

List comprehension with conditional statement if

Syntax

```
[<item> for <item> in <listIterable> <if_statement>]
```

Example : List of Even Numbers from 1 to 20

Using Normal For loop

```
result = []
for i in range(1,21):
  if i%2==0:
    result.append(i)
print(result)
```

Using List Comprehension

```
result = [i for i in range(21) if i%2==0]
print(result)
```

Output of above two programs

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

List comprehension with conditional statement if and else statement

```
[<if_block> if<condition> else <else_block> for <item> in <iterable_obj>]
```

Where

If_block : Added to the final list if condition successed.
Item : Element from iterable object.
Else_block :Added to final List if condition fails.
Iterable_obj : Any iterable object such as list.

Example : Create a list of odd and even string from 1 to 10

Using Normal For Loop

```
result = []
for i in range(1,11):
  if i%2 == 0:
    result.append("Even")
  else:
    result.append("Odd")

print(result)
```

Using List Comprehension

```
result = ["Even" if i%2==0 else "Odd" for i in range(1,10)]
print(result)
```

Output

```
['Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']
```

2.6 Lambda Function On List

Lambda function used to create a small anonymous function without a name. Lambda is a powerful python function with some limited capabilities. Lambda function is also referred to as an anonymous function. Lambda function can accept any number of elements as input but only one expression is evaluated and returned.

Lambda Function can be used with Following python built-in functions

- map
- filter
- Reduce

2.6.1 map

map() is a Python built-in function used to apply a specific function on every element of an iterable object such as list, tuple, strings, etc . *map()* returns a mapped object after applying the function to every element from iterable object. *map()* can be used with a normal function or lambda function.

Syntax

```
map(func, *iterables)
```

Where

func : A normal function or a Lambda function.

Iterbales : Any iterable Object such as list, string, tuple etc..

Examples

We are going to practice here multiple Examples using **map()** function with the help of normal user-defined function and a lambda function.

Example : Add 5 To every list Element (normal function)

```
# Function to add 5 to given input element
def Add(a):
  return a+5

# A Sample list
lst = [1,2,3,4]
# Apply Add Function to lst with map()
result = map(Add,lst)
print(f" Result of Mapped Function :{result}")
# Display original List
print(f" Original List : {lst}")
# Type cast map object to list
final_result = list(result)
print(f" After Applying Map Function(Add) : {final_result}")
```

Example : Add 5 To every list Element (lambda function)

```
# A Sample list
lst = [1,2,3,4]
# Apply Add Function to lst with map()
result = map(lambda a : a+5,lst)
print(f" Result of Mapped Function :{result}")
# Display original List
print(f" Original List : {lst}")
# Type cast map object to list
final_result = list(result)
print(f" After Applying Map Function(Add) : {final_result}")
```

Output of above Examples

```
Result of Mapped Function :<map object at 0x7ffaeb8714a8>
Original List : [1, 2, 3, 4]
After Applying Map Function(Add) : [6, 7, 8, 9]
```

Example : Convert a String Numbers List to int Numbers

```
# A Sample strings numbers list
lst = ['1','2','3','4']

# Convert it into int
result = map(int,lst)


# Display original List
print(f" Original List : {lst}")
# Type cast map object to list

final_result = list(result)
print(f" Int List(After Conversion) : {final_result}")
```

Output

```
Original List : ['1', '2', '3', '4']
Int List(After Conversion) : [1, 2, 3, 4]
```

Example : Addition of Two numbers List

```
# Define Two Lists
lst1 = [1,2,3,4]
lst2 = [5,6,7,8]

# Use of Multiple Parameter as input
Addition = map(lambda a,b:a+b,lst1,lst2)


print(f" List 1 : {lst1} \n List 2 : {lst2}")

# Convert mapped object to list
final_result = list(Addition)
print(f" Addition of List1 and List2 : {final_result}")
```

Output

```
List 1 : [1, 2, 3, 4]
List 2 : [5, 6, 7, 8]
Addition of List1 and List2 : [6, 8, 10, 12]
```

2.6.2 filter

filter() is a python built-in method used to filter out records from an iterable object based on condition evaluated from a Function . *filter()* function is applicable to every element of an iterable object. *filter()* methods added the element to the final list only if applied function returns *True* Otherwise that element is excluded .

Syntax

```
filter(function or None, iterable)
```

Where

function	:	Any user defined normal function or lambda function.
iterable	:	any iterable Object such as list, tuple, strings etc…

Examples : Filter Odd Numbers

```
# Define a  List
lst1 = [1,2,3,4,5,6,7,8]

# Filter out odd Numbers
even_numbers = filter(lambda a:a%2,lst1)

print(f" List 1 : {lst1}")
# Convert mapped object to list
final_result = list(even_numbers)
print(f"Filtered List(Odd) : {final_result}")
```

Output

```
List 1 : [1, 2, 3, 4, 5, 6, 7, 8]
Filtered List(Odd) : [1, 3, 5, 7]
```

Example : Filter Uppercase letters

```
# Define a  string
text = "PyThOn"

# Filter out Capital Letters
even_numbers = filter(lambda a:a.isupper(),text)

print(f" text : {text}")
# Convert mapped object to list
final_result = list(even_numbers)
print(f"Uppercase Letters : {final_result}")
```

Output

```
text : PyThOn
Uppercase Letters : ['P', 'T', 'O']
```

Example : Filter Even numbers from given input list

```
# Define a  List
lst1 = [1,2,3,4,5,6,7,8]

# Filter out Even Numbers
even_numbers = filter(lambda a:a%2==0,lst1)
print(even_numbers)
```

Output

```
[2, 4, 6, 8]
```

2.6.3 reduce

reduce() function is not a part of python built-in function you need to import it from **functools**. *reduce()* is a function used to reduce an iterable object to a single value by combining elements via a supplied function.

Syntax

```
from functools import reduce
reduce(function, sequence[, initial])
```

Where

function : Any lambda or normal user defined function.

