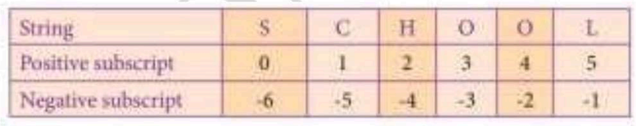
STRING

- A string is a group/ a sequence of characters.
- This is how we declare a string. We can use a pair of single or double quotes. Every string object is of the type _str‘.
- strings are immutable, which means we can‘t change an existing string. The bestwe can do is creating a new string that is a variation on the original:

ACCESSING CHARACTERS IN STRING

- In Python, individual characters of a String can be accessed by using the method of Indexing.
- Once you define a string, python allocate an index value for its each character.
- These index values are otherwise called as subscript which are used to access and manipulate the strings.
- The subscript can be positive or negative integer numbers.
- The positive subscript 0 is assigned to the first character and n-1 to the last character, where n is the number of characters in the string.
- The negative index assigned from the last character to the first character in reverse order begins with -1.



String	S	C	H	O	O	L
Positive subscript	0	1	2	3	4	5
Negative subscript	-6	-5	-4	-3	-2	-1

<u>Example 1</u> : Program to access each character with its positive subscript of a giving string

```
str1 = input ("Enter a string: ")
index=0
for i in str1:
print ("Subscript[",index,"] : ", i)
index + = 1
```

<u>Output</u>

```
Enter a string: welcome
Subscript [ 0 ] : w
Subscript [ 1 ] : e
Subscript [ 2 ] : l
Subscript [ 3 ] : c
Subscript [ 4 ] : o
Subscript [ 5 ] : m
Subscript [ 6 ] : e
```

Example 2 : Program to access each character with its negative subscript of a giving string

```
str1 = input ("Enter a string: ")
index=-1
while index >= -(len(str1)):
print ("Subscript[",index,"] : " + str1[index])
index += -1
```

Output

```
Enter a string: welcome
Subscript [ -1 ] : w
Subscript [ -2 ] : m
Subscript [ -3 ] : o
Subscript [ -4 ] : c
Subscript [ -5 ] : l
Subscript [ -6 ] : e
Subscript [ -7 ] : w
```

- While accessing an index out of the range will cause an IndexError.
- Only Integers are allowed to be passed as an index, float or other types that will cause a TypeError.

STR() FUNCTION

The str() function is used to convert the specified value into a string.

Syntax of str()

str(object, encoding='utf-8', errors='strict')

Here, encoding and errors parameters are only meant to be used when the object type is bytes or bytearray.

str() Parameters

The str() method takes three parameters:

- object - whose string representation is to be returned
- encoding - that the given byte object needs to be decoded to (can be UTF-8, ASCII, etc)
- errors - a response when decoding fails (can be strict, ignore, replace, etc)

str() Return Value

The str() method returns:
a printable string representation of a given object string representation of a given byte object in the provided encoding
Example 1: Python() String

```
# string representation of Luke
name = str('Lalit')
print(name)
```
output: Lalit

```
# string representation of an integer 40
age = str(40)
print(age)
```
Output: 40

```
# string representation of a numeric string 7ft
height = str('7ft')
print(height)
Run Code
```
Output: 7ft

In the above example, we have used the str() method with different types of arguments like string, integer, and numeric string.

OPERATIONS ON STRING

String concatenation

- String Concatenation is a very common operation in programming. String Concatenation is the technique of combining two strings.
- Python String Concatenation can be done using various ways.

- **String Concatenation using + Operator**
  - It's very easy to use the + operator for string concatenation.
  - This operator can be used to add multiple strings together.
  - The arguments must be a string. Here, The + Operator combines the string that is stored in the var1 and var2 and stores in another variable var3.

Note: Strings are immutable, therefore, whenever it is concatenated, it is assigned to a new variable.

```
# Defining strings
var1 = "Hello "
var2 = "World"

# + Operator is used to combine strings
var3 = var1 + var2
print(var3)
```

Output: Hello World

- **String Concatenation using join() Method**
  - The join() method is a string method and returns a string in which the elements of the sequence have been joined by str separator.
  - This method combines the string that is stored in the var1 and var2.
  - It accepts only the list as its argument and list size can be anything.

```
var1 = "Hello"
var2 = "World"

# join() method is used to combine the strings
print("".join([var1, var2]))

# join() method is used here to combine
# the string with a separator Space(" ")
var3 = " ".join([var1, var2])
print(var3)
```

Output

```
HelloWorld
Hello World
```

- **String Concatenation using % Operator**
  - We can use the % operator for string formatting, it can also be used for string concatenation.
  - It's useful when we want to concatenate strings and perform simple formatting.
  - The %s denotes string data type. The value in both the variable is passed to the string %s and becomes –Hello World‖.

```
var1 = "Hello"
var2 = "World"
# % Operator is used here to combine the string
print("% s % s" % (var1, var2))
```

Output

```
Hello World
```

- **String Concatenation using format() function**
  - str.format() is one of the string formatting methods in Python, which allows multiple substitutions and value formatting.
  - It concatenate elements within a string through positional formatting.
  - The curly braces {} are used to set the position of strings.
  - The first variable stores in the first curly braces and the second variable stores in the second curly braces. Finally, it prints the value –Hello World‖.

```
var1 = "Hello"
var2 = "World"

# format function is used here to
# combine the string
print("{} {}".format(var1, var2))

# store the result in another variable
var3 = "{} {}".format(var1, var2)
print(var3)
```

Output

```
Hello World
Hello World
```

- **String Concatenation using (, comma)**
  - –,‖ is a great alternative to string concatenation using –+‖.
  - when you want to include single whitespace. Use a comma when you want to combine data types with single whitespace in between.

```
var1 = "Hello"
var2 = "World"

# , to combine data types with a single whitespace.
print(var1, var2)
```

Output

```
Hello World
```

STRING SLICING

- A segment of a string is called a slice.
- Subsets of strings can be taken using the slice operator **([ ] and [:])** with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- **Syntax [Start: stop: steps]**

- Slicing will start from index and will go up to **stop** in **step** of steps.
- Default value of start is 0
- Stop is last index of list
- And for step default is 1

Example 1:

```
str = 'Hello World!'

print str    # Prints complete string

print str[0] # Prints first character of the string

print str[2:5] # Prints characters starting from 3rd to 5th

print str[2:] # Prints string starting from 3rd character print
```

Output:

```
Hello World!

H

llo
```

llo World!

Example 2:

- x='computer'

x[1:4]

output: 'omp'

- x[:5]

'compu'

- x[-1]

'r'

- x[-3:]

'ter'

- x[:-2]

'comput'

- x[::-2]

'rtpo'

TRAVERSING A STRING

- Traversing just means to process every character in a string, usually from left end to right end
- One way to write a traversal is with **a while loop**:

```
fruit = "fruit"
index = 0
while index < len(fruit):
letter = fruit[index]
  print(letter)
```

```
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

- Another way to write a traversal is with a for loop:

```
fruit = "peach"
for char in fruit:
 print(char)
```

Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.

ESCAPE SEQUENCE

- Escape characters or sequences are illegal characters for Python and never get printed as part of the output.
- When backslash is used in Python programming, it allows the program to escape the next characters.
- Escape sequences allow you to include special characters in strings. To do this, simply add a backslash (\) before the character you want to escape.

Following would be the syntax for an escape sequence

Syntax:

\Escape character

Escape character	**Function**	**Example Code**	**Result**
\n	The new line character helps the programmer to	txt = –Good\nmorning‖	Good morning

	insert a new line before or after a string.	print(txt)	
\\	This escape sequence allows the programmer to insert a backslash into the Python output.	txt = ―Good\\morning‖ print(txt)	Good\morning
\b	This escape sequence provides backspace to the Python string. It is inserted by adding a backslash followed by ―b‖. ―b‖ here represents backslash.	txt = ―Good \bmorning‖ print(txt)	Good morning
\‘	It helps you to add a single quotation to string	txt = ―Good\‘morning‖ print(txt)	Good‘morning
\t	Gives a tab space equivalent to 8 normal spaces	txt= ―Good\tmorning‖ print(txt)	Good morning
\‖	Prints a double quote inside a string enclosed with double-quotes.	txt=‖Good\‖morning‖ print(txt)	Good‖morning

RAW STRING

A raw string can be used by prefixing the string with r or R, which allows for backslashes to be included without the need to escape them.

Example:

```
print(r"Backslashes \ don't need to be escaped in raw strings.")
```

Output:

```
Backslashes \ don't need to be escaped in raw strings.
```

But keep in mind that unescaped backslashes at the end of a raw string will cause and error:

```
print(r"There's an unescaped backslash at the end of this string\")
```

Output:

```
 File "main.py", line 1
  print(r"There's an unescaped backslash at the end of this string\")
                                                                  ^
SyntaxError: EOL while scanning string literal
```

UNICODE STRING

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world.

Example

```
#!/usr/bin/python

print u'Hello, world!'
```

Output

When the above code is executed, it produces the following result −

```
Hello, world!
```

STRING METHODS

- split()

split() method splits the string according to delimiter str (space if not provided)

Syntax:
String.split()

Example:

```
string="Nikhil Is Learning"
string.split()

['Nikhil', 'Is', 'Learning']
```

- count()

count() method counts the occurrence of a string in another string

Syntax:
String.count()

Example:

```
string='Nikhil  Is Learning'
string.count('i')
        3
```

❖ find()

Find() method is used for finding the index of the first occurrence of a string in another string.

Syntax:

String.find(―string‖)

Example:

```
string="Nikhil  Is Learning"
string.find('k')
2
```

❖ swapcase()

converts lowercase letters in a string to uppercase and viceversa

Syntax:

String.find(―string‖)

Example:

```
string="HELLO"
string.swapcase()
'hello'
```

❖ startswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

Syntax:

String.startswith(―string‖)

Example:

```
string="Nikhil  Is Learning"
string.startswith('N')
True
```

❖ endswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) ends with substring str; returns true if so and false otherwise.

Syntax:

String.endswith(-string‖)

Example:

string="Nikhil Is Learning"

string.startswith('g')

True

❖ isalnum():

Isalnum() method returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

Syntax:

String.isalnum()

Example:

string="123alpha"
string.isalnum()
True

❖ isalpha():

isalpha() method returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

Syntax:

String.isalpha()

Example:

string="nikhil"
string.isalpha()
True

❖ isdigit():

isdigit() returns true if string contains only digits and false otherwise.

Syntax:

String.isdigit()

Example:

```
string="123456789"
string.isdigit()
```

True

- ❖ Lower()

Convert the string with all the upper case letter to lower case.

Syntax:

String.lower()

Example:

```
String="HELLO"
String.lower()
```

Output:

```
hello
```

- ❖ Upper()

Convert the string with all the lower case letter to upper case.

Syntax:

String.upper()

Example:

```
String="hello"
String.upper()
```

Output:

```
HELLO
```

- ❖ replace()

replace() method replaces all occurrences of old in string with new or at most max occurrences if max given.

Syntax:

String.replace(old str,new str)

Example:

```
string="Nikhil  Is Learning"
string.replace('Nikhil','Neha')'Neha  Is Learning'
```

LIST

- Lists are used to store multiple items in a single variable.
- List items are ordered, changeable, and allow duplicate values.
- To use a list, you must declare it first. Do this using square brackets and separatevalues with commas.
- List items are indexed, the first item has index [0], the second item has index [1] etc.
- A list need not be homogeneous; that is, the elements of a list do not all have to be of the same type.

Example:

```
list1=[1,2,3,'A','B',7,8,[10,11]]
print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

Properties of Lists

Python Lists are ordered and mutable (or changeable). Let‘s understand this further:

- **Ordered**: There is a defined order of elements within a list. New elements are added to the end of the list.
- **Mutable**: We can modify the list i.e add, remove and change elements of a python list.

Creating python list

- A Python list is created using the square brackets, within which, the elements of a list are defined.

- The elements of a list are separated by commas.

Creating a list

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

Avengers = ['hulk', 'iron-man', 'Captain', 'Thor']
```

Access Items

- List items are indexed and you can access them by referring to the index number:

Example:Second item of the list:

```
 List = ["apple", "banana", "cherry"]
print(List[0])    output:apple
print(List[1])    output:banana
print(List[2])     output:cherry
```

Negative Indexing

Negative indexing means start from the end.
-1 refers to the last item, -2 refers to the second last item etc.

Example:Print the last item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new list with the specified items.

Example

```
list = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(list[2:5])
cherry,orange,kiwi
```

OPERATIONS ON A LIST

1. Access elements of a list

We can access the elements of a python list using the indexes — [0], [1], etc. The index starts at 0, so the index of the _n‘th element of a list is _n-1‘.To access consecutive elements of a list, we can use colon ‗:‘ and set the range of indexes like this [1:5]. The last index within the range is not considered in the output, so [1:5] will give elements from indexes 1 to 4.We can use the negative index to access elements from the last. Index -1 will give the last element, index -2 will give 2nd last element, etc.

Example:

```
mix_list = [1, 'Jerry', 'Science', 79.5, True]
print(mix_list[0])
```
output: 1
```
print(mix_list[3])
```
output: 79.5

2. Adding (or Appending) element

Elements can be added (or appended) to the python lists using the append and insert function.

Using the insert function, we can add elements at any position of a list. The function takes 2 arguments: 1) Index at which we want to insert the element. 2) The element that we want to insert.

The append function is used to add elements to the end of a list.

Example:

```
sub_list = ['Python', 'Perl', 'Java', 'C++']
sub_list.insert(3,'R')
print(sub_list)
```
output: ['Python', 'Perl', 'Java', 'R', 'C++']

```
# Using append
sub_list.append('Django')
print(sub_list)
```
output: ['Python', 'Perl', 'Java', 'R', 'C++', 'Django']

3. Sorting Lists

We can sort the elements of the list in ascending or descending order. In the below code, we are sorting the subject list in ascending and descending order (using ―reverse = True‖).

Example:

```
sub_list = ['Python', 'Perl', 'Java', 'C++']

# ascending order
sub_list.sort()
print(sub_list)
```

```
['C++', 'Java', 'Perl', 'Python']

# descending order
sub_list.sort(reverse = True)
print(sub_list)
['Python', 'Perl', 'Java', 'C++']
```

4. Update elements of a list

Because lists are changeable, we can update the elements of the lists using the index of the elements.

For example, in the code below we are updating _Science‘ to _Computer‘ using the index of _Science‘.

Example:

```
mix_list = [1, 'Jerry', 'Science', 79.5, True]
mix_list[2] = 'Computer'
print(mix_list)
```

Output: [1, 'Jerry', 'Computer', 79.5, True]

5. Delete elements of a list

We can easily delete the element of a list using the very intuitive remove function.

Suppose, we want to delete _Java‘ from our subject list, then we can delete it using the remove function and provide _Java‘ as an argument.

Example:

```
sub_list = ['Python', 'Perl', 'Java', 'C++']

#remove Java
sub_list.remove('Java')
print(sub_list)
```

Output: ['Python', 'Perl', 'C++']

6. Pop the elements of a list

The pop function is used to remove or pop out the last element from the list and output the remaining elements.

Example:

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

```
num_list.pop()
print(num_list)
```

Output: [1, 2, 3]

7. Length/Number of elements in a list

We can find the total number of elements in a list using the length function.

Example

```
num_list = [1, 3, 4, 5]

#length of a list
print(len(num_list))
```

Output: 4

8. Maximum element within a list

We can easily find the maximum of the numbers which are present within a list using the very intuitive max function.

Max function only work on homogenous list i.e. lists having elements of the same data type.

Example

```
num_list = [1, 3, 4, 5]
max(num_list)
```

Output: 5

9. Concatenate lists

We can easily concatenate 2 lists using the ‗+' operator. (Concatenation is like appending 2 lists).

```
num_list1 = [1, 3, 4, 5]
num_list2 = [5, 6, 7]
print(num_list1 + num_list2)
```

Output: [1, 3, 4, 5, 5, 6, 7]

IMPLEMENTATION OF STACK AND QUEUE USING LIST

STACK IN PYTHON

- A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner.

- In stack, a new element is added at one end and an element is removed from that end only.
- The insert and delete operations are often called push and pop.

The functions associated with stack are:

- empty() – Returns whether the stack is empty
- size() – Returns the size of the stack
- peek() – Returns a reference to the topmost element of the stack
- push(a) – Inserts the element ‗a' at the top of the stack
- pop() – Deletes the topmost element of the stack

Implementation:

- There are various ways from which a stack can be implemented in Python.
- Python‗s built-in data structure list can be used as a stack.
- Instead of push(), append() is used to add elements to the top of the stack while pop() removes the element in LIFO order.

Python program to demonstrate stack implementation

```
stack = []

stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)
```

Output

```
Initial stack
['a', 'b', 'c']

Elements popped from stack:
c
b
a

Stack after elements are popped:
[]
```

QUEUE IN PYTHON

- queue is a linear data structure that stores items in First In First Out (FIFO) manner.
- With a queue the least recently added item is removed first.
- A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

Operations associated with queue are:

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

Implementation

- List is a Python's built-in data structure that can be used as a queue.
- Instead of enqueue() and dequeue(), append() and pop() function is used.
- Lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring O(n) time.

Python program to demonstrate queue implementation using list

```
queue = []

queue.append('a')
queue.append('b')
queue.append('c')

print("Initial queue")
print(queue)

print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))

print("\nQueue after removing elements")
print(queue)
```

Output:

```
Initial queue
['a', 'b', 'c']

Elements dequeued from queue
a
b
c

Queue after removing elements
[]
```

NESTED LIST

- A list within another list is referred to as a nested list in Python.
- We can also say that a list that has other lists as its elements is a nested list.
- When we want to keep several sets of connected data in a single list, this can be helpful.

Here is an illustration of a Python nested list:

MyList = [[22, 14, 16], ["Joe", "Sam", "Abel"], [True, False, True]]

- To access the elements in this nested list we use indexing.
- To access an element in one of the sublists, we use two indices – the index of the sublist and the index of the element within the sublist.

<u>Example of the nested list:</u>

```
MyList = [[22, 14, 16], ["Joe", "Sam", "Abel"], [True, False, True]]
print(MyList[0])
[22, 14, 16]
print(MyList[0][1])
output: 14
MyList[0][1] = 20
print(MyList)
 output:[[22, 20, 16], ["Joe", "Sam", "Abel"], [True, False, True]]
```

Add items to a Nested list

- To add new values to the end of the nested list, use append() method.

```
L = ['a', ['bb', 'cc'], 'd']
L[1].append('xx')
print(L)
Output: ['a', ['bb', 'cc', 'xx'], 'd']
```

- When you want to insert an item at a specific position in a nested list, use insert() method.

