# Module -2

**Python Collection Objects, Classes**

#### Strings

* Creating and Storing Strings,
* Basic String Operations,
* Accessing Characters in String by Index Number,
* String Slicing and Joining,
* String Methods,
* Formatting Strings,

#### Lists

* Creating Lists,
* Basic List Operations,
* Indexing and Slicing in Lists,
* Built-In Functions Used on Lists,
* List Methods
* Sets, Tuples and Dictionaries.

#### Files

* reading and writing files.

#### Class

* Class Definition
* Constructors
* Inheritance
* Overloading

ROOPA.H.M, Dept of MCA, RNSIT Page 1

**Strings**

A string consists of a sequence of characters, which includes letters, numbers, punctuation marks and spaces.

## Creating and Storing Strings

* A string can be created by enclosing text in single and double quotes. A triple quote can be used for multi-line strings. Here are some examples:

>>> S='hello'

>>> Str="hello"

>>> M="""This is a multiline String across two lines"""

* Sometimes we may want to have a string that contains backslash and don't want it to be treated as an escape character. Such strings are called *raw string.* In Python raw string is created by prefixing a string literal with 'r' or 'R'. Python raw string treats backslash (\) as a literal character.

>>> s= **r**" world health \n organization"

>>> print(s)

world health \n organization

## Basic String Operations

**String Concatenation**

Concatenation can be done in various ways, some are as follows:

|  |  |
| --- | --- |
| >>> str1 = "Hello"  >>> str2 = 'there'  >>> str3 = str1 + str2  >>> print(str3) Hellothere | Strings can be concatenated by using + operator |
| >>> str= 'this' 'is' 'python' 'class'  >>> print(str) thisispythonclass | Strings can be concatenated by placing strings side by side. |

## The in operator

* + The in operator of Python is a Boolean operator which takes two string operands.
  + It returns True, if the first operand appears as a substring in second operand, otherwise returns False.

ROOPA.H.M, Dept of MCA, RNSIT Page 2

|  |
| --- |
| **Ex:1**  if 'pa' in “roopa:  print('Your string contains “pa”.') |
| **Ex:2**  if ';' not in “roopa”:  print('Your string does not contain any semicolons.') |
| **Ex:3 we can avoid writing longer codes like this** if t=='a' or t=='e' or t=='i' or t=='o' or t=='u': **instead we can write this code with ‘in’ operator**  if t in 'aeiou': |

## String comparison

* + Basic comparison operators like < (less than), > (greater than), == (equals) etc. can be applied on string objects.
  + Such comparison results in a Boolean value True or False.
  + Internally, such comparison happens using ASCII codes of respective characters.
  + List of ASCII values for some of the character set

A – Z : 65 – 90 a – z : 97 – 122 0 – 9 : 48 – 57

Space : 32 Enter Key : 13

### Examples are as follows:

|  |
| --- |
| **Ex:1**  if Name == ‘Ravindra’: print(‘Ravindra is selected.’) |
| **Ex:2**  if word < ' Ravindra':  print('Your name,' + word + ', comes before Ravindra.’)  elif word > ' Ravindra':  print('Your name,' + word + ', comes after Ravindra.’)  else:  print('All right, Ravindra.') |

ROOPA.H.M, Dept of MCA, RNSIT Page 3

## Traversal through a string with a loop

* + Extracting every character of a string one at a time and then performing some action on that character is known as traversal.
  + A string can be traversed either using while loop or using for loop in different ways.

***Consider an example using while loop:***

|  |  |
| --- | --- |
| st="Roopa" index=0  while index < len(st): print(st[index],  end="\t")  index+=1 | *Output:*  R o o p a |

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="grapes" for char in fruit:  print(char,end="\t") | *Output:*  g r a p e s |

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.

**Accessing Characters in String by Index Number**

* We can get at any single character in a string using an index specified in square brackets
* The index value must be an integer and starts at zero.
* The index value can be an expression that is computed.

Example : str= “ good morning”

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Character | g | o | O | d |  | m | o | r | n | i | n | g |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

>>> word = 'Python'

>>> word[0] # character in position 0 'P'

>>> word[5] # character in position 5 'n'

ROOPA.H.M, Dept of MCA, RNSIT Page 4

* + Python supports negative indexing of string starting from the end of the string as shown below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| character | g | o | O | d |  | m | o | r | n | i | n | g |
| index | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

>>> word = 'Python'

>>> word[-1] # last character 'n'

>>> word[-2] # second-last character 'o'

>>> word[-6] 'P'

## String Slicing and Joining String slices

* + A segment or a portion of a string is called as slice.
  + Only a required number of characters can be extracted from a string using colon (:) symbol.
  + The basic syntax for slicing a string :

|  |
| --- |
| String\_name[*start* **:** *end* **:** *step*] |
| Where,  *start* **:** the position from where it starts  *end* **:** the position where it stops(excluding end position)  *step****:*** also known as *stride*, is used to indicate number of steps to be incremented after extracting first character. The default value of *stride* is 1. |

* + If start is not mentioned, it means that slice should start from the beginning of the string.
  + If the end is not mentioned, it indicates the slice should be till the end of the string.
  + If the both are not mentioned, it indicates the slice should be from the beginning till the end of the string.

Examples: s = ”abcdefghij”

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| characters | a | b | C | D | e | f | g | h | i | j |
| Reverse index | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

ROOPA.H.M, Dept of MCA, RNSIT Page 5

|  |  |  |  |
| --- | --- | --- | --- |
| **Slicing Code Result Description**  s[2:5] cde characters at indices 2, 3, 4  s[ :5] abcde ﬁrst ﬁve characters  s[5: ] fghij characters from index 5 to the end  s[-2:] ij last two characters  s[ : ] abcdefghij entire string  s[1:7:2] bdf characters from index 1 to 6, by twos  s[ : :-1] jihgfedcba a negative step reverses the string  By the above set of examples, one can understand the power of string slicing and of Python script. The slicing is a powerful tool of Python which makes many task simple pertaining to data types like strings, Lists, Tuple, Dictionary etc.  **String Methods**   * Strings are an example of Python objects. * An object contains both data (the actual string itself) and methods, which are eﬀectively functions that are built into the object and are available to any instance of the object. * Python provides a rich set of built-in classes for various purposes. Each class is enriched with a useful set of utility functions and variables that can be used by a Programmer. * The built-in set of members of any class can be accessed using the dot operator as shown–   *objName.memberMethod(arguments)*   * The dot operator always binds the member name with the respective object name. This is very essential because, there is a chance that more than one class has members with same name. To avoid that conflict, almost all Object-oriented languages have been designed with this common syntax of using dot operator. * Python has a function called dir which lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods. | | | |
|  | >>> stuff = 'Hello world'  >>> type(stuff) <class 'str'>  >>> dir(stuff)  ['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'] | |  |
|  | ROOPA.H.M, Dept of MCA, RNSIT | Page 6 |  |

* *help* function can used to get some simple documentation on a method.

>>> **help(msg.casefold)**

Help on built-in function casefold:

casefold() method of builtins.str instance

Return a version of the string suitable for caseless comparisons..

* This is built-in help-service provided by Python. Observe the className.memberName format while using help.
* The methods are usually called using the object name. This is known as *method invocation*. We say that a method is invoked using an object.

|  |  |
| --- | --- |
| **Method description** | **Example** |
| **capitalize()**  Return a capitalized version of the string. More specifically, make the first character have upper case and the rest lower case. | >>> msg="bengaluru"  >>> print(msg.capitalize()) Bengaluru |
| **find()**  S.find(sub[, start[, end]]) -> int  Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].  Optional arguments start and end are interpreted as in slice notation.  Return -1 on failure. | >>> st='hello'  >>> i=st.find('l')  >>> print(i) 0  >>> print(st.find(‘x’))  -1  >>> st="calender of Feb.cal2019"  >>> i= st.find('cal')  >>> print(i) 0  >>> i=st.find('cal',10,20)  >>> print(i) 16 |