Sequence : any sequence iterable list

Example

Consider

```
Lst1 = [1,2,3,4,5]
```

Output Should be

```
15
```

Reduce Calculate the Sequence if reduce(lambda a,b :a+b,Lst1)

Res = ((((1+2)+3)+4)+5)

Res= 15

Example

```
# Import reduce from functools
from functools import reduce

Lst1 = [1,2,3,4,5]

result = reduce(lambda a,b :a+b,Lst1)

print(f"Addition of All Elements in Lst1 is : {result}")
```

Output

```
Addition of All Elements in Lst1 is : 15
```

Chapter - 03

TUPLE

3. Tuple.

3.1 Introduction.

We have already learned string and list now we are going to focus on tuples. *A tuple* is similar to a list but the only difference is that a *tuple* is an immutable object where a *list* is a mutable object. Mutable means we can modify the object after its creation and immutable objects are not allowed to modify after its creation. *A tuple* is a collection of ordered and unchangeable (Immutable) items.

Syntax

1. Blank Tuple

```
variable_name = ()
--------OR---------
Variable_name = tuple()
```

2. Single elements

```
variable_name = (elem1,)
------OR------
Variable_name = elem1,
```

3. Multiple Elements

```
variable_name = (elem1,elem2,elem3...)
------OR---------
variable_name = elem1,elem2,elem3...
```

Example : Single Elements

```
tuple_var = 1, #Or (1,)
```

Example : Multiple Elements

```
tuple_var = (1,2,3,4,5)
type= type(tuple_var)

print(f'Tuple Example: {tuple_var}')
print(f'Type of tuple_var : {type}')
```

Output

```
Tuple Example: (1, 2, 3, 4, 5)
Type of tuple_var : <class 'tuple'>
```

Example : String Values

```
tuple_var = ("Hello","Python")
type= type(tuple_var)

print(f'Tuple Example: {tuple_var}')
print(f'Type of tuple_var : {type}')
```

Output

```
Tuple Example: ('Hello', 'Python')
Type of tuple_var : <class 'tuple'>
```

Example : Without round brackets

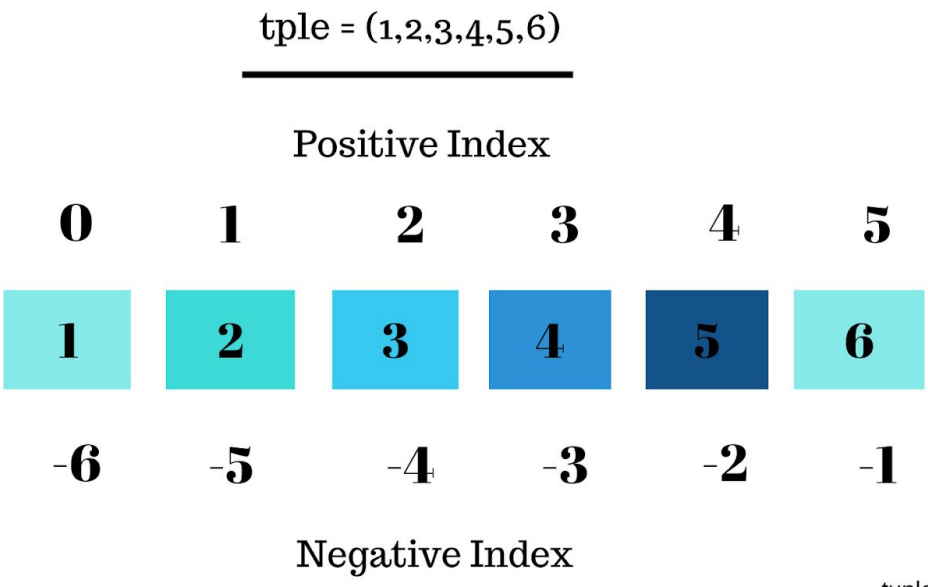
```
tuple_var = "Hello","Python"
tuple_var2 = 5,00,000.00
tuple_var3 = 100,
tuple_var4 = "Python",
```

Above all are the example of tuples,

3.2 Access Elements

3.2.1 Single Element

To Access single element from a *tuple*, we can use a positive or negative index after the tuple name. The first element is placed at **0** index and the last index is placed at **-1** index.



tuple Indexing

Syntax

```
tuple[index]
```

Where

Index : Positive or negative integer number. It will return an item from tuple on a given position. Raises an exception if given index is out of range value.

Example : Positive and Negative Index

```
colors = ("Pink","Yellow","Black","Green")
first = colors[0]
last = colors[-1]

print(f'Original Tuple: {colors}')
print(f'First Element : {first}')
print(f'Last Element  : {last}')
```

Output

```
Original Tuple: ('Pink', 'Yellow', 'Black', 'Green')
First Element : Pink
Last Element  : Green
```

Example : Out of Range Exception

```
colors = ("Pink","Yellow","Black","Green")
error = colors[10]
print(error)
```

