INDEX

Exception Handling

-->In any programming language there are 2-types of errors are possible
1).Syntax Errors
2).Runtime Errors

1).Syntax Errors:

The errors which are occurs because of invalid syntax are called as syntax errors.

Ex:

```
x=10
if x==10
	print("Hello")
SyntaxError: invalid syntax
```

Ex:

```
print "Hello"
SyntaxError: Missing parentheses in call to 'print'.
```

Note:
Programmer is responsible to correct these errors. Once all the syntax errors are corrected then only program execution will be started.

2).Runtime Errors:

-->Also called as exceptions.
-->While executing the program if something goes wrong because of end user input or programming logic or memory peoblem etc we will get Runrime Errors.

Ex:
```
print(10/0)==>ZeroDivisionError: division by zero

print(10/"ten")==>TypeError: unsupported operand type(s) for /: 'int' and 'str'

x=int(input("Enter Number:"))
print(x)
D:\pythonclasses>py test.py
Enter some umber:ten
ValueError: invalid literal for int() with base 10: 'ten'
```

Note: Exception handling concept applicable for Runtime Errors but not for syntax errors.

What is an Exception:
====================

An unwanted and unexpected event that distrubs normal flow of the program is called as an exception.

Ex:

ZeroDivisionError
TypeError
ValueError
FileNOtFoundError
EOFError
TyrePuncheredError
SleepingError

-->It is highly recommended to handle exceptions. The main objective of exception handling is Graceful Termination of the program(i.e we should not block our resources and we should not miss anything).

-->Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Ex:

For example our programming requirement is reading data from the remote file locating at london. At the runtime if london file is not available then the program should not be terminated abnormally. We have to provide local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

```
try:
        read data from remote file locating at london
except FileNotFoungError:
        use local file and continue rest of the program normally.
```

Q:What is an Exception?
Q:What is purpose of Exception Handling?
Q:What is the meaning of Exception Handling?

Default Exception Handling in Python:

-->Every Exception in python is an object. For every exception type the corresponding classes are available.

-->Whenever an exception occurs PVM will create the corresponding exception object and will check for handling code. If the handling code is not available then python interpreter terminates the program abnormally and prints corresponding exception information to the console.

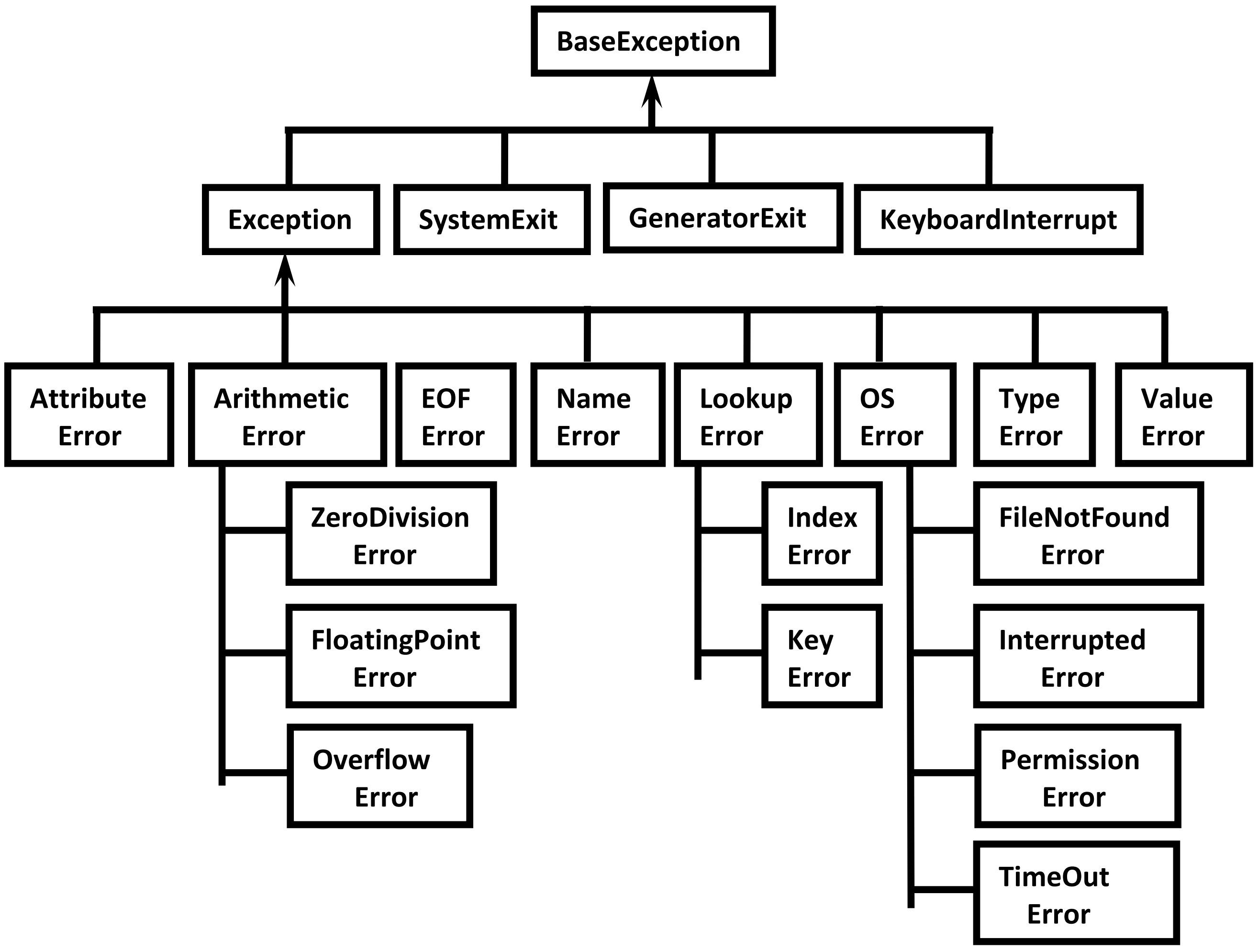
-->The rest of the program won't be executed.

Ex:

```
1) print("Hello")
2) print(10/0)
3) print("Hi")
```

```
o/p:D:\pythonclasses>py test.py
Hello
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

Python's Exception Hierarchy:



-->Every Exception in python is a class.

-->All exception classes are child classes of BaseException i.e every exception class extends BaseException either directly or indirectly. Hence BaseException acts as root for python Exception Hierarchy.

-->Most of the times being a programmer we have to concentrate exception and its child classes.

Customized Exception Handling by using try-except:

-->It is highly recommended to handle exceptions.

-->The code which may raise exception is called as risky code and we have to take risky code inside the try block. The corresponding handling code we have to take inside except block.

Syn:

```
try:
	Risky code
except XXXXXX:
	Handling code/alternative code
```

without try-except:

```
print("stmt-1")
print(10/0)
print("stmt-3")
```

```
o/p:D:\pythonclasses>py test.py
stmt-1
ZeroDivisionError: division by zero
```

-->Abnormal termination/Non-Graceful Termination

with try-except:

```
print("stmt-1")
try:
	print(10/0)
except ZeroDivisionError:
	print(10/2)
print("stmt-3")
```

```
o/p:
stmt-1
5.0
stmt-3
```
-->Normal Termination/Graceful Termination.

Control flow in try-except block:

Ex:

```
try:
        stmt-1
        stmt-2
        stmt-3
except XXX:
        stmt-4
stm-5
```

case-1: If there is no Exception.
    1,2,3,5 Noraml Termination.
Case-2: If an exception is raised at stmt-2 and corresponding except block is matched.
    1,4,5 Normal Termination.
Case-3: If an exception is raised at stmt-2 and corresponding except block is not matched.
    1, Abnormal Termination.
Case-4: If an exception is raised at stmt-4 or stmt-5 then it always abnormal termination.

Conclusions:
----
1).within the try block if anywhere exception raised then rest of the try block won't be executed eventhough we handled that exception. Hence we have to take only risky code inside the try block and length of the try block should be as less as possible.

2).In addition to try block, there may be a chance of raising exception inside except block and finally block also.

3).If any statement which is not part of try block raises an exception then it is always abnormal termination.

How to print exception information:

```
try:
        print(10/0)
except ZeroDivisionError as msg:
        print("exception raised and its description is :",msg)
```

o/p:
```
exception raised and its description is : division by zero
```

try with multiple except blocks:

-->The way of handling exception is varied exception to exception. Hence for every exception type a separate except block we have to provide. i.e try with multiple except block is possible and recommended.
-->If try with multiple except blocks are available then based on raised exception the corresponding except block will be executed.

Ex:

```
try:
    x=int(input("Enter First Number:"))
    y=int(input("Enter Second Number:"))
    print(x/y)
except ZeroDivisionError:
    print("can't devide with zero")
except ValueError:
    print("please provide int value only")
```

```
o/p:D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:2
5.0

o/p:D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:0
can't devide with zero

o/p:D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:ten
please provide int value only
```

-->If try multiple except blocks available then the order of these except blocks is important. Python interpreter will always consider top to bottom until matched except block identified.

Ex:

```
try:
        x=int(input("Enter First Number:"))
        y=int(input("Enter Second Number:"))
        print(x/y)
except ArithmeticError:
        print("ArithmeticError")
except ZeroDivisionError:
        print("can't devide with zero")
```

```
o/p:D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:0
ArithmeticError
```

Ex:

```
try:
        x=int(input("Enter First Number:"))
        y=int(input("Enter Second Number:"))
        print(x/y)
except ZeroDivisionError:
        print("can't devide with zero")
except ArithmeticError:
        print("ArithmeticError")
```

```
o/p:D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:0
can't devide with zero
```

Single except block that can handle multiple exceptions:

-->We can write single except block that can be handle multiple diferent types of exceptions.