```
L = ['a', ['bb', 'cc'], 'd']
L[1].insert(0,'xx')
print(L)
Output: ['a', ['xx', 'bb', 'cc'], 'd']
```

- You can merge one list into another by using extend() method.

```
L = ['a', ['bb', 'cc'], 'd']
L[1].extend([1,2,3])
print(L)
Output:  ['a', ['bb', 'cc', 1, 2, 3], 'd']
```

Remove items from a Nested List

If you know the index of the item you want, you can use pop() method. It modifies the list and returns the removed item.

```
L = ['a', ['bb', 'cc', 'dd'], 'e']
x = L[1].pop(1)
print(L)
Output: ['a', ['bb', 'dd'], 'e']
# removed item
print(x)
Output: cc
```

- If you don‘t need the removed value, use the del statement.

```
L = ['a', ['bb', 'cc', 'dd'], 'e']
del L[1][1]
print(L)
Output: ['a', ['bb', 'dd'], 'e']
```

- If you‘re not sure where the item is in the list, use remove() method to delete it by value.

```
L = ['a', ['bb', 'cc', 'dd'], 'e']
L[1].remove('cc')
print(L)
Output: ['a', ['bb', 'dd'], 'e']
```

Find Nested List Length
You can use the built-in len() function to find how many items a nested sublist has.

```
L = ['a', ['bb', 'cc'], 'd']
print(len(L))
Output: 3
print(len(L[1]))
Output : 2
```

DICTIONARY

- A dictionary is a kind of data structure that stores items in key-value pairs.
- A key is a unique identifier for an item, and a value is the data associated with that key.
- Dictionaries often store information.
- Dictionaries are mutable in Python, which means they can be changed after they are created.
- They are also unordered, indicating the items in a dictionary are not stored in any particular order.

CREATING A DICTIONARY

- Dictionaries are created using curly braces {}.
- The key is on the left side of the colon (:) and the value is on the right.
- A comma separates each key-value pair.
- will use the built-in dictionary data type to create a Python dictionary.
- This type stores all kinds of data, from integers to strings to lists.
- The dictionary data type is similar to a list but uses keys instead of indexes to look up values.
- You use the dict() function in Python to create a dictionary.
- This function takes two arguments:
  - The first argument is a list of keys.
  - The second argument is a list of values.

Example : createing a dictionary using the dict() function

- **Empty dictionary**

```
my_dict = {}
```

- **Dictionary with integer keys**

```
my_dict = {1: 'apple', 2: 'ball'}
```

- **Dictionary with mixed keys**

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

ACCESSING ELEMENTS OF A DICTIONARY

- In Python, dictionaries are accessed by key, not by index.
- This means you can't access a dictionary element by using its position in the dictionary. Instead, you must use the dictionary key.
- There are two ways to access a dictionary element in Python.
  - The first is by using the get() method. This method takes two arguments: the dictionary key and a default value. If the key is in the dictionary, the get() method will return the value associated with that key. The get() method will return the default value if the key is not in the dictionary.
  - The second way to access a dictionary element is using the [] operator. This operator takes the dictionary key as an argument and returns the value associated with the key value. If the key value is not in the dictionary, the [] operator will raise a KeyError.

GET VS [] FOR RETRIEVING ELEMENTS

```
my_dict = {'name': 'Jack', 'age': 26}
print(my_dict['name'])
print(my_dict.get('age'))
print(my_dict.get('address'))
Output: Jack
Output: 26
Output: KeyError
```

OPERATIONS ON THE DICTIONARY

- A dictionary is mutable, we can add new values, and delete and update old values.

- **Accessing the values of dictionary**
  - The dictionary's values are accessed by using key.
  - Consider a dictionary of networking ports.
  - In order to access the dictionary's values, the key is considered.

```
Port = {80: ―HTTP‖, 23: ―Telnet‖, 443: ―HTTPS‖}
```

```
print(port[80])
Output: 'HTTP'
print(port[443])
Output: 'HTTPS'
```

If the key is not found, then the interpreter shows the preceding error.

- **Deleting an item from the dictionary**
  - del keyword is used to delete the entire dictionary or the dictionary's items.

Syntax :del dict[key]

Considering the following code snippet for example:

```
port = {80: "HTTP", 23 : "Telnet", 443 : "HTTPS"}
del port[23]
print(port)
Output: {80: 'HTTP', 443: 'HTTPS'}
```

Syntax to delete the entire dictionary:

del dict_name

Consider the following example:

```
port = {80: "HTTP", 23 : "Telnet", 443 : "HTTPS"}
del port
print(port)
Traceback (most recent call last):
File "<pyshell#12>", line 1, in <module> port
NameError: name 'port' is not defined
```

The preceding error shows that the port dictionary has been deleted.

- **Updating the values of the dictionary**
  - To update the dictionary, just specify the key in the square bracket along with the dictionary name and assigning new value.

Syntax:

dict[key] = new_value

Example:

port = {80: "HTTP", 23 : "SMTP‖, 443 : "HTTPS"}

In the preceding dictionary, the value of port 23 is "SMTP", but in reality, port number 23 is for telnet protocol.

Let's update the preceding dictionary with the following code:

port = {80: "HTTP", 23 : "SMTP", 443 : "HTTPS"}

print(port)

Output: {80: 'HTTP', 443: 'HTTPS', 23: 'SMTP'}

port[23] = "Telnet"

print(port)

Output:{80: 'HTTP', 443: 'HTTPS', 23: 'Telnet'}

- **Adding an item to the dictionary**

Item can be added to the dictionary just by specifying a new key in the square brackets along with the dictionary.

syntax

dict[new_key] = value

Example:

port = {80: "HTTP", 23 : "Telnet"}

port[110]="POP"

print(port)

Output:{80: 'HTTP', 110: 'POP', 23: 'Telnet'}

<u>Other dictionary functions:</u>

Similar to lists and tuples, built-in functions available for dictionary.

- **len()-** to find the number of items that are present in a dictionary.

Example:

port = {80: "http", 443: "https", 23:"telnet"}

print(len(port))

Output: 3

- **max()-**It returns the key with the maximum worth.

Example:

dict1 = {1:"abc",5:"hj", 43:"Dhoni", ("a","b"):"game", "hj":56}

max(dict1)

Output:('a', 'b')

- **min()-** It returns the dictionary's key with the lowest worth.

Example:

dict1={1: 'abc', (1, 3): 'kl', 5: 'hj', 43: 'Dhoni', 'hj':56}

min(dict1)

Output: 1

Built in functions on dictionary

clear(): Removes all key-value pairs from the dictionary, making it empty.

Example:

my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

print(len(my_dict))

Output: 3

get(key[, default]): Returns the value associated with the given key. If the key is not found, it returns the optional default value (or None if not specified).

Example

my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

age = my_dict.pop('age')

print(age) # Output: 25

print(my_dict)

Output: {'name': 'John', 'city': 'New York'}

country = my_dict.pop('country', 'Unknown')

print(country)

Output: Unknown

keys(): Returns a list (or an iterable) containing all the keys in the dictionary.

Example:

my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

keys = my_dict.keys()

print(keys)

Output: dict_keys(['name', 'age', 'city'])

pop(key[, default]): Removes and returns the value associated with the given key. If the key is not found, it returns the optional default value (or raises a KeyError if not specified).

Example:

my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

age = my_dict.pop('age')

print(age)

Output: 25

print(my_dict)

Output: {'name': 'John', 'city': 'New York'}

popitem(): Removes and returns an arbitrary key-value pair as a tuple.

Example:

my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

item = my_dict.popitem()

print(item)

Output: ('city', 'New York')

print(my_dict)

Output: {'name': 'John', 'age': 25}

setdefault(key[, default]): Returns the value associated with the given key. If the key is not found, it adds the key with the optional default value (or None if not specified) to the dictionary.

Example:

```
my_dict = {'name': 'John', 'age': 25}
city = my_dict.setdefault('city', 'Unknown')
print(city)
```

Output: Unknown

print(my_dict)

Output: {'name': 'John', 'age': 25, 'city': 'Unknown'}

```
age = my_dict.setdefault('age', 30)
print(age)  # Output: 25 (existing value is returned)
```

values(): Returns a list (or an iterable) containing all the values in the dictionary.

Example

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
values = my_dict.values()
print(values)
```

Output: dict_values(['John', 25, 'New York'])

DICTIONARY METHODS

- In Python, several built-in methods allow us to manipulate dictionaries.

- These methods are useful for adding, removing, and changing the values of dictionary.
-

Clear() Method

- Removes all the elements from the dictionary
- Syntax: dictionary.clear()
- Here, clear() removes all the items present in the dictionary.
- The clear() method doesn't take any parameters.

```
first_dict = {
  "name": "ChampCode",
  "founder": "Advaith",
  "address":"Mysuru,Karnataka",
  "Contact no": –8897868975‖,
  "Email": "Champcode@gmail.com",
}
first_dict.clear()
print(first_dict)
```

Output: {}

Values() Method

The values method accesses all the values in a dictionary. Like the keys() method, it returns the values in a tuple.

Syntax: dictionary.values()

values() method returns a view object that displays a list of all values in a given dictionary.

```
first_dict = {
  "name": "ChampCode",
  "founder": "Advaith",
  "address":"Mysuru,Karnataka",
  "Contact no": –8897868975‖,
  "Email": "Champcode@gmail.com",
}
dict_values = first_dict.values()
print(dict_values)
```

Output: dict_values('ChampCode', 'Advaith', 'Mysuru,Karnataka', –8897868975‖, ‖Champcode@gmail.com)

Items() Method

The items() method returns all the entries of the dictionary in a list. In the list is a tuple representing each of the items.