Output

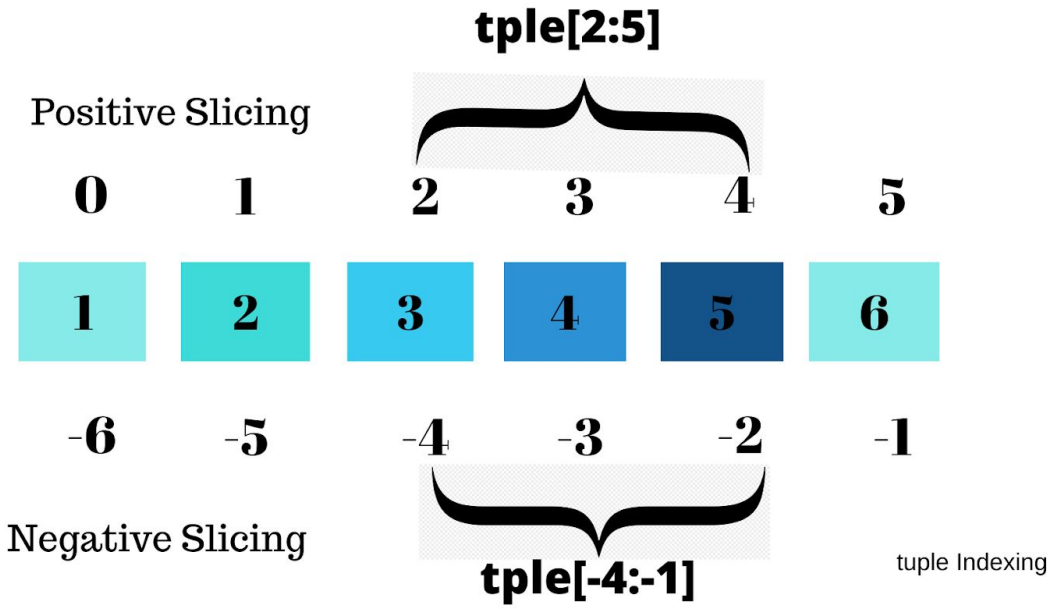
```
Traceback (most recent call last):
    error = colors[10]
IndexError: tuple index out of range
```

Some Other Examples

```
colors = ("Pink","Yellow","Black","Green")
colors[1] : Yellow
colors[2] : Black
colors[3] : Green
```

3.2.2 Slicing

Slicing is the way to Access multiple elements from iterables with the help of *start* and *end* range. It will return a blank tuple when the slicing range does not match.



Syntax

```
Tuple[start:end]
```

Where

start : integer positive or negative Number,Included in result.

end : integer positive or negative Number excluded from final result.

Example:

```
tple = (1,2,3,4,5,6)

# Fetch 3,4,5 Using Positive Index
pos = tple[2:5]
```

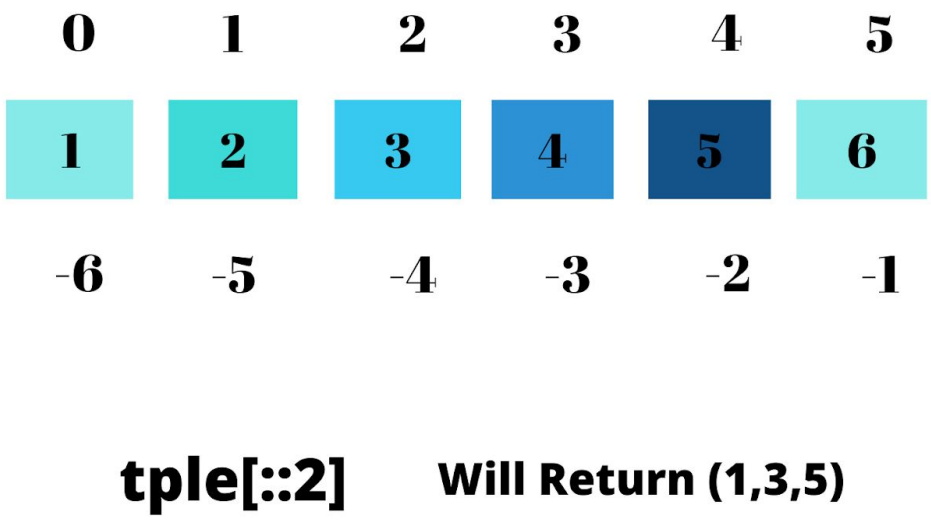
```
# Fetch 3,4,5 Using Negative Index
neg = tple[-4:-1]

print(f'Using tple[2:5]   :{pos}')
print(f'Using tple[-4:-1] :{neg}')
```

Output

```
Using tple[2:5]   :(3, 4, 5)
Using tple[-4:-1] :(3, 4, 5)
```

3.2.3 Extended Slicing

Extended slicing is similar to slicing but contains a third parameter for step or difference



Syntax

```
Tuple[start:end:step]
```

Where

start: integer positive and negative Number, included in the result, optional

end: integer positive negative Number excluded from result, optional

Example

```
tple = (1,2,3,4,5,6)

# Fetch Every Second Elements
sliced_data = tple[::2]

print(f'original tuple :{tple}')
print(f'Extended Slice Using tple[::2] : {sliced_data}')
```

Output

```
original tuple :(1, 2, 3, 4, 5, 6)
Extended Slice Using tple[::2] : (1, 3, 5)
```

Example : Fetch Every second Elements from 0th index to 4

```
tple = (1,2,3,4,5,6)

# Fetch Every second Elements from 0th index to 4
sliced_data = tple[0:4:2]

print(f'original tuple ":{tple}')
print(f'Extended Slice Using tple[0:4:2] : {sliced_data}')
```

Output

```
original tuple ":(1, 2, 3, 4, 5, 6)
Extended Slice Using tple[0:4:2] : (1, 3)
```

3.3 Tuple Unpacking

Python Support tuple assignment where right hand side values are assigned to left hand side variables.

3.3.1 Tuple Packing

Storing multiple values into one single tuple (placing values into one tuple) called as *tuple Packing.*

Syntax

```
<tuple_name> = (elem1,elem2,elem3)
```

Example

```
colors = ("Red","Green","Blue")
```

3.3.2 Tuple Unpacking

Extracting values from tuple to back into variables this process is called *tuple unpacking*.

Syntax

```
Elem1,elem2,... = tuple
```

Note : No of elements defined at left hand side should be equal to no of element in right hand side tuple .