```
except(Exception1,Exception2,Exception3.....)
except(Exception1,Exception2,Exception3.....) as msg
```

-->Parenthesis are mandatory ans this group of exceptions internally considered as tuple.

Ex:

```
try:
        x=int(input("Enter First Number:"))
        y=int(input("Enter Second Number:"))
        print(x/y)
except(ZeroDivisionError,ValueError) as msg:
        print("plz provide valid numbers only the problem is:",msg)
```

```
o/p:D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:0
plz provide valid numbers only the problem is: division by zero

D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:ten
plz provide valid numbers only the problem is: invalid literal for int() with base 10: 'ten'
```

Default except block:

-->We can use default except block to handle any type of exception.
-->In default except block generally we can print normal error messages.

Syn:
```
        except:
                statements
```
Ex:

```
try:
        x=int(input("Enter First Number:"))
        y=int(input("Enter Second Number:"))
        print(x/y)
except ZeroDivisionError:
        print("ZeroDivisionError: can't divide with zero")
except:
        print("DefaultExcept: plz provide valid input")
```

```
o/p:D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:0
ZeroDivisionError: can't divide with zero
```

```
D:\pythonclasses>py test.py
Enter First Number:10
Enter Second Number:ten
DefaultExcept: plz provide valid input
```

Note:
If try with multiple except blocks available then the default except block should be last, otherwise we will get a Syntax Error.

Ex:

```
try:
        print(10/0)
except:
        print("Default Except")
except ZeroDivisionError:
        print("ZeroDivisionError")
```

o/p:SyntaxError: default 'except:' must be last

Note:
The folowing are various combinations of except block.
1.except ZeroDivisionError:
2.except ZeroDivisionError as msg:
3.except(ZeroDivisionError,ValueError):
4.except(ZeroDivisionError,ValueError) as msg:
5.except:

finally block:

-->It is not recommended to maintain clean up code(Resource Deallocation code or Resource Releasing code) inside the try block because there is no guarantee for the execution of every statement inside try block always.
-->It is not recommended to maintain cleanup code inside except block, because if there is no exception then except block won't be executed.
-->Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.
-->Hence the main purpose of finally block is to maintain clean up code.

Syn:

```
try:
        Risky Code
except:
        Handling Code
finally:
        Cleanup Code
```

-->The speciality of finally block is it will be executed always whether the exception raised or not raised whether the exception handled or not handled.

Case-1: If there is no exception

```
try:
        print("try")
except:
        print("except")
finally:
        print("finally")
```

o/p:

```
try
finally
```

Case-2: If there is an exception raised but handled.

```
try:
        print("try")
        print(10/0)
except ZeroDivisionError:
        print("except")
finally:
        print("finally")
```

o/p:

```
try
except
finally
```

Case-3: If there is an exception raised but not handled.

```
try:
            print("try")
            print(10/0)
except NameError:
            print("except")
finally:
            print("finally")
```

o/p:
```
try
finally
ZeroDivisionError: division by zero
```
Note:
There is only one situation where finally block won't be executed ie whenever we are using os._exit(0) function.

-->Whenever we are using os._exit(0) function then PVM itself will be shutdown. In this particular case finally won't be executed.

Ex:

```
import os
try:
            print("try")
            os._exit(0)
except NameError:
            print("except")
finally:
            print("finally")
```

o/p:
```
try
```

Note:
os._exit(0):
-->where o represents status code and it indicates normal termination.
-->There are multiple status codes are possible.

Control flow of try except finally:

```
try:
        stmt-1
        stmt-2
        stmt-3
except:
        stmt-4
finally:
        stmt-5
stmt-6
```

Case-1: If there is no exception

    1,2,3,5,6 Normal Termination

Case-2: If an exception raised in stmt-2 and the corresponding except block matched.

    1,4,5,6 Normal Termination

Case-3:If an exception raised in stmt-2 but the corresponding except block not matched.

    1,5 Abnormal Termination.

Case-4: An exception raised at stmt-5 or stmt-6 then it is always abnormal termination

Case-5: An exception raised at stmt-4 then it always abnormal termination but before that finally block will be executed.

Nested try-except-finally block:

-->We can take try-except-finally blocks inside try or except or finally blocks i.e nesting of try-except-finally is possible.

Syn:

```
try:
        ----------
        ----------
        ----------
        try:
                --------
                --------
                --------
        except:
                --------
                --------
                --------
```

```
except:
        ---------
        ---------
        ---------
```

-->General risky code we have to take inside outer try block and too much risky code we have to take inner try block. Inside inner try block if any exception raised then inner except block is responsible to handle. If unable to handle then outer except block is responsible to handle.

Ex:

```
try:
        print("Outer try block")
        try:
                print("Innner try block")
                print(10/0)
        except ZeroDivisionError:
                print("Inner except block")
        finally:
                print("Inner finally block")
except:
        print("Outer except block")
finally:
        print("Outer finally block")
```

```
o/p:D:\pythonclasses>py test.py
Outer try block
Innner try block
Inner except block
Inner finally block
Outer finally block
```

Control flow in nested try-except-finally:

```
try:
        stmt-1
        stmt-2
        stmt-3
        try:
                stmt-4
                stmt-5
                stmt-6
```

```
        except X:
                stmt-7
        finally:
                stmt-8
        stmt-9
except Y:
        stmt-10
finally:
        stmt-11
stmt-12
```

Case-1:If there is no exception
1,2,3,4,5,6,8,9,11,12 normal termination.

Case-2:If an exception raised on stmt-2 and the corresponding except block matched.
1,10,11,12 normal termination.
Case-3:If an exception raised in stmt-2 and the corresponding except block not matched.
1,11 abnormal termination.

Case-4:If an exception raised at stmt-5 and inner except block matched.
1,2,3,4,7,8,9,11,12 normal termination.

Case-5:If an exception raised at stmt-5 and inner except block not matched but outer except block matched.
1,2,3,4,8,10,11,12 normal termination.

Case-6:If an exception raised at stmt-5 and inner except block and outer except block not matched.
1,2,3,4,8,11 abnormal termination.

Case-7:If an exception raised at stmt-7 and corresponding except block matched.
1,2,3,.,.,.,8,10,11,12 normal termination.

Case-8:If an exception raised at stmt-7 and corresponding except block not matched.
1,2,3,.,.,.,8,11 abnormal termination.

Case-9:If an exception raised at stmt-8 and corresponding except block matched.
1,2,3,.,.,.,.,10,11,12 normal termination.

Case-10:If an exception raised at stmt-8 and corresponding except block not matched.
1,2,3,.,.,.,.,11 abnormal termination.

Case-11:If an exception raised at stmt-9 and corresponding except block matched.
123,.,.,.,.,8,10,11,12 normal termination.

Case12:If an exception raised at stmt-9 and corresponding except block not matched.
1,2,3,.,.,.,.,8,11 abnormal termination.

Case-13:If an exception raised at stmt-10 then it always abnormal termination but before abnormal termination finally(stmt-11) will be executed.

Case-14:If an exception raised at stmt-11 or stmt-12 then it always abnormal termination

Note: If the control entered into try block then compulsary finally block will be executed. If the control not entered into try block then finally block won't be executed.

else block with try-except-finally:

-->We can use else block with try-except-finally block.
-->else block will be executed if and only if there are no exceptions inside try block.

Syn:

```
try:
        Risky code
except:
        wiil be executed if exception inside try
else:
        will be executed if there is no exception inside try
finally:
        will be executed whether exception raised or not raised handled or not handled.
```

Ex:

```
try:
        print("try")
        print(10/0)-->1
except:
        print("except")
else:
        print("else")
finally:
        print("finally")
```

-->If we comment line-1 then else block will be executed bcoz there is no exception inside try. In this case the output is:

```
try
else
finally
```

-->If we are not commenting line-1 then else block won't be executed bcoz there is an exception inside try block. In this case output is:

```
try
except
finally
```

Various possible combinations of try-except-finally:
----

1).Whenever we are writting try block, compulsary we should write except or finally block. i.e witout except or finally block we can't write try block.

2).Whenever we are writting except block, compulsory we should write try block. i.e except without try is always invalid.

3).Whenever we are writting finally block, compulsory we should write try block. i.e finally without try is always invalid.

4).We can write multiple except blocks for the same try, but we can't write multiple finally blocks for the same try.

5).Whenever we are writting else block compulsory except block should be there. i.e without except block we can't write else block.

6).in try-except-finally order is important.

7).We can define try-except-else-finally inside try,except,finally blocks. i.e nesting of try-except-else-finally is always possible.

Possible combinations:
----

1)

```
try:
        print("Hello....")
```

2)

```
except:
        print("Hi.....")
```

3)

```
else:
        print("Hi.....")
```

4)

```
finally:
        print("Hi.......")
```

5)

```
try:
        print("try")
except:
        print("except")
```

6)

```
try:
        print("try")
        print(10/0)
finally:
        print("finally")
```

7)

```
try:
        print("try")
except:
        print("except")
else:
        print("else")
```

8)

```
try:
        print("try")
else:
        print("else")
```

9)

```
try:
    print("try")
else:
    print("else")
finally:
    print("finally")
```

10)

```
try:
    print("try")
except XXX:
    print("except1")
except YYY:
    print("except2")
```

11)

```
try:
    print("try")
except:
    print("except")
else:
    print("else")
else:
    print("else")
```

12)

```
try:
    print("try")
except:
    print("except")
finally:
    print("finally")
finally:
    print("finally")
```

13)

```
try:
    print("try")
print("Hello....")
```

```
except:
    print("except")
```

14)

```
try:
    print("try")
except:
    print("except")
print("Hello")
except:
    print("except")
```

15)

```
try:
    print("try")
except:
    print("except")
print("Hello")
finally:
    print("finally")
```

16)

```
try:
    print("try")
except:
    print("except")
print("Hello")
else:
    print("else")
```

17)

```
try:
    print("try")
except:
    print("except")
try:
    print("try")
except:
    print("except")
```

18)

```
try:
    print("try")
except:
    print("except")
try:
    print("try")
finally:
    print("finally")
```

19)

```
try:
    print("try")
except:
    print("except")
if 10>20:
    print("if")
else:
    print("else")
```

20)

```
try:
    print("try")
    try:
        print("inner try")
    except:
        print("inner except")
    finally:
        print("inner finally")
except:
    print("except")
```

21)

```
try:
    print("try")
except:
    print("except")
    try:
        print("inner try")
    except:
        print("inner except")
```

```
    finally:
        print("inner finally")
```

22)

```
try:
    print("try")
except:
    print("except")
finally:
    print("finally")
    try:
        print("inner try")
    except:
        print("inner except")
    finally:
        print("inner finally")
```

23)

```
try:
    print("try")
except:
    print("except")
try:
    print("try")
else:
    print("else")
```

24)

```
try:
    print("try")
    try:
        print("inner try")
except:
    print("except")
```

25)

```
try:
    print("try")
else:
    print("else")
except:
```

```
        print("except")
finally:
        print("finally")
```

Types of Exceptions:

-->In python there are 2-types of exceptions are possible.
1).Predefined Exception
2).User Defined Exception

1).Predefined Exception:

-->Also known as inbuilt exceptions.
-->The exceptions which are raised automatically by python virtual machine whenever a partucular event occurs, are called as pre defined exceptions.

Ex:Whenevr we are trying to perfrom Division by zero, automatically python will raise ZeroDivisionError.

```
print(10/0)==>ZeroDivisionError
```

Ex: Whenevr we are trying to convert input value to int type and if input value is not int value python will raise ValueError automatically.