ROOPA.H.M, Dept of MCA, RNSIT Page 7

|  |  |
| --- | --- |
| **strip()**  Return a copy of the string with leading and trailing whitespace remove.  **strip(chars)**  If chars is given, remove characters specified as arguments at both end  **rstrip()**  to remove whitespace at right side  **lstrip()**  to remove whitespace at left side | >>> st=" hello world "  >>> st1 = st.strip()  >>> print(st1) hello world  >>> st="###Hello##“  >>> st1=st.strip('#’)  >>> print(st1) Hello |
| **casefold()**  Return a version of the string suitable for caseless comparisons. | >>> first="india"  >>> second="INDIA"  >>>first.casefold()== second.casefold()  True |
| **split(“separator”)**  The split() method returns a list of strings after breaking the given string by the specified separator. *separator* is a delimiter. The string splits at this specified separator. If is not provided then any white space is a separator. | >>>str="this is python"  >>> var=str.split()  >>> print(var) ['this','is','pyhton']  >>> s="abc,def,ght,ijkl"  >>> print(s.split(",")) ['abc', 'def', 'ght', 'ijkl'] |
| **startswith(prefix, start, end)**  Return *True* if Str starts with the specified prefix, *False* otherwise. With optional start, test Str beginning at that position. With optional end, stop comparing Str at that position. Prefix can also be a tuple of strings  to try. | >>> str="logical"  >>> print(str.startswith("L"))  False  #case sensitive, hence false |

ROOPA.H.M, Dept of MCA, RNSIT Page 8

|  |  |
| --- | --- |
| **join()**  Concatenate any number of strings. The string whose method is called is inserted in between each given string. The result is returned as a new string. | >>> s1 = 'abc'  >>> s2 = '123'  >>> print(s1.join(s2)) 1abc2abc3  >>>s1 = '-'  >>>s2 = 'abc'  >>>print(s1.join(s2)) a-b-c |

* Other string methods are:

|  |  |
| --- | --- |
| **Method** | **Description** |
| lower() | returns a string with every letter of the original in lowercase |
| upper() | returns a string with every letter of the original in uppercase |
| replace(x,y) | returns a string with every occurrence of x replaced by y |
| count(x) | counts the number of occurrences of x in the string |
| index(x) | returns the location of the ﬁrst occurrence of x |
| isalpha() | returns True if every character of the string is a letter |

## Formatting Strings

Strings can be formatted in different ways, using :

* + format operator
  + f-string
  + format( ) function

**Format operator**

* The *format operator*, % allows us to construct strings, replacing parts of the strings with the data stored in variables.
* When applied to integers, % is the modulus operator. But when the first operand is a string,

% is the format operator.

* For example, the format sequence “%d” means that the operand should be formatted as an

integer (d stands for “decimal”):

**Example 1:**

>>> camels = 42

>>>'%d' % camels

*'42'*

ROOPA.H.M, Dept of MCA, RNSIT Page 9

* A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

**Example 2 :**

>>> camels = 42

>>> 'I have spotted %d camels.' % camels 'I have spotted 42 camels.'

* If there is more than one format sequence in the string, the second argument has to be a

tuple. Each format sequence is matched with an element of the tuple, in order.

* The following example uses “%d” to format an integer, “%g” to format a floating point number

, and “%s” to format a string:

**Example 3:**

>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels') 'In 3 years I have spotted 0.1 camels.'

## f-strings

Formatted strings or f-strings were introduced in Python 3.6. A f-string is a string literal that is prefixed with “f”. These strings may contain replacement fields, which are expressions enclosed within curly braces {}. The expressions are replaced with their values.

Example : >>>a=10

>>>print(f”the value is {a}”) the value is 10

## Format function

**format()** : is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.

Two types of Parameters:

* positional\_argument
* keyword\_argument
* Positional argument: It can be integers, floating point numeric constants, strings, characters and even variables.
* Keyword argument : They is essentially a variable storing some value, which is passed as parameter.

ROOPA.H.M, Dept of MCA, RNSIT Page 10

*# To demonstrate the use of formatters with positional key arguments.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Positional arguments are placed in order | | >>>print("{0} college{1} ".format("RNSIT","EC")) |  | department |
| RNSIT college EC department | | |
| Reverse the | index | >>>print("{1} department | {0} | college |
| numbers with | the | ".format("RNSIT","EC”)) |  |  |
| parameters of | the |  |  |  |
| EC department RNSIT college | | |
| placeholders |  |  | | |
| Positional arguments | | >>>print("Every {} should know the use of {} {} | | |
| are not specified. By | | python programming and {}".format("programmer", | | |
| default it starts | | "Open", "Source", "Operating Systems")) | | |
| positioning from zero | | Every programmer should know the | use of | Open Source |
|  | | programming and Operating Systems |  |  |
| Use the index numbers | | >>>print("Every {3} should know the use of {2} {1} | | |
| of the values to change | | programming and {0}" .format("programmer", "Open", | | |
| the order that they | | "Source", "Operating Systems")) | | |
| appear in the string | | Every Operating Systems should know the use of Source | | |
|  | | Open programming and programmer | | |
| Keyword arguments are | | print("EC department {0} ‘D’ | section | {college}" |
| called by their keyword | | .format("6", college="RNSIT")) |  |  |
| Name | | EC department 6 ‘D’ section RNSIT | | |

## Lists

**A list is a sequence**

### A list is an ordered sequence of values.

* It is a data structure in Python. The values inside the lists can be of any type (like integer, float, strings, lists, tuples, dictionaries etc) and are called as elements or items.

### The elements of lists are enclosed within square brackets.

ROOPA.H.M, Dept of MCA, RNSIT Page 11

## Creating Lists

### There are various ways of creating list:

* + Creating a simple list:

**L = [1,2,3]**

### Use square brackets to indicate the start and end of the list, and separate the items

by commas.

### Empty list is equivalent of 0 or ' '. The empty list [ ] can be created using list function or empty square brackets.

**a = [ ]**

**l=list()**

* + Long lists If you have a long list to enter, you can split it across several lines, like

### below:

**nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,**

**18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,]**

* + We can use ***eval(input( ))*** to allow the user to enter a list. Here is an example:

|  |  |
| --- | --- |
| L = eval(input('Enter a list: ')) print('The first element is ', L[0]) | **Output:**  Enter a list: [5,7,9] The first element is 5 |

* + Creating list using *list* and *range* function

|  |  |
| --- | --- |
| num=list(range(10))  print(num) | **Output:**  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] |

* + - Nested lists can be created as follows:

|  |  |
| --- | --- |
| a = ['a', 'b', 'c'] n = [1, 2, 3]  x = [a, n] print(x) | Output:  [['a', 'b', 'c'], [1, 2,  3]] |
| col=[23,[9.3,11.2,],[‘good’], []]  print(col) | [23,[9.3,11.2],['good'],[]] |

## Basic List Operations

### The plus (+) operator concatenates lists in the same way it concatenates strings. The following shows some experiments in the interactive shell with list concatenation:

ROOPA.H.M, Dept of MCA, RNSIT Page 12

|  |  |
| --- | --- |
| a=[10, 20, 30]  a = a + [1, 3, 5]  print(a) a += [10]  print(a) | Output:  [10, 20, 30, 1, 3, 5]  [10, 20, 30, 1, 3, 5, 10] |

### The statement a = a + [1, 3, 5] reassigns a to the new list [10, 20, 30, 1, 3, 5].

* + - * + The statement a += [10] updates a to be the new list [10, 20, 30, 1, 3, 5, 10].
        + Similarly, the \* operator repeats a list a given number of times:

|  |  |
| --- | --- |
| lt=[0]\*20 print(lt) | Output:  [0, 0, 0, 0] |
| print([1, 2, 3] \* 3) | [1, 2, 3, 1, 2, 3, 1, 2, 3] |
| print(["abc"]\*5) | ['abc','abc','abc','abc','abc'] |

## List Aliasing

### When an object is assigned to other using assignment operator, both of them will refer to same object in the memory. The association of a variable with an object is called as *reference*.

|  |
| --- |
| a = [10, 20, 30, 40]  b = [10, 20, 30, 40]  print('Is ', a, ' equal to ', b, '?', sep='', end=' ') print(**a == b**)  print('Are ', a, ' and ', b, ' aliases?', sep='', end=' ') print(**a is b**)  c = [100, 200, 300, 400]  d = c # *creating alias*  print('Is ', c, ' equal to ', d, '?', sep='', end=' ') print(**c == d**)  print('Are ', c, ' and ', d, ' aliases?', sep='', end=' ')  print(**c is d**) |
| **Output:**  Is [10, 20, 30, 40] equal to [10, 20, 30, 40]? True  Are [10, 20, 30, 40] and [10, 20, 30, 40] aliases? False  Is [100, 200, 300, 400] equal to [100, 200, 300, 400]? True  Are [100, 200, 300, 400] and [100, 200, 300, 400] aliases? True |