Example

```
colors = ("Red","Green","Blue")
r,g,b=colors
print(r,g,b)
```

Output

```
Red Green Blue
```

Python support * as an optional parameter while doing tuple unpacking it will help to pack more elements into one single variable. Only one starred expression used in right right hand side

Consider following tuple

```
num = (1,2,3,4,5,6,7)
```

Now Create three different variable and assign values as follows

start = 1

end = 7

other = [2,3,4,5,6,7]

We can Use normal python indexing to get value and assign it

```
num = (1,2,3,4,5,6,7)

start = num[0]
end = num[-1]
other = num[1:len(num)-1]

print(f'original Data : {num}')
print(f'Start :{start} \nEnd :{end} \nOther:{other}')
```

Using Tuple Unpacking

```
num = (1,2,3,4,5,6,7)
start,*other,end=num

print(f'original Data : {num}')
print(f'Start :{start} \nEnd :{end} \nOther:{other}')
```

Output

```
original Data : (1, 2, 3, 4, 5, 6, 7)
Start :1
End :7
Other:(2, 3, 4, 5, 6)
```

In the above example *start* variable will be assigned first element from *num* and *end* variable will assign last element from num tuple. *other* variable will get assigned all remaining values from the *num* tuple .

```
num = (1,2,3,4,5,6,7)
start,*other,secondlast,end=num
print(start,other,secondlast,end)
```

Output

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

3.4 Operation on Tuple

3.4.1 Concatenate

To **concatenate** two tuples we can use the plus(+) mathematical operator .

Syntax

```
Tuple1 + Tuple2
```

Example

```
tuple1 = (1,2)
tuple2 = (3,4)

result_tuple = tuple1 + tuple2

print(f' Tuple1 :{tuple1}\n tuple2 :{tuple2}\n
tuple1+tuple2:{result_tuple}')
```

Output

```
Tuple1 :(1, 2)
tuple2 :(3, 4)
tuple1+tuple2:(1, 2, 3, 4)
```

3.4.2 Update Tuple Value

Tuple is an Immutable object so after tuple object creation it does not allow to modify that object. If tuples contain any mutable object then we can modify that part of tuple which is mutable.

Example

```
tuple1 = (1,2)
tuple1[1]=0
```

In the above example it will create an exception `'tuple' object does not support item assignment` because tuple is an immutable object.

Example : Updating tuple which contains Mutable object

```
tuple1 = (1,2,[3,4,5])
print(f' Original Tuple :{tuple1}')

tuple1[2][2]=9
print(f' Tuple After Updation :{tuple1}')
```

Output

```
Original Tuple :(1, 2, [3, 4, 5])
Tuple After Updation :(1, 2, [3, 4, 9])
```

In the above Example the third element of tuple is a list which is mutable so we can modify it even if it is in tuple.

Note: tuple object can not be modified after its creation but you can modify it using some tricks.

Steps

1. Convert tuple object to list.
2. Modify the element of the converted list.
3. Now again convert the list to tuple.

3.4.3 Delete Tuple

Tuple does not support item deletion but we can delete the tuple object itself.

Syntax

```
del tuple
```

Example

```
tuple1 = (1,2,[3,4,5])
del tuple1
```

3.5 Methods

- Index
- count

3.5.1 index

index() method returns a first lower index of specified value . It will raise an exception when specified value is not in the tuple.

Syntax

```
tuple.index(value,start,stop)
```

Where

value : search text.

tuple : tuple object.

start : integer optional starting position index.

end : integer optional ending position index.

Example

```
tuple1 = (1,2,3,4,5)

#search index of 3
result = tuple1.index(3)

print(f' index of 3 is {result}')
```

Output

```
index of 3 is 2
```

Example : Find index in specific range index

```
tuple1 = (1,2,3,4,5,5,3,4,5,6,7,7)

#search index of 3 in 4 to 8
result = tuple1.index(3,4,8)

print(f' index of 3 between 4 to 8 index  is {result}')
```

Output

```
 index of 3 between 4 to 8 index  is 6
```

3.5.2 count

count() method returns number of occurrences of a given element in tuple.

Syntax

```
tuple.count(element)
```

Example

```
tuple1 = (1,2,3,4,7,5,5,3,4,5,6,7,7)

# count of 7
count_7 = tuple1.count(7)

print(f'7 Appears {count_7} times in tuple')
```

Output

```
7 Appears 3 times in tuple
```

Chapter - 04

SET

4. Set

4.1 Introduction

Set is an abstract data type used to store unique elements, without maintaining any order and set is unindexed so the items will appear in random order. Set is an immutable python object and it cannot contain mutable objects. Set created using curly braces ({}) or using built in method set().

Syntax

Using curly braces

```
set_variable = {element1,element2...}
```

Using Built in Methods

```
set_variable = set(<iterator>)
```

Where

iterator : Any iterable object

Example

```
set_variable = {1,2,3,4,5,6,6}
print(set_variable)
```

Output

```
{1, 2, 3, 4, 5, 6}
```

In the above example set_variable contains **6** two times but in output contains only one 6. Because a set contains only unique elements.

Example Create empty Set

```
set_variable = {}
types = type(set_variable)
print(types)
```

Output

```
<class 'dict'>
```

Here {} create an empty dictionary to create an empty set object using python built in method **set().**

Example

```
set_variable = set()
Types = type(set_variable)
print(types)
```

Output

```
<class 'set'>
```

Example a String Iterator

```
set_variable = set("Hello")
print(set_variable)
```

Output

```
{'l', 'e', 'H', 'o'}
```

Example Set with mutable object

```
set_var = {1,2,3,[4,5]}
print(set_var)
```

Output

```
Traceback (most recent call last):
    set_var = {1,2,3,[4,5]}
TypeError: unhashable type: 'list'
```

4.2 Accessing Set Element

Set elements can not be accessed by referring to index because set items have no index .

Example

```
set_var = {1,2,3,4,5}
for i in set_var:
  print(i)
```

Output

```
1
2
3
4
5
```

Example Check element in set

```
set_var = {4,6,8,2,1}
result = 'Found' if 6 in set_var else 'Not-found'
print(result)
```

Output

```
Found
```

4.3 Set Basic Operations

4.3.1 Insertion methods

4.3.1.1 add

add() method is used to insert a single object element to a given set. *add()* method adds an element to a given set only if that element is not already in the set .

