PYTHON MODULE 2

2.1 Functions

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

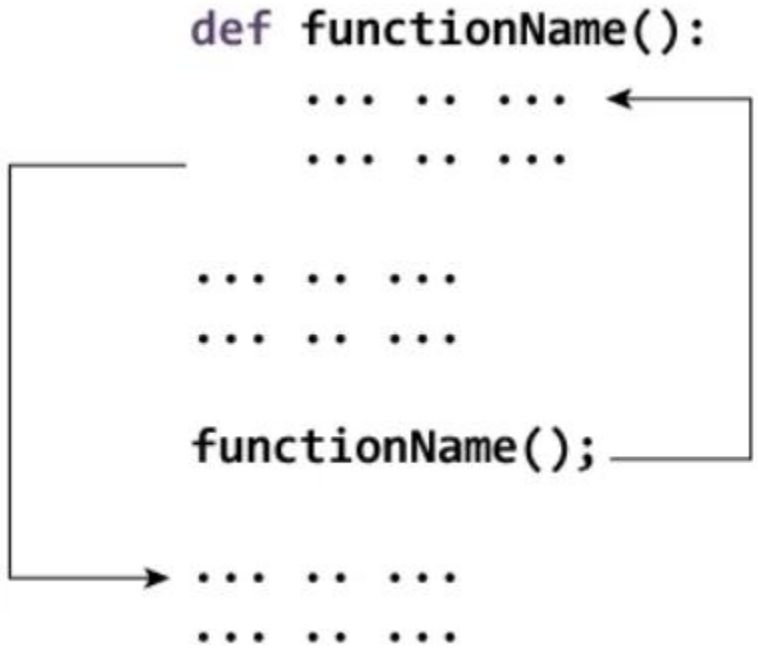
Syntax of Function

```
def function_name(parameters):
        """docstring"""
        statement(s)
```

Above shown is a function definition that consists of the following components.

1. Keyword def that marks the start of the function header.
2. A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of the function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

How Function works in Python?



Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

Built-in Functions**:**

Built-in functions are the functions that are already written or defined in python. We only need to remember the names of built-in functions and the parameters used in the functions. As these functions are already defined so we do not need to define these functions. Below are some **built-in** functions of Python.

Built-in Functions used in Python

Built-in Functions			
A abs() aiter() all() any() anext() ascii() **B** bin() bool() breakpoint() bytearray() bytes() **C** callable() chr() classmethod() compile() complex() **D** delattr() dict() dir() divmod()	**E** enumerate() eval() exec() **F** filter() float() format() frozenset() **G** getattr() globals() **H** hasattr() hash() help() hex() **I** id() input() int() isinstance() issubclass() iter()	**L** len() list() locals() **M** map() max() memoryview() min() **N** next() **O** object() oct() open() ord() **P** pow() print() property()	**R** range() repr() reversed() round() **S** set() setattr() slice() sorted() staticmethod() str() sum() super() **T** tuple() type() **V** vars() **Z** zip() **_** __import__()

User-Defined Functions:

The functions defined by a programmer to reduce the complexity of big problems and to use that function according to their need. This type of functions is called **user-defined functions**.

```
#Example of user defined function

x = 3
y = 4
def add():
   print(x+y)

add()
```

Output:

7

Note: User-defined functions are used commonly when we build large projects or applications in our upcoming articles. It is good practice to define functions for larger problems so that our code will remain combined.

Understanding Variable Scope

Local and Global variables: We have already discussed variables in our previous topics. Local and Global variables play a vital role while working with functions. Many times, if you are not aware of local and global variables term, then there will be a chance that once you will surely be stuck into the problem while solving complex programs. Let's discuss each of this term one by one

Global Variable in Python: Like it, names defines "Global" which means it can be accessible everywhere. Now listen carefully when we define a function, then we can also create a variable inside our function.

It is also possible that one **variable** is already there outside of a **function**. Now we have two variables: One is inside the function and the second one is outside of the function. Let's see the below-given diagram to make it more clear:

```
#Local and Global variable

x = 23              #This is the Global Variable

def fnc_name():
  y = 24            #This is Local Variable
  print(y)

fnc_name()


```

The variable defined outside of a function is called "Global" and the variable defined inside a function is called "Local" variable.

Important: Scope of the Global variable is everywhere inside the program. We can access the global variable even inside the function also. But the scope of the local variable is limited. It is only accessible inside the function only where it has been defined or declared.
We cannot use a local variable outside of the function. If we do so, then the compiler will generate an error as a *variable is not defined*.

Example 1: What if we try to access a local variable outside of function? Let's see with the help of an example:

```
#Local and Global variable

x = 23          #This is the Global Variable

def fnc_name():
 y = 24         #This is Local Variable
 print(y)

fnc_name()
print(y)
```

Output:

```
#Output
#Error: variable not defined
```

Question: If we define a local variable and a global variable with the same name.When we print the variable inside a function, which variable will be printed "Local" or "Global".

Answer: Local variable has the highest preference inside the function. It will first check for the local variable inside function. If local variable is present inside function, then it will go with the local variable else it will go with a global variable.

Example 2:

```
#Local and Global variable

x = 23          #This is the Global Variable

def fnc_name():
  x = 24        #This is Local Variable
  print(y)

fnc_name()
```

Output: 24

Global Keyword in Python: Global keyword is the keyword used to change the current scope of variable ie. to convert the scope of a local variable to the global variable. We already know that global variable is accessible everywhere inside the program. But the local variable is only accessible inside the function only.

But what if we want to make the local variable of function to be the global variable. For that purpose, we use the "Global" keyword. That's it nothing else.

```
#Local and Global variable

x = 23              #This is the Global Variable

def fnc_name():
  y = 24            #This is Local Variable
  print(y)

fnc_name()


```

You can easily differentiate between the **local and global variable** from the above given example. Here inside the **fnc_name(),** variable y is the local variable and it cannot be accessed outside of the function. If we want to access it outside we need to make it global first. For that purpose, we use the "Global" keyword.

Example elaborating the use of global keyword:

```
#Elaborating the use of Global Keyword

x = "Global Variable"
def fncn():
   z = "local variable z"
   global y
   y = "local variable y"

fncn()
print(x)  # X is global variable so it can be accessed anywhere
print(y)  # Y has been converted from local to global using "Global Keyword"
#print(z)  # Z is still local variable hence it generate error if we try to access outside of function
```

Output:

```
#Output
#Global variable
#local variable Y
```

Importance of well documented code.

Why Documenting Your Code Is So Important

When you write code, you write it for two primary audiences: your users and your developers (including yourself). Both audiences are equally important

Commenting vs Documenting Code

In general, commenting is describing your code to/for developers. The intended main audience is the maintainers and developers of the Python code. In conjunction with well-written code, comments help to guide the reader to better understand your code and its purpose and design:

Documenting code is describing its use and functionality to your users. While it may be helpful in the development process, the main intended audience is the users. The following section describes how and when to comment your code.

Documenting Your Python Code Base Using Docstrings

Docstrings Background
Documenting your Python code is all centered on docstrings. These are built-in strings that, when configured correctly, can help your users and yourself with your project's documentation. Along with docstrings, Python also has the built-in function help() that prints out the objects docstring to the console.

Use lambda expressions, iterators, and generators.

Lambda Function in Python: Lambda function is also known as an anonymous function. Anonymous function means, function without a name. We used the "def" keyword to define a function but the "lambda" keyword is used to define the lambda function.

Syntax of Lambda Function:

```
lambdaarguments : expression
```

Important note:

- The lambda function can have multiple arguments but it should have only single expressions.
- Lambda function is limited to a single expression only.
- Lambda function does not have any name.
- In the Lambda function, there is no need to write the return function.

Example 1: Program to find the square of five using **normal function**.

```
Square of 5 using normal function

x =5
deffncn():
i= x*x
print(i)

fncn()
```

Output:

```
#Output
#25
```

Example 2: Program to find the square of five using normal function.

```
Lambda function with single argument

x =lambdaa : a*a
result =x(5)
print(result)
```

Output:

```
#Output
#25
```

Here how it works

Let's take the same example, Program to find the square of a number(5).