```
x=int("ten")==>ValueError
```

2).User Defined Exceptions:

-->Also known as Cistomized Exceptions or programatic Exceptions.
-->Some times we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions are called as user defined exceptions or customized exceptions.
-->Programmer is responsible to define these exceptions and python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.

Ex:

InSufficientFundsException
InvalidInputException
TooYoungException
TooOldException

How to Define and Raised Customized Exceptions:
--
-->Every exception in python is a class that extends Exception class either directly or indirectly.

Syn:

```
class classname(parent Exception class name):
        def __init__(self,arg):
                self.msg=arg
```

Ex:

```
class TooYoungException(Exception):
        def __init__(self,arg):
                self.msg=arg
class TooOldException(Exception):
        def __init__(self,arg):
                self.msg=arg
age=int(input("Enter Age:"))
if age>60:
        raise TooYoungException("Plz wait some more time you will get best match soon!!!!")
elif age<18:
        raise TooOldException("Your age already crossed marriage age...no chance of getting marriage")
else:
        print("You will get match details soon by email......")
```

Note:
raise keyword is best suitable for customized exceptions but not for pre defined exceptions.

Logging Module

Logging the Exceptions:
=====================
-->It is highly recommended to store complete application flow and exceptions information to a file. This process is called as logging.
-->The main advantages of logging are:
1).We can use log files while perfroming debugging.
2).We can provide statsics like number of requests per day etc...
-->To implement logging, Python provides one inbuilt module logging.

logging levels:

-->Depending on type of information, logging data is divided according to the following 6-levels in python.

1).CRITICAL==>50==>Represents a very seroius problem that needs hogh attention.
2).ERROR==>40==>Represents a seroius error.
3).WARNING==>30==>Represents a warning message, some caution needed. It is an alert to the programmer.
4).INFO==>20==>Represents a message with some important information.
5).DEBUG==>10==>Represents a message with debugging iformation.
6).NOTSET==>0==>Represents that level is not set.

How to implement logging:

-->To perform logging, first we required to create a file to store messages and we have to specify which level messages we have to store.
-->We can do this by using basicConfig() function of logging module.
-->logging.basicConfig(filename='log.txt',level=logging.WARNING)
-->The above line will create a file log.txt and we can store either WARNING level or higher level messages to that file.
-->After creating log file, we can write messages to that file by using the following methods.

```
logging.debug(message)
logging.info(message)
logging.warning(message)
logging.error(message)
logging.critical(message)
```

Scenario:

w.a.p to create a log file and write WARNING and higher level messages.

```
import logging
logging.basicConfig(filename='log.txt',level=logging.WARNING)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

log.txt:

```
WARNING:root:This is warning message
ERROR:root:This is error message
CRITICAL:root:This is critical message
```

Note:

In the above program only WARNING and higher level messages will be written to log file. If we set level as DEBUG then all the messages will be written to log file.

```
import logging
logging.basicConfig(filename='log.txt',level=logging.DEBUG)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

log.txt:

```
DEBUG:root:This is debug message
INFO:root:This is info message
WARNING:root:This is warning message
ERROR:root:This is error message
CRITICAL:root:This is critical message
```

Note:

We can perform log messages to include date and time, ip address of the client etc at advanced level

How to write python program exceptions to log file:

-->By using the following function we can write exceptions information to th elog file.

logging.exception(msg)

Ex:

Python program to write exception information to the log file.

```
import logging
logging.basicConfig(filename='mylog.txt',level=logging.INFO)
logging.info("A new request came:")
try:
        x=int(input("Enter first number:"))
        y=int(input("Enter second number:"))
        print(x/y)
except ZeroDivisionError as msg:
        print("can't divide with zero")
        logging.exception(msg)
except ValueError as msg:
        print("Enter only int values")
        logging.exception(msg)
logging.info("Request processing completed")
```

```
D:\pythonclasses>py test.py
Enter first number:10
Enter second number:2
5.0

D:\pythonclasses>py test.py
Enter first number:10
Enter second number:0
can't divide with zero

D:\pythonclasses>py test.py
Enter first number:10
Enter second number:ten
Enter only int values
```

mylog.txt:

```
INFO:root:A new request came:
INFO:root:Request processing completed
INFO:root:A new request came:
ERROR:root:division by zero
```

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(x/y)
ZeroDivisionError: division by zero
INFO:root:Request processing completed
INFO:root:A new request came:
ERROR:root:invalid literal for int() with base 10: 'ten'
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    y=int(input("Enter second number:"))
ValueError: invalid literal for int() with base 10: 'ten'
INFO:root:Request processing completed
```

Assertions

Debugging python program by using assert keyword:
--
-->The process of identifying and fixing the bugs is called as debugging. Very common way of debugging is to use print() statement. But the problem with the print() statement is after fixing the bug, compulsory we have to delete the extra added print() statements, otherwise these will be executed at the runtime which creates performance problems and disturbs console output.
-->To overcome this problem we should go for assert statement. The main advantage of assert statement over print() statement is after fixing bugs we are not required to delete assert statements. Based on our requirement we can enable or disable assert statements.
-->Hence the main purpose of assertionis to perform debugging, usually we can perform debugging either in development or in test environments but not in production environmnet. Hence assertion concept is applicable only for dev and test environments but not for production environment.

Types of assert statements:

-->There are 2-types of assert statements.
1).Simple Version
2).Augmented Version

1).Simple Version:
assert condition_expression

2).Augmented Version:
assert condition_expression, message
-->conditional_expression will be evaluated and if it is true then the program will be continued.
-->If it is false then the program will be terminated by raising AssertionError.
-->By seeing AssertionError, programmer can analyze the code and can fix the problem.

Ex:

```
def squareit(x):
        return x**x
assert squareit(2)==4,"The square of 2 should be 4"
assert squareit(3)==9,"The square of 3 should be 9"
```

```
5) assert squareit(4)==16,"The square of 4 should be 16"
6) print(squareit(2))
7) print(squareit(3))
8) print(squareit(4))
```

```
o/p:D:\pythonclasses>py test.py
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    assert squareit(4)==4,"The square of 4 should be 16"
AssertionError: The square of 4 should be 16
```

```
1) def squareit(x):
2)          return x*x
3) assert squareit(2)==4,"The square of 2 should be 4"
4) assert squareit(3)==9,"The square of 3 should be 9"
5) assert squareit(4)==16,"The square of 4 should be 16"
6) print(squareit(2))
7) print(squareit(3))
8) print(squareit(4))
```

o/p:

```
4
9
16
```

Note: To disable assert statements py -O text.py

Exception Handling Vs assertions:

-->Assertions concept can be used to alert the programmer to resolve development time errors.

-->Exception Handling can be used to handle run time errors.

File Handling

-->As part of the programming requirement, we have to store our data permanently for future purpose. For this requirement we should go for files.
-->Files are are very common permenant storage areas to store our data.

Types of Files:

-->There are 2-tyes of files.
    1).Text Files
    2).Binary Files

1).Text Files:

-->Usually we can use text files to store characters data.
Ex: abc.txt

2).Binary Files:

-->Usually we can use binary files to store binary data like images, video files,audio files etc...

Opening a file:

-->Before performing any operation(like read or write) on the file, first we have to open that file. For this we should use python's inbuilt function open().
-->But the time of the open, we have to specify mode, which represents the purpose of opening file.

Syn: f=open(filename,mode)

-->The allowed modes in python are

1).r-->Open an exidting file for read operations. The filepointer is positioned at the beginning ofthe file. If the specified file does not exist then we will get FileNotFoundError. This is default mode.

2).w-->Open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create the file.

3).a-->Open an existing file for append operation. It won't override existing data. If the specified file is not already available then this mode will create a new file.

4).r+-->To read and write data into the file. The prevoius data in the file will not be deleted. The file pointer is placed at the beggining of the file.

5).w+-->To write andread data. It will override existing data.

6).a+-->To append and read data from the file. It won't override existing data.

7).x-->To open a file in exclusive creating mode for write operation. Ifthe file already exists then we will get FileExistError.

Note:

All the baove modes are applicable for text files. Ifthe above modes suffixed with 'b' then these represents for binary files.

Ex:rb,wb,ab,r+b,w+b,a+b,xb

Ex:f=open("abc.txt","w")
-->We are pening abc.txt file for writting data.

Closing a File:

-->After completing our operations on the file, it is highly recommended to close the file. For this we have to use close() function.

```
f.close()
```

Various properties of File Object:

-->Once we opened a file and we got file object, we can get various details related to that file byusing its properties.

name-->Name of the opened file.
mode-->Mode in which the file is opened.
closed-->returns boolean value indicates that file is closed or not.
readable()-->Returns a boolean value indicates that whether file is readable or not.
writable()-->Returns a boolean value indicates that whether file is writable or not.

Ex:

```
f=open("abc.txt","w")
print("File Name:",f.name)
print("File Mode:",f.mode)
print("Is File Readable:",f.readable())
print("Is File Writable:",f.writable())
print("Is File Closed:",f.closed)
f.close()
print("Is File Closed:",f.closed)
```

```
o/p:D:\pythonclasses>py test.py
File Name: abc.txt
File Mode: w
Is File Readable: False
Is File Writable: True
Is File Closed: False
Is File Closed: True
```

Writting data to text file:

-->We can write character data to the text files by using the following 2-methods.
1).write(str)
2).writelines(list of lines)

Ex:

```
f=open("abcd.txt","w")
f.write("Durga\n")
f.write("Software\n")
f.write("Solutions\n")
print("Data written to the file successfully")
f.close()
```

abcd.txt:

```
Durga
Software
Solutions
```

Note: In the above program, data present in the file will be overridden everytime if we are run the program. Instead of overriding if we want to append operation then we should open the file as follows. f=open("abcd.txt","a")

Ex-2:

```
f=open("abcd.txt","w")
list=["sunny\n","bunny\n","chinny\n","pinny"]
f.writelines(list)
print("List of lines written to the file successfully")
f.close()
```

abcd.txt:

```
sunny
bunny
chinny
pinny
```

Note:

while writting data by using write() methods, compulsory we have to provide line separator(\n), otherwise total data should be written in a single line.

Reading character data from the text files:

-->We can read character data from the text file by using following methods.

read()==>To read all data from the file.

read(n)==>To read 'n' characters from the file.

readline()==>To read only one line.

readlines()==>To read all lines into a list.

Ex-1: To readtotal data from the file

```
f=open("abcd.txt","r")
data=f.read()
print(data)
f.close()
```

o/p:

```
sunny
bunny
chinny
pinny
```

Ex-2:To read only first 10 characters

```
f=open("abcd.txt","r")
data=f.read(10)
print(data)
f.close()
```

o/p:
```
sunny
bunn
```

Ex-3:To read data line by line

```
f=open("abcd.txt",'r')
line1=f.readline()
print(line1,end='')
line2=f.readline()
print(line2,end='')
line3=f.readline()
print(line3,end='')
f.close()
```

o/p:
```
sunny
bunny
chinny
```

Ex-4:To read all lines into list:

```
f=open("abcd.txt","r")
lines=f.readlines()
print(type(lines))
for line in lines:
  print(line,end='')
f.close()
```

o/p:
```
D:\pythonclasses>py test.py
<class 'list'>
sunny
bunny
chinny
pinny
```

Ex-5:

```
f=open("abcd.txt","r")
print(f.read(3))
print(f.readline())
print(f.read(4))
print("Remaining Data")
print(f.read())
f.close()
```

```
o/p:D:\pythonclasses>py test.py
sun
ny

bunn
Remaining Data
y
chinny
pinny
```

The with statement:

-->The with statement can be used while opening a file. we acn use this to group file operation statements with a block.
-->The advantage of with statement is it will take care closing of file, after completing all operations automatically even in case of exceptions also, and we are not required to close explicitly.