Syntax

dictionary.items()

Example:

```
first_dict = {
   "name": "ChampCode",
   "founder": "Advaith",
   "address": "Mysuru,Karnataka",
   "Contact no": –8897868975‖,
   "Email": "Champcode@gmail.com",
}
items = first_dict.items()
print(items)
```

Output: dict_items[('name',"ChampCode"), ("founder","Advaith"), (address": "Mysuru,Karnataka")("Contact no": –8897868975‖), ("Email": "Champcode@gmail.com",)]

keys() Method

The keys() returns all the keys in the dictionary. It returns the keys in a tuple – another Python data structure.

```
first_dict = {
   "name": "ChampCode",
  "founder": "Advaith",
  "address": "Mysuru,Karnataka",
  "Contact no": –8897868975‖,
  "Email": "Champcode@gmail.com",
}
items = first_dict.keys()
print(items)
```

Output: dict_keys('name', 'founder', 'address', 'Contact no', 'Email',)

Pop() Method

The pop() method removes a key-value pair from the dictionary. To make it work, you need to specify the key inside its parentheses.

```
first_dict = {
  "name": "ChampCode",
  "founder": "Advaith",
  "address": "Mysuru,Karnataka",
  "Contact no": –8897868975‖,
  "Email": "Champcode@gmail.com",
}
first_dict.pop("Contact no")
print(first_dict)
```

Output: { "name": "ChampCode", "founder": "Advaith","address": "Mysuru,Karnataka", "Email": "Champcode@gmail.com" }

You can see the Contact no key and its value have been removed from the dictionary.

Popitem() Methods

The popitem() method works like the pop() method. The difference is that it removes the last item in the dictionary.

```
first_dict = {
  "name": "ChampCode",
  "founder": "Advaith",
  "address": "Mysuru,Karnataka",
  "Contact no": –8897868975‖,
  "Email": "Champcode@gmail.com"
}
first_dict.popitem()
print(first_dict)
```

Output: { "name": "ChampCode", "founder": "Advaith","address": "Mysuru,Karnataka", "Contact no": –8897868975‖ }

You can see that the last key-value pair ("Email": "Champcode@gmail.com") has been removed from the dictionary.

Update() Method

The update() method adds an item to the dictionary.

You have to specify both the key and value inside its braces and surround it with curly braces.

```
first_dict = {
  "name": "ChampCode",
  "founder": "Advaith",
  "address": "Mysuru,Karnataka",
  "Contact no": –8897868975‖,
  "Email": "Champcode@gmail.com"
}
first_dict.update({―Alternative_contactno:‖ –7854658674‖} )
print(first_dict)
```

Output: {"name": "ChampCode", "founder": "Advaith","address": "Mysuru,Karnataka", "Contact no": –8897868975‖, "Email":‖ Champcode@gmail.com,‖,‖ Alternative_contactno:‖ –7854658674‖}

The new entry has been added to the dictionary.

Copy() Method

The copy() method does what its name implies – it copies the dictionary into the variable specified.

```
first_dict = {
   "name": "ChampCode",
   "founder": "Advaith",
   "address": "Mysuru,Karnataka",
   "Contact no": ―8897868975‖,
   "Email": "Champcode@gmail.com"
}
second_dict = first_dict.copy()
print(second_dict)
```

Output: { "name": "ChampCode", "founder": "Advaith","address": "Mysuru,Karnataka", "Contact no": ―8897868975‖, "Email": "Champcode@gmail.com"}

<u>POPULATING A DICTIONARY</u>

populating a dictionary refers to adding or assigning values to key-value pairs in the dictionary. There are several ways to populate a dictionary in Python.
Here are a few examples:

1.Assignment:

```
my_dict = {}  # Empty dictionary

# Assigning values to keys
my_dict['name'] = 'John'
my_dict['age'] = 25
my_dict['city'] = 'New York'

print(my_dict)
Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

2.Dictionary literal:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
print(my_dict)
```

Output: {'name': 'John', 'age': 25, 'city': 'New York'}

3.Using the dict() constructor:

```
my_dict = dict(name='John', age=25, city='New York')
print(my_dict)
Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

4.Using the fromkeys() method:

```
keys = ['name', 'age', 'city']
values = ['John', 25, 'New York']
my_dict = dict.fromkeys(keys, None)
for i, key in enumerate(keys):
    my_dict[key] = values[i]
print(my_dict)
Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

5.Using a loop to populate the dictionary:

```
keys = ['name', 'age', 'city']
values = ['John', 25, 'New York']

my_dict = {}
for i in range(len(keys)):
    my_dict[keys[i]] = values[i]

print(my_dict)
Output: {'name': 'John', 'age': 25, 'city': 'New York'}
```

These are some common methods to populate a dictionary in Python. Choose the method that best suits your needs and the structure of the data you want to store in the dictionary.

TRAVERSING A DICTIONARY

- Traversing a dictionary means iterating over its keys, values, or key-value pairs to access and process the data stored in the dictionary.
- Here are several ways to traverse a dictionary in Python:

1.Iterate over keys:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for key in my_dict:
    print(key)
```

Output:

```
name
age
city
```

2.Iterate over values:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for value in my_dict.values():
    print(value)
```

Output:

```
John
25
New York
```

3.Iterate over key-value pairs (using items()):

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for key, value in my_dict.items():
    print(key, value)
```

Output:

```
name John
age 25
city New York
```

4.Accessing values using keys:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for key in my_dict:
    value = my_dict[key]
    print(key, value)
```

Output:

```
name John
age 25
city New York
```

5.Using the get() method:

```
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}
for key in my_dict:
    value = my_dict.get(key)
    print(key, value)
```

Output:

```
name John
age 25
city New York
```

TUPLE

- A tuple is an ordered collection of elements.
- It is similar to a list, but unlike lists, tuples are immutable, meaning they cannot be modified once created.
- Tuples are defined using parentheses () and separating the elements with commas.
- Tuples can contain elements of different data types, and they can even store other data structures such as lists or dictionaries.
- Tuples can also be empty, containing no elements at all

Here's an example of a tuple:

```
my_tuple = (1, 2, 3, 'a', 'b', 'c')
```

In the above example, my_tuple is a tuple that contains integer values (1, 2, 3) as well as string values ('a', 'b', 'c').

TUPLE OPERATION

Tuples are immutable sequences in Python, and they support various operations. Here are some common operations that can be performed on tuples:

1.Accessing Elements:

You can access individual elements of a tuple using indexing or slicing.

```
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple[0])  # Output: 1
print(my_tuple[1:4])  # Output: (2, 3, 4)
```

2.Concatenation:

You can concatenate two or more tuples using the + operator.

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
```

```
concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)
```

Output: (1, 2, 3, 4, 5, 6)

3.Repetition:

You can repeat a tuple multiple times using the * operator.

```
my_tuple = (1, 2, 3)
repeated_tuple = my_tuple * 3
print(repeated_tuple)
```

Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)

4.Length: You can determine the length of a tuple using the len() function.

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

print(len(my_tuple))
```

Output: 5

5.Membership Test: You can check if an element exists in a tuple using the in keyword.

```
my_tuple = (1, 2, 3, 4, 5)
print(3 in my_tuple) # Output: True
print(6 in my_tuple)  # Output: False
```

6.Tuple Unpacking: You can assign the elements of a tuple to multiple variables simultaneously.

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

7.Iterating Over a Tuple: You can use a loop to iterate over the elements of a tuple.

```
my_tuple = (1, 2, 3, 4, 5)
for element in my_tuple:
    print(element)
```

TUPLE METHODS

Tuples in Python are immutable sequences and have a few built-in methods for various operations. Here are some commonly used tuple methods:

1.count():

Returns the number of occurrences of a specific value in a tuple.

```
my_tuple = (1, 2, 2, 3, 3, 3)
count = my_tuple.count(3)
print(count)
```

Output: 3

2.index():

Returns the index of the first occurrence of a value in a tuple.

Example

```
my_tuple = (10, 20, 30, 40, 50)
index = my_tuple.index(30)
print(index)
```

Output: 2

3.len():

Returns the number of elements in a tuple.

Example

```
my_tuple = (1, 2, 3, 4, 5)
length = len(my_tuple)
print(length)
```

Output: 5

4.sorted():

Returns a new sorted list from the elements of the tuple.

Example

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

```
sorted_tuple = sorted(my_tuple)
print(sorted_tuple)
```

Output: [1, 2, 3, 4, 5]

5.max():

Returns the largest element in a tuple.

Example

```
my_tuple = (10, 5, 20, 30)
maximum = max(my_tuple)
print(maximum)
```

Output: 30

6.min():

Returns the smallest element in a tuple.

Example

```
my_tuple = (10, 5, 20, 30)
minimum = min(my_tuple)
print(minimum)
```

Output: 5

7.any():

Returns True if at least one element in the tuple is True. Returns False otherwise.

Example

```
my_tuple = (False, False, True, False)
result = any(my_tuple)
print(result)
```

Output: True

8.all(): Returns True if all elements in the tuple are True. Returns False otherwise.

Example

```
my_tuple = (True, True, False, True)
result = all(my_tuple)
```

```
print(result)
```

Output: False

BUILTIN FUNCTION IN TUPLE

Tuples in Python have several built-in functions that can be used to perform various operations. Here are some commonly used built-in functions for tuples:

1.len(tuple):

Returns the length (number of elements) of the tuple.