Step 1). a : a*a
just say a i want the square of a*a ie. (a:a*a)

Step 2). lambda a : a*a
Now we are working with a lambda function, so we need to name our arguments as lambda ie (lambda a:a*a)

Step 3). x = lambda a : a*a
Now we will assign or store the value in a variable x ie. (x = lambda a:a*a)

So our complete lambda function is : **x = lambda a:a*a**
The rest of the process is the same to call a function by passing arguments. Here we have used only one argument that's why we only pass one argument ie. **x(5)** .
Next we print the output accordingly.

Example 3: In this example I will show you how to use more than one argument in lambda function. Lambda function is also called as **lambda expression**.

Program to find sum of two numbers using lambda expression:

```
Lambda function with two arguments

x =lambdaa,b:a+b
result =x(5,7)
print(result)
```

Output:

```
#Output
#12
```

Note: A very important point to remember is that lambda function can have any number of arguments but have single expression.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a Python **iterator object** must implement two special methods, `__iter__()` and `__next__()`, collectively called the **iterator protocol**.
An object is called **iterable** if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.
The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

Iterating Through an Iterator

We use the `next()` function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise the `StopIteration` Exception. Following is an example.

```
# define a list
my_list = [4, 7, 0, 3]

# get an iterator using iter()
my_iter = iter(my_list)

# iterate through it using next()

# Output: 4
print(next(my_iter))

# Output: 7
print(next(my_iter))

# next(obj) is same as obj.__next__()

# Output: 0
print(my_iter.__next__())

# Output: 3
print(my_iter.__next__())

# This will raise error, no items left
```

```
next(my_iter)
```

Output

```
4
7
0
3
Traceback (most recent call last):
  File "<string>", line 24, in <module>
    next(my_iter)
StopIteration
```

A more elegant way of automatically iterating is by using the for loop. Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

```
>>>for element inmy_list:
... print(element)
...
4
7
0
3
```

Working of for loop for Iterators

As we see in the above example, the `for` loop was able to iterate automatically through the list.

In fact the `for` loop can iterate over any iterable. Let's take a closer look at how the `for` loop is actually implemented in Python.

Is actually implemented as.

```
# create an iterator object from that iterable
```

```
iter_obj = iter(iterable)

# infinite loop
whileTrue:
try:
# get the next item
        element = next(iter_obj)
# do something with element
exceptStopIteration:
# ifStopIteration is raised, break from loop
break
```

So internally, the `for` loop creates an iterator object, `iter_obj` by calling `iter()` on the iterable.

Ironically, this `for` loop is actually an infinite while loop.

Inside the loop, it calls `next()` to get the next element and executes the body of the `for` loop with this value. After all the items exhaust, `StopIteration` is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

Python Infinite Iterators

It is not necessary that the item in an iterator object has to be exhausted. There can be infinite iterators (which never ends). We must be careful when handling such iterators.

Here is a simple example to demonstrate infinite iterators.

The built-in function `iter()` can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel.

```
>>>int()
0

>>>inf = iter(int,1)
```

```
>>>next(inf)
0
>>>next(inf)
0
```

We can see that the int() function always returns 0. So passing it as iter(int,1) will return an iterator that calls int() until the returned value equals 1. This never happens and we get an infinite iterator.

A sample run would be as follows.

```
>>>a = iter(InfIter())
>>>next(a)
1
>>>next(a)
3
>>>next(a)
5
>>>next(a)
7
```

And so on...

The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory..

Generators in Python

There is a lot of work in building an iterator in Python. We have to implement a class with __iter__() and __next__() method, keep track of internal states, and raise StopIteration when there are no values to be returned.

This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Create Generators in Python

It is fairly simple to create a generator in Python. It is as easy as defining a normal function, but with a `yield` statement instead of a `return` statement.
If a function contains at least one `yield` statement (it may contain other `yield` or `return` statements), it becomes a generator function.
Both `yield` and `return` will return some value from a function.
The difference is that while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and Normal function

Here is how a generator function differs from a normal function.

- Generator function contains one or more `yield` statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

Here is an example to illustrate all of the points stated above. We have a generator function named my_gen() with several yield statements.

```
# A simple generator function
defmy_gen():
    n = 1
print('This is printed first')
# Generator function contains yield statements
yield n

    n += 1
print('This is printed second')
yield n

    n += 1
print('This is printed at last')
yield n
```

An interactive run in the interpreter is given below. Run these in the Python shell to see the output.

```
>>># It returns an object but does not start execution immediately.
>>>a = my_gen()

>>># We can iterate through the items using next().
>>>next(a)
This is printed first
1
>>># Once the function yields, the function is paused and the control is transferred to the caller.

>>># Local variables and theirs states are remembered between successive calls.
>>>next(a)
This is printed second
2

>>>next(a)
This is printed at last
3

>>># Finally, when the function terminates, StopIteration is raised automatically on further calls.
```

```
>>>next(a)
Traceback (most recent call last):
...
StopIteration
>>>next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that the value of variable `n` is remembered between each call.

Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.

To restart the process we need to create another generator object using something like `a = my_gen()`.

One final thing to note is that we can use generators with for loops directly.

This is because a `for` loop takes an iterator and iterates over it using `next()` function. It automatically ends when `StopIteration` is raised.

Check here to know how a for loop is actually implemented in Python.

```
# A simple generator function
defmy_gen():
    n = 1
print('This is printed first')
# Generator function contains yield statements
yield n

    n += 1
print('This is printed second')
yield n

    n += 1
print('This is printed at last')
```

```
yield n


# Using for loop
for item inmy_gen():
print(item)
```
Run Code

When you run the program, the output will be:

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Python Generator Expression

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

Similar to the lambda functions which create anonymous functions,

generator expressions create anonymous generator functions.

```
# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
list_ = [x**2for x inmy_list]

# same thing can be done using a generator expression
# generator expressions are surrounded by parenthesis ()
generator = (x**2for x inmy_list)

print(list_)
```

```
print(generator)
```

Output

```
[1, 9, 36, 100]
<generator object <genexpr> at 0x7f5d4eb4bf50>
```

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object, which produces items only on demand.

Here is how we can start getting items from the generator:

```
# Initialize the list
my_list = [1, 3, 6, 10]

a = (x**2for x inmy_list)
print(next(a))

print(next(a))

print(next(a))

print(next(a))

next(a)
```
Run Code

When we run the above program, we get the following output:

```
1
9
36
100
Traceback (most recent call last):
  File "<string>", line 15, in <module>
StopIteration
```

Generator expressions can be used as function arguments. When used in such a way, the round parentheses can be dropped.

```
>>>sum(x**2for x inmy_list)
146

>>>max(x**2for x inmy_list)
100
```

2.2 Advanced concepts using python lists and built-in functions.

Working with python lists and built-in functions

Create Python Lists

In Python, a list is created by placing elements inside square brackets [], separated by commas.

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

A list can have any number of items and they may be of different types (integer, float, string, etc.).

```
# empty list
my_list = []

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

A list can also have another list as an item. This is called a nested list.

```
# nested list
```

```
my_list = ["mouse", [8, 4, 6], ['a']]
```

Access List Elements

There are various ways in which we can access the elements of a list.