Ex:

```
with open("abcd.txt","w") as f:
 f.write("Durrgs\n")
 f.write("Software\n")
 f.write("Solutions\n")
 print("Is file is closed:",f.closed)
print("Is file is closed:",f.closed)
```

```
o/p:D:\pythonclasses>py test.py
Is file is closed: False
Is file is closed: True
```

The tell() and seek() methods:
----

tell():

We can use tell() method to return current position of the cursor(file pointer) from beginning of the file.[can u please tell current position]

The position(index) of first character in file is zero just like string index.

Ex:

```
f=open("abc.txt","r")
print(f.tell())
print(f.read(3))
print(f.tell())
print(f.read(5))
print(f.tell())
```

abc.txt:

```
Durga
Software
Solutions
```

```
o/p:D:\pythonclasses>py test.py
0
Dur
3
ga
So
9
```

seek():

We can use seek() method to move cursor(file pointer)to specifed location.
[can u please seek the cursor to a particular location]

f.seek(offset,fromwhere)

offset represents the number of positions.

-->The allowed values for second attribute(from where) are:

0-->From beginning of file(default value)
1-->From current position
2-->From end of the file

Note:

Python-2 supports all 3-values but python-3 supports only zero

Ex:

```
data="All Students Are BAD"
f=open("abc.txt",'w')
f.write(data)
with open("abc.txt","r+") as f:
  text=f.read()
  print(text)
  print("The current position:",f.tell())
  f.seek(17)
  print("The current position:",f.tell())
  f.write("GEMS!!!!!")
  f.seek(0)
  text=f.read()
  print("Data After Modification:")
  print(text)
```

```
o/p:D:\pythonclasses>py test.py
All Students Are BAD
The current position: 20
The current position: 17
Data After Modification:
All Students Are GEMS!!!!!
```

How to check a particular file exists or not?

-->We can use os library to get information about files in our computer.
-->os module has path module, which cintains isfile() function to check whether a particular file exists or not?

Syn: **os.path.isfile(filename)**

Q:w.a.p to check whether the given file exist or not. If it is available then print its content.

```
import os,sys
fname=input("Enter File Name:")
if os.path.isfile(fname):
  print("File Exists:",fname)
  f=open(fname,'r')
else:
```

```
    print("File does not exist",fname)
    sys.exit(0)
print("The content of file is:")
data=f.read()
print(data)
```

```
o/p:D:\pythonclasses>py test.py
Enter File Name:a.txt
File does not exist a.txt

D:\pythonclasses>py test.py
Enter File Name:abc.txt
File Exists: abc.txt
The content of file is:
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
```

Note:

sys.exit(0)==>To exit system without executing rest of the program.
argument represents status code. 0 means normal termination and it is the default value.

Q: w.a.p to print the number of line, words and characterspresent in given file?

```
import os,sys
fname=input("Enter File Name:")
if os.path.isfile(fname):
    print("File Exists:",fname)
    f=open(fname,'r')
else:
    print("File does not exist",fname)
    sys.exit(0)
lcount=wcount=ccount=0
for line in f:
    lcount=lcount+1
    ccount=ccount+len(line)
    words=line.split()
    wcount=wcount+len(words)
print("The Number Of Lines:",lcount)
print("The Number Of Characers:",ccount)
```

```
print("The Number Of Words:",wcount)
```

o/p:

```
D:\pythonclasses>py test.py
Enter File Name:abc.txt
File Exists: abc.txt
The Number Of Lines: 6
The Number Of Characers: 161
The Number Of Words: 24
```

abc.txt:

```
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
All Students Are GEMS!!!!!
```

Handling Binary Data:

->It is very common requirement to read or write binary data like images, videos files,audio files etc....

Q:w.a.p to read image file and write to a new image file.

```
f1=open("sunny.jpg",'rb')
f2=open("newpic.jpg",'wb')
bytes=f1.read()
f2.write(bytes)
print("New image is available with the name:newpic.jpg")

```

o/p:

```
D:\pythonclasses>py test.py
New image is available with the name:newpic.jpg
```

Handling CSV Files:

CSV==>Comma Separated Values
-->As the part of programming, it is very common requirement to write and read data wrt csv files. Python provides csv module to handle csv files.

<u>**writting data to csv file:**</u>

```
import csv
with open("emp.csv","w",newline='') as f:
 w=csv.writer(f)#Returns csv writer object pointing to f
 w.writerow(["ENO",'ENAME',"ESAL","EADDR"])
 n=int(input("Enter Number of Empolyees:"))
 for i in range(n):
        eno=input("Enter Employee No:")
        ename=input("Enter Employee Name:")
        esal=input("Enter Employee Sal:")
        eaddr=input("Enter Employee Address:")
        w.writerow([eno,ename,esal,eaddr])
print("Total employees data written to csv file successfully")
```

Note: Observe the difference with newline attribute and without.

```
with open("emp.csv","w",newline='') as f:
with open("emp.csv","w") as f:
```

Note:If we are not using newline attribute then in the csv file blank lines will be included between data. To prevent these blank lines, newline attribute is required in python-3, but in python-2 just we can specify mode as 'wb' and we are not required touse newline attribute.

Reading data from csv file:

```
import csv
f=open("emp.csv","r")
r=csv.reader(f)#Returns csv reader object
print(type(r))
data=list(r)
#print(data)
for line in data:
 for word in line:
        print(word,"\t",end='')
 print()
```

```
o/p:D:\pythonclasses>py test.py
<class '_csv.reader'>
ENO    ENAME   ESAL   EADDR
100    Mahesh  1000    Hyd
101    Durga   2000    Hyd
```

Zipping and Unzipping files:

-->It is very common requirement to zip and unzip files.
-->The main advantages are:

1).To improve memory utilization.
2).We can reduce transport time.
3).We can improve performance.

-->To perform zip and unzip operations, python contains one in-built module zip file. This module contains ZipFile class.

To create zip file:

-->We have to create ZipFile class object with name of the zip file, mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.

```
f=ZipFile("files.zip","w",ZIP_DEFLATED)
```

-->Once we created ZipFile object, we can add files by using write() method.

```
f.write(filename)
```

Ex:

```
from zipfile import *
f=ZipFile("files.zip","w",ZIP_DEFLATED)
f.write("cricketers.txt")
f.write("heroes.txt")
f.write("social.txt")
f.close()
print("files.zip file created successfully")
```

To perform Unzip operation:

-->We have to create ZipFile object as follows.

```
f=ZipFile("files.zip","w",ZIP_STORED)
```

-->ZIP_STORED represents unzip operation. This is default value and hence we are not required to specify.
-->Once we created ZipFile object for unzip operation, we can get all file names present in that zip file by using namelist() method.

```
name=f.namelist()
```

Ex:

```
from zipfile import *
f=ZipFile("files.zip","r",ZIP_STORED)
f.extractall()
names=f.namelist()
print(names)
for name in names:
 print("File Name:",name)
 print("The content in the file is:")
 f1=open(name,'r')
 print(f1.read())
 print()
```

Working with directories:

-->It isvery common requirement to perform operations for directories like
1).To know current working directory.
2).To create a new directory.
3).To remove an existing directory.
4).To rename a directory.
5).To list contents of the directory
etc........

-->To perform these operations pythonprovides in-built module os, which contains several methods to perform directory related operations.

Q.To know current working directory:

```
import os
cwd=os.getcwd()
print("Current working directory:",cwd)
```

Q.To create a sub directory in the current working directory:

```
import os
os.mkdir("mahesh")
print("My sub directory created cwd")
```

Q.To create sub directory in my sub directory:

```
        cwd
                |-mahesh
                        |-mysub-2
```

```
import os
os.mkdir("mahesh/mysub")
print("My sub2 directory created inside mahesh")
```

Note: Assume mahesh already present in cwd.

Q:Tocreate multiple directories like sub1 in tht sub2 in that sub3:

```
import os
os.makedirs("sub1/sub2/sub3")
print("sub1 sub2 sub3 directories created")
```

Q:To remove a directory:

```
import os
os.rmdir("mahesh/mysub")
print("mysub dir was deleted")
```

Q:To remove multiple sub directories in the path:

```
import os
os removedirs("sub1/sub2/sub3")
print("All 3 directories sub1,sub2,sub3 removed")
```

Q:To rename a directory:

```
import os
os.rename("mahesh","maheshdasari")
print("mahesh directory renamed as maheshdasari")
```

Q:Toknow content of directory:

-->os module provides listdir() to list out the contents of the specified directory. It won't display the contents of sub directory.

```
import os
print(os.listdir("."))
```

```
o/p:D:\pythonclasses>py test.py
['abc.txt', 'abcd.txt', 'add.py', 'com', 'cricketers.txt', 'emp.csv', 'files.zip', 'heroes.txt',
'log.txt', 'maheshdasari', 'maheshmath.py', 'module1.py', 'mylog.txt', 'newpic.jpg', 'pack1',
'rand.py', 'sample.py', 'social.txt', 'str.py', 'sub1', 'sunny.jpg', 'test.py', 'Test1.class',
'Test1.java', '__pycache__']
```

-->The aboveprogram diplays contents of current working directory but not contents of sub directories.

-->If we want to the content of a directory including sub directories then we should go for walk() function.

Q: To know contents of directories including sub directoris:

-->We have to use walk() function.
[can you please walk in the directory so that we can aware all contents of that directory]
-->os.walk(path,topdown=True,onerror=None,followlinks=False)
-->It returns an iterator object whose contents can be displayed by using for loop.

path-->Directory path.
topdown==True-->Travel from top to bottom
onerror==None-->on error detected which function has to execute.
followlinks=True-->To visit directories pointed by symbol links.

Ex:To display all contents of current directory including sub directories.

```
import os
for dirpath,dirnames,filenames in os.walk("D:\\"):
 print("Current Directory path:",dirpath)
 print("Directories:",dirnames)
 print("Files:",filenames)
 print()
```

```
o/p:Current Directory path: .\com
Directories: ['durgasoft', '__pycache__']
Files: ['module1.py', '__init__.py']

Current Directory path: .\com\durgasoft
Directories: ['__pycache__']
Files: ['module2.py', '__init__.py']
```

```
Current Directory path: .\com\durgasoft\__pycache__
Directories: []
Files: ['module2.cpython-37.pyc', '__init__.cpython-37.pyc']

Current Directory path: .\com\__pycache__
Directories: []
Files: ['module1.cpython-37.pyc', '__init__.cpython-37.pyc']
```

Note: To display contents of a particular directory, we have to provide that directory name as argument to walk() function.

os.walk("directoryname")

Q: What is difference between listdir() and walk() functions?
--

In case of listdir(), we will get contents of specified directory but not sub directory contents. But in the case of walk() function we will get contents of specified directory and its sub directories also.