Example:

```
my_tuple = (1, 2, 3, 'a', 'b', 'c')
print(len(my_tuple))
```

Output: 6

2.tuple(iterable):

Converts an iterable object (such as a list, string, or another tuple) into a tuple.

Example:

```
my_list = [1, 2, 3, 4, 5]
converted_tuple = tuple(my_list)
print(converted_tuple)
```

Output: (1, 2, 3, 4, 5)

3.max(tuple):

Returns the largest element in the tuple.

Example:

```
my_tuple = (10, 5, 20, 15)
print(max(my_tuple))
```

Output: 20

4.min(tuple):

Returns the smallest element in the tuple.

Example:

```
my_tuple = (10, 5, 20, 15)
print(min(my_tuple))
```

Output: 5

5.sum(tuple):

Returns the sum of all the elements in the tuple (if they are numbers).

Example

```
my_tuple = (1, 2, 3, 4, 5)
print(sum(my_tuple))
```

Output: 15

6.sorted(tuple):

Returns a new tuple with the elements sorted in ascending order.

Example:

```
my_tuple = (5, 2, 8, 1, 3)
sorted_tuple = sorted(my_tuple)
print(sorted_tuple)
```

Output: (1, 2, 3, 5, 8)

7.tuple.count(value):

Returns the number of occurrences of a specific value in the tuple.

Example:

```
my_tuple = (1, 2, 3, 2, 4, 2)
print(my_tuple.count(2))
```

Output: 3

8.tuple.index(value[, start[, end]]):

Returns the index of the first occurrence of a value in the tuple.

Example:

```
my_tuple = (1, 2, 3, 2, 4, 2)
print(my_tuple.index(2))
```

SET:

- a set is an unordered collection of unique elements. I
- t is defined by enclosing a comma-separated sequence of elements inside curly braces {}.

Example,

```
s = {1, 2, 3, 4, 5}
```

Sets have the following characteristics:

- **Unordered:** The elements in a set are not stored in any particular order, and their positions can change.
- **Unique elements**: A set cannot contain duplicate elements. If you try to add a duplicate element to a set, it will only be stored once.
- **Mutable:** You can add or remove elements from a set after it is created.
- **Membership test**: Sets are useful for membership testing. You can quickly check if an element is present in a set using the in operator.

Here's an example that demonstrates some basic operations with sets:

```
my_set = {1, 2, 3, 4, 5}

print(len(my_set))        # Output: 5

my_set.add(6)
print(my_set)           # Output: {1, 2, 3, 4, 5, 6}

my_set.remove(2)
print(my_set)           # Output: {1, 3, 4, 5, 6}

print(3 in my_set)       # Output: True
print(2 in my_set)       # Output: False
```

OPERATIONS ON SETS

Sets in Python provide various operations to perform common set operations like union, intersection, difference, and more. Here are some of the commonly used operations on sets:

1. Union: The union of two sets set1 and set2 contains all the unique elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print(union_set)
Output: {1, 2, 3, 4, 5}
```

Alternatively, you can use the | operator to perform the union operation:

```
union_set = set1 | set2
```

2. Intersection: The intersection of two sets set1 and set2 contains the elements that are common to both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1.intersection(set2)
print(intersection_set)
Output: {3}
```

Alternatively, you can use the & operator to perform the intersection operation:

```
intersection_set = set1 & set2
```

3. Difference: The difference between two sets set1 and set2 contains the elements that are in set1 but not in set2.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
```

```
difference_set = set1.difference(set2)
print(difference_set)
Output: {1, 2}
```

Alternatively, you can use the - operator to perform the difference operation:

```
difference_set = set1 - set2
```

4. Symmetric Difference: The symmetric difference between two sets set1 and set2 contains the elements that are in either set1 or set2, but not both.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set)
Output: {1, 2, 4, 5}
```

Alternatively, you can use the ^ operator to perform the symmetric difference operation:

```
symmetric_difference_set = set1 ^ set2
```

These are just a few examples of the operations you can perform on sets in Python. Sets also provide methods like isdisjoint(), issubset(), issuperset(), and more to perform additional set operations and comparisons.

BUILT IN FUNCTION ON SETS

Sets in Python provide several built-in functions to perform operations and manipulations on sets. Here are some of the commonly used built-in functions for sets:

1. len():

Returns the number of elements in a set.

Example

```
my_set = {1, 2, 3, 4, 5}
print(len(my_set))   # Output: 5
```

2. add():

Adds an element to a set.

Example

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)
```

Output: {1, 2, 3, 4}

3. remove(): Removes an element from a set. Raises a KeyError if the element is not found.

Example

```
my_set = {1, 2, 3, 4}
my_set.remove(3)
print(my_set)
```

Output: {1, 2, 4}

4. discard(): Removes an element from a set if it is present. Does not raise an error if the element is not found.

Example

```
my_set = {1, 2, 3, 4}
my_set.discard(3)
print(my_set)
```

Output: {1, 2, 4}

5. pop(): Removes and returns an arbitrary element from a set. Raises a KeyError if the set is empty.

Example

```
my_set = {1, 2, 3, 4}
removed_element = my_set.pop()
print(removed_element)
```

Output: The removed element (e.g., 1)

```
print(my_set)
```

Output: The modified set without the removed element

6. clear():

Removes all elements from a set, making it empty.

Example

```
my_set = {1, 2, 3, 4}
my_set.clear()
print(my_set)
Output: set()
```

7. copy(): Creates a shallow copy of a set.

Example

```
my_set = {1, 2, 3}
copy_set = my_set.copy()
print(copy_set)
```

Output: {1, 2, 3}

8. issubset():

Checks if a set is a subset of another set.

Example

```
set1 = {1, 2, 3}
set2 = {1, 2, 3, 4, 5}
print(set1.issubset(set2))
```

Output: True

9. issuperset():

Checks if a set is a superset of another set.

Example

```
set1 = {1, 2, 3, 4, 5}
set2 = {1, 2, 3}
print(set1.issuperset(set2))
```

Output: True

10. isdisjoint():

Checks if two sets have no common elements.

Example

```
set1 = {1, 2, 3}
set2 = {4, 5, 6}
print(set1.isdisjoint(set2))
```

Output: True

SET METHODS

Sets in Python have several built-in methods that provide various operations and manipulations. Here are some commonly used set methods:

1. add():

Adds an element to the set.

Example

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)
```

Output: {1, 2, 3, 4}

2. remove():

Removes an element from the set. Raises a KeyError if the element is not found.

Example

```
my_set = {1, 2, 3, 4}
my_set.remove(3)
print(my_set)
```

Output: {1, 2, 4}

3. discard():

Removes an element from the set if it is present. Does not raise an error if the element is not found.

Exampe

```
my_set = {1, 2, 3, 4}
my_set.discard(3)
print(my_set)
```

Output: {1, 2, 4}

4. pop():

Removes and returns an arbitrary element from the set. Raises a KeyError if the set is empty.

Example

```
my_set = {1, 2, 3, 4}
removed_element = my_set.pop()
print(removed_element)
```

Output: The removed element (e.g., 1)

```
print(my_set)
```

Output: The modified set without the removed element

5. clear(): Removes all elements from the set, making it empty.

Example

```
my_set = {1, 2, 3, 4}
my_set.clear()
print(my_set)
```

Output: set()

6. copy():

Creates a shallow copy of the set.

Example

```
my_set = {1, 2, 3}
copy_set = my_set.copy()
print(copy_set)
```

Output: {1, 2, 3}

7. union():

Returns a new set that is the union of the set and one or more other sets.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print(union_set)
```

Output: {1, 2, 3, 4, 5}

8. intersection(): Returns a new set that is the intersection of the set and one or more other sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1.intersection(set2)
print(intersection_set)
```

Output: {3}

9. difference(): Returns a new set that contains the elements in the set but not in one or more other sets.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1.difference(set2)
print(difference_set)
```

Output: {1, 2}

10. symmetric_difference():

Returns a new set that contains the elements that are in either the set or another set, but not both.

Example

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_set =
set1.symmetric_difference(set2)
print(symmetric_difference_set)
```

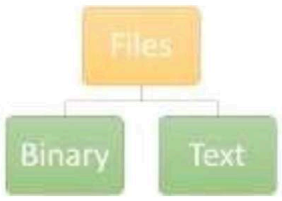
Output: {1, 2, 4, 5}

FILE HANDLING

- A file is a named location on the system storage which stores related data for future purposes.
- The data can be anything like a simple text file, video or audio file, or any complex executable programs.
- Files consist of three important parts as listed below.
- The **header** holds the metadata of the file i.e., the details like name, size, type, etc of file.
- **Data** is the contents in the file like texts, pictures, audio, etc.
- **EOF** indicates the end of the file.
- when it comes to accessing a file we must know where we have stored the file. For that, we use the File path which gives the route or location of the file. The file path contains 3 parts as listed below.
- The folder path indicates the location of the folder in which the file resides.
- Filename indicates the actual name of the file.
- File extension shows the type of file.
- Types of Files in python

Types of files

There are two types of files in python. They are:



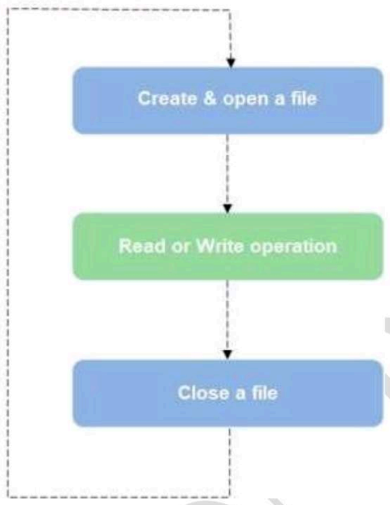
- **Binary Files** are files that contain non-text values like images, audios etc. Usually, binary files contain the objects in the form of zeros and ones.