List Index

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.
Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.
Nested lists are accessed using nested indexing.

```
my_list = ['p', 'r', 'o', 'b', 'e']

# first item
print(my_list[0])  # p

# third item
print(my_list[2])  # o

# fifth item
print(my_list[4])  # e

# Nested List
n_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing
print(n_list[0][1])

print(n_list[1][3])

# Error! Only integer can be used for indexing
print(my_list[4.0])
```

Run Code

Output

```
p
o
e
a
5
Traceback (most recent call last):
  File "<string>", line 21, in <module>
TypeError: list indices must be integers or slices, not float
```

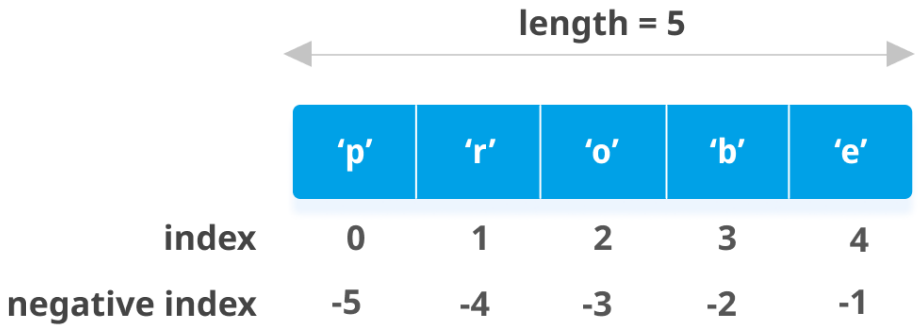
Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing in lists
my_list = ['p','r','o','b','e']

# last item
print(my_list[-1])

# fifth last item
print(my_list[-5])
```
Run Code

Output

```
e
p
```



List indexing in Python

List Slicing in Python

We can access a range of items in a list by using the slicing operator :.

```
# List slicing in Python

my_list = ['p','r','o','g','r','a','m','i','z']

# elements from index 2 to index 4
print(my_list[2:5])

# elements from index 5 to end
print(my_list[5:])

# elements beginning to end
print(my_list[:])
```

Run Code

Output

```
['o', 'g', 'r']
['a', 'm', 'i', 'z']
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Note: When we slice lists, the start index is inclusive but the end index is exclusive. For example, `my_list[2: 5]` returns a list with elements at index 2, 3 and 4, but not 5.

Add/Change List Elements

Lists are mutable, meaning their elements can be changed unlike string or tuple.

We can use the assignment operator = to change an item or a range of items.

```
# Correcting mistake values in a list
odd = [2, 4, 6, 8]

# change the 1st item
odd[0] = 1

print(odd)

# change 2nd to 4th items
odd[1:4] = [3, 5, 7]

print(odd)
Run Code
```

Output

```
[1, 4, 6, 8]
[1, 3, 5, 7]
```

We can add one item to a list using the append() method or add several items using the extend() method.

```
# Appending and Extending lists in Python
odd = [1, 3, 5]

odd.append(7)

print(odd)

odd.extend([9, 11, 13])

print(odd)
Run Code
```

Output

```
[1, 3, 5, 7]
[1, 3, 5, 7, 9, 11, 13]
```

We can also use + operator to combine two lists. This is also called concatenation.

The * operator repeats a list for the given number of times.

```
# Concatenating and repeating lists
odd = [1, 3, 5]

print(odd + [9, 7, 5])

print(["re"] * 3)
```

Run Code

Output

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

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

```
# Demonstration of list insert() method
odd = [1, 9]
odd.insert(1,3)

print(odd)

odd[2:2] = [5, 7]

print(odd)
```

Run Code

Output

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

Delete List Elements

We can delete one or more items from a list using the Python del statement. It can even delete the list entirely.

```
# Deleting list items
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# delete one item
delmy_list[2]

print(my_list)

# delete multiple items
delmy_list[1:5]

print(my_list)

# delete the entire list
delmy_list

# Error: List not defined
print(my_list)
```

Run Code

Output

```
['p', 'r', 'b', 'l', 'e', 'm']
['p', 'm']
Traceback (most recent call last):
  File "<string>", line 18, in <module>
NameError: name 'my_list' is not defined
```

We can use `remove()` to remove the given item or `pop()` to remove an item at the given index.

The `pop()` method removes and returns the last item if the index is not provided.

This helps us implement lists as stacks (first in, last out data structure).

And, if we have to empty the whole list, we can use the `clear()` method.

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']
```

```
print(my_list)

# Output: 'o'
print(my_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']
print(my_list)

# Output: 'm'
print(my_list.pop())

# Output: ['r', 'b', 'l', 'e']
print(my_list)

my_list.clear()

# Output: []
print(my_list)
```
Run Code

Output

```
['r', 'o', 'b', 'l', 'e', 'm']
o
['r', 'b', 'l', 'e', 'm']
m
['r', 'b', 'l', 'e']
[]
```

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
>>>my_list = ['p','r','o','b','l','e','m']
>>>my_list[2:3] = []
>>>my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>>my_list[2:5] = []
>>>my_list
['p', 'r', 'm']
```

Python List Methods

Python has many useful list methods that makes it really easy to work with lists. Here are some of the commonly used list methods.

Methods	Descriptions
append()	adds an element to the end of the list
extend()	adds all elements of a list to another list
insert()	inserts an item at the defined index
remove()	removes an item from the list
pop()	returns and removes an element at the given index
clear()	removes all items from the list
index()	returns the index of the first matched item
count()	returns the count of the number of items passed as an argument
sort()	sort items in a list in ascending order
reverse()	reverse the order of items in the list

copy()	returns a shallow copy of the list

```
# Example on Python list methods

my_list = [3, 8, 1, 6, 8, 8, 4]

# Add 'a' to the end
my_list.append('a')

# Output: [3, 8, 1, 6, 8, 8, 4, 'a']
print(my_list)

# Index of first occurrence of 8
print(my_list.index(8))   # Output: 1

# Count of 8 in the list
print(my_list.count(8))  # Output: 3
Run Code
```

Naturally construct lists with list comprehension

List Comprehension: Elegant way to create Lists

List comprehension is an elegant and concise way to create a new list from an existing list in Python.

A list comprehension consists of an expression followed by for statement inside square brackets.
Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x inrange(10)]
print(pow2)
Run Code
```

Output

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

This code is equivalent to:

```
pow2 = []
for x inrange(10):
   pow2.append(2 ** x)
```

A list comprehension can optionally contain more `for` or [if statements](). An optional `if` statement can filter out items for the new list. Here are some examples.

```
>>>pow2 = [2 ** x for x inrange(10) if x >5]
>>>pow2
[64, 128, 256, 512]
>>>odd = [x for x inrange(20) if x % 2 == 1]
>>>odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>>[x+yfor x in ['Python ','C '] for y in ['Language','Programming']]
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

Other List Operations in Python

List Membership Test

We can test if an item exists in a list or not, using the keyword `in`.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True
print('p'inmy_list)
```

```
# Output: False
print('a'inmy_list)

# Output: True
print('c'notinmy_list)
```
Run Code

Output

```
True
False
True
```

Iterating Through a List

Using a `for` loop we can iterate through each item in a list.

```
for fruit in ['apple','banana','mango']:
print("I like",fruit)
```
Run Code

Output

```
I like apple
I like banana
I like mango
```

Understanding of tuples and differences with python list

Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses `()`, separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples

# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)

# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

Run Code

Output

```
()
(1, 2, 3)
(1, 'Hello', 3.4)
('mouse', [8, 4, 6], (1, 2, 3))
```

A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"
print(my_tuple)

# tuple unpacking is also possible
a, b, c = my_tuple

print(a)   # 3
print(b)   # 4.6
print(c)     # dog
```

Run Code

Output

```
(3, 4.6, 'dog')
3
4.6
dog
```

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("hello")
print(type(my_tuple))  #<class 'str'>

# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple))  #<class 'tuple'>

# Parentheses is optional
my_tuple = "hello",
print(type(my_tuple))  #<class 'tuple'>
```

Run Code

Output

```
<class 'str'>
<class 'tuple'>
<class 'tuple'>
```

Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
# Accessing tuple elements using indexing
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # 'p'
print(my_tuple[5])   # 't'

# IndexError: list index out of range
# print(my_tuple[6])

# Index must be an integer
# TypeError: list indices must be integers, not float
# my_tuple[2.0]

# nested tuple
```

```
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])      # 's'
print(n_tuple[1][1])      # 4
```
Run Code

Output

```
p
t
s
4
```

---

2. Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing for accessing tuple elements
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

Output

```
t
p
```

3. Slicing

We can access a range of items in a tuple by using the slicing operator colon :.