Syntax

```
set.add(element)
```

Where

element : Any element which you want to add in the set.

Example

```
set_var = {1,2}

print(f' Set : {set_var}')

#Add 3 to set_var
set_var.add(3)
set_var.add(3)

print(f' Set after adding 3  : {set_var}')
```

Output

```
Set : {1, 2}
Set after adding 3  : {1, 2, 3}
```

In the above example we have added 3 two times but the set stores only unique elements so the second 3 is ignored while adding in the set.

4.3.1.2 update

update() methods are similar to *add()* method only difference is that *update()* adds multiple elements at a single time where add adds only single element to set. Similar to add update also adds elements to the set if that element does not exist in the set.

Syntax

```
set.update(<iterator>)
```

Example

```
set_var = {1,2}

print(f' Set : {set_var}')

#Add 3,4,5 to set_var
set_var.update([3,4,5])

print(f' Set after adding 3,4,5  : {set_var}')
```

Output

```
Set : {1, 2}
Set after adding 3,4,5  : {1, 2, 3, 4, 5}
```

Example String

```
color={"red","green","blue"}
#Add "black" and "orange" to set
color.update(["black","orange"])
print(color)
```

Output

```
{'orange', 'red', 'green', 'blue', 'black'}
```

4.3.2 Deletion Methods

4.3.2.1 remove

remove() methods remove the element from set if the element is present in the set . If the element is not present in the list it will raise an exception. Set does not support indexing , *remove()* method accepts elements which we want to remove from the set.

Syntax

```
set.remove(<element>)
```

Example

```
color = {"red","green","blue"}
print(f' list of colors : {color}')

#remove "red"
color.remove("red")

print(f' list of colors after removing red : {color}')
```

Output

```
list of colors : {'red', 'green', 'blue'}
list of colors after removing red : {'green', 'blue'}
```

Example: To Create Key Error

```
color = {"red","green","blue"}
color.remove("yellow")
print(f' list of colors after removing yellow : {color}')
```

Output

```
color.remove("yellow")
KeyError: 'yellow'
```

4.3.2.2 discard

discard() method removes an element from set only if that element is present in the set. If an element is not present in the list nothing is returned even an exception .

Syntax

```
set.discard(<element>)
```

Example

```
color = {"red","green","blue"}
print(f' list of colors : {color}')

#remove "red"
color.discard("red")

print(f' list of colors after removing red : {color}')
```

Output

```
list of colors : {'blue', 'red', 'green'}
list of colors after removing red : {'blue', 'green'}
```

Example When Element does not exists

```
color = {"red","green","blue"}
color.discard("yellow")
print(f' list of colors after removing yellow : {color}')
```

Output

```
list of colors after removing yellow : {'green', 'blue', 'red'}
```

4.3.2.3 pop

pop() method removes any random element from the set and returns that element.

Syntax

```
set.pop()
```

Example

```
color = {"red","green","blue"}
print(f' list of colors: {color}')

color.pop()
print(f' list of colors after pop : {color}')
```

Output

```
 list of colors: {'green', 'red', 'blue'}
 list of colors after pop : {'red', 'blue'}
```

Output of the above example changes every time when you execute.

4.3.3 Update methods

Set do not have any methods to update the element of set

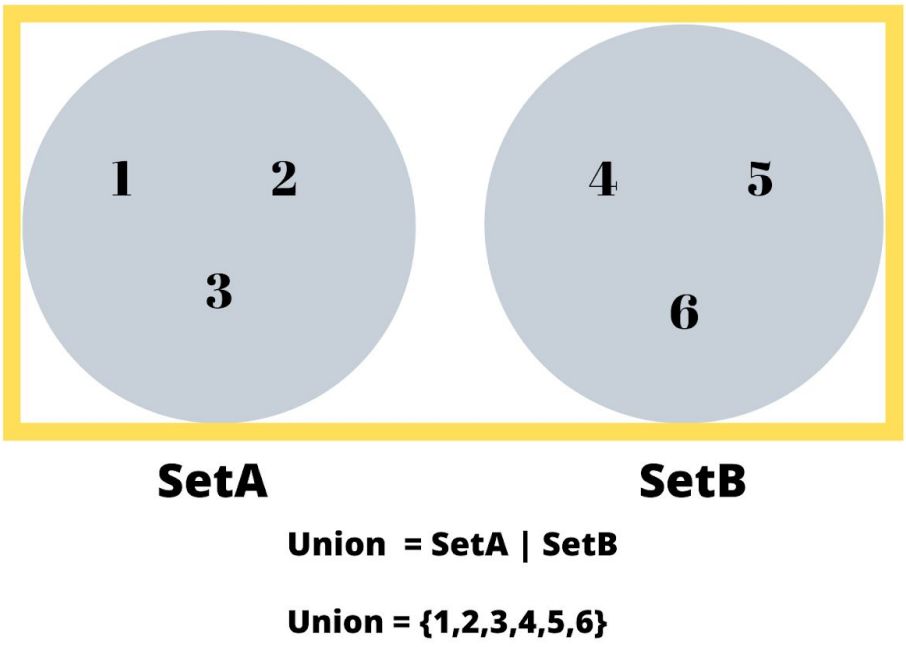
4.4 Set Core Theoretical Operations

Sets can be used to carry out mathematical set operations following a different operation on set can be performed

1. Union (|)
2. Intersection (&)
3. difference (-)
4. symmetric difference (^)

4.4.1 union

The *union* of a collection of sets is the set of all elements in the collection. The *union* of *seta* and *setb* is the set of all elements that are in *seta* and *setb*.