ROOPA.H.M, Dept of MCA, RNSIT Page 13

|  |  |  |  |
| --- | --- | --- | --- |
| * The assignment statement (c=d) causes variables c and d to refer to the same list object. We say that c and d are aliases. In other words, there are two references to the same object in the memory. * An object with more than one reference has more than one name, hence we say that object is aliased. If the aliased object is mutable, changes made in one alias will reflect the other.   **Indexing and Slicing in Lists Indexing**   * The syntax for accessing the elements of a list is the same as for accessing the characters of a string: the bracket operator. The expression inside the brackets speciﬁes the index. Indexing starts from 0. * *heterolist.py* demonstrates that lists may be heterogeneous; that is, a list can hold elements of varying types. Accessing elements of the list with their index.   collection=[24.2,4,'word',eval,19,-0.03,'end'] 24.2  4  print(collection[0]) word  print(collection[1]) <built-in function eval>  print(collection[2]) 19  print(collection[3]) -0.03  print(collection[4]) end  [24.2,4,'word',  print(collection[5]) <built-in function eval>,  print(collection[6]) 19, -0.03, 'end']  print(collection)   * Accessing elements with negative index   names=["roopa","shwetha","rajani"] Output: print(names[-2]) shwetha  print(names[-1]) rajani   * Accessing the elements within inner list can be done by double-indexing. The inner list is treated as a single element by outer list. the first index indicates position of inner list inside outer list, and the second index means the position of particular value within inner list. | | | |
|  | ls=[[1,2],['EC','CS']]  print(ls[1][0]) | Output:  EC |  |
|  |  | Page 14 |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

**List slicing**

### We can make a new list from a portion of an existing list using a technique known as slicing. A list slice is an expression of the form

**list [ begin : end : step ]**

* + - * If the begin value is missing, it defaults to 0.

### If the end value is missing, it defaults to the length of the list.

* + - * The default step value is 1.

|  |  |
| --- | --- |
| lst =[10,20,30,40,50,60,70,80,90,100]  print(lst) | Output: [10,20,30,40,50,60,70,80,90,100] |
| print(lst[**0:3**]) | [10, 20, 30] |
| print(lst[**4:8**]) | [50, 60, 70, 80] |
| print(lst[**2:5**]) | [30, 40, 50] |
| print(lst[**-5:-3**]) | [60, 70] |
| print(lst[**:3**]) | [10, 20, 30] |
| print(lst[**4:**]) | [50, 60, 70, 80, 90, 100] |
| print(lst[**:**]) | [10,20,30,40,50,60,70,80,90,100] |
| print(lst[**-100:3**]) | [10, 20, 30] |
| print(lst[**4:100**]) | [50, 60, 70, 80, 90, 100] |
| print(lst[**2:-2:2**]) | [30, 50, 70] |
| print(lst[**::2**]) | [10, 30, 50, 70, 90] |

### A *begin* value less than zero is treated as zero.

Ex: lst[**-100:3**] *#here -100 is treated as 0*

* + - An *end* value greater than the length of the list is treated as the length of the list. Ex: lst[**4:100**] *# here 100 is treated as len(lst)*

ROOPA.H.M, Dept of MCA, RNSIT Page 15

* + - A slice operator on the left side of an assignment can update multiple elements:

|  |  |
| --- | --- |
| t = ['a','b','c','d','e','f'] t[1:3] = ['x', 'y']  print(t) | Output:  ['a', 'x', 'y', 'd', 'e', 'f'] |

## Built-In Functions Used on Lists

### There are several built-in functions that operate on lists. Here are some useful ones:

|  |  |
| --- | --- |
| **Function** | **Description** |
| **len** | returns the number of items in the list |
| **sum** | returns the sum of the items in the list |
| **min** | returns the minimum of the items in the list |
| **max** | returns the maximum of the items in the list |

Example demonstrating the use of above functions

|  |  |
| --- | --- |
| nums = [3, 41, 12, 9, 74,  15]  print("length:",len(nums)) print(max(nums)) print(min(nums)) print(sum(nums))  print(sum(nums)/len(nums)) | Output: length: 6  74  3  154  25.666666666666668 |

The sum( ) function only works when the list elements are numbers. The other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

## Ex: Program to read the data from the user and to compute sum and average of those numbers

### In this program, we initially create an empty list. Then, we are taking an infinite while loop. As every input from the keyboard will be in the form of a string, we need to convert x into float type and then append it to a list. When the keyboard input is a string ‘done’, then the loop is going to get terminated. After the loop, we will find the

ROOPA.H.M, Dept of MCA, RNSIT Page 16

average of those numbers with the help of built-in functions sum() and len().

|  |  |
| --- | --- |
| ls= list() while (True):  x= input('Enter a number: ') if x== 'done':  break x= float(x) ls.append(x)  average = sum(ls) / len(ls) print('Average:', average) | Output:  Enter a number: 2 Enter a number: 3 Enter a number: 4 Enter a number: 5 Enter a number: done Average: 3.5 |

## List Methods

### There are several built-in methods in list class for various purposes. They are as follows:

|  |  |
| --- | --- |
| **append:** The append() method adds a single item to the existing list. It doesn't return a new list; rather it modifies the original list. | |
| fruits = ['apple', 'banana', 'cherry'] fruits.append("orange")  print("fruits after appending:",fruits) | fruits after appending: ['apple', 'banana', 'cherry', 'orange'] |
| **extend:** the extend() method takes a single argument (a list) and adds it to the end. | |
| t1 = ['a', 'b', 'c'] t2 = ['d', 'e']  t1.extend(t2)  print("t1+t2 :",t1) | t1+t2 : ['a', 'b', 'c', 'd', 'e'] |
| **count:** Returns the number of times a given element appears in the list. Does not modify the list | |
| ls=[1,2,5,2,1,3,2,10]  print("count of 2 :" ,ls.count(2)) | count of 2 : 3 |
| **insert :** Inserts a new element before the element at a given index. Increases the length of the  list by one. Modiﬁes the list. | |
| ls=[3,5,10]  ls.insert(1,"hi")  print("ls after inserting element:",ls) | ls after inserting element: [3,'hi,5,10] |
| **index :** Returns the lowest index of a given element within the list. Produces an error if the element does not appear in the list. Does not modify the list. | |

ROOPA.H.M, Dept of MCA, RNSIT Page 17

|  |  |
| --- | --- |
| ls=[15, 4, 2, 10, 5, 3, 2, 6]  print(ls.index(2)) print(ls.index(2,3,7))  #*finds the index of 2 between the position 3 to 7* | 2  6 |
| **reverse:** Physically reverses the elements in the list. The list is modiﬁed. | |
| ls=[4,3,1,6]  ls.reverse() print("reversed list :" ,ls) | reversed list : [6, 1, 3, 4] |
| **sort:** Sorts the elements of the list in ascending order. The list is modiﬁed. | |
| ls=[3,10,5, 16,-2]  ls.sort()  print("list in ascending order: ",ls)  ls.sort(reverse=True)  print("list in descending order: ",ls) | ls in ascending order:[-2,3,5,10,16]  ls in descending order:[16,10,5,3,-2] |
| **clear:** This method removes all the elements in the list and makes the list empty | |
| ls=[1,2,3]  ls.clear() print(ls) | [] |

**Deleting elements**

### There are several ways to delete elements from a list. Python provides few built-in methods for removing elements as follows:

* + **pop()** : This method deletes the last element in the list, by default or removes the item at specified index p and returns its value. Index is passed as an argument to pop(). pop modiﬁes the list and returns the element that was removed.

|  |  |
| --- | --- |
| my\_list = ['p','r','o','b','l','e','m']  print('poped element is:',my\_list.pop()) print(my\_list)  print('poped element is:',my\_list.pop(1)) print(my\_list) | Output:  poped element is: **m**  ['p','r','o','b','l','e']  poped element is: **r**  ['p', 'o', 'b', 'l', 'e'] |

### **remove():** This method can be used, if the index of the element is not known, then

ROOPA.H.M, Dept of MCA, RNSIT Page 18

### the value to be removed can be specified as argument to this function.

|  |  |
| --- | --- |
| my\_list = ['p','r','o','p','e','r'] my\_list.remove('r')  print(my\_list) | Output:  ['p', 'o', 'p', 'e', 'r'] |

— Note that, this function will remove only the first occurrence of the specified value, but not all occurrences.

### — Unlike *pop()* function, the *remove()* function will not return the value that has been deleted.

* + **del:** This is an operator to be used when more than one item to be deleted at a time. Here also, we will not get the items deleted.