- **Text Files** are files that contain text values that are structured into lines. A text file in short contains multiple lines of text values. Each line ends with a special character referred to as the End of Line.

When you want to work with files, you need to perform the 3 basic operations which are listed below in the order of processing.

- Open a file
- Read or write operation
- Close a file

Flow chart for file handling



- The above flow chart gives you the workflow of file handling in python.
- Initially, we need to create a file. This can be done manually by the user wherever he needs and save it with a valid filename and extension. Also, python supports file creation which you will encounter in future sessions.

Modes of files.

- **R(Read Default mode):-** Open file for reading.It shows an error if the file does not exist

- **W(Write):-**Opens file in write-only mode to overwrite. Creates a file if it does not exist
- **A(Append):-**Appends the file at the end of the file without trimming the existing. Creates a file if it does not exist.
- **X(Create):-** Creates the defined file. Error if the file already exists.
- **B(Binary):-** Opens file in binary mode
- **T(Text):-**Opens file in text mode. Default mode.
- **R+(Read and Write):-**Opens file for both reading and writing
- **RB(Binary read):-**Opens file in read and binary mode
- **RB+(Binary read and write):-** Opens file in binary mode for reading and writing
- **W+(Read and write)**:-Opens file in writing and reading mode
- **WB(Binary write):-**Opens file in writing and binary mode
- **WB+(Binary read and write)**:-Opens file in binary mode for writing and reading
- **A+(Append read and write):-**Open file in appending mode for reading and writing.
- **Ab(Binary append)**:-Opens in append mode and binary mode
- **Ab+(Binary append read and write):-**Opens file in read write binary mode for appending

You can use several attributes to get the details of the file once after a file object gets created. Some of the attributes are listed below.

.closed:returns True when the file is closed otherwise returns false

.mode: returns the mode of the file with which file is opened.

.name: returns the name of the file.

PYTHON FILE HANDLING OPERATIONS

Most importantly there are 4 types of operations that can be handled byPython on files:

- Open
- Read
- Write
- Close

Other operations include:

- Rename
- Delete

PYTHON CREATE AND OPEN A FILE

- Python has an in-built function called open() to open a file.
- It takes a minimum of one argument as mentioned in the below syntax
- The open method returns a file object which is used to access the write, read and other in-built methods.

Syntax:

```
file_object = open(file_name, mode)
```

- Here, file_name is the name of the file or the location of the file that you want to open, and file_name should have the file extension included as well. Which means in **test.txt** – the term test is the name of the file and .txt is the extension of the file.
- The mode in the open function syntax will tell Python as what operation you want to do on a file.

- **'r' – Read Mode:** Read mode is used only to read data from the file.

- **'w' – Write Mode:** This mode is used when you want to write data into the file or modify it. Remember write mode overwrites the data present in the file.
- **'a' – Append Mode:** Append mode is used to append data to the file. Remember data will be appended at the end of the file pointer.
- **'r+' – Read or Write Mode:** This mode is used when we want to write or read the data from the same file.
- **'a+' – Append or Read Mode:** This mode is used when we want to read data from the file or append the data into the same file.

Note: The above-mentioned modes are for opening, reading or writing text files only.

While using binary files, we have to use the same modes with the letter **'b'** at the end. So that Python can understand that we are interacting with binary files.

- **'wb' –** Open a file for write only mode in the binary format.
- **'rb' –** Open a file for the read-only mode in the binary format.
- **'ab' –** Open a file for appending only mode in the binary format.
- **'rb+' –** Open a file for read and write only mode in the binary format.
- **'ab+' –** Open a file for appending and read-only mode in the binary format.

<u>Example 1:</u>

fo = open("C:/Documents/Python/test.txt", "r+")

- In the above example, we are opening the file named „test.txt" present at the location „C:/Documents/Python/" and we are opening the same file in a read-write mode which gives us more flexibility.

<u>Example 2:</u>

fo = open("C:/Documents/Python/img.bmp", "rb+")

- In the above example, we are opening the file named „img.bmp" present at the location "C:/Documents/Python/", But, here we are trying to open the binary file.

PYTHON READ FROM FILE

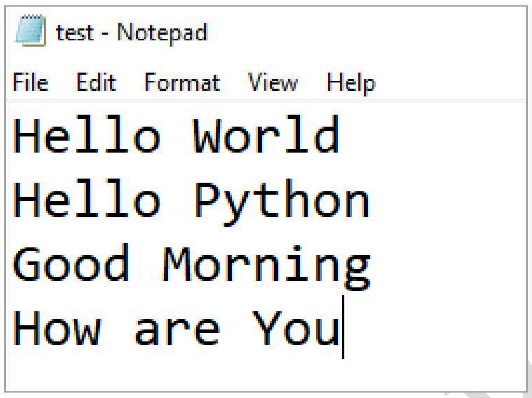
In order to read a file in python, we must open the file in read mode.

There are three ways in which we can read the files in python.

- read([n])
- readline([n])
- readlines()

Here, n is the number of bytes to be read. First,

create a sample text file as shown below.



```
Hello World
Hello Python
Good Morning
How are You
```

Example 1:

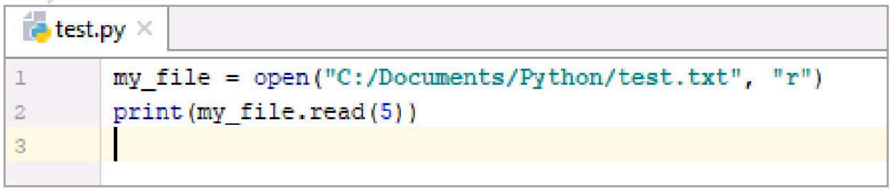
my_file = open(“C:/Documents/Python/test.txt”, “r”)

print(my_file.read(5))

Output:

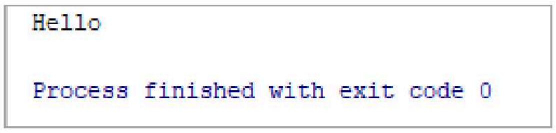
Hello

Here we are opening the file test.txt in a read-only mode and are reading only the first 5 characters of the file using the my_file.read(5) method.



```
my_file = open("C:/Documents/Python/test.txt", "r")
print(my_file.read(5))
```

Output:



```
Hello

Process finished with exit code 0
```

Reading the entire file at once

filename = “C:/Documents/Python/test.txt”

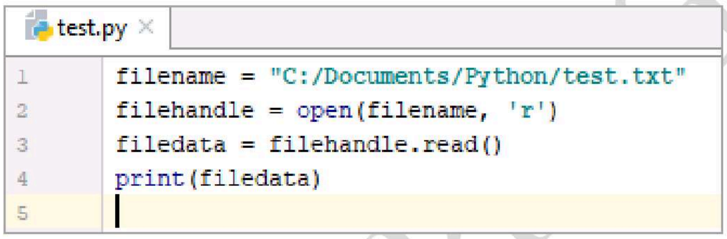
filehandle = open(filename, „r‟)

filedata = filehandle.read()

print(filedata)

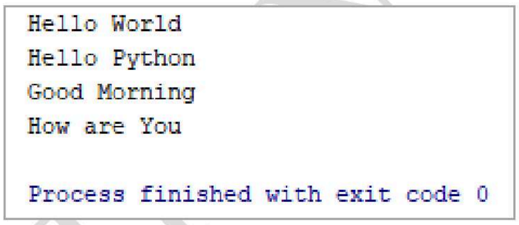
Output:

Hello World
Hello Python
Good Morning
How are You



```
filename = "C:/Documents/Python/test.txt"
filehandle = open(filename, 'r')
filedata = filehandle.read()
print(filedata)
```

Output:



```
Hello World
Hello Python
Good Morning
How are You

Process finished with exit code 0
```

PYTHON WRITE TO FILE

- In order to write data into a file, we must open the file in write mode.
- We need to be very careful while writing data into the file as it overwrites the content present inside the file that you are writing, and all the previous data will be erased.

We have two methods for writing data into a file as shown below.

- write(string)
- writelines(list)

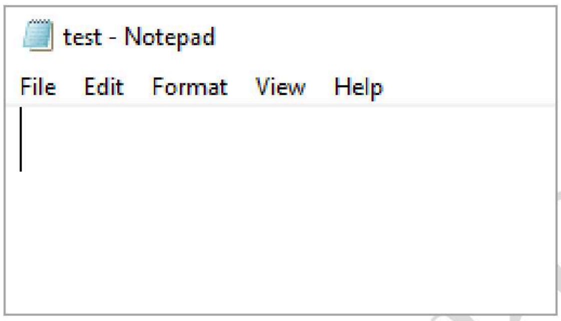
Example 1:

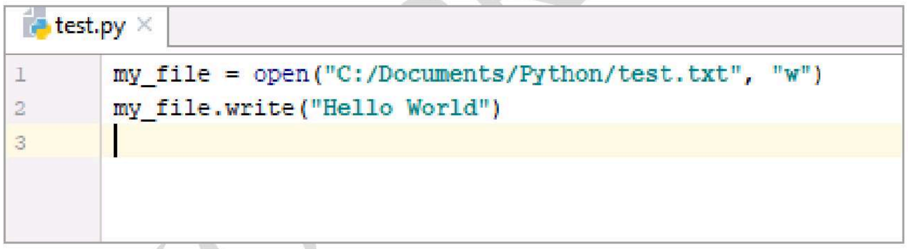
my_file = open(“C:/Documents/Python/test.txt”, “w”)

my_file.write(“Hello World”)

The above code writes the String „Hello World‟ into the „test.txt‟ file.