Image(4.1 Uninon)

Syntax

Using | Character

```
Seta | Setb
```

Using built-in function union()

```
seta.union(setb)
```

Example

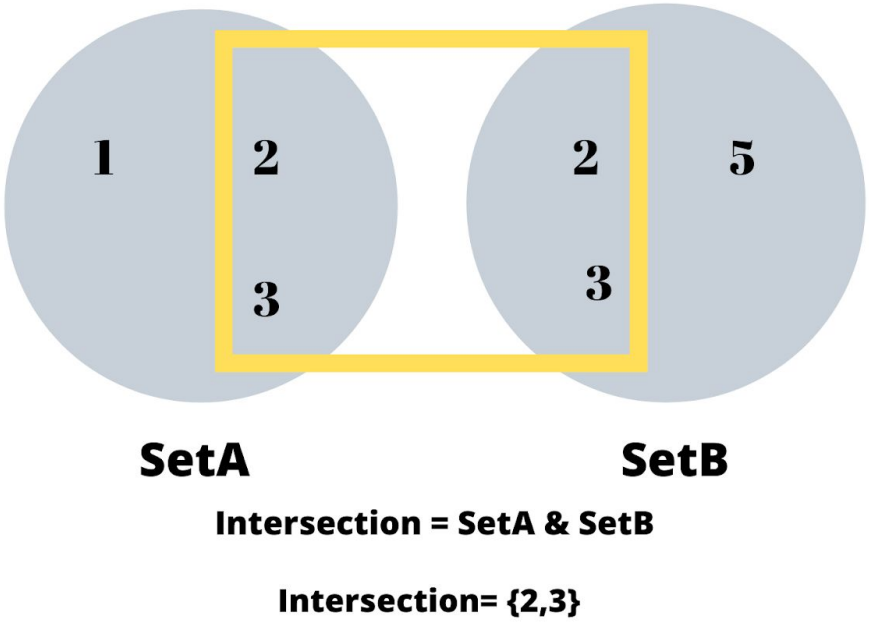
```
#Initialize Two Set
A={1,3,2}
B={6,5,4}
print(f'SetA:  {A} \nSetB:  {B}')

#using built in method .union()
result=A.union(B)
print(f'Union using Function :  {result}')

#Using | operator
result1=A | B
print(f'Union using       | :  {result}')
```

Output

```
SetA:  {1, 2, 3}
SetB:  {5, 4, 6}
Union using Function :  {1, 2, 3, 4, 5, 6}
Union using       |  :  {1, 2, 3, 4, 5, 6}
```

4.4.2 intersection

The new set is constructed with common elements from given set collections.



(Image 4.2 : Intersection)

Syntax

Using & Character

```
Seta & Setb
```

Using built-in function intersection()

```
seta.intersection(setb)
```

Example

```
#Initialize Two Set
A={1,2,3}
B={3,2,5}

print(f' SetA  : {A} \n SetB  : {B}')

#using built in method .intersection()
result=A.intersection(B)
print(f' intersection A.intersection(B) :{result}')


#Using & operator
result1=A & B
print(f' intersection A & B    :{result1}')
```

Output

```
 SetA  : {1, 2, 3}
 SetB  : {3, 2, 5}
 intersection A.intersection(B) :{2, 3}
 intersection A & B    :{2, 3}
```

4.4.3 difference

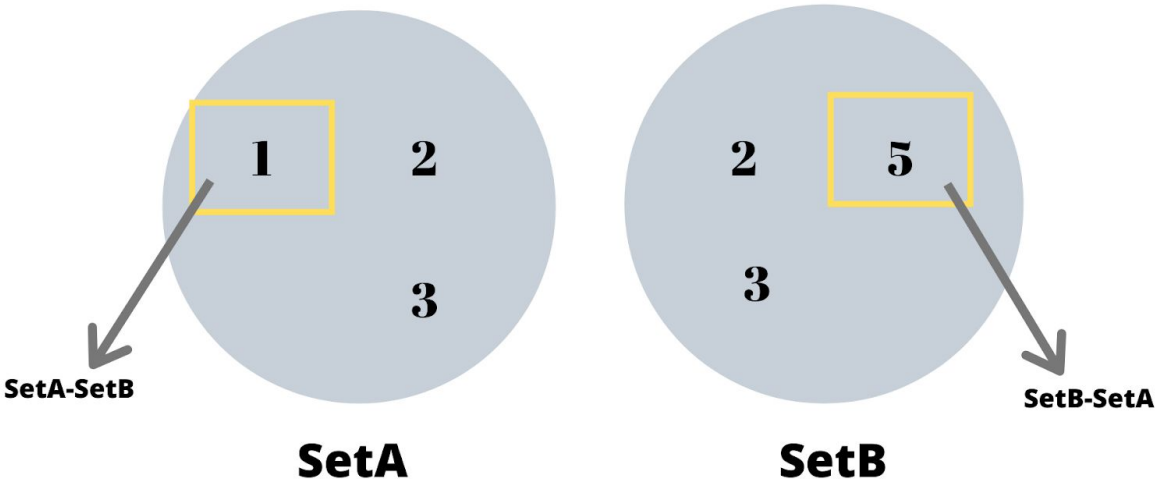
Subtraction of Given Set . Difference can be fetched using **'-'** operator or built in method difference.

Using - Operator

```
setA - setB
```

Using Built in Function

```
setA.difference(setB)
```



Set Difference

(Image 4.3 : Difference)

Consider SetA and SetB set collection.

SetA - SetB : The elements included in setA, but not included in setB

SetB - SetA : The elements included in setB, but not included in setA.

Example

```
A={1,2,3}
B={2,3,5}

print(f' A:{A} \n B:{B}')
#using built in method .difference()
result  = A.difference(B)
resultb = B.difference(A)

print(f' A.difference(B) :{result} \n B.difference(A) :{resultb}')

#Using - operator
result1  = A - B
result1b = B - A
print(f' A - B :{result} \n B - A:{resultb}')
```

Output

```
A:{1, 2, 3}
B:{2, 3, 5}
A.difference(B) :{1}
B.difference(A) :{5}
A - B :{1}
B - A:{5}
```

4.4.4 symmetric difference

The **symmetric_difference()** method returns a set that contains all items from both sets, but not the items that are present in both sets .