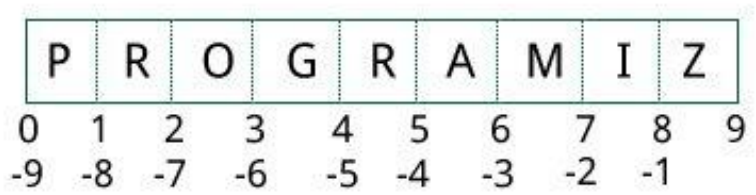
```
# Accessing tuple elements using slicing
my_tuple = ('p','r','o','g','r','a','m','i','z')

# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

Run Code

Output

```
('r', 'o', 'g')
('p', 'r')
('i', 'z')
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.



Element Slicing in Python

Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like a list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

```
# Changing tuple values
my_tuple = (4, 2, 3, [6, 5])


# TypeError: 'tuple' object does not support item assignment
# my_tuple[1] = 9

# However, item of mutable element can be changed
my_tuple[3][0] = 9# Output: (4, 2, 3, [9, 5])
print(my_tuple)

# Tuples can be reassigned
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple)
Run Code
```

Output

```
(4, 2, 3, [9, 5])
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

We can use `+` operator to combine two tuples. This is called **concatenation**.

We can also **repeat** the elements in a tuple for a given number of times using the `*` operator.

Both `+` and `*` operations result in a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

Run Code

Output

```
(1, 2, 3, 4, 5, 6)
('Repeat', 'Repeat', 'Repeat')
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.

Deleting a tuple entirely, however, is possible using the keyword del.

```
# Deleting tuples
my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# can't delete items
# TypeError: 'tuple' object doesn't support item deletion
# del my_tuple[3]

# Can delete an entire tuple
delmy_tuple

# NameError: name 'my_tuple' is not defined
print(my_tuple)
```

```
Run Code
```

Output

```
Traceback (most recent call last):
  File "<string>", line 12, in <module>
NameError: name 'my_tuple' is not defined
```

Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Some examples of Python tuple methods:

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # Output: 2
print(my_tuple.index('l'))  # Output: 3
Run Code
```

Output

```
2
3
```

Other Tuple Operations

1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword `in`.

```
# Membership test in tuple
my_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation
print('a'inmy_tuple)
print('b'inmy_tuple)

# Not in operation
print('g'notinmy_tuple)
Run Code
```

Output

```
True
False
True
```

2. Iterating Through a Tuple

We can use a `for` loop to iterate through each item in a tuple.

```
# Using a for loop to iterate through a tuple
for name in ('John', 'Kate'):
print("Hello", name)
Run Code
```

Output

```
Hello John
Hello Kate
```

Differences with lists

- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Deep dive into sets, dictionaries and work with loops

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).

However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

Creating Python Sets

A set is created by placing all the items (elements) inside curly braces `{}`, separated by comma, or by using the built-in `set()` function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# Different types of sets in Python
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

Run Code

Output

```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```

Try the following examples as well.

```
# set cannot have duplicates
# Output: {1, 2, 3, 4}
my_set = {1, 2, 3, 4, 3, 2}
print(my_set)

# we can make set from a list
# Output: {1, 2, 3}
my_set = set([1, 2, 3, 2])
print(my_set)

# set cannot have mutable items
# here [3, 4] is a mutable list
# this will cause an error.

my_set = {1, 2, [3, 4]}
```

Run Code

Output

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

```
Traceback (most recent call last):
  File "<string>", line 15, in <module>
my_set = {1, 2, [3, 4]}
TypeError: unhashable type: 'list'
```

Creating an empty set is a bit tricky.

Empty curly braces `{}` will make an empty dictionary in Python. To make a set without any elements, we use the `set()` function without any argument.

```
# Distinguish set and dictionary while creating empty set

# initialize a with {}
a = {}

# check data type of a
print(type(a))

# initialize a with set()
a = set()

# check data type of a
print(type(a))
```

Run Code

Output

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

Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initializemy_set
my_set = {1, 3}
print(my_set)

# my_set[0]
# if you uncomment the above line
# you will get an error
# TypeError: 'set' object does not support indexing

# add an element
# Output: {1, 2, 3}
my_set.add(2)
print(my_set)

# add multiple elements
# Output: {1, 2, 3, 4}
my_set.update([2, 3, 4])
print(my_set)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
my_set.update([4, 5], {1, 6, 8})
print(my_set)
```

Run Code

Output

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

Removing elements from a set

A particular item can be removed from a set using the methods discard() and remove().

The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

The following example will illustrate this.

```
# Difference between discard() and remove()

# initializemy_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)

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

# discard an element
# not present in my_set
```

```
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)

# remove an element
# not present in my_set
# you will get an error.
# Output: KeyError

my_set.remove(2)
```
Run Code

Output

```
{1, 3, 4, 5, 6}
{1, 3, 5, 6}
{1, 3, 5}
{1, 3, 5}
Traceback (most recent call last):
  File "<string>", line 28, in <module>
KeyError: 2
```

Similarly, we can remove and return an item using the pop() method.

Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all the items from a set using the clear() method.

```
# initializemy_set
# Output: set of unique elements
my_set = set("HelloWorld")
print(my_set)

# pop an element
# Output: random element
print(my_set.pop())

# pop another element
my_set.pop()
print(my_set)

# clearmy_set
```

```
# Output: set()
my_set.clear()
print(my_set)
Run Code
```

Output

```
{'H', 'l', 'r', 'W', 'o', 'd', 'e'}
H
{'r', 'W', 'o', 'd', 'e'}
set()
```

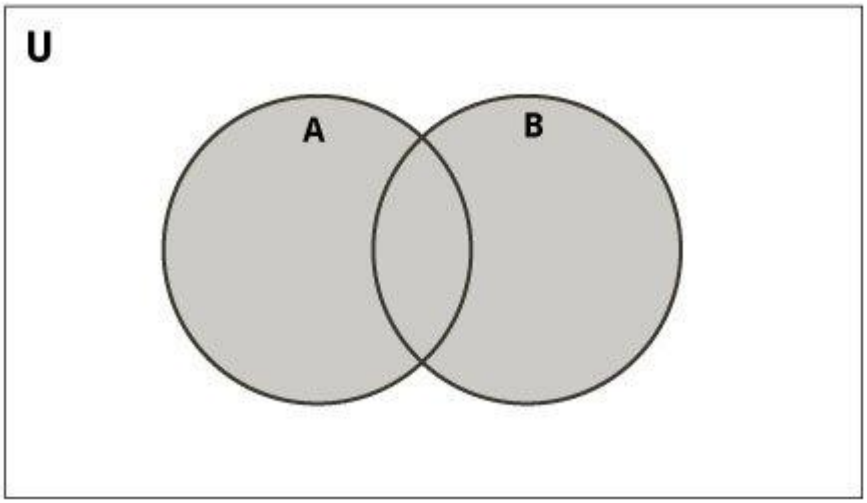
Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

```
>>>A = {1, 2, 3, 4, 5}
>>>B = {4, 5, 6, 7, 8}
```

Set Union



Set Union in Python

Union of `A` and `B` is a set of all elements from both sets.

Union is performed using `|` operator. Same can be accomplished using the `union()` method.