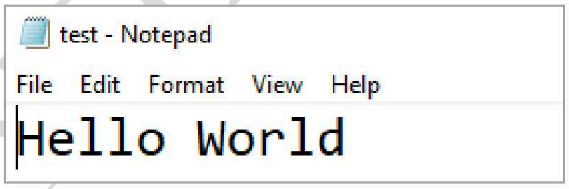
Before writing data to a test.txt file:





```
my_file = open("C:/Documents/Python/test.txt", "w")
my_file.write("Hello World")
```

Output:



```
Hello World
```

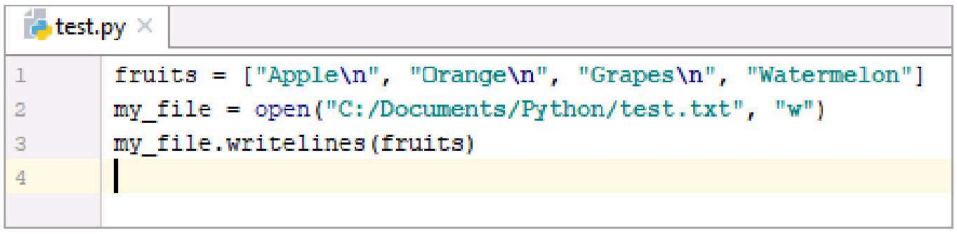
Example 2:

fruits = [“Apple\n”, “Orange\n”, “Grapes\n”, “Watermelon”]

my_file = open(“C:/Documents/Python/test.txt”, “w”)

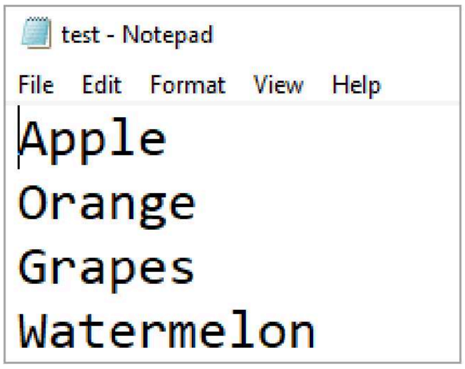
my_file.writelines(fruits)

The above code writes a **list of data** into the „test.txt" file simultaneously.



```
fruits = ["Apple\n", "Orange\n", "Grapes\n", "Watermelon"]
my_file = open("C:/Documents/Python/test.txt", "w")
my_file.writelines(fruits)
```

Output:



```
Apple
Orange
Grapes
Watermelon
```

PYTHON APPEND TO FILE

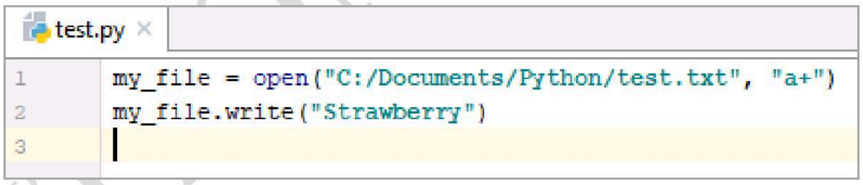
- To append data into a file we must open the file in „a+" mode so that we will have access to both the append as well as write modes.

Example 1:

my_file = open("C:/Documents/Python/test.txt", "a+")

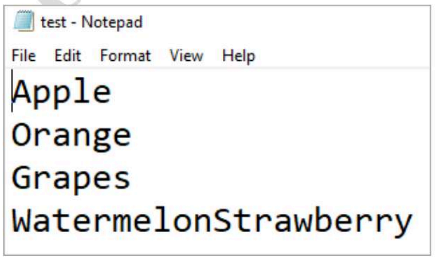
my_file.write ("Strawberry")

The above code appends the string „Apple" at the **end** of the „test.txt" file.



```
my_file = open("C:/Documents/Python/test.txt", "a+")
my_file.write("Strawberry")
```

Output:



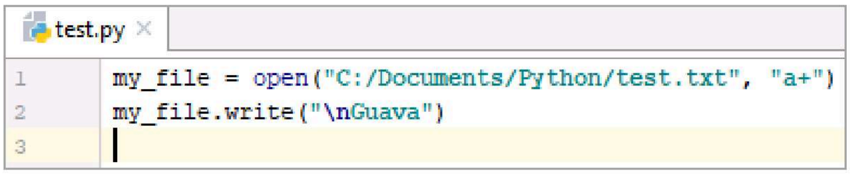
```
Apple
Orange
Grapes
WatermelonStrawberry
```

Example 2:

my_file = open("C:/Documents/Python/test.txt", "a+")

my_file.write ("\nGuava")

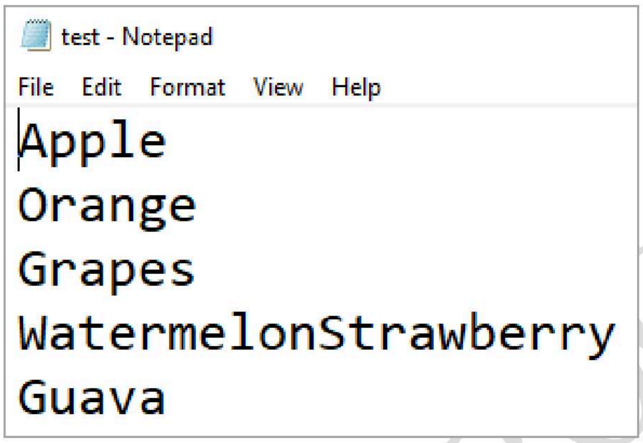
The above code appends the string „Guava‟ at the end of the „test.txt‟ file **in a new line**.



test.py

```
my_file = open("C:/Documents/Python/test.txt", "a+")
my_file.write("\nGuava")
```

Output:



test - Notepad

File Edit Format View Help

```
Apple
Orange
Grapes
WatermelonStrawberry
Guava
```

PYTHON CLOSE FILE

- In order to close a file, we must first open the file. In python, we have an in-built method called close() to close the file which is opened.
- Whenever you open a file, it is important to close it, especially, with write method. Because if we don‟t call the close function after the write method then whatever data we have written to a file will not be saved into the file.

Example 1:

my_file = open("C:/Documents/Python/test.txt", "r")

print(my_file.read())

my_file.close()

Example 2:

my_file = open("C:/Documents/Python/test.txt", "w") my_file.write(“Hello World”)

my_file.close()

PYTHON RENAME OR DELETE FILE

- Python provides us with an “os” module which has some in-built methods that would help us in performing the file operations such as renaming and deleting the file.

In order to use this module, first of all, we need to import the “os” module in our program and then call the related methods.

rename() method:

This rename() method accepts two arguments i.e. the current file name and the new file name.

Syntax:

os.rename(current_file_name, new_file_name)

Example 1:

import os

os.rename(“test.txt”, “test1.txt”)

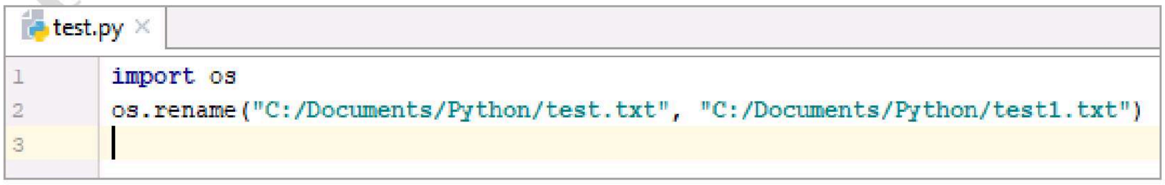
Here „test.txt‟ is the current file name and „test1.txt‟ is the new file name.

You can specify the location as well as shown in the below example.

Example 2:

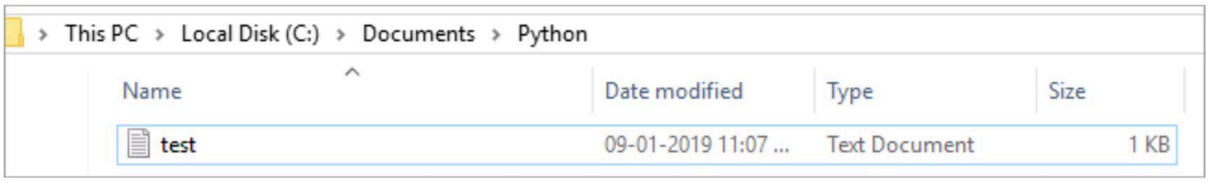
import os

os.rename(“C:/Documents/Python/test.txt”, “C:/Documents/Python/test1.txt”)



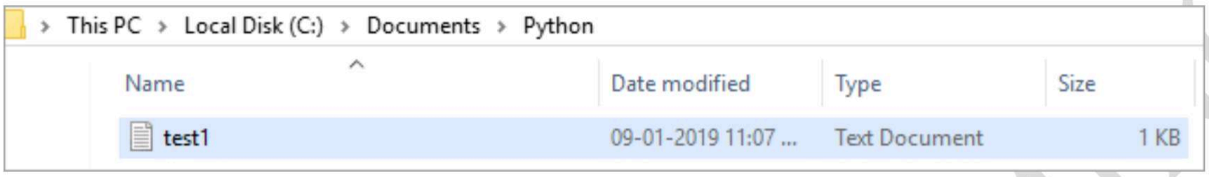
```
import os
os.rename("C:/Documents/Python/test.txt", "C:/Documents/Python/test1.txt")
```

Before Renaming the file:


> This PC > Local Disk (C:) > Documents > Python

Name	Date modified	Type	Size
test	09-01-2019 11:07 ...	Text Document	1 KB

After executing the above program


> This PC > Local Disk (C:) > Documents > Python

Name	Date modified	Type	Size
test1	09-01-2019 11:07 ...	Text Document	1 KB

remove() method:

We use the remove() method to delete the file by supplying the file name or the file location that you want to delete.