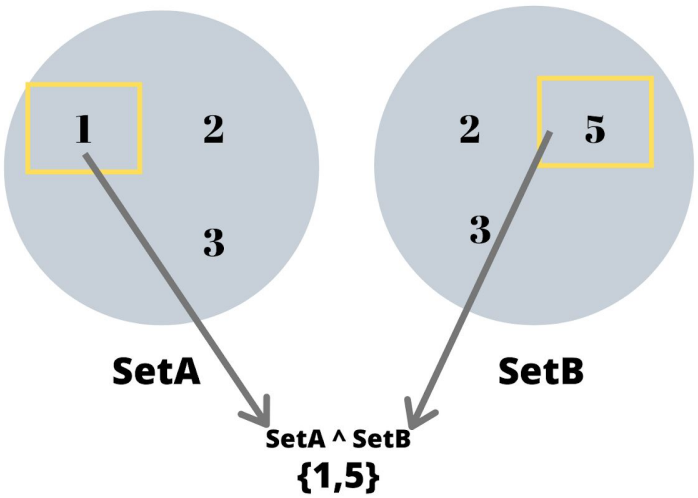
Syntax :

Using ^

```
SetA ^ SetB
```

Using Built-in Function

```
SetA.symmetric_difference(SetB)
```



Example

```
#Initialize Two Different Sets
A={1,2,3}
B={2,3,5}

#using built in method .symmetric_difference()
result=A.symmetric_difference(B)
print(f'symmetric_difference :  {result}')

#Using ^ operator
result1=A ^ B
print(f'A ^ B :  {result1}')
```

Output

```
symmetric_difference :  {1, 5}
A ^ B :  {1, 5}
```

4.5 Set Methods

4.5.1 Basic Methods

4.5.1.1 clear

clear() method removes all elements from the set.

Syntax

```
set.clear()
```

Example

```
set_var = {1,2}
set_var.clear()
print(set_var)
```

Output

```
set()
```

4.5.1.2 copy

copy() method returns a shallow copy of a given set .

Syntax

```
set.copy()
```

Example

```
a = {1,2}

# Create a copy
b = a.copy()

b.add(3)
print(f'a={a}\nb={b}')
```

Output

```
a={1, 2}
b={1, 2, 3}
```

4.5.2 Boolean Methods

4.5.2.1 isdisjoint

isdisjoint() returns *True* if two sets do not have any common element otherwise return *False*.

Syntax

```
seta.isdisjoint(setb)
```

Example

```
a = {1,2}
b = {3,4}

result = a.isdisjoint(b)
print(f'a & b are disjoint set :{result}')
```

Output

```
a & b are disjoint set :True
```

4.5.2.2 issubset

issubset() method returns *True* if another set contains this subset otherwise it will return *False*.

Syntax

```
seta.issubset(setb)
```

Example

```
a = {1,2}
b = {3,4,1,2}

result = a.issubset(b)
print(f'a is subset of b :{result}')
```

Output

```
a is subset of b :True
```

```
a = {1,2}
b = {3,4,1,2}
result = b.issubset(a)
print(f'b is subset of a :{result}')
```

Output

```
b is subset of a :False
```

4.5.2.3 issuperset

A set **A** is a **superset** of another set **B** if all elements of the set **B** are elements of set **A.**

Syntax

```
seta.issuperset(setb)
```

Example

```
a = {1,2}
b = {3,4,1,2}

result = a.issuperset(b)
print(f'a is superset of b :{result}')
```

Output

```
a is superset of b :False
```

Example

```
a = {1,2}
b = {3,4,1,2}

result = b.issuperset(a)
print(f'b is superset of a :{result}')
```

Output

```
b is superset of a :True
```

Chapter - 05

DICTIONARY

5. Dictionary

5.1 Introduction

Any language dictionary contains alphabetically sorted unique words with their meaning in the same manner python dictionary holds the values and keys where keys are unique and immutable, values have no constraints they can be mutable or duplicate.

Other data types hold and single elements where the dictionary holds **key: value** pair. A dictionary is a collection which is unordered, changeable.

Syntax

Empty dictionary using {}

```
dict_var = {}
```

Empty dictionary using keyword

```
dict_var = dict()
```

Dictionary with elements

```
dict_var = {'key':'value','key2':'value'....}
```

Example

```
colors = {'R':'Red','B':'Blue'}
print(colors)
```

Output

```
{'R': 'Red', 'B': 'Blue'}
```

Above the dictionary contains two elements, the first one key is **R** and the value of that key is **Red.** The second key is **B** and the value is **Blue.**

Example : with different elements

```
colors = {(1,2):'Red','B':[3,4,5,6]}
print(colors)
```

Output

```
{(1, 2): 'Red', 'B': [3, 4, 5, 6]}
```

Example : key as mutable object

```
colors = {[1,2]:'Red'}
print(colors)
```

Output

```
colors = {[1,2]:'Red'}
TypeError: unhashable type: 'list'
```

5.2 Access dictionary elements

5.2.1 Access Single elements

5.2.1.1 Using key name

To access elements from a dictionary we need a *key* name instead of index so the position of the *key* in the dictionary doesn't matter.

Syntax

```
dict[key]
```

Example

```
mydict = {"name":"jhon","age":20, "subject":["math","English"]}
name=mydict["name"]

print(f'Name :{name}')
```

Output

```
Name :jhon
```

5.2.1.2 Using get built-in function

get() is a python dictionary method that returns the value of a given key from a dictionary object and returns *None* if *key* does not exist in the dictionary.