```
# Set union method
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
# Output: {1, 2, 3, 4, 5, 6, 7, 8}
print(A | B)
```

Run Code

Output

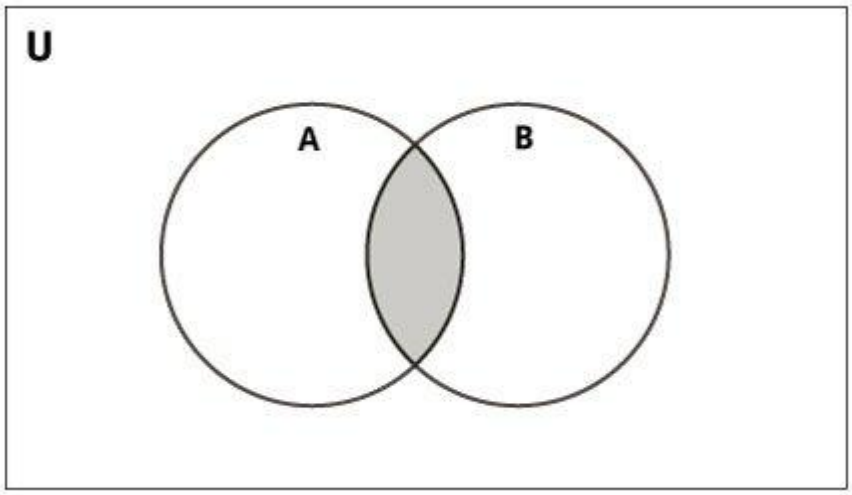
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Try the following examples on Python shell.

```
# use union function
>>>A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}

# use union function on B
```

```
>>>B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection



Set Intersection in Python

Intersection of A and B is a set of elements that are common in both the sets. Intersection is performed using & operator. Same can be accomplished using the intersection() method.

```
# Intersection of sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
# Output: {4, 5}
print(A & B)
```

Run Code

Output

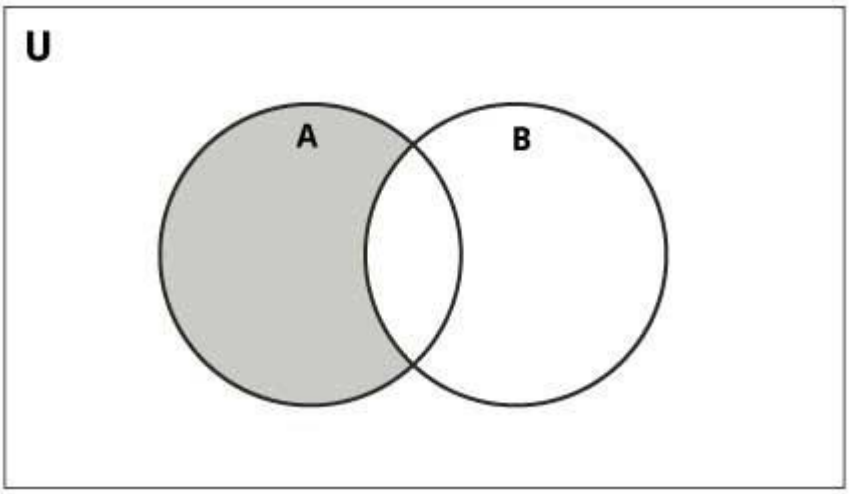
```
{4, 5}
```

Try the following examples on Python shell.

```
# use intersection function on A
```

```
>>>A.intersection(B)
{4, 5}

# use intersection function on B
>>>B.intersection(A)
{4, 5}
```

Set Difference



Set Difference in Python

Difference of the set `B` from set `A`(`A` - `B`) is a set of elements that are only in `A` but not in `B`. Similarly, `B` - `A` is a set of elements in `B` but not in `A`.

Difference is performed using `-` operator. Same can be accomplished using the `difference()` method.

```
# Difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use - operator on A
# Output: {1, 2, 3}
print(A - B)
```

Run Code

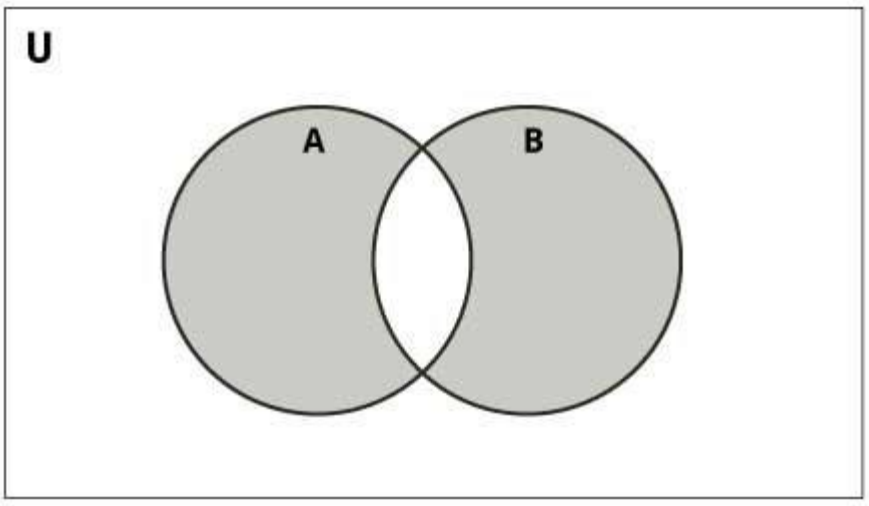
Output

```
{1, 2, 3}
```

Try the following examples on Python shell.

```
# use difference function on A
>>>A.difference(B)
{1, 2, 3}

# use - operator on B
>>>B - A
{8, 6, 7}

# use difference function on B
>>>B.difference(A)
{8, 6, 7}
```

Set Symmetric Difference



Set Symmetric Difference in Python

Symmetric Difference of `A` and `B` is a set of elements in `A` and `B` but not in both (excluding the intersection).

Symmetric difference is performed using `^` operator. Same can be accomplished using the method `symmetric_difference()`.

```
# Symmetric difference of two sets
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use ^ operator
# Output: {1, 2, 3, 6, 7, 8}
print(A ^ B)
```

Run Code

Output

```
{1, 2, 3, 6, 7, 8}
```

Try the following examples on Python shell.

```
# usesymmetric_difference function on A
>>>A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}

# usesymmetric_difference function on B
>>>B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Other Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:

Method	Description
add()	Adds an element to the set

clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns `True` if two sets have a null intersection
issubset()	Returns `True` if another set contains this set
issuperset()	Returns `True` if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises `KeyError` if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a `KeyError`
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another

union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Other Set Operations

Set Membership Test

We can test if an item exists in a set or not, using the `in` keyword.

```
# in keyword in a set
# initializemy_set
my_set = set("apple")

# check if 'a' is present
# Output: True
print('a'inmy_set)

# check if 'p' is present
# Output: False
print('p'notinmy_set)
Run Code
```

Output

```
True
False
```

Iterating Through a Set

We can iterate through each item in a set using a `for` loop.

```
>>>for letter in set("apple"):
... print(letter)
...
a
p
e
l
```

Built-in Functions with Set

Built-in functions like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` etc. are commonly used with sets to perform different tasks.

Function	Description
all()	Returns `True` if all elements of the set are true (or if the set is empty).
any()	Returns `True` if any element of the set is true. If the set is empty, returns `False`.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).

sum()	Returns the sum of all elements in the set.

Python dictionary is an unordered collection of items. Each item of a dictionary has a `key/value` pair.
Dictionaries are optimized to retrieve values when the key is known.

Python Dictionary

In this tutorial, you'll learn everything about Python dictionaries; how they are created, accessing, adding, removing elements from them and various built-in methods.

Python Dictionaries to Store key/value Pairs

Python dictionary is an unordered collection of items. Each item of a dictionary has a `key/value` pair.
Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces `{}` separated by commas.
An item has a `key` and a corresponding `value` that is expressed as a pair (**key: value**).
While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# 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]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

As you can see from above, we can also create a dictionary using the built-in `dict()` function.

Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses `keys`. Keys can be used either inside square brackets `[]` or with the `get()` method.
If we use the square brackets `[]`, `KeyError` is raised in case a key is not found in the dictionary. On the other hand, the `get()` method returns `None` if the key is not found.

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack
print(my_dict['name'])

# Output: 26
```

```
print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error
# Output None
print(my_dict.get('address'))

# KeyError
print(my_dict['address'])
```
Run Code

Output

```
Jack
26
None
Traceback (most recent call last):
  File "<string>", line 15, in <module>
    print(my_dict['address'])
KeyError: 'address'
```

Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}

# update value
my_dict['age'] = 27
```

```
#Output: {'age': 27, 'name': 'Jack'}
print(my_dict)

# add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```
Run Code

Output

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

Removing elements from Dictionary

We can remove a particular item in a dictionary by using the `pop()` method. This method removes an item with the provided `key` and returns the `value`.