Syntax:

os.remove(file_name)

Example 1:

```
import os
os.remove("test.txt")
```

Here „test.txt" is the file that you want to remove.

Similarly, we can pass the file location as well to the arguments as shown in the below example

Example 2:

```
import os
os.remove("C:/Documents/Python/test.txt")
```

Writing and Reading Data from a Binary File

Binary files store data in the binary format (0"s and 1"s) which is understandable by the machine. So when we open the binary file in our machine, it decodes the data and displays in a human-readable format.

Example:

#Let‟s create some binary file.

my_file = open(“C:/Documents/Python/bfile.bin”, “wb+”)

message = “Hello Python”

file_encode = message.encode(“ASCII”)

my_file.write(file_encode)

my_file.seek(0)

bdata = my_file.read()

print(“Binary Data:”, bdata)

ntext = bdata.decode(“ASCII”)

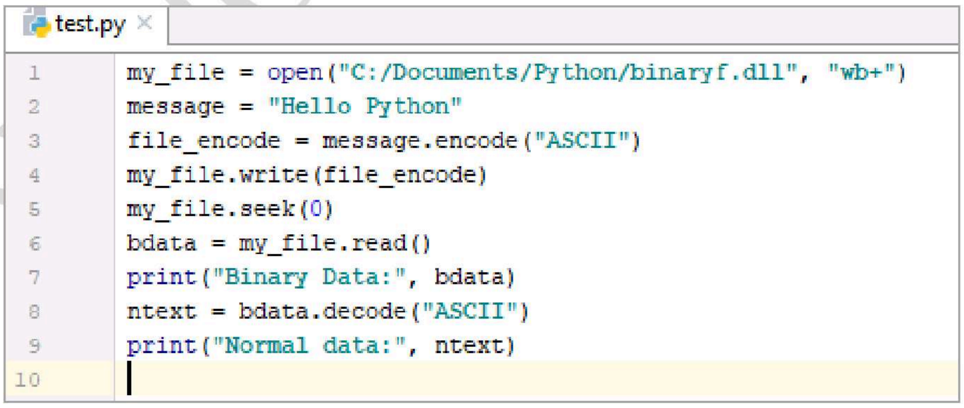
print(“Normal data:”, ntext)

In the above example, first we are creating a binary file **‘bfile.bin’** with the read and write access and whatever data you want to enter into the file must be encoded before you call the write method.

Also, we are printing the data without decoding it, so that we can observe how the data exactly looks inside the file when it‟s encoded and we are also printing the same data by decoding it so that it can be readable by humans.

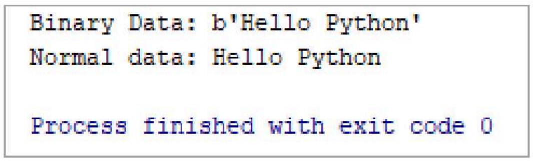
Output:

Binary Data: b‟Hello Python‟
Normal data: Hello Python



```
my_file = open("C:/Documents/Python/binaryf.dll", "wb+")
message = "Hello Python"
file_encode = message.encode("ASCII")
my_file.write(file_encode)
my_file.seek(0)
bdata = my_file.read()
print("Binary Data:", bdata)
ntext = bdata.decode("ASCII")
print("Normal data:", ntext)
```

Output:



```
Binary Data: b'Hello Python'
Normal data: Hello Python

Process finished with exit code 0
```

File I/O Attributes

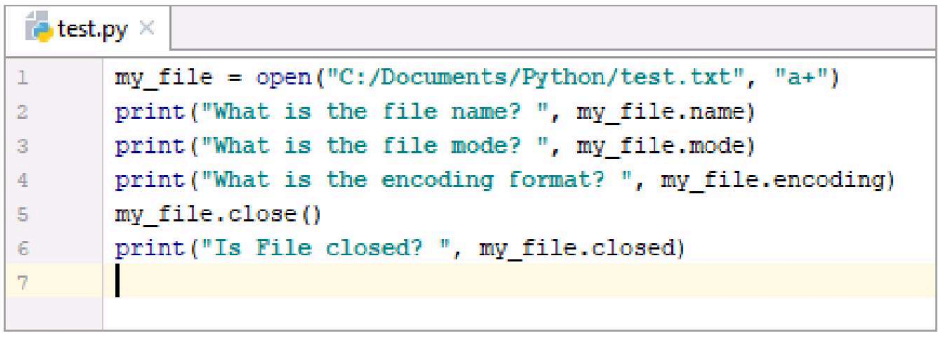
Attribute	Description
Name	Return the name of the file
Mode	Return mode of the file
Encoding	Return the encoding format of the file
Closed	Return true if the file closed else returns false

Example:

```
my_file = open("C:/Documents/Python/test.txt", "a+")
print("What is the file name? ", my_file.name)
print("What is the file mode? ", my_file.mode)
print("What is the encoding format? ", my_file.encoding)
print("Is File closed?", my_file.closed)
my_file.close()
print("Is File closed?", my_file.closed)
```

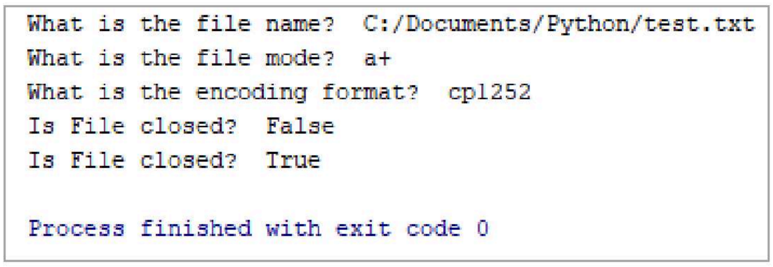
Output:

```
What is the file name? C:/Documents/Python/test.txt
What is the file mode? r
What is the encoding format? cp1252
Is File closed? False
Is File closed? True
```


test.py

```
my_file = open("C:/Documents/Python/test.txt", "a+")
print("What is the file name? ", my_file.name)
print("What is the file mode? ", my_file.mode)
print("What is the encoding format? ", my_file.encoding)
my_file.close()
print("Is File closed? ", my_file.closed)
```

Output:



```
What is the file name?  C:/Documents/Python/test.txt
What is the file mode?  a+
What is the encoding format?  cp1252
Is File closed?  False
Is File closed?  True

Process finished with exit code 0
```

FILE PATH AND NAME

- In file handling, the file path and name refer to the location and name of a file that you want to read from or write to.
- The file path specifies the directory or folder structure where the file is located, and the file name is the actual name of the file itself.
- The file path can be either an absolute path or a relative path. An absolute path provides the full directory structure starting from the root directory, while a relative path is defined relative to the current working directory of the program.

Here are a few examples to illustrate the file path and name in file handling using Python:

1. Reading from a file with an absolute file path:

file_path = "/path/to/file.txt" with open(file_path, "r") as file:

data = file.read() print(data)

2. **Reading from a file with a relative file path:**

file_path = "data/file.txt" with open(file_path, "r") as file:

data = file.read()

print(data)

- In the above example, the file **file.txt** is located in the **data** folder, which is in the current working directory.

3. Writing to a file with an absolute file path:

file_path = "/path/to/output.txt" with open(file_path, "w") as file:
file.write("Hello, world!")

4. **Writing to a file with a relative file path:**

file_path = "output.txt" with open(file_path, "w") as file:

file.write("Hello, world!")

In this case, the file **output.txt** will be created in the current working directory.

It's important to note that the actual file name should be included in the file path. The file name should have the appropriate file extension to indicate the file type (e.g., **.txt**, **.csv**, **.json**, etc.). Additionally, ensure that you have the necessary permissions to read from or write to the specified file path.

FORMAT OPERATOR

- In file handling, the format operator is typically used to specify the formatting of data when reading from or writing to a file.
- In Python, the format operator is represented by the % symbol. It allows you to format strings by replacing placeholders with corresponding values.
- The format operator is often used in conjunction with the % formatting codes, which specify the type and format of the values being inserted.

Here's a basic example that demonstrates the usage of the format operator in file handling with Python:

```
# Writing to a file using format operator

name = "John"

age = 25

 with open("example.txt", "w") as file:

   file.write("Name: %s, Age: %d" % (name, age))
```

Reading from a file using format operator

```
with open("example.txt", "r") as file: data = file.read()

  print(data)
```

- In this example, the %s and %d are format codes.
- The %s is used to insert a string value, and %d is used to insert an integer value.
- The values (name, age) are provided in a tuple, and they replace thecorresponding format codes in the string.

The output of this code will be:

```
Name: JohnAge: 25
```