Syntax

```
dict.get(key)
```

Example

```
mydict = {"name":"jhon","age":20, "subject":["math","English"]}
name=mydict.get("name")

print(f'Name :{name}')
```

Output

```
Name :jhon
```

5.2.2 Access Keys

To access all keys from a given dictionary use *keys()* method . *keys()* return a *dict_keys* class object .

Syntax

```
dict.keys()
```

Example

```
dict_var = {'name': 'jhon','age': 20,'subjects': ['math','english'] }

# Get all keys
keys = dict_var.keys()

print(f' Available keys {keys}')
```

Output

```
Available keys dict_keys(['name', 'age', 'subjects'])
```

5.2.3 Access values

To access all values from a given dictionary use *values()* method. *values()* return a *dict_values* class object .

Syntax

```
dict.values()
```

Example

```
dict_var = {'name': 'jhon','age': 20,'subjects': ['math','english'] }

# Get all keys
values = dict_var.values()

print(f' Available values {values}')
```

Output

```
Available values dict_values(['jhon', 20, ['math', 'english']])
```

5.2.4 iterate using dict.items()

keys() method used to get onlys key and *values()* method is used to get only values. *dict.items()* method is used to iterate through keys and values .

Syntax:

```
dict.items()
```

Example

```
dict_var = {'name': 'jhon','age': 20,'subjects': ['math','english'] }

for key,value in dict_var.items():
  print(key,value)
```

Output

```
name jhon
age 20
subjects ['math', 'english']
```

5.3 Dictionary Operations

5.3.1 Insert New value

5.3.1.1 Single Element

To add a single new element to a dictionary use the following syntax . if that *key* already exists then the existing value of that *key* is updated .

Syntax

```
Dict[key] = values
```

Example

```
dict_var = {'name': 'jhon','age': 20,'subjects': ['math','english'] }

# add new key year with value 2020
dict_var['year'] = 2020

print(dict_var)
```

Output

```
{'name': 'jhon', 'age': 20, 'subjects': ['math', 'english'], 'year':
2020}
```

5.3.1.2 Multiple Element

To insert multiple elements or another dictionary use *update()* method to perform operation .

Syntax

```
dict.update(another_dictionary)
```

Example

```
dict_var = {'name': 'jhon','age': 20,'subjects': ['math','english'] }

dict_2 ={'year': 2020, 'class' : 'A'}

# Update dict_var
dict_var.update(dict_2)

print(dict_var)
```

Output

```
{'name': 'jhon', 'age': 20, 'subjects': ['math', 'english'], 'year':
2020, 'class': 'A'}
```

5.3.2 Update Element

Insertion of element and updation of values both are the same while inserting a new element if you pass a such *key* which already exists in the dictionary then the value of that *key* is updated with a new value.

Example

```
dict_var = {'name': 'jhon','age': 20,'subjects': ['math','english'] }

# Update name value
dict_var['name'] = 'Roy'

print(dict_var)
```

Output

```
{'name': 'Roy', 'age': 20, 'subjects': ['math', 'english']}
```

5.3.3 delete elements

5.3.3.1 pop

pop() method removes the element from the dictionary and returns that element . if *key* does not exist then raise an *keyerror*.

Syntax

```
dict.pop(key)
```

Example

```
dict_var = {'name': 'jhon','age': 20}

# remove name key
dict_var.pop('name')

print(dict_var)
```

Output

```
{'age': 20}
```

5.3.3.2 popitem

popitem() removes an arbitrary element from the dictionary if the dictionary is empty and raises an error .

Syntax

```
dict.popitem()
```

Example

```
dict_var = {'name': 'jhon','age': 20}

# remove arbitrary item
dict_var.popitem()

print(dict_var)
```

Output

```
{'age': 20}
```

5.4 Dictionary Methods

5.4.1 clear

Remove all elements from the dictionary .

Syntax

```
dict.clear()
```

Example

```
dict_var = {'name': 'jhon','age': 20}

# remove all elements
dict_var.clear()

print(dict_var)
```

Output

```
{}
```

5.4.2 copy

copy() method creates a shallow copy of a given dictionary.

Syntax

```
dict.copy()
```

Example

```
dict_var = {'name': 'jhon','age': 20}

D1 = dict_var.copy()

print(D1)
```

Output

```
{'name': 'jhon', 'age': 20}
```

5.4.3 fromkeys

fromkeys() method returns a new dictionary where keys from iterable and values equal to value.

Syntax:

dict.fromkeys(iterable,value=None)

Where

Iterable : any iterable

Value : default value is None

Example

```
dict_var = {'name': 'jhon','age': 20}

new_dict = dict_var.fromkeys(dict_var)
print(new_dict)
```

Output

```
{'name': None, 'age': None}
```

Example with value

```
dict_var = {'name': 'jhon','age': 20}

new_dict = dict_var.fromkeys(dict_var,5)
print(new_dict)
```

Output

```
{'name': 5, 'age': 5}
```

5.4.4 setdefault

setdefault() method is similar to get method returns value of given *key* if *key* does not exist in the dictionary it will return a default value.

Syntax

```
dict.setdefault(key,'defaultvalue')
```

Example

```
mydict1 = {'name': 'jhon','year': 2020}
mydict2 = {'name': 'jhon'}

# Print dictionary
print(f'mydict1 :{mydict1} \nmydict2 :{mydict2}')

#set default year as 2020
year = mydict1.setdefault('year','2020')
year1 = mydict2.setdefault('year','2020')


# print year from both dictionary
print(f' year from mydict1 :{year}')
print(f' year from mydict2 :{year1}')
```

Output

```
mydict1 :{'name': 'jhon', 'year': 2020}
mydict2 :{'name': 'jhon'}
year from mydict1 :2020
year from mydict2 :2020
```

In above example *mydict1* contains year so **mydict1.setdefault('year','2020')** Return 2020 as output that key value . But in the second example mydict2 does ot contain year key and value **mydict2.setdefault('year','2020')** return default value 2020 because key does not exist in the dictionary.