The `popitem()` method can be used to remove and return an arbitrary `(key, value)` item pair from the dictionary. All the items can be removed at once, using the `clear()` method.

We can also use the `del` keyword to remove individual items or the entire dictionary itself.

```
# Removing elements from a dictionary

# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value
# Output: 16
print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)
```

```
# remove an arbitrary item, return (key,value)
# Output: (5, 25)
print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}
print(squares)

# remove all items
squares.clear()

# Output: {}
print(squares)

# delete the dictionary itself
del squares
```

Output

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
{1: 1, 2: 4, 3: 9}
{}
Traceback (most recent call last):
  File "<string>", line 30, in <module>
    print(squares)
NameError: name 'squares' is not defined
```

Python Dictionary Methods

Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Returns the value of the key. If the key does not exist, returns d (defaults to None).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the key and returns its value or d if key is not found. If d is not provided and the key is not found, it raises KeyError.
popitem()	Removes and returns an arbitrary item (**key, value**). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of d and returns d (defaults to None).
update([other])	Updates the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Returns a new object of the dictionary's values

Here are a few example use cases of these methods.

```
# Dictionary Methods
marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}
print(marks)

for item inmarks.items():
print(item)

# Output: ['English', 'Math', 'Science']
print(list(sorted(marks.keys())))
Run Code
```

Output

```
{'Math': 0, 'English': 0, 'Science': 0}
('Math', 0)
('English', 0)
('Science', 0)
['English', 'Math', 'Science']
```

2.3 Python Scripting

Read and write files, handle errors, and import local scripts

File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the **File Handle** in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. **Read Only ('r') :** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises the I/O error. This is also the default mode in which a file is opened.
2. **Read and Write ('r+'):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.

3. **Write Only ('w') :** Open the file for writing. For the existing files, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
4. **Write and Read ('w+')** : Open the file for reading and writing. For an existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
5. **Append Only ('a')**: Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read ('a+') :** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Opening a File

It is done using the open() function. No module is required to be imported for this function.

File_object = open(r"File_Name","Access_Mode")

The file should exist in the same directory as the python program file else, the full address of the file should be written in place of the filename. Note: The **r** is placed before the filename to prevent the characters in the filename string to be treated as special characters. For example, if there is \temp in the file address, then \t is treated as the tab character, and an error is raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in the same directory and the address is not being placed.

- Python

```
# Open function to open the file "MyFile1.txt"
# (same directory) in append mode and
file1 =open("MyFile.txt","a")

# store its reference in the variable file1
# and "MyFile2.txt" in D:\Text in file2
file2 =open(r"D:\Text\MyFile2.txt","w+")
```

Closing a file

close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode. File_object.close()

- Python

```
# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 =open("MyFile.txt","a")
file1.close()
```

Writing to a file

There are two ways to write in a file.

1. **write() :** Inserts the string str1 in a single line in the text file.

File_object.write(str1)

1. **writelines() :** For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.

File_object.writelines(L) for L = [str1, str2, str3]

Reading from a file

There are three ways to read data from a text file.

1. **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

File_object.read([n])

1. **readline() :** Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

File_object.readline([n])

1. **readlines() :** Reads all the lines and return them as each line a string element in a list.

File_object.readlines()

Note: '\n' is treated as a special character of two bytes

- Python3

```
# Program to show various ways to read and
# write data in a file.
file1 =open("myfile.txt","w")
L =["This is Delhi \n","This is Paris \n","This is London \n"]
```

```
# \n is placed to indicate EOL (End of Line)
file1.write("Hello \n")
file1.writelines(L)
file1.close() #to change file access modes

file1 =open("myfile.txt","r+")

print("Output of Read function is ")
print(file1.read())
print()

# seek(n) takes the file handle to the nth
# bite from the beginning.
file1.seek(0)

print( "Output of Readline function is ")
print(file1.readline())
print()

file1.seek(0)

# To show difference between read and readline
print("Output of Read(9) function is ")
print(file1.read(9))
print()

file1.seek(0)

print("Output of Readline(9) function is ")
print(file1.readline(9))

file1.seek(0)
# readlines function
print("Output of Readlines function is ")
print(file1.readlines())
print()
file1.close()
```

Output:

Output of Read function is

Hello

This is Delhi

This is Paris

This is London

Output of Readline function is

Hello

Output of Read(9) function is

Hello

Th

Output of Readline(9) function is

Hello

Output of Readlines function is

['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']

Appending to a file

- Python3

```
# Python program to illustrate
# Append vs write mode
file1 =open("myfile.txt","w")
L =["This is Delhi \n","This is Paris \n","This is London \n"]
file1.writelines(L)
file1.close()

# Append-adds at last
file1 =open("myfile.txt","a")#append mode
file1.write("Today \n")
```

```
file1.close()

file1 =open("myfile.txt","r")
print("Output of Readlines after appending")
print(file1.readlines())
print()
file1.close()

# Write-Overwrites
file1 =open("myfile.txt","w")#write mode
file1.write("Tomorrow \n")
file1.close()

file1 =open("myfile.txt","r")
print("Output of Readlines after writing")
print(file1.readlines())
print()
file1.close()
```

Output:

Output of Readlines after appending

['This is Delhi \n', 'This is Paris \n', 'This is London \n', 'Today \n']

Output of Readlines after writing

['Tomorrow \n']

Python Exception Handling Using try, except and finally statement

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Some of the common Exception Errors are :

- **IOError:** if the file can't be opened
- **KeyboardInterrupt:** when an unrequired key is pressed by the user
- **ValueError:** when built-in function receives a wrong argument
- **EOFError:** if End-Of-File is hit without reading any data
- **ImportError:** if it is unable to find the module

Difference between Syntax Error and Exceptions

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Exceptions: Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Example: Let us try to access the array element whose index is out of bound and handle the corresponding exception.

- Python3

```
# Python program to handle simple runtime error
#Python 3

a =[1, 2, 3]
try:
    print("Second element = %d"%(a[1]))

    # Throws error since there are only 3 elements in array
    print("Fourth element = %d"%(a[3]))

except:
    print("An error occurred")
```

In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax for adding specific exceptions are –

```
try:
    # statement(s)
```

```
except IndexError:
    # statement(s)
except ValueError:
    # statement(s)
```

Example: Catching specific exception in Python

```
# Program to handle multiple errors with one
# except statement
# Python 3

def fun(a):
    if a < 4:

        # throws ZeroDivisionError for a = 3
        b = a/(a-3)

    # throws NameError if a >= 4
    print("Value of b = ", b)

try:
    fun(3)
    fun(5)

# note that braces () are necessary here for
# multiple exceptions
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

Output

ZeroDivisionError Occurred and Handled

If you comment on the line fun(3), the output will be

NameError Occurred and Handled

The output above is so because as soon as python tries to access the value of b, NameError occurs.

Try with Else Clause

In python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Example: Try with else clause

- Python3

```
# Program to depict else clause with try-except

# Python 3

# Function which returns a/b

def AbyB(a , b):

    try:

        c =((a+b) /(a-b))

    except ZeroDivisionError:

        print("a/b result in 0")

    else:

        print(c)


# Driver program to test above function

AbyB(2.0, 3.0)

AbyB(3.0, 3.0)
```

Output:

-5.0

a/b result in 0

Finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after normal termination of try block or after try block terminates due to some exception.