Running other programs from python program:

-->os module contains system() function to run the programs and commands.
-->It is exactly same as system() function in C-Language.

Syn: os.system("command string")

-->The argument is any command which is executing from DOS.

Ex:

```
import os
os.system("dir *.py")
os.system("py abc.py")
```

How to get information about file:
--

-->We can get statistics of a file like size, last accessed time, last modified time etc by using stat() function of os module.

stats=os.stat("abc.txt")

-->The statistics of a file includes the following parameters.
st_mode==>Protection Bits
st_ino==>Inode number
st_dev==>device
st_nlink==>no of hard links
st_uid==>userid of owner
st_gid==>groupid of owner

st_size==>size of the file in bytes
st_atime==>Time of last access
st_mtime==>Time of most recent moification

Note:
at_atime,st_mtime and st_ctime returns the time as number of milliseconds since Jan 1st 1970, 12:00AM(Epoche Time Standard). By using datetime module fromtimestamp() function, we can get exact date and time.

Q:To print all statics of file abc.txt:

```
import os
stats=os.stat("abc.txt")
print(stats)
```

```
o/p:D:\pythonclasses>py test.py
os.stat_result(st_mode=33206, st_ino=3096224743875076, st_dev=1014225080,
st_nlink=1, st_uid=0, st_gid=0, st_size=166, st_atime=1542099518, st_mtime=1542166071,
st_ctime=1542099518)
```

Q:To print specified properties:

```
import os
from datetime import *
stats=os.stat("abc.txt")
print("File size in Bytes:",stats.st_size)
print("File Last Accessed Time:",datetime.fromtimestamp(stats.st_atime))
print("File Last Modifies Time:",datetime.fromtimestamp(stats.st_mtime))
```

```
o/p:D:\pythonclasses>py test.py
File size in Bytes: 166
File Last Accessed Time: 2018-11-13 14:28:38.715755
File Last Modifies Time: 2018-11-14 08:57:51.052870
```

Pickling and Unpickling of Objects:

-->Sometimes we have to write total state of object to the file and we have to read total object from the file.
-->The process of writing state object to the file is called as pickling and the process of reading state of an object from the file is called as unpickling.
-->We can implement pickling and unpickling by using pickle module of python.
-->pickle module contains dump() function to perform pickling

pickle.dump(object,f)
-->pickle module contains load() function to perform unpickling.
pickle.load(f)

Q:Create an employee class accept data and display data.

```
class Employee:
 def __init__(self,eno,ename,eaddr):
        self.eno=eno
        self.ename=ename
        self.eaddr=eaddr
e=Employee(100,"Mahesh","Hyd")
print(e.eno,e.ename,e.eaddr)
```

o/p:D:\pythonclasses>py test.py
100 Mahesh Hyd

Q:Writting and reading state of object by using pickle module:

```
import pickle
class Employee:
 def __init__(self,eno,ename,eaddr):
        self.eno=eno
        self.ename=ename
        self.eaddr=eaddr
 def display(self):
        print(self.eno,"\t",self.ename,"\t",self.eaddr)
with open("emp.dat","wb") as f:
 e=Employee(100,"Mahesh","Hyd")
 pickle.dump(e,f)
 print("Pickiling of Employee object completed......")
with open("emp.dat","rb") as f:
 obj=pickle.load(f);
 print("Priniting Employee information after unpickling...")
 obj.display()
```

Q: writing multiple employee objects to the file:

----

emp.py:

```
class Employee:
  def __init__(self,eno,ename,eaddr):
        self.eno=eno
        self.ename=ename
        self.eaddr=eaddr
  def display(self):
        print(self.eno,"\t",self.ename,"\t",self.eaddr)
```

pick.py:

```
import emp,pickle
f=open("emp.dat","wb")
n=int(input("Enter the number of employees:"))
for i in range(n):
  eno=int(input("Enter Employee Number:"))
  ename=input("Enter Employee Name:")
  eaddr=input("Enter Employee Address:")
  e=emp.Employee(eno,ename,eaddr)
  pickle.dump(e,f)
print("Employee Objects pickled successfully")
```

```
o/p:D:\pythonclasses>py pick.py
Employee Objects pickled successfully
```

unpick.py:

```
import emp,pickle
f=open("emp.dat","rb")
print("Employee deatils:")
while True:
  try:
        obj=pickle.load(f)
        obj.display()
  except EOFError:
        print("All employees completed")
        break
f.close()
```

```
o/p:D:\pythonclasses>py unpick.py
Employee deatils:
100     Mahesh      Hyd
102     Durga           Hyd
104     Sunny           Hyd
All employees completed
```

Python's Object Oriented Programming (OOP's)

===

What is class:

-->In python every thing is an object. To create objects we required some Model or plan or Blue print, which is nothing but a class.
-->We can write a class to represent properties(attributes) and actions(behaviour) ofobject.
-->Properties can be represented by variables.
-->Actions can be represented by methods.
-->Hence class contains both variables and methods.

How to define a class?

-->We can define a class by using class keyword.

Syn:

```
class className:
        '''documentation string'''
        variables:Instance, static and local variables
        methods:Instance, static, class methods
```

-->Documentation string represents description of the class. Within the class doc string is always optional. We can get doc string by using following 2-ways.

1).print(classname.__doc__)
2).help(classname)

Ex:

```
class Student:
    '''This is student class with required data'''
print(Student.__doc__)
help(Student)
```

-->With in the python classes we can represent data by using variables.
-->There are 3-types of variables are allowed.

1).Instance Variables(Object Level Variables)
2).Static Variables(Class Level Variables)
3).Local Variables(Method Level Variables)

-->Within the python class we can represent operations by using methods. The following are various types of allowed methods.

1).Instance Methods
2).Class Methods
3).Static Methods

Example for a class:

```
class Student:
        '''Developed by mahesh for python demo'''
        def __init__(self):
                self.name="Mahesh"
                self.age=30
                self.marks=100
        def talk(self):
                print("Hello I Am:",self.name)
                print("My Age Is:",self.age)
                print("My Marks Are:",self.marks)
```

What is an object?

-->Physicalexistance of class is nothing but object. We can create any number of objects for a class.
Syn:

ReferenceVariable=ClassName()

Ex:

s=Student()

What is a Reference Variable?

-->The variable which can be used to refer an object is called as referenece variable.
-->By using reference variable, we can access properties and methods of an object.

Ex:

w.a.p to create a Student class and creates an object to it. Call the method talk() to display student details.

```
class Student:

        def __init__(self,name,rollno,marks):
                self.name=name
                self.age=rollno
                self.marks=marks

        def talk(self):
```

```
        print("Hello My Name Is:",self.name)
        print("My Roll No:",self.rollno)
        print("My Marks Are:",self.marks)

s=Student("Mahesh",101,90)
s.talk()
```

self Variable:

-->self is a default variable which is always pointing to current object(like this keyword in java).
-->By using self we can access instance variables and instance methods of object.

Note:
----
1).self should be first parameter inside constructor.

```
def __init__(self):
```

2).self should be first parameter inside instance methods.

```
def talk(self):
```

Constructor Concept:
===================
-->Constructor is a special method in Python.
-->The name of the constructor should be __init__(self).
-->Constructor will be executed automatically at the time of object creation.
-->The main purpose of constructor is to declare and initialize instance variables.
-->Per object constructor will be executed once.
-->Constructor can take atleast one argument(atleast self).
-->Constructor is optional and if we are not providing any constructor python will provide default constructor.

Ex:

```
def __init__(self,name,rollno,marks):
        self.name=name
        self.age=rollno
        self.marks=marks
```

Program to demonistrate constructor will execute only once per object:

```
class Test:
        def __init__(self):
                print("Constructor exeuction...")
                def m1(self):
                        print("Method execution...")
                        t1=Test()
                        t2=Test()
                        t3=Test()
                        t1.m1()
```

Output:

```
Constructor exeuction...
Constructor exeuction...
Constructor exeuction...
Method execution...
```

Program:

```
class Student:
        """ This is student class with required data'''
        def __init__(self,x,y,z):
                self.name=x
                self.rollno=y
                self.marks=z
                def display(self):
                        print("Student Name:{}\nRollno:{}
\nMarks:{}".format(self.name,self.rollno,self.marks))
                        s1=Student("Mahesh",101,80)
                        s1.display()
                        s2=Student("Sunny",102,100)
                        s2.display()
```

Output:

```
Student Name:Mahesh
Rollno:101
Marks:80
Student Name:Sunny
Rollno:102
Marks:100
```

How to create list of objects in constructor:

```
class Movie:
    def __init__(self,name,hero,heroine,rating):
        self.name=name
        self.hero=hero
        self.heroine=heroine
        self.rating=rating

    def info(self):
        print("Movie Name:",self.name)
        print("Hero Name:",self.hero)
        print("Heroine Name:",self.heroine)
        print("Movie Rating :",self.rating)
        print()
movies=[Movie('Spider','Mahesh','Anusha',30),Movie('Bahubali','Prabhas','Anusha'
,30),Movie('Geethagovindam','Vijay','Preethi',90)]
for movie in movies:
    movie.info()
```

Ex: without constructor

```
class Movie:
    def info(self):
        print("Movie Name:",self.name)
        print("Hero Name:",self.hero)
        print("Heroine Name:",self.heroine)
        print("Movie Rating :",self.rating)
        print()
m=Movie()
m.name="Spider"
m.hero="Mahesh"
m.heroine="Rakul"
m.rating="90"
m.info()
```

Ex:with normal method

```
class Movie:
        def vd(self,name,hero,heroine,rating):
                self.name=name
                self.hero=hero
                self.heroine=heroine
                self.rating=rating
        def info(self):
                print("Movie Name:",self.name)
                print("Hero Name:",self.hero)
                print("Heroine Name:",self.heroine)
                print("Movie Rating :",self.rating)
                print()
m=Movie()
m.vd("Spider","Mahesh","Rakul","90")
m.info()
```

Difference between Methods and Constructors:

Method	Constructor
1. Name of method can be any name.	1. Constructor name should be always __init__
2. Method will be executed, if we call that method.	2. Constructor will be executed automatically at the time of object creation.
3. Per object, method can be called any number of times.	3. Per object, Constructor will be executed only once.
4. Inside method we can write and initialize instance variables.	4. Inside Constructor we have to declare business logic.