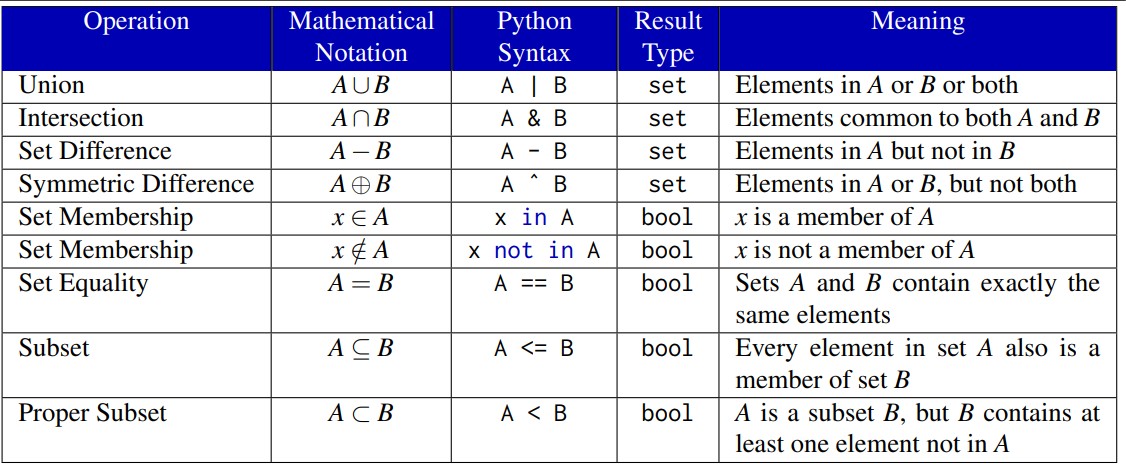
|  |  |
| --- | --- |
| my\_list = ['p','r','o','b','l','e','m'] del my\_list[2]  *#deletes the element at position 2*  print(my\_list) | ['p', 'r', 'b', 'l', 'e', 'm'] |
| del my\_list[1:5]  *#deletes the elements between position 1 and 5*  print(my\_list) | ['p', 'm'] |
| del my\_list  *#deletes the entire list*  print(my\_list) | NameError:  name 'my\_list' is not defined |
| *#Deleting all odd indexed elements of a list*  t=['a', 'b', 'c', 'd', 'e']  del t[1::2] print(t) | ['a', 'c', 'e'] |

**Sets**

* Python provides a data structure that represents a mathematical set. As with mathematical sets, we use curly braces { } in Python code to enclose the elements of a literal set.
* Python distinguishes between set literals and dictionary literals by the fact that all the items in a dictionary are colon-connected (:) key-value pairs, while the elements in a set are simply values.
* Unlike Python lists, sets are unordered and may contain no duplicate elements. The following

ROOPA.H.M, Dept of MCA, RNSIT Page

|  |  |
| --- | --- |
|  | |
| interactive sequence demonstrates these set properties: | |
| >>> S = {10, 3, 7, 2, 11} | |
| >>> S {2, 11, 3, 10, 7} | |
| >>> T = {5, 4, 5, 2, 4, 9} | |
| >>> T {9, 2, 4, 5} | |
| * We can make a set out of a list using the set conversion function: | |
| >>> L = [10, 13, 10, 5, 6, 13, 2, 10, 5] | |
| >>> S = set(L) | |
| >>> S | |
| {10, 2, 13, 5, 6} | |
| As we can see, the element ordering is not preserved, and duplicate elements appear only | |
| once in the set. | |
| * Python set notation exhibits one important difference with mathematics: the expression { } | |
| does not represent the empty set. In order to use the curly braces for a set, the set must | |
| contain at least one element. The expression set( ) produces a set with no elements, and | |
| thus represents the empty set. | |
| * Python reserves the { } notation for empty dictionaries. Unlike in mathematics, all sets in | |
| python must be finite. Python supports the standard mathematical set operations of | |
| intersection, union, set difference, and symmetric difference. Table below shows the python | |
| syntax for these operations. | |
| **Example :** The following interactive sequence computes the union and intersection and two | |
| sets and tests for set membership: | |
| ROOPA.H.M, Dept of MCA, RNSIT | Page 20 |



## Tuples

* + Tuples are one of Python's simplest and most common collection types. A tuple is a sequence of values much like a list.
  + The values stored in a tuple can be any type, and they are indexed by integers.
  + A tuple is an immutable list of values.

## Tuple creation

There are many ways to a tuple. They are as follows

|  |  |  |
| --- | --- | --- |
| **Creation** | **Output** | **Comments** |
| **t ='a','b','c','d','e'**  **print(t)** | **('a','b','c','d','e')** | Syntactically, a tuple is a comma-separated list of  values. |
| **t=('a','b','c','d','e')**  **print(t)** | **('a','b','c','d','e')** | Although not necessary, it is common to enclose  tuples in parentheses |
| **t1=tuple() print(t1)**  **t2=()**  **print(t2)** | **()**  **()** | Creating an empty tuple |
| **t=tuple('lupins') print(t)**  **t = tuple(range(3)) print(t)** | **('l','u','p','i','n','s')**  **(0, 1, 2)** | Passing arguments to tuple function |
| **t1 = ('a',)**  **print(type(t1))**  **t2 = ('a')**  **print(type(t2))** | **<class 'tuple’>**  **<class 'str’>** | To create a tuple with a single element, you have to include the ﬁnal comma. |
| **t = (1, "Hello", 3.4)**  **print(t)** | **(1, "Hello", 3.4)** | tuple with mixed datatypes |
| **t=("mouse",[8,4,6],(1,2,3))**  **print(my\_tuple)** | **("mouse",[8,4,6],(1,2,3))** | Creating nested tuple |

ROOPA.H.M, Dept of MCA, RNSIT Page 21

## Accessing Elements in a Tuple

index operator [ ] to access an item in a tuple where the index starts from 0

|  |  |
| --- | --- |
| College =('r','n','s','i','t') print(College[0]) print(College[4])  #*accessing elements of nested tuple* n\_tuple =("program", [8, 4, 6],(1, 2, 3)) print(n\_tuple[0][3])  print(n\_tuple[1][1])  *#tuple slicing is possible*  t = ('a', 'b', 'c', 'd', 'e’)  print(t[1:3]) | Output:  **r t g 4**  **('b', 'c’)** |

In the above example, *n\_tuple* is a tuple with mixed elements i.e., string, list , or tuple

## Tuples are immutable

* One of the main diﬀerences between lists and tuples in Python is that tuples are immutable, that is, one cannot add or modify items once the tuple is initialized.

For example:

**>>> t = (1, 4, 9)**

**>>> t[0] = 2**

Traceback (most recent call last): File "<stdin>", line 1, in <module>

TypeError: **'tuple' object does not support item assignment**

* Similarly, tuples don't have ***.****append* and ***.****extend* methods as list does. Using **+=** is possible, but it changes the binding of the variable, and not the tuple itself:

**>>> t = (1, 2)**

**>>> q = t**

**>>> t += (3, 4)**

**>>>t**

**(1, 2, 3,4)**

**>>>**

**q (1, 2)**

ROOPA.H.M, Dept of MCA, RNSIT Page 22

* + Be careful when placing mutable objects, such as lists, inside tuples. This may lead to very confusing outcomes when changing them. For example:

>>> t = (1, 2, 3, [1, 2, 3]) (1, 2, 3, [1, 2, 3])

*# the below statement both raise an error and change the contents of the list within the tuple:*

>>> t[3] += [4, 5]

**TypeError: 'tuple' object does not support item assignment**

>>> t

(1, 2, 3, [1, 2, 3, 4, 5])

* You can use the += operator to "append" to a tuple - this works by creating a new tuple with the new element you "appended" and assign it to its current variable; the old tuple is not changed, but replaced.

>>>t = ('A',)+ t[1:]

>>> print(t)

('A', 'b', 'c', 'd', 'e')

## Comparing tuples

### Tuples can be compared using operators like >, <, >=, == etc.

* + Python starts by comparing the ﬁrst element from each sequence. If they are equal, it goes on to the next element, and so on, until it ﬁnds elements that diﬀer. Subsequent elements are not considered (even if they are really big).

|  |  |
| --- | --- |
| **Example** | **Description** |
| >>>(0, 1, 2) < (0, 3, 4)  True | Step1:0==0 Step2:1 < 3 true  *Comparison stops at this stage*  Step3:2 < 4 ignored |
| >>>(0, 1, 2000000) < (0, 3, 4)  True | Step1:0==0 Step2:1 < 3 true  *Comparison stops at this stage*  Step3:2000000 < 4 ignored |

* + When we use relational operator on tuples containing non-comparable types, then TypeError

ROOPA.H.M, Dept of MCA, RNSIT Page 2

|  |  |  |
| --- | --- | --- |
| will be thrown.  >>>(1,'hi')<('hello','world')  TypeError: '<' not supported between instances of 'int' and 'str'   * The sort function works the same way. It sorts primarily by ﬁrst element, but in the case of a   tie, it sorts by second element, and so on. This pattern is known as **DSU**  —**Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,  —**Sort** the list of tuples using the Python built-in sort, and  —**Undecorate** by extracting the sorted elements of the sequence  Consider a program of sorting words in a sentence from longest to shortest, which illustrates DSU property. | | |
|  | txt = 'this is an example for sorting tuple' words = txt.split()  t = list()  for word in words: t.append((len(word), word))  print(t) *#displays unsorted list*  t.sort(reverse=True)  print(t) *#displays sorted list*  res = list()  for length, word in t: res.append(word)  print(res) #displays sorted list with only words but not length | |
|  | Output: [(4,'this'),(2,'is'),(2,'an'),(7,'example'),(3,'for'),(7,'sorting'),(5,'tuple')]  [(7,'sorting'),(7,'example'),(5,'tuple'),(4,'this'),(3,'for'),(2,'is'),(2,'an')]  ['sorting', 'example', 'tuple', 'this', 'for', 'is', 'an'] | |
| In the above program,   * The ﬁrst loop builds a list of tuples, where each tuple is a word preceded by its length. * sort compares the ﬁrst element, length, ﬁrst, and only considers the second element to break | | |
|  | ROOPA.H.M, Dept of MCA, RNSIT | Page 24 |

ties. The keyword argument reverse=True tells sort to go in decreasing order.

— The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in reverse alphabetical order, so “sorting”

appears before “example” in the list.

**Tuple assignment** (***Packing and Unpacking Tuples***)

* + One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows to assign more than one variable at a time when the left side is a sequence.

### Ex:

>>>x, y, z = 1, 2, 3

>>> print(x) #prints 1

>>> print(y) #prints 2

>>> print(y) #prints 3

* + when we use a tuple on the left side of the assignment statement, we omit the parentheses,

but the following is an equally valid syntax:

>>> m = [ 'have', 'fun' ]

>>> (x, y) = m

>>> x

>>> y

#prints

'have'

#prints 'fun'

* + A particularly clever application of tuple assignment allows us to swap the values of two variables in a single statement.

>>> a=10

>>> b=20

>>> a, b = b, a

>>> print(a, b)

#prints 20 10

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

* + The number of variables on the left and the number of values on the right must be the same:

>>> a, b = 1, 2, 3

**ValueError: too many values to unpack**

ROOPA.H.M, Dept of MCA, RNSIT Page 25

* + The symbol \_ can be used as a disposable variable name if we want only few elements of a tuple, acting as a placeholder:

>>>a = 1, 2, 3, 4

>>>**\_**, x, y, **\_** = a

>>>print(x)

>>>print(y)

#prints 2

#prints 3

* + Sometimes we may be interested in using only few values in the tuple. This can be achieved by using *variable-length argument tuples* (variable with a \*preﬁx) can be used as a catch-all variable, which holds multiple values of the tuple.

>>>first, **\*more**, last = (1, 2, 3, 4, 5)

>>>print(first) #prints 1

>>>print(more) #prints [2,3,4]

>>>print(last) #prints 5

* More generally, the right side can be any kind of sequence (string, list, or tuple). For example, to split an email address into a user name and a domain. Code is as follows:

>>>addr = 'monty@python.org'

>>> **uname, domain = addr.split('@')**

>>> print(uname) #prints monty

>>> print(domain) #prints python.org

## Dictionaries and tuples

* Dictionaries have a method called ***items*** that returns a list of tuples, where each tuple is a

*key-value* pair:

>>> d = {'a':10, 'b':1, 'c':22}

>>> t = list(**d.items()**)

>>> print(t)

[('b', 1), ('a', 10), ('c', 22)]

* As dictionary may not display the contents in an order, we can use sort() on lists and then print in required order.
* However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key.

ROOPA.H.M, Dept of MCA, RNSIT Page 26

## Dictionaries

* Dictionary is a set of key: value pairs, with the requirement that the keys are unique (within one dictionary).
* Dictionary is a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a *key-value* pair.
* Unlike other sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type such as strings, numbers, tuples(if they contain only strings, numbers, or tuples).
* Dictionaries are mutable, that is, they are modifiable.
* A pair of braces creates an empty dictionary: { }.

## d = { }

* The function dict creates a new dictionary with no items.

## empty\_d = dict()

* Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary.

Ex: a dictionary that maps from English to Spanish words

|  |  |
| --- | --- |
| eng2sp={} eng2sp[**'one'**] = 'uno' print(eng2sp) eng2sp[**'two'**] = 'dos' print(eng2sp) eng2sp[**'three'**]= 'tres'  print(eng2sp) | Output:  {'one': 'uno'}  {'one': 'uno', 'two': 'dos'}  {'one':'uno','two':'dos','three':'tres'} |

ROOPA.H.M, Dept of MCA, RNSIT Page 27

* + A dictionary can be initialized at the time of creation itself.

|  |  |
| --- | --- |
| eng2sp ={'one':'uno','two':'dos', 'three': 'tres'}  print(eng2sp) | **Output:**  {'one':'uno','three':'tres', 'two': 'dos'} |

Notice the output, the order of items in a dictionary is not same as its creation. As dictionary members are not indexed over integers, the order of elements inside it may vary.

## Accessing elements of dictionary

* To access an element with in a dictionary, we use square brackets exactly as we would with a list. In a dictionary every key has an associated value.

## >>> print(eng2sp['two']) 'dos'

The key 'two' always maps to the value “dos” so the order of the items doesn’t matter.

* If the key isn’t in the dictionary, you get an exception:

## >>> print(eng2sp['four']) KeyError: 'four'

**Length of the dictionary**

The *len* function works on dictionaries; it returns the number of *key-value* pairs:

## >>> num\_word = {1: 'one', 2: 'two', 3:'three'}

**>>>len(num\_word) 3**

## in operator with dictionaries

The *in* operator works on dictionaries to check whether something appears as a key in the dictionary (but not the value).

>>>eng2sp ={'one':'uno','two':'dos','three':'tres'}

>>> 'one' in eng2sp True

>>> 'uno' in eng2sp

False

#searching key

#searching value

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

ROOPA.H.M, Dept of MCA, RNSIT Page 28

The in operator uses different algorithms for lists and dictionaries.

— For *lists*, it searches the elements of the list in order i.e., linear search. As the list gets longer, the search time gets longer in direct proportion.

— For *dictionaries*, Python uses an algorithm called a hash table that has a remarkable property: the in operator takes about the same amount of time no matter how many items are in the dictionary

## Dictionary as a set of counters

### Assume that we need to count the frequency of alphabets in a given string. There are different ways to do it .

* Create 26 variables to represent each alphabet. Traverse the given string and increment the corresponding counter when an alphabet is found.
* Create a list with 26 elements (all are zero in the beginning) representing alphabets. Traverse the given string and increment corresponding indexed position in the list when an alphabet is found.
* Create a dictionary with characters as keys and counters as values. When we find a

character for the first time, we add the item to dictionary. Next time onwards, we increment the value of existing item. Each of the above methods will perform same task, but the logic of implementation will be different. Here, we will see the implementation using dictionary.

|  |
| --- |
| word = 'vishveshwarayya' # given string  d = dict() #creates empty dictionary  for c in word: #extracts each character in the string if c not in d: #if character is not found  d[c] = 1 #initialize counter to 1 else:  d[c] = d[c] + 1 #otherwise, increment counter  print(d) |
| Output:  {'v':2, 'i':1, 's':2, 'h':2, 'e':1, 'w':1, 'a':3, 'r':1, 'y':2} |

ROOPA.H.M, Dept of MCA, RNSIT Page 29

It can be observed from the output that, a dictionary is created here with characters as keys and frequencies as values. Note that, here we have computed *histogram* of counters.

* Dictionary in Python has a method called get(), which takes key and a default value as two arguments. If key is found in the dictionary, then the get() function returns corresponding value, otherwise it returns default value. For example,

|  |  |
| --- | --- |
| month = {'jan':1, 'march':3, 'june':6} print(**month.get('jan', 0)**)  print(**month.get('october', 'not found')**) | Output: 1  not found |

In the above example, when the get() function is taking '*jan*' as argument, it returned corresponding value, as *1* is found in *month* directory . Whereas, when get() is used with '*october*' as key, the default value ‘*not found*’ (passed as second argument) is returned.

* The function get() can be used effectively for calculating frequency of alphabets in a string. Here is the modified version of the program –

|  |
| --- |
| word = 'vishveshwarayya' d = dict()  for c in word:  d[c] = **d.get(c,0)** + 1 print(d) |
| Output:  {'v':2, 'i':1, 's':2, 'h':2, 'e':1, 'w':1, 'a':3, 'r':1, 'y':2} |

In the above program, for every character *c* in a given string, we will try to retrieve a value. When the *c* is found in d, its value is retrieved, 1 is added to it, and restored. If *c* is not found, 0 is taken as default and then 1 is added to it.

#### Looping and dictionaries

* + When a for-loop is applied on dictionaries, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:

|  |  |
| --- | --- |
| names ={'chuck':1, 'annie':42, 'jan':100} for key in names:  print(**key, names[key]**) | Output: chuck 1  annie 42  jan 100 |

* + If we wanted to ﬁnd all the entries in a dictionary with a value above ten, we could write the following code:

ROOPA.H.M, Dept of MCA, RNSIT Page 30

|  |  |
| --- | --- |
| names ={'chuck':1, 'annie':42, 'jan':100} for key in names:  if **names[key] > 10** :  print(key, names[key]) | Output: annie 42  jan 100 |

The for loop iterates through the *keys* of the dictionary, so we must use the index operator to retrieve the corresponding *value* for each *key* that are greater than 10.

* Sometimes we may want to access key-value pair together from the dictionary, it can be done by using **items()** method as follows:

|  |  |
| --- | --- |
| names ={'chuck':1,'annie':42,'jan':100} for k, v in **names.items()**:  print(k, v) | Output: chuck 1  annie 42  jan 100 |

## Sorting the dictionary elements

If we want to print the keys in alphabetical order, ﬁrst make a list of the keys in the dictionary using the *keys method* available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out *key-value* pairs in sorted order as follows:

|  |  |
| --- | --- |
| names ={'chuck':1, 'annie':42, 'jan':100} lst = **list(names.keys())**  print(lst)  **lst.sort()**  print("Elements in alphabetical order:") for key in lst:  print(key, names[key]) | Output:  ['chuck', 'annie', 'jan'] Elements in alphabetical order: annie 42  chuck 1  jan 100 |

## FILES

* + File is a named location on disk to store related information.
  + Data for python program can come from difference sources such as keyboard, text file, web server, database.
  + Files are one such sources which can be given as input to the python program. Hence handling files in right manner is very important.
  + Primarily there are two types of files
    - *Text ﬁles :* A text ﬁle can be thought of as a sequence of lines without any images, tables etc. These files can be create by using some text editors.
    - *Binary ﬁles* : These files are capable of storing text, image, video, audio, database ﬁles, etc which contains the data in the form of bits.

ROOPA.H.M, Dept of MCA, RNSIT Page 31

**Opening ﬁles**

* + To perform read or write operation on a ﬁle, ﬁrst ﬁle must be opened.
  + Opening the ﬁle communicates with the operating system, which knows where the data for

each ﬁle is stored.

* + A file can be opened using a built-in function open( ).
  + The syntax of open( ) function is as below :

**fhand= open(“filename”, “mode”)**

Here,

*filename* -> is name of the file to be opened. This string may be just a name of the file, or it may include pathname also. Pathname of the file is optional when the file is stored in current working directory.

*mode* -> This string specifies an access mode to use the file i.e., for reading, writing, appending etc.

*fhand* -> It is a reference to a file object, which acts as a handler for all further operations on files.

* + If mode is not specified , by default, open( ) uses mode ‘r’ for reading.

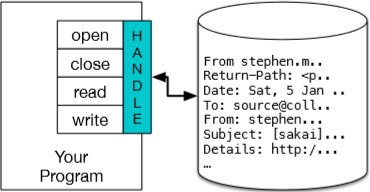
>>> fhand = open('sample.txt')

>>>print(fhand)

<\_io.TextIOWrapper name='sample.txt' mode='r' encoding='cp1252'>

Note: In this example, we assume the ﬁle ‘sample.txt‘ stored in the same folder that you are in when you start Python. Otherwise path of the file has to be passed.

fhand = open('c:/users/roopa/desktop/sample.txt')

* + If the open is successful, the operating system returns a ﬁle handle.
  + The ﬁle handle is not the actual data contained in the ﬁle, but instead it is a “handle” that we can use to read the data. You are given a handle if the requested ﬁle exists and you have the proper permissions to read the ﬁle.

ROOPA.H.M, Dept of MCA, RNSIT Page 32

* + If the ﬁle does not exist, open will fail with a traceback and you will not get a handle to access

the contents of the ﬁle:

>>> fhand = open('fibo.txt')

Traceback (most recent call last): File "<stdin>", line 1, in <module> FileNotFoundError: [Errno 2] No such file or directory: 'fibo.txt'

* + List of modes in which files can be opened are given below :

|  |  |
| --- | --- |
| Mode | Meaning |
| **r** | Opens a file for reading purpose. If the specified file does not exist in the  specified path, or if you don‟t have permission, error message will be displayed. This is the default mode of open() function in Python. |
| **w** | Opens a file for writing purpose. If the file does not exist, then a new file with the given name will be created and opened for writing. If the file  already exists, then its content will be over-written. |
| **a** | Opens a file for appending the data. If the file exists, the new content will  be appended at the end of existing content. If no such file exists, it will be created and new content will be written into it. |
| **r+** | Opens a file for reading and writing. |
| **w+** | Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and  writing. |
| **a+** | Opens a file for both appending and reading. The file pointer is at the end  of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and |
| **rb** | Opens a file for reading only in binary format |
| **wb** | Opens a file for writing only in binary format |
| **ab** | Opens a file for appending only in binary format |

## Reading files

* Once the specified file is opened successfully. The open( ) function provides handle which is a refers to the file.
* There are several ways to read the contents of the file

##### using the ﬁle handle as the sequence in for loop.

|  |
| --- |
| **sample.txt** |
| Python is a high level programming language it is introduced by Guido van rossam.  Python is easy to learn and simple to code an application |

ROOPA.H.M, Dept of MCA, RNSIT Page 33

|  |
| --- |
| **Print\_count.py** |
| fhand = open('sample.txt') count = 0  for line in fhand: count = count + 1  print('Line :', count,line) print('Line Count:', count) fhand.close() |
| **Output:** |
| Line : 1 Python is a high level programming language Line : 2 it is introduced by Guido van rossam.  Line : 3 Python is easy to learn and simple to code an application  Line Count: 3 |

* + In the above example, ‘for’ loop simply counts the number of lines in the ﬁle and prints them out.
  + When the ﬁle is read using a for loop in this manner, Python takes care of splitting the data in the ﬁle into separate lines using the newline character. Python reads each line through the newline and includes the newline as the last character in the line variable for each iteration of the for loop.
  + Notice the above output, there is a gap of two lines between each of the output lines. This is because, the new-line character \n is also a part of the variable *line* in the loop, and the print() function has default behavior of adding a line at the end. To avoid this double-line spacing, we can remove the new-line character attached at the end of variable *line* by using built-in string function rstrip() as below –

## print("Line: ",count, line.rstrip())

* + Because the for loop reads the data one line at a time, it can eﬃciently read and count the lines in very large ﬁles without running out of main memory to store the data. The above program can count the lines in any size ﬁle using very little memory since each line is read, counted, and then discarded.

##### The second way of reading a text ﬁle loads the entire ﬁle into a string :

* + If you know the ﬁle is relatively small compared to the size of your main memory, you can read the whole ﬁle into one string using the read method on the ﬁle handle.

ROOPA.H.M, Dept of MCA, RNSIT Page 34

* + In this example, the entire contents (all 140 characters) of the ﬁle sample.txt are read directly into the variable *content*. We use string slicing to print out the ﬁrst 7 characters of the string data stored in variable *content*.
  + When the ﬁle is read in this manner, all the characters including all of the lines and newline characters are one big string in the variable *content*. It is a good idea to store the output of read as a variable because each call to read exhausts the resource.

#### Letting the user choose the ﬁle name

* Editing Python code every time to process a diﬀerent ﬁle is tedious work. It would be more usable to ask the user to enter the ﬁle name string each time the program runs so they can use the program on diﬀerent ﬁles without changing the Python code.
* This is quite simple to do by reading the ﬁle name from the user using input as follows:

|  |
| --- |
| fname = input('Enter the file name: ') fhand = open(fname)  count = 0  for line in fhand:  if line.startswith('ro'): count = count + 1  print('There were', count, ‘lines that starts with ro in', fname)  fhand.close() |
| **Output :**  Enter the file name: name.txt  There were 3 lines that starts with ro in name.txt |

* We read the ﬁle name from the user and place it in a variable named fname and open that

ﬁle. Now we can run the program repeatedly on diﬀerent ﬁles.

ROOPA.H.M, Dept of MCA, RNSIT Page 35

## Writing files

* To write a ﬁle, you have to open it with mode “w” as a second parameter:

>>> fout = open('output.txt', 'w')

>>> print(fout)

<\_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>

* If the ﬁle already exists, opening it in write mode clears out the old data and starts fresh, so

be careful! If the ﬁle doesn’t exist, a new one is created.

* The write method of the ﬁle handle object puts data into the ﬁle, returning the number of

characters written.

>>> line1 = "This here's the wattle,\n"

>>> fout.write(line1) 24

* We must make sure to manage the ends of lines as we write to the ﬁle by explicitly inserting the newline character when we want to end a line. The print statement automatically appends a newline, but the write method does not add the newline automatically.

>>> line2 = 'the emblem of our land.**\n'**

>>> fout.write(line2) 24

* When you are done writing, you have to close the ﬁle to make sure that the last bit of data is

physically written to the disk so it will not be lost if the power goes oﬀ.

>>> fout.close( )

* We could close the ﬁles which we open for read as well, but we can be a little sloppy if we are only opening a few ﬁles since Python makes sure that all open ﬁles are closed when the program ends. When we are writing ﬁles, we want to explicitly close the ﬁles so as to leave nothing to chance.
* To avoid such chances, the *with* statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

|  |
| --- |
| **with** open("test.txt", 'w') as f : f.write("my first file\n") f.write("This file\n\n")  f.write("contains three lines\n") |
| my first file  This file These lines are written into file  “test.txt”  contains three lines |

ROOPA.H.M, Dept of MCA, RNSIT Page 36

**Classes and objects**

* Python is an object-oriented programming language. “object-oriented programming” uses programmer deﬁned types to organize both code and data.
* Almost everything in Python is an object, with its properties and methods.
* Class is a basis for any object oriented programming language.

## Class definition

* We have used many of Python’s built-in types; now we are going to define a new type.
* Class is a user-defined data type which binds data and functions together into single entity.
* Class is just a prototype (or a logical entity/blue print) which will not consume any memory.
* An object is an instance of a class and it has physical existence. One can create any number of objects for a class.
* A class can have a set of variables (also known as attributes, member variables) and member functions (also known as methods).
* Class − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members and methods, accessed via dot notation.
* Class can defined with following syntax

**class** ClassName:

'Optional class documentation string' class\_suite

* Class can be created using keyword *class*
* The class has a documentation string, which can be accessed via ClassName. doc .
* The class\_suite consists of all the component statements defining class members, data attributes and functions.
* As an example, we will create a class called Point .

class Point:

pass

#creating empty class

Here, we are creating an empty class without any members by just using the keyword pass

within it.

* Class can be created only with documentation string as follows:

|  |  |
| --- | --- |
| class Point:  """Represents a point in 2-D space."""  Print(Point) | Output:  <class  ' main .Point'> |

ROOPA.H.M, Dept of MCA, RNSIT Page 37

Because Point is deﬁned at the top level, its “full name” is main .Point. The term main indicates that the class Point is in the main scope of the current module.

* Creating a new object is called ***instantiation***, and the object is an ***instance*** of the class. Class can have any number of instances. Object of the class can be created as follows:

|  |  |
| --- | --- |
| **blank=Point()**  print(blank) | Output:  < main .Point object at 0x03C72070> |

Here *blank* is not the actual object, rather it contains a reference to Point .When we print an object, Python tells which class it belongs to and where it is stored in the memory. Observe the output ,It clearly shows that, the object occupies the physical space at location *0x03C72070*(hexadecimal value).

**Attributes**

* + An object can contain named elements known as **attributes**. One can assign values to these attributes using dot operator. For example, (0,0) represents the origin, and coordinates (x,y) represents some point. so we can assign two attributes **x** and **y** for the object *blank* of a class **Point** as below:

|  |  |
| --- | --- |
| >>> blank.x = 3.0  >>> blank.y = 4.0 | *Object Diagram* |

A state diagram that shows an object and its attributes is called an **object diagram**.

The variable *blank* refers to a Point object, which contains two attributes. Each attribute

refers to a ﬂoating-point number.

* + We can read the value of an attribute using the same syntax:

>>> blank.y

4.0

*# read the value of an attribute y*

>>> x = blank.x *# Attribute x of an object can be assigned to other variables*

>>> x

3.0

The expression *blank.x* means, “Go to the object *blank* refers to and get the value of x”. In the example, we assign that value to a variable named *x*. There is no conﬂict between the variable *x* and the attribute *x*.

ROOPA.H.M, Dept of MCA, RNSIT Page 38

* + Classes in Python have two types of attributes:

**Class attribute** − A variable that is shared by all instances of a class. They are common to all the objects of that class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

**Instance attribute** − instance attributes defined for individual objects.. Attributes of one instance are not available for another instance of the same class.

Following example demonstrate the usage of *instance attribute* and *class attribute*:

class Flower:

‘’’folwers and it behaviour’’’

color = 'white' *# class attribute shared by all instances*

>>> lotus = Flower()

>>> rose = Flower()

>>> lotus.color 'white'

>>> rose.color 'white'

>>> rose.usedIn=’bouquet’

>>> rose.usedIn

’bouquet’

>>> lotus.usedIn

*# shared by all Flower object # shared by all Flower object*

*#defining instance attribute*

*#specific to rose object*

AttributeError: 'flower' object has no attribute 'usedin'

Here, the attributes *usedIn* created is available only for the object rose, but not for lotus. Thus, *usedIn* is instance attribute but not class attribute. We can use attributes with dot notation as part of any expression.

>>> '(%g, %g)' % (blank.x, blank.y) '(3.0, 4.0)'

>>> sum = **blank.x** + **blank.y**

>>> sum 5.0

* + - We can pass an *instance as an argument* in the usual way.

def print\_**point(p)**:

*# p is an alias of blank*

print('(%g, %g)' %(p.x, p.y))

>>> print\_point(**blank**) *#reference of object is sent to p*

**(3.0, 4.0)**

ROOPA.H.M, Dept of MCA, RNSIT Page 39

**Example** :

*Program to create a class Point representing a point on coordinate system. Implement following functions –*

* *A function read\_point() to receive x and y attributes of a Point object as user input*
* *A function distance() which takes two objects of Point class as arguments and computes the Euclidean distance between them.*

|  |
| --- |
| import math  class Point:  """ class Point representing a coordinate point"""  def read\_point(p):  p.x=float(input("x coordinate:")) p.y=float(input("y coordinate:"))  def print\_point(p): print("(%g,%g)"%(p.x, p.y))  def distance(p1,p2):  d=math.sqrt((p1.x-p2.x)\*\*2+(p1.y-p2.y)\*\*2) return d  p1=Point() *#create first object*  print("Enter First point:")  read\_point(p1) *#read x and y for p1*  p2=Point() *#create second object*  print("Enter Second point:")  read\_point(p2) *#read x and y for p2*  dist=distance(p1,p2) *#compute distance*  print("First point is:") print\_point(p1) *#print p1* print("Second point is:") print\_point(p2) *#print p2*  print("Distance is: %g" %(distance(p1,p2))) *#print d* |
| Output:  **Enter First point:**  x coordinate:10 y coordinate:20  **Enter Second point:**  x coordinate:3 y coordinate:5  First point is: (10,20) Second point is:(3,5)  **Distance is: 16.5529** |

ROOPA.H.M, Dept of MCA, RNSIT Page 40

In the above program, we have used 3 functions which are not members of the class:

* *read\_point(p)* to read the input through keyboard for x and y values.
* *print\_point()* to display point object in the form of ordered-pair.
* *distance(p1,p2*) to find the Euclidean distance between two points (x1,y1) and (x2,y2).

The formula is 

since all these functions does not belong to class, they are called as normal functions without dot notation.

## Constructor method

* Python uses a special method called a **constructor method**. Python allows you to define only one constructor per class. Also known as the init () method, it will be the first method definition of a class and its syntax is

#### def init (self, parameter\_1, parameter\_2, …., parameter\_n):

**statement(s)**

* The init () method defines and initializes the instance variables. It is invoked as soon as an object of a class is instantiated.
* The init () method for a newly created object is automatically executed with all of its parameters .
* The init () method is indeed a special method as other methods do not receive this treatment. The parameters for init () method are initialized with the arguments that you had passed during instantiation of the class object.
* Class methods that begin with a double underscore ( ) are called special methods as they have special meaning. The number of arguments during the instantiation of the class object should be equivalent to the number of parameters in init () method (excluding the self parameter).
* Example:

class Person:

def init (self, name, age): self.name = name

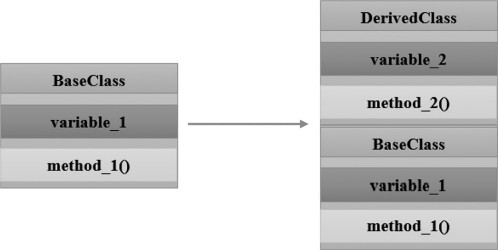
self.age = age p1 = Person("John", 36)

print(p1.name) print(p1.age)

ROOPA.H.M, Dept of MCA, RNSIT Page 41

# Inheritance

* Inheritance enables new classes to receive or inherit variables and methods of existing classes. Inheritance is a way to express a relationship between classes. If you want to build a new class, which is already similar to one that already exists, then instead of creating a new class from scratch you can reference the existing class and indicate what is different by overriding some of its behavior or by adding some new functionality.
* A class that is used as the basis for inheritance is called a *superclass* or *base* class. A class that inherits from a base class is called a *subclass* or *derived* class. The terms parent class and child class are also acceptable terms to use respectively.
* A derived class inherits variables and methods from its base class while adding additional variables and methods of its own. Inheritance easily enables reusing of existing code. Class BaseClass, on the left, has one variable and one method.
* Class DerivedClass, on the right, is derived from BaseClass and contains an additional variable and an additional method.



* The syntax for a derived class definition looks like this:

## class DerivedClassName(BaseClassName):

### <statement-1>

.

### .

<statement-N>

* To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

ROOPA.H.M, Dept of MCA, RNSIT Page 42

# Overloading

## Operator overloading

* + Normally operators like +,-,/,\*, works fine with built-in datatypes.
  + Changing the behavior of an operator so that it works with programmer-deﬁned types(class) is called operator overloading.
  + Basic operators like +, -, \* etc. can be overloaded. To overload an operator, one needs to write a method within user-defined class. The method should consist of the code what the programmer is willing to do with the operator.
  + Let us consider an example to overload + operator to add two Time objects by defining

add method inside the class.

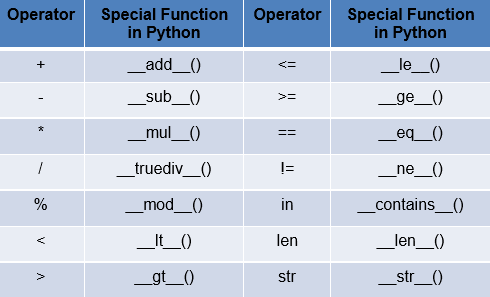
ROOPA.H.M, Dept of MCA, RNSIT Page 43

In the above example,

 when the statement **t3=t1+t2** is used, it invokes a special method add () written inside the class. Because, internal meaning of this statement is t3 = t1. add (t2)

 Here, t1 is the object invoking the method. Hence, self inside add () is the reference (alias) of t1. And, t2 is passed as argument explicitly.

Python provides a special set of methods which have to be used for overloading operator. Following table shows gives a list of operators and their respective Python methods for overloading.



## Example program

### This program demonstrates creating or defining a class and its object, init method and operator overloading concept by overloading + operator by redefining add function.

ROOPA.H.M, Dept of MCA, RNSIT Page 44

|  |  |
| --- | --- |
| class Point:  def **init (self,a=0,b=0)**: self.x=a  self.y=b  def **str (self)**:  return "(%d,%d)"%(self.x, self.y)  def **add (self, p2)**: p3=Point() p3.x=self.x+p2.x p3.y=self.y+p2.y return p3  p1=Point(10,20) p2=Point() print("P1 is:",p1)  print("P2 is:",p2)  p4=p1+p2 print("Sum is:",p4) | Output:  **P1 is: (10,20)**  **P2 is: (0,0)**  **Sum is: (10,20)** |

ROOPA.H.M, Dept of MCA, RNSIT Page 45

|  |  |
| --- | --- |
| **Question Bank** | |
| ***Q.***  ***No.*** | ***Questions*** |
|  | **LISTS** |
| 1 | What are lists? Lists are mutable. Justify the statement with examples. |
| 2 | How do you create an empty list? Give example. |
| 3 | Explain + and \* operations on list objects with suitable examples. |
| 4 | Discuss various built-in methods in lists. |
| 5 | Implement a Python program using Lists to store and display the average of N integers accepted  from the user. |
| 6 | What are the different ways of deleting elements from a list? Discuss with suitable functions. |
| 7 | How do you convert a list into a string and vice-versa? Illustrate with examples. |
| 8 | Write a short note on: a) Parsing lines b) Object Aliasing |
| 9 | Write the differences between  a. sort() and sorted()   1. append() and extend() 2. join() and split() |
| 10 | When do we encounter TypeError, ValueError and IndexError? |
| 11 | What are identical and equivalent objects? How are they identified? give examples. |
| 12 | Discuss different ways of traversing a list. |
| 13 | Discuss the following List operations and functions with examples :  1. Accessing, Traversing and Slicing the List Elements  2. + (concatenation) and \* (Repetition)   1. append, extend, sort , remove and delete 2. len, sum, min and max 3. split and join |
|  | **DICTIONARY and TUPLES** |
| 1 | How tuples are created in Python? Explain different ways of accessing and creating them. |
| 2 | Write a Python program to read all lines in a file accepted from the user and print all email  addresses contained in it. Assume the email addresses contain only non-white space characters. |
| 3 | List merits of dictionary over list. |
| 4 | Explain dictionaries. Demonstrate with a Python program. |
| 5 | Compare and contrast tuples with lists. |
| 6 | Define a dictionary type in Python. Give example. |
| 7 | Explain get() function in dictionary with suitable code snippet. |
| 8 | Discuss the dictionary methods keys() and items() with suitable programming examples. |

ROOPA.H.M, Dept of MCA, RNSIT Page 46

|  |  |
| --- | --- |
| 9 | Explain various steps to be followed while debugging a program that uses large datasets. |
| 10 | Briefly discuss key-value pair relationship in dictionaries. |
| 11 | Define a tuple. Give an example to illustrate creation of a tuple |
| 12 | What is mutable and immutable objects? Give examples. |
| 13 | Explain List of Tuples and Tuple of Lists. |
| 14 | How do you create an empty tuple and a single element tuple? |
| 15 | Explain DSU pattern with respect to tuples. Give example |
| 16 | How do you create a tuple using a string and using a list? Explain with example. |
| 17 | Explain the concept of tuple-comparison. How tuple-comparison is implemented in sort()  function? Discuss. |
| 18 | Write a short note on: Tuple assignment  Dictionaries and Tuples |
| 19 | How tuples can be used as a key for dictionaries? Discuss with example. |
| 20 | Discuss pros and cons of various sequences like lists, strings and tuples. |
| 21 | Explain the following operations in tuples: Sum of two tuples  Slicing operators |
| 22 | Discuss the Tuple Assignment with example .Explain how swapping can be done using tuple assignment. Write a Python program to input two integers a and b , and swap those numbers . Print both input and swapped numbers. |
|  | **SETS** |
| 23 | Explain how to create an empty set. |
| 24 | List the merits of sets over list. Demonstrate it with example. |
| 25 | Write a program to create an intersection, union, set difference, and symmetric  difference of sets. |
|  | **Classes and Objects** |
| 26 | Define class and object. Given an example for creating a class and an object of that class. |
| 27 | What are attributes? Explain with an example and respective object diagram. |
| 28 | Write a program to create a class called Rectangle with the help of a corner point, width and height. Write following functions and demonstrate their working:   1. To find and display center of rectangle 2. To display point as an ordered pair 3. To resize the rectangle |

ROOPA.H.M, Dept of MCA, RNSIT Page 47

|  |  |
| --- | --- |
| 29 | What is a Docstring? Why are they written? |
| 30 | What do you mean by “instance as returning value”? Explain with an example. |
| 31 | What is inheritance? Demonstrate it with example. |
| 32 | When do we encounter AttributeError? |
| 33 | How do you find the memory address of an instance of a class? |
| 34 | Differentiate instance attribute and class attribute |
| 35 | Difference between pure function and modifier. write a python program to find duration of the event if start and end time is given by defining class TIME. |
| 36 | Write a program to create a class called ‘Time’ to represent time in HH:MM:SS format. Perform following operations:  i. T3=T1+T2  ii. T4=T1+360  iii. T5=130+T1  Where T1,T2,T3,T4,T5 are object of time class |
| 37 | Explain init and str method with an example program |
| 38 | Write a program to add two point objects by overloading + operator. Overload \_ \_str\_ \_( ) to display point as an ordered pair. |
| 39 | Define polymorphism. Demonstrate polymorphism with function to find histogram to count the number of times each letters appears in a word and in sentence. |
| 40 | Using datetime module write a program that gets the current date and prints the day of the week. |
| 41 | What does the keyword self in python mean? Explain with an example. |

ROOPA.H.M, Dept of MCA, RNSIT Page 48