Syntax:

```
try:

    # Some Code....


except:

    # optional block
```

```
    # Handling of exception (if required)

else:
    # execute if no exception

finally:
    # Some code .....(always executed)
```

Example:

- Python3

```
# Python program to demonstrate finally


# No exception Exception raised in try block
try:
    k =5//0   # raises divide by zero exception.
    print(k)


# handleszerodivision exception
exceptZeroDivisionError:
    print("Can't divide by zero")


finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

Output:

```
Can't divide by zero
This is always executed
```

Some of the common built-in exceptions are other than above mention exceptions are:

Exception	Description
IndexError	When the wrong index of a list is retrieved.
AssertionError	It occurs when the assert statement fails
AttributeError	It occurs when an attribute assignment is failed.
ImportError	It occurs when an imported module is not found.
KeyError	It occurs when the key of the dictionary is not found.
NameError	It occurs when the variable is not defined.
MemoryError	It occurs when a program runs out of memory.
TypeError	It occurs when a function and operation are applied in an incorrect type.

Reading and loading text, CSV data files using python

A CSV (Comma Separated Values) format is one of the most simple and common ways to store tabular data. To represent a CSV file, it must be saved with the **.csv** file extension.
Let's take an example:

If you open the above CSV file using a text editor such as sublime text, you will see:

```
SN, Name, City
1, Michael, New Jersey
2, Jack, California
```

As you can see, the elements of a CSV file are separated by commas. Here, , is a delimiter.
You can have any single character as your delimiter as per your needs.

> **Note:** The csv module can also be used for other file extensions (like: **.txt**) as long as their contents are in proper structure.

Working with CSV files in Python

While we could use the built-in `open()` function to work with CSV files in Python, there is a dedicated `csv` module that makes working with CSV files much easier.
Before we can use the methods to the `csv` module, we need to import the module first using:

```
import csv
```

Reading CSV files Using csv.reader()

To read a CSV file in Python, we can use the `csv.reader()` function. Suppose we have a `csv` file named **people.csv** in the current directory with the following entries.

Name	Age	Profession
Jack	23	Doctor

Miller 22 Engineer

Let's read this file using `csv.reader()`:

Example 1: Read CSV Having Comma Delimiter

```
import csv
withopen('people.csv', 'r') as file:
    reader = csv.reader(file)
for row in reader:
print(row)
```

Output

```
['Name', 'Age', 'Profession']
['Jack', '23', 'Doctor']
['Miller', '22', 'Engineer']
```

Here, we have opened the **people.csv** file in reading mode using:

```
withopen('people.csv', 'r') as file:
    .. .. ...
```

Then, the `csv.reader()` is used to read the file, which returns an iterable `reader` object.

The `reader` object is then iterated using a `for` loop to print the contents of each row.

In the above example, we are using the `csv.reader()` function in default mode for CSV files having comma delimiter.

However, the function is much more customizable.

Suppose our CSV file was using **tab** as a delimiter. To read such files, we can pass optional parameters to the `csv.reader()` function. Let's take an example.

Example 2: Read CSV file Having Tab Delimiter

```
import csv
withopen('people.csv', 'r',) as file:
   reader = csv.reader(file, delimiter = '\t')
for row in reader:
print(row)
```

Notice the optional parameter `delimiter = '\t'` in the above example.

The complete syntax of the `csv.reader()` function is:

```
csv.reader(csvfile, dialect='excel', **optional_parameters)
```

As you can see from the syntax, we can also pass the dialect parameter to the `csv.reader()` function. The `dialect` parameter allows us to make the function more flexible.

Writing CSV files Using csv.writer()

To write to a CSV file in Python, we can use the `csv.writer()` function.

The `csv.writer()` function returns a `writer` object that converts the user's data into a delimited string. This string can later be used to write into CSV files using the `writerow()` function. Let's take an example.

Example 3: Write to a CSV file

```
import csv
withopen('protagonist.csv', 'w', newline='') as file:
```

```
    writer = csv.writer(file)
writer.writerow(["SN", "Movie", "Protagonist"])
writer.writerow([1, "Lord of the Rings", "Frodo Baggins"])
writer.writerow([2, "Harry Potter", "Harry Potter"])
```

When we run the above program, a **protagonist.csv** file is created with the following content:

```
SN,Movie,Protagonist
1,Lord of the Rings,Frodo Baggins
2,HarryPotter,Harry Potter
```

In the above program, we have opened the file in writing mode.

Then, we have passed each row as a list. These lists are converted to a delimited string and written into the CSV file.

Example 4: Writing multiple rows with writerows()

If we need to write the contents of the 2-dimensional list to a CSV file, here's how we can do it.

```
import csv
csv_rowlist = [["SN", "Movie", "Protagonist"], [1, "Lord of the Rings", "Frodo Baggins"],
            [2, "Harry Potter", "Harry Potter"]]
withopen('protagonist.csv', 'w') as file:
    writer = csv.writer(file)
writer.writerows(csv_rowlist)
```

The output of the program is the same as in **Example 3**.

Here, our 2-dimensional list is passed to the `writer.writerows()` method to write the content of the list to the CSV file.

Example 5: Writing to a CSV File with Tab Delimiter

```
import csv
withopen('protagonist.csv', 'w') as file:
    writer = csv.writer(file, delimiter = '\t')
writer.writerow(["SN", "Movie", "Protagonist"])
writer.writerow([1, "Lord of the Rings", "Frodo Baggins"])
writer.writerow([2, "Harry Potter", "Harry Potter"])
```

Notice the optional parameter `delimiter = '\t'` in the `csv.writer()` function.

The complete syntax of the `csv.writer()` function is:

```
csv.writer(csvfile, dialect='excel', **optional_parameters)
```

Similar to `csv.reader()`, you can also pass dialect parameter the `csv.writer()` function to make the function much more customizable. To

Python csv.DictReader() Class

The objects of a `csv.DictReader()` class can be used to read a CSV file as a dictionary.

Example 6: Python csv.DictReader()

Suppose we have the same file **people.csv** as in **Example 1**.

Name	Age	Profession

Jack	23	Doctor
Miller	22	Engineer

Let's see how `csv.DictReader()` can be used.

```
import csv
withopen("people.csv", 'r') as file:
csv_file = csv.DictReader(file)
for row incsv_file:
print(dict(row))
```

Output

```
{'Name': 'Jack', ' Age': ' 23', ' Profession': ' Doctor'}
{'Name': 'Miller', ' Age': ' 22', ' Profession': ' Engineer'}
```

As we can see, the entries of the first row are the dictionary keys. And, the entries in the other rows are the dictionary values.

Here, `csv_file` is a `csv.DictReader()` object. The object can be iterated over using a `for` loop. The `csv.DictReader()` returned an `OrderedDict` type for each row. That's why we used `dict()` to convert each row to a dictionary.
Notice that, we have explicitly used the dict() method to create dictionaries inside the `for` loop.

```
print(dict(row))
```

Note: Starting from Python 3.8, csv.DictReader() returns a dictionary for each row, and we do not need to use `dict()` explicitly.

The full syntax of the `csv.DictReader()` class is:

```
csv.DictReader(file, fieldnames=None, restkey=None, restval=None, dialect='excel', *args, **kwds)
```

Python csv.DictWriter() Class

The objects of `csv.DictWriter()` class can be used to write to a CSV file from a Python dictionary.

The minimal syntax of the `csv.DictWriter()` class is:

```
csv.DictWriter(file, fieldnames)
```

Here,

- `file` - CSV file where we want to write to
- `fieldnames` - a `list` object which should contain the column headers specifying the order in which data should be written in the CSV file

Example 7: Python csv.DictWriter()

```
import csv

withopen('players.csv', 'w', newline='') as file:
    fieldnames = ['player_name', 'fide_rating']
    writer = csv.DictWriter(file, fieldnames=fieldnames)

writer.writeheader()
writer.writerow({'player_name': 'Magnus Carlsen', 'fide_rating': 2870})
writer.writerow({'player_name': 'Fabiano Caruana', 'fide_rating': 2822})
writer.writerow({'player_name': 'Ding Liren', 'fide_rating': 2801})
```

The program creates a **players.csv** file with the following entries:

```
player_name,fide_rating
Magnus Carlsen,2870
```

```
Fabiano Caruana,2822
Ding Liren,2801
```

The full syntax of the `csv.DictWriter()` class is:

```
csv.DictWriter(f, fieldnames, restval='', extrasaction='raise', dialect='excel', *args, **kwds)
```

Using the Pandas library to Handle CSV files

Pandas is a popular data science library in Python for data manipulation and analysis. If we are working with huge chunks of data, it's better to use pandas to handle CSV files for ease and efficiency.

Before we can use pandas, we need to install it.
Once we install it, we can import Pandas as:

```
import pandas as pd
```

To read the CSV file using pandas, we can use the `read_csv()` function.