Types of Variables:
=================
Inside Python class 3 types of variables are allowed:

1. Instance Variables (Object Level Variables)
2. Static Variables (Class level Variables)
3. Local Variables (Method Level Variables)

1. Instance Variables :
====================
If the value of a variable is varied from object to object then such type of variables are called Instance Variables.

For every object a separate copy of instance variables will be created.

Where we can declare Instance Variables ?
====================================

1. Inside constructor by using self variable.
2. Inside Instance Method by using self variable.
3. Outside of the class by using object reference variable.

1. Inside constructor by using self variable.

We can declare instance variables inside a constructor by using self keyword.
Once we create the object, automatically these variables will be added to the object.

Ex:

```
class Employee:
        def __init__(self):
                self.eno=100
                self.ename="Mahesh"
e=Employee()
print(e.__dict__)
```

output:
```
{'eno':100,'ename':'Mahesh'}
```

2. Inside Instance Method by using self variable.

We can also declare instance variables inside instance method by using self variable.
If any instance variable declared inside instance method, that instance variable will be added once we call that method.

Ex:

```
class Test:
        def __init__(self):
                self.a=10
```

```
          self.b=20
              def m1(self):
              self.c=30
              t=Test()
              t.m1()
              print(t.__dict__)
```

Output:
{'a': 10, 'b': 20, 'c': 30}

3. Outside of the class by using object reference variable.
--

We can also add instance variables outside of a class to a particular object.

```
class Test:
        def __init__(self):
        self.a=10
        self.b=20
        def m1(self):
        self.c=30
        t=Test()
        t.m1()
        t.d=40
        print(t.__dict__)
```

Output:
{'a': 10, 'b': 20, 'c': 30, 'd': 40}

How to access Instance Variables
=============================
We can access instance variables with in the class by using self variable and outside of the class by using object reference.

Ex:

```
class Test:
        def __init__(self):
        self.a=10
        self.b=20
        def display(self):
            print(self.a)
            print(self.b)
```

```
        t=Test()
        t.display()
        print(t.a,t.b)
```

Output:
```
10
20
10 20
```

How to delete instance variable from the object :
==

1. Within a class we can delete instance variable as follows:

    del self.variableName

2. From outside of class we can delete instance variables as follows :
    del objectreference.variableName

Ex:

```
class Test:
    def __init__(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40
    def m1(self):
        del self.d
t=Test()
print(t.__dict__)
t.m1()
print(t.__dict__)
```

Output:
```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
```

Note: The instance variables which are deleted from one object,will not be deleted from other objects.

```
class Test:
        def __init__(self):
                self.a=10
                self.b=20
                self.c=30
                self.d=40
t1=Test()
t2=Test()
del t1.a
print(t1.__dict__)
print(t2.__dict__)
```

Output:
```
{'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

-->If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

```
class Test:
        def __init__(self):
                self.a=10
                self.b=20
t1=Test()
t1.a=333
t1.b=999
t2=Test()
print("t1:",t1.a,t1.b)
print("t2:",t2.a,t2.b)
```

output:
```
333  999
10  20
```

1. Static Variables :
=================

If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.

For total class only one copy of static variable will be created and shared by all the objects of that class.

We can access staic variables either by class name of by object reference. But recommended to use class name.

Instance Variable vs Static Variable:
=================================

Note : In the case of instance variables for every object a separate copy will be created, but in the case of static variables for total class only one copy will be created and shared by every object of that class.

```
class Test:
        x=10
        def __init__(self):
                self.y=20
t1=Test()
t2=Test()
print("t1:",t1.x,t1.y)
print("t2:",t2.x,t2.y)
Test.x=333
t1.y=999
print("t1:",t1.x,t1.y)
print("t2:",t2.x,t2.y)
```

output:
```
10      20
10      20
333     999
333     20
```

Various places to declare static variables:
=====================================
1.In general we can declare within the class directly but from out side of any method.
2.Inside constructor by using class name.
3.Inside instance method by using class name.
4.Inside classmethod by using either class name or cls variable.
5.Inside static method by using class name.

```
class Test:
        a=10
        def __init__(self):
                Test.b=20
        def m1(self):
                Test.c=30
        @classmethod
```

```
    def m2(cls):
        cls.d1=40
        Test.d2=50
    @staticmethod
    def m3():
        Test.e=60
t1=Test()
print(Test.__dict__)
t1.m1()
print(Test.__dict__)
Test.m2()
print(Test.__dict__)
Test.m3()
print(Test.__dict__)
Test.f=90
print(Test.__dict__)
```

How to access static variables:

===========================

1. inside constructor: by using either self or classname

2. inside instance method: by using either self or classname

3. inside class method: by using either cls variable or classname

4. inside static method: by using classname

5. From outside of class: by using either object reference or classname.

```
class Test:
    a=10
    def __init__(self):
        print(self.a)
        print(Test.a)
    def m1(self):
        print(Test.a)
        print(self.a)
    @classmethod
    def m2(cls):
        print(Test.a)
        print(cls.a)
    @staticmethod
    def m3():
        print(Test.a)

t=Test()
print(Test.a)
t.m1()
```

```
t.m2()
t.m3()
```

Where we can modify the value of static variable:
===
Anywhere either with in the class or outside of class we can modify by using classname.
But inside class method, by using cls variable.

```
class Test:
    a=777
    @classmethod
    def m1(cls):
        cls.a=333
    @staticmethod
    def  m2():
        Test.a=999
print(Test.a)
Test.m1()
print(Test.a)
Test.m2()
print(Test.a)
```

output:

```
777
333
999
```

******
If we change the value of static variable by using either self or object reference variable:

If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed,just a new instance variable with that name will be added to that particular object.

Ex-1:

```
class Test:
    a=10
    def m1(self):
        self.a=333
t1=Test()
t1.m1()
```

```
print(Test.a)
print(t1.a)
```

output:
```
10
333
```

Ex-2:

```
class Test:
        x=10
        def __init__(self):
                self.y=20
t1=Test()
t2=Test()
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
t1.x=333
t1.y=999
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
```

output:
```
10      20
10      20
333     999
10      20
```

Ex-3:

```
class Test:
        a=10
        def __init__(self):
                self.b=20
t1=Test()
t2=Test()
Test.a=333
t1.b=999
print('t1:',t1.a,t1.b)
print('t2:',t2.a,t2.b)
```

output:
```
333    999
333    20
```

Ex-4:

```
class Test:
        a=10
        def __init__(self):
                self.b=20
        def m1(self):
                self.a=333
                self.b=999
t1=Test()
t2=Test()
t1.m1()
print('t1:',t1.a,t1.b)
print('t2:',t2.a,t2.b)
```

output:
```
333    999
10     20
```

Ex-5:

```
class Test:
        a=10
        def __init__(self):
                self.b=20
        @classmethod
        def m1(cls):
                cls.a=333
                cls.b=999
t1=Test()
t2=Test()
t1.m1()
print('t1:',t1.a,t1.b)
print('t2:',t2.a,t2.b)
print(Test.a,Test.b)
```

output:
```
333     20
333     20
333     999
```

How to delete static variables of a class:
=====================================

We can delete static variables from anywhere by using the following syntax

```
del classname.variablename
```

But inside classmethod we can also use cls variable

```
del cls.variablename
```

```
class Test:
    a=10
    @classmethod
    def m1(cls):
        del cls.a
Test.m1()
print(Test.__dict__)
```

Ex:

```
class Test:
    a=10
    def __init__(self):
        Test.b=20
        del Test.a
    def m1(self):
        Test.c=30
        del Test.b
    @classmethod
    def m2(cls):
        Test.d=40
        del Test.c
    @staticmethod
    def m3():
        Test.e=50
        del Test.d
print(Test.__dict__)
t=Test()
print(Test.__dict__)
t.m1()
```

```
print(Test.__dict__)
t.m2()
print(Test.__dict__)
t.m3()
print(Test.__dict__)
```

****** Note: By using object reference variable/self we can read static variables, but we cannot modify or delete.**

If we are trying to modify, then a new instance variable will be added to that particular object. t1.a = 70

If we are trying to delete then we will get error.

Ex:

```
class Test:
    a=10
    t1=Test()
    del t1.a ===>AttributeError: a
```

We can modify or delete static variables only by using classname or cls variable.

```
import sys
class Customer:
    ''' Customer class with bank operations.....'''
    bankname="MaheshBank"
    def __init__(self,name,balance=0.0):
        self.name=name
        self.balance=balance
    def deposit(self,amt):
        self.balance=self.balance+amt
    def withdraw(self,amt):
        if amt>self.balance:
            print("Insufficient Funds...can't perform this operation")
            sys.exit()
        self.balance=self.balance-amt
        print("Balance after withdraw:",self.balance)
print("Welcome to",Customer.bankname)
name=input("Enter Your Name:")
c=Customer(name)
while True:
    print("d-Deposit\nw-Withdraw\ne-Exit")
    option=input("Enter your option:")
```

```
    if option=='d' or option=='D':
        amt=float(input("Enter amount:"))
        c.deposit(amt)
    elif option=='w' or option=='W':
        amt=float(input("Enter amount:"))
        c.withdraw(amt)
    elif option=='e' or option=='E':
        print("Thanks for Banking!!!!!!!!!")
        sys.exit()
    else:
        print("Invalid option....Plz choose a valid option")
```

output:

```
D:\pythonclasses>py test.py
Welcome to MAHESHBANK
Enter Your Name:Mahesh
d-Deposit
w-Withdraw
e-exit
Choose your option:d
Enter amount:10000
Balance after deposit: 10000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:d
Enter amount:20000
Balance after deposit: 30000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:w
Enter amount:2000
Balance after withdraw: 28000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:r
Invalid option..Plz choose valid option
d-Deposit
w-Withdraw
e-exit
Choose your option:e
Thanks for Banking
```

Local variables:
==============

Sometimes to meet temporary requirements of programmer,we can declare variables inside a method directly,such type of variables are called local variable or temporary variables.

Local variables will be created at the time of method execution and destroyed once method completes.

Local variables of a method cannot be accessed from outside of method.

Example:

```
class Test:
    def m1(self):
        a=1000
        print(a)
        def m2(self):
            b=2000
            print(b)
            t=Test()
            t.m1()
            t.m2()
```

Output:
1000 2000

Example 2:

```
class Test:
    def m1(self):
        a=1000
        print(a)
        def m2(self):
            b=2000
            print(a) #NameError: name 'a' is not defined
            print(b)
            t=Test()
            t.m1()
            t.m2()
```

Types of Methods:

================

Inside Python class 3 types of methods are allowed
1. Instance Methods
2. Class Methods
3. Static Methods

1. Instance Methods:

==================

Inside method implementation if we are using instance variables then such type of methods are called instance methods. Inside instance method declaration,we have to pass self variable.

```
def m1(self):
```

By using self variable inside method we can able to access instance variables.

Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

```
class Student:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    def display(self):
        print("Hi",self.name)
        print("Your marks are:",self.marks)
    def grade(self):
        if self.marks>=60:
            print("You got First Grade")
        elif self.marks>=50:
            print("You got Second Grade")
        elif self.marks>=35:
            print("You got Third Grade")
        else:
            print("You  are failed")
n=int(input("Enter number of students:"))
for i in range(n):
    name=input("Enter Name:")
    marks=int(input("Enter Marks:"))
    s=Student(name,marks)
    s.display()
    s.grade()
    print()
```

ouput:

```
D:\pythonclasses>py test.py
Enter number of students:2
Enter Name:Mahesh
Enter Marks:90
Hi Mahesh Your Marks are: 90
You got First Grade

Enter Name:Sunny
Enter Marks:12
Hi Sunny Your Marks are: 12
You are Failed
```

Setter and Getter Methods:
========================
We can set and get the values of instance variables by using getter and setter methods.

Setter Method:
=============

Setter methods can be used to set values to the instance variables. setter methods also known as mutator methods.

Syntax:

```
	def setVariable(self,variable):
		self.variable=variable
```

Example:

```
	def setName(self,name):
		self.name=name
```

Getter Method:
==============
Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.

Syntax:

```
	def getVariable(self):
		return self.variable
```

Example:

```
	def getName(self):
		return self.name
```

Program:

```
class Student:
        def setName(self,name):
                self.name=name

        def getName(self):
                return self.name

        def setMarks(self,marks):
                self.marks=marks

        def getMarks(self):
                return self.marks
l=[]
n=int(input("Enter Number of students:"))
for i in range(n):
        s=Student()
        name=input("Enter Name:")
        s.setName(name)
        marks=int(input("Enter Marks:"))
        s.setMarks(marks)
        l.append(s)

for s in l:
        print("Studnet Name:",s.getName())
        print("Student Marks:",s.getMarks())
```

output:

```
D:\pythonclasses>py test.py
Enter Number of students:2
Enter Name:Mahesh
Enter Marks:90
Enter Name:Durga
Enter Marks:100

Studnet Name: Mahesh
Student Marks: 90
Studnet Name: Durga
Student Marks: 100
```

2. Class Methods:
=================

Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.

We can declare class method explicitly by using @classmethod decorator. For class method we should provide cls variable at the time of declaration

We can call classmethod by using classname or object reference variable.

Demo program:

```
class Animal:
    legs=4
    @classmethod
    def walk(cls,name):
        print("{} walks with {}legs......".format(name,cls.legs))
Animal.walk("Dog")
Animal.walk("Cat")
```

```
o/p:D:\pythonclasses>py test.py
Dog walks with 4legs......
Cat walks with 4legs......
```

w.a.p to track the number of objects created for a class

```
class Test:
    count=0
    def __init__(self):
        Test.count=Test.count+1
    @classmethod
    def no_of_objects(cls):
        print("The number of objects created for test class:",cls.count)
t1=Test()
t2=Test()
Test.no_of_objects()
t3=Test()
t4=Test()
t5=Test()
Test.no_of_objects()
```

```
o/p:D:\pythonclasses>py test.py
The number of objects created for test class: 2
The number of objects created for test class: 5
```

3. Static Methods:
=================

In general these methods are general utility methods. Inside these methods we won't use any instance or class variables. Here we won't provide self or cls arguments at the time of declaration.

We can declare static method explicitly by using @staticmethod decorator We can access static methods by using classname or object reference

Note: In general we can use only instance and static methods.Inside static method we can access class level variables by using class name.

class methods are most rarely used methods in python.

Program:

```
class MaheshMath:
        @staticmethod
        def add(x,y):
                print("The sum is:",x+y)

        @staticmethod
        def product(x,y):
                print("The sum is:",x*y)

        @staticmethod
        def average(x,y):
                print("The sum is:",(x+y)/2)

MaheshMath.add(10,20)
MaheshMath.product(10,20)
MaheshMath.average(10,20)
```

```
o/p:D:\pythonclasses>py test.py
The sum is: 30
The sum is: 200
The sum is: 15.0
```

Passing members of one class to another class:

We can access members of one class inside another class.

Ex:

```
class Employee:
        def __init__(self,eno,ename,esal):
                self.eno=eno
                self.ename=ename
                self.esal=esal
        def display(self):
                print("Employee Number:",self.eno)
                print("Employee Name:",self.ename)
                print("Employee Salary:",self.esal)

class Test:
        def modify(emp):
                emp.esal=emp.esal+10000
                emp.display()

e=Employee(100,"Mahesh",10000)
Test.modify(e)
```

```
o/p:D:\pythonclasses>py test.py
Employee Number: 100
Employee Name: Mahesh
Employee Salary: 20000
```

Inner classes:
=============
Sometimes we can declare a class inside another class,such type of classes are called inner classes.

Without existing one type of object if there is no chance of existing another type of object,then we should go for inner classes.

Example:
Without existing Car object there is no chance of existing Engine object. Hence Engine class should be part of Car class.

Class Car:
    .....
class Engine:
......

Example:
Without existing university object there is no chance of existing Department object

class University:
.....
class Department:
......

eg3: Without existing Human there is no chance of existin Head. Hence Head should be part of Human.

class Human:

class Head:

Note: Without existing outer class object there is no chance of existing inner class object. Hence inner class object is always associated with outer class object.

program:

```
class Outer:
        def __init__(sel):
                print("Outer class object creation")

        class Inner:
                def __init__(self):
                        print("Inner class object creation")
                def m1(self):
                        print("Inner class method")

o=Outer()
i=o.Inner()
i.m1()
```

Note:

The following are various possible syntaxes for calling inner class method
1. o=Outer() i=o.Inner() i.m1()

2. i=Outer().Inner() i.m1()
3. Outer().Inner().m1()

Ex:

```
class Human:
    def __init__(self):
        self.name="Sunny"
        self.head=self.Head()
        self.brain=self.head.Brain()

    def display(self):
        print("Hello",self.name)
        self.head.talk()
        self.brain.think()

    class Head:
        def talk(self):
            print("Talking...........")

        class Brain:
            def think(self):
                print("Thinking??????")
h=Human()
h.display()
```

```
o/p:D:\pythonclasses>py test.py
Hello Sunny
Talking...........
Thinking??????
```

Ex:

```
class Person:
    def __init__(self,name,dd,mm,yyyy):
        self.name=name
        self.dob=self.Dob(dd,mm,yyyy)

    def display(self):
        print("Name:",self.name)
        self.dob.display()

```

```
        class Dob:
                def __init__(self,dd,mm,yyyy):
                        self.dd=dd
                        self.mm=mm
                        self.yyyy=yyyy

                def display(self):
                        print("DOB={}/{}/{}".format(self.dd,self.mm,self.yyyy))

p=Person("Sunny",15,8,1947)
p.display()
```

```
o/p:D:\pythonclasses>py test.py
Name: Sunny
DOB=15/8/1947
```

Nested Methods:

Nested Method:A method inside the method.
Purpose: To define method specific repetedly required functionality.

Ex:

```
class Test:
        def m1(self):
                def calc(a,b):
                        print("The sum is:",a+b)
                        print("The product is:",a*b)
                        print("The division is:",a/b)
                        print("The difference is:",a-b)
                        print()

                calc(10,20)
                calc(100,200)
                calc(100,200)

t=Test()
t.m1()
```

Garbage Collection

In old languages like C++, programmer is responsible for both creation and destruction of objects.Usually programmer taking very much care while creating object, but neglecting destruction of useless objects. Because of his neglectance, total memory can be filled with useless objects which creates memory problems and total application will be down with Out of memory error.

But in Python, We have some assistant which is always running in the background to destroy useless objects.Because this assistant the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.

Hence the main objective of Garbage Collector is to destroy useless objects.

If an object does not have any reference variable then that object eligible for Garbage Collection.

How to enable and disable Garbage Collector in our program:
===

By default Gargbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of gc module.

1. gc.isenabled()
Returns True if GC enabled

2. gc.disable() To disable GC explicitly

3. gc.enable() To enable GC explicitly

Example:

```
import gc
class Test:
        print(gc.isenabled())
        gc.disable()
        print(gc.isenabled())
        gc.enable()
        print(gc.isenabled())
```

Destructors:
============
Destructor is a special method and the name should be __del__ Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc).
Once destructor execution completed then Garbage Collector automatically destroys that object.

Note: The job of destructor is not to destroy object and it is just to perform clean up activities.

Example:

```
import time
class Test:
	def __init__(self):
		print("Object Initialization.....")
	def __del__(self):
		print("Fullfilling last wish and performing clean up activities....")
t1=Test()
t1=None
time.sleep(10)
print("End of application")
```

```
o/p:D:\pythonclasses>py test.py
Object Initialization.....
Fullfilling last wish and performing clean up activities....
End of application
```

Note: If the object does not contain any reference variable then only it is eligible fo GC. ie if the reference count is zero then only object eligible for GC

Example:

```
import time
class Test:
	def __init__(self):
		print("Object Initialization.....")
	def __del__(self):
		print("Fullfilling last wish and performing clean up activities....")
t1=Test()
t2=t1
t3=t2
```

```
del t1
time.sleep(10)
print("object not yest destroyed after deleting t1")
del t2
print("object not yest destroyed after deleting t2")
del t3
print("By this time object will be destroyed")
print("End of application")
```

```
o/p:D:\pythonclasses>py test.py
Object Initialization.....
object not yest destroyed after deleting t1
object not yest destroyed after deleting t2
Fullfilling last wish and performing clean up activities....
By this time object will be destroyed
End of application
```

Example:
=========

```
import time
class Test:
        def __init__(self):
                print("Constructor Execution.........")
        def __del__(self):
                print("Destructor Execution......")

list=[Test(),Test(),Test()]
del list
time.sleep(5)
print("End of application")
```

```
o/p:D:\pythonclasses>py test.py
Constructor Execution.........
Constructor Execution.........
Constructor Execution.........
Destructor Execution......
Destructor Execution......
Destructor Execution......
End of application
```

How to find the number of references of an object:
==

sys module contains getrefcount() function for this purpose.

sys.getrefcount(objectreference)

Example:

```
import sys
class Test:
pass
t1=Test()
t2=t1
t3=t1
t4=t1
print(sys.getrefcount(t1))
```

Output 5

Note: For every object, Python internally maintains one default reference variable self.

OOP's--PART-2

Agenda:

-->Inheritance
-->Has-A Relationship
-->IS-A Relationship
-->IS-A vs HAS-A Relationship
-->Composition vs Aggregation

-->Types of Inheritance
  -->Single Inheritance
  -->Multi Level Inheritance
  -->Hierarchical Inheritance
  -->Multiple Inheritance
  -->Hybrid Inheritance
  -->Cyclic Inheritance

-->Method Resolution Order (MRO)
-->super() Method

Using members of one class inside another class:

We can use members of one class inside another class by using the following ways

1. By Composition (Has-A Relationship)
2. By Inheritance (IS-A Relationship)

1. By Composition (Has-A Relationship):

-->By using Class Name or by creating object we can access members of one class inside another class is nothing but composition (Has-A Relationship).

-->The main advantage of Has-A Relationship is Code Reusability.

Demo Program-1:

```
class Engine:
        a=10
        def __init__(self):
                self.b=20
        def m1(self):
                print("Engine Specific Functionality")
class Car:
        def __init__(self):
                self.engine=Engine()
        def m2(self):
                print("Car using Engine class functionality")
                print(self.engine.a)
                print(self.engine.b)
                self.engine.m1()
c=Car()
c.m2()
```

```
o/p:D:\pythonclasses>py test.py
Car using Engine class functionality
10
20
Engine Specific Functionality
```

Demo Program-2:

```
class Car:
        def __init__(self,name,model,color):
                self.name=name
                self.model=model
                self.color=color
        def getinfo(self):
                print("Car Name:{}, Model:{},color:{}".format(self.name,self.model,self.color))

class Employee:
        def __init__(self,ename,eno,car):
                self.ename=ename
                self.eno=eno
                self.car=car
        def empinfo(self):
```

```
        print("Employee Name:",self.ename)
        print("Employee Number:",self.eno)
        print("Employee Car Info:")
        self.car.getinfo()

c=Car("Innova","2.5v","Grey")
e=Employee("Mahesh",101,c)
e.empinfo()
```

```
o/p:D:\pythonclasses>py test.py
Employee Name: Mahesh
Employee Number: 101
Employee Car Info:
Car Name:Innova, Model:2.5v, color:Grey
```

-->In the above example Employee Has-A car reference and hence Employee class can access all memebers of Car class.

Demo Program-3:

```
class X:
    a=10
    def __init__(self):
        self.b=20
    def m1(self):
        print("m1 method of X class")

class Y:
    c=30
    def __init__(self):
        self.d=40
    def m2(self):
        print("m2 method of Y class")
    def m3(self):
        x=X()
        print(x.a)
        print(x.b)
        x.m1()
        print(Y.c)
        print(self.d)
        self.m2()
        print("m3 method in Y class")
y1=Y()
y1.m3()
```

```
o/p:D:\pythonclasses>py test.py
10
20
m1 method of X class
30
40
m2 method of Y class
m3 method in Y class
```

2. By Inheritance(IS-A Relationship):

-->What ever variables, methods and constructors available in the parent class by default available to the child classes and we are not required to rewrite.

-->Hence the main advantage of inheritance is Code Reusability and we can extend existing functionality with some more extra functionality.

Syntax : class childclass(parentclass):

Demo Program:

```
class P:
        a=10
        def __init__(self):
                self.b=20
        def m1(self):
                print("Parent Instance method")
        @classmethod
        def m2(cls):
                print("Parent class method")
        @staticmethod
        def m3():
                print("Parent5 static method")
class C(P):
        pass

c=C()
print(c.a)
print(c.b)
c.m1()
c.m2()
c.m3()
```

```
o/p:D:\pythonclasses>py test.py
10
20
Parent Instance method
Parent class method
Parent5 static method
```

Ex:

```
class P:
        10 methods
class C(P):
        5 methods
```

-->In the above example Parent class contains 10 methods and these methods automatically available to the child class and we are not required to rewrite those methods(Code Reusability) Hence child class contains 15 methods.

Note:
What ever members present in Parent class are by default available to the child class through inheritance.

Demo Program:

```
class P:
        def m1(self):
                print("Parent class method")
class C(P):
        def m2(self):
                print("Child class method")
c=C()
c.m1()
c.m2()
```

```
o/p:D:\pythonclasses>py test.py
Parent class method
Child class method
```

-->What ever methods present in Parent class are automatically available to the child class and hence on the child class reference we can call both parent class methods and child class methods.

-->Similarly variables also

```
class P:
        a=10
        def __init__(self):
                self.b=20
class C(P):
        c=30
        def __init__(self):
                super().__init__()===>Line-1
                self.d=30
c1=C() 1
print(c1.a,c1.b,c1.c,c1.d)
```

o/p:
10 20 30 40

-->If we comment Line-1 then variable b is not available to the child class.

Demo Program:

```
class Person:
        def __init__(self,name,age):
                self.name=name
                self.age=age
        def eat_n_drink(self):
                print("Drink Beer and Eat Biryani")

class Employee(Person):
        def __init__(self,name,age,eno,sal):
                super().__init__(name,age)
                self.eno=eno
                self.sal=sal
        def work(self):
                print("Coding Python is vsery easy just like drinking chilled beer")
        def emp_info(self):
                print("Employee Name:",self.name)
                print("Employee Age:",self.age)
                print("Employee Number:",self.eno)
                print("Employee Salary:",self.sal)

e=Employee("Mahesh",50,100,100000)
e.eat_n_drink()
```

```
e.work()
e.emp_info()
```

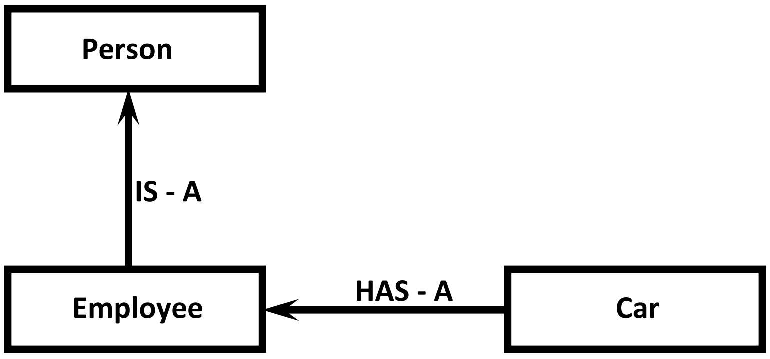
```
o/p:D:\pythonclasses>py test.py
Drink Beer and Eat Biryani
Coding Python is vsery easy just like drinking chilled beer
Employee Name: Mahesh
Employee Age: 50
Employee Number: 100
Employee Salary: 100000
```

IS-A vs HAS-A Relationship:

-->If we want to extend existing functionality with some more extra functionality then we should go for IS-A Relationship.

-->If we dont want to extend and just we have to use existing functionality then we should go for HAS-A Relationship.

Eg: Employee class extends Person class Functionality
But Employee class just uses Car functionality but not extending



Program:

```
class Car:
        def __init__(self,name,model,color):
                self.name=name
                self.model=model
                self.color=color
        def get_info(self):
                print("\tCar name:{}\n\tModel:{}\n\tColor={}".format(self.name,self.model,self.color))

```

```
class Person:
        def __init__(self,name,age):
                self.name=name
                self.age=age
        def eat_n_drink(self):
                print("Drink Beer and Eat Biryani")

class Employee(Person):
        def __init__(self,name,age,eno,sal,car):
                super().__init__(name,age)
                self.eno=eno
                self.sal=sal
                self.car=car
        def work(self):
                print("Coding Python is vsery easy just like drinking chilled beer")
        def emp_info(self):
                print("Employee Name:",self.name)
                print("Employee Age:",self.age)
                print("Employee Number:",self.eno)
                print("Employee Salary:",self.sal)
                print("Employee Car Info:")
                self.car.get_info()

c=Car("Innova","2.5v","Grey")
e=Employee("Mahesh",50,100,100000,c)
e.eat_n_drink()
e.work()
e.emp_info()
```

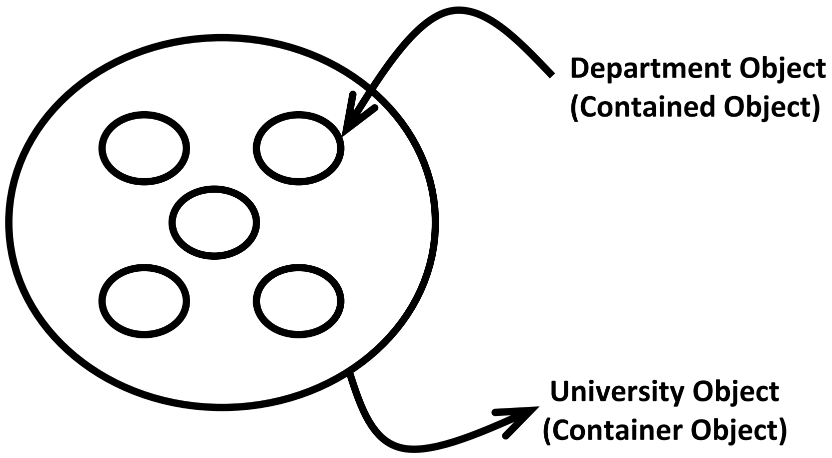
```
o/p:D:\pythonclasses>py test.py
Drink Beer and Eat Biryani
Coding Python is vsery easy just like drinking chilled beer
Employee Name: Mahesh
Employee Age: 50
Employee Number: 100
Employee Salary: 100000
Employee Car Info:
        Car name:Innova
        Model:2.5v
        Color=Grey
```

-->In the above example Employee class extends Person class functionality but just uses Car class functionality.

Composition vs Aggregation:
==========================

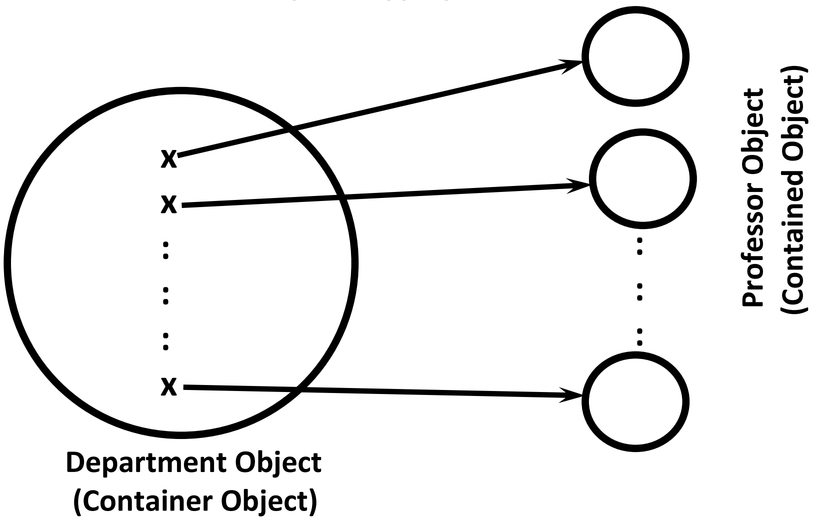
Composition:

-->Without existing container object if there is no chance of existing contained object then the container and contained objects are strongly associated and that strong association is nothing but Composition.
Ex: University contains several Departments and without existing university object there is no chance of existing Department object. Hence University and Department objects are strongly associated and this strong association is nothing but Composition.