```
import pandas as pd
pd.read_csv("people.csv")
```

Here, the program reads **people.csv** from the current directory.

To write to a CSV file, we need to call the `to_csv()` function of a DataFrame.

```
import pandas as pd

# creating a data frame
df = pd.DataFrame([['Jack', 24], ['Rose', 22]], columns = ['Name', 'Age'])

# writing data frame to a CSV file
df.to_csv('person.csv')
```

Here, we have created a DataFrame using the `pd.DataFrame()` method. Then, the `to_csv()` function for this object is called, to write into **person.csv**.

Using modules from python standard library and third-party libraries

The Python Standard Library is a collection of script modules accessible to a Python program to simplify the programming process and removing the need to rewrite commonly used commands. They can be used by 'calling/importing' them at the beginning of a script.

The following are among the most important:

- time
- sys
- os
- math
- random
- pickle
- urllib
- re
- cgi
- socket

A Python library is a collection of related modules. It contains bundles of code that can be used repeatedly in different programs. It makes Python Programming simpler and convenient for the programmer. As we don't need to write the same code again and

again for different programs. Python libraries play a very vital role in fields of Machine Learning, Data Science, Data Visualization, etc.

Python standard library

The Python Standard Library contains the exact syntax, semantics, and tokens of Python. It contains built-in modules that provide access to basic system functionality like I/O and some other core modules. Most of the Python Libraries are written in the C programming language. The Python standard library consists of more than 200 core modules. All these work together to make Python a high-level programming language. Python Standard Library plays a very important role. Without it, the programmers can't have access to the functionalities of Python. But other than this, there are several other libraries in Python that make a programmer's life easier.

Use of Libraries in Python Program

As we write large-size programs in Python, we want to maintain the code's modularity. For the easy maintenance of the code, we split the code into different parts and we can use that code later ever we need it. In Python, *modules* play that part. Instead of using the same code in different programs and making the code complex, we define mostly used functions in modules and we can just simply import them in a program wherever there is a requirement. We don't need to write that code but still, we can use its functionality by importing its module. Multiple interrelated modules are stored in a library. And whenever we need to use a module, we import it from its library. In Python, it's a very simple job to do due to its easy syntax. We just need to use **import**.
Let's have a look at exemplar code:

- Python3

```
# Importing math library

import math


A =16

print(math.sqrt(A))
```

Output

4.0

Here in the above code, we imported the math library and used one of its methods i.e. sqrt (square root) without writing the actual code to calculate the square root of a number. That's how a library makes the programmers' job easier. But here we needed only the sqrt method of math library, but we imported the whole library. Instead of this, we can also import specific items from a library module.

Importing specific items from a library module

As in the above code, we imported a complete library to use one of its methods. But we could have just imported "sqrt" from the math library. Python allows us to import specific items from a library.

Let's look at an exemplar code :

- Python3

```
# Importing specific items

frommath importsqrt, sin


A =16

B =3.14

print(sqrt(A))

print(sin(B))
```

Output

4.0

0.0015926529164868282

In the above code, we can see that we imported only "sqrt" and "sin" methods from the math library.

Debugging Python Code

Debugging means the complete control over the program execution. Developers use debugging to overcome program from any bad issues. So debugging is a healthier

process for the program and keeps the diseases bugs far away. Python also allows developers to debug the programs using pdb module that comes with standard Python by default. We just need to import pdb module in the Python script. Using pdb module, we can set breakpoints in the program to check the current status. We can Change the flow of execution by using jump, continue statements . Let's understand debugging with a Python program.

Example:

- Python3

```
# Program to print Multiplication

# table of a Number

n=5

forx inrange(1,11) :

  print( n , '*', x , '=', n*x )
```

Output:

```
5 * 1 = 5

5 * 2 = 10

5 * 3 = 15

5 * 4 = 20

5 * 5 = 25

5 * 6 = 30

5 * 7 = 35

5 * 8 = 40

5 * 9 = 45

5 * 10 = 50
```

This program simply print multiplication table but now we need to debug the loop steps using set_trace() function call to pdb module .

Example:

- Python3

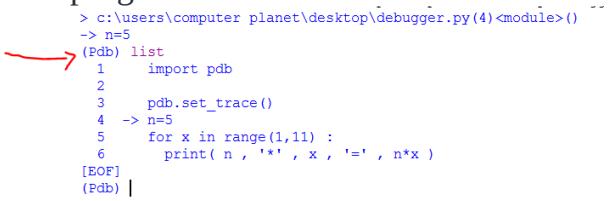
```
# Python Program to print Multiplication Table
# We want to debug the for loop so we use
# set_trace() call to pdb module

import pdb


# It means , the Start of Debugging Mode
pdb.set_trace()

n=5
for x in range(1,11) :
  print( n , '*', x , '=', n*x )
```

Output:

```
--Return--
> <ipython-input-16-301815242d6c>(8)<module>()->None
-> pdb.set_trace() # It means , the Start of Debugging Mode
(Pdb)
```

Basic Commands to use Python Debugger

list command to see entire program .



list command

list 3 , 6 to see the program lines from 3 to 5 only .

```
(Pdb) list 3,6
  3     pdb.set_trace()
  4  -> n=5
  5     for x in range(1,11) :
  6       print( n , '*' , x , '=' , n*x )
(Pdb)
```

list lines command in pdb debugger

break command to stop the program execution at particular line .

```
(Pdb) break 5
Breakpoint 1 at c:\users\computer planet\desktop\debugger.py:5
(Pdb)
```

break command in pdb debugger

continue command to see the next step in the loop .

```
(Pdb) continue
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb) continue
5 * 1 = 5
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb) continue
5 * 2 = 10
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb) continue
5 * 3 = 15
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb) continue
5 * 4 = 20
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb)
```

jump command allows us to go on any particular line in the program .

```
(Pdb) jump 11
*** Jump failed: line 11 comes after the current code block
(Pdb) jump 5
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb)
```

jump command in pdb debugger

pp command to see variable value at the current position in the program .

```
(Pdb) continue
5 * 4 = 20
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb) jump 11
*** Jump failed: line 11 comes after the current code block
(Pdb) jump 5
> c:\users\computer planet\desktop\debugger.py(5)<module>()
-> for x in range(1,11) :
(Pdb) pp x
4
(Pdb)
```

pp command in pdb debugger

disable command to disable the current line output and we can use continue command to skip this line in program. We use quit or exit command to come outside the debugging mode .

Conclusion

Debugging helps developers to analyze programs line by line. Developers see the every interpreted line by using debugging mode in programs. Python comes with by default debugger that is easy to import and use. So it is good to start with debugger when confused about execution of large loops, current variable values and all .

2.4Python Classes

Introduction to object-oriented programming (OOPs) using python

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

A parrot is an object, as it has the following properties:

- name, age, color as attributes
- singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

```
class Parrot:

  pass
```

Here, we use the `class` keyword to define an empty class `Parrot`. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, `obj` is an object of class `Parrot`.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

Creating Class and Object in Python

Example 1: Creating Class and Object in Python

```
classParrot:

# class attribute
    species = "bird"

# instance attribute
def__init__(self, name, age):
        self.name = name
self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
Run Code
```

Output

```
Blu is a bird
Woo is also a bird
```

```
Blu is 10 years old
Woo is 15 years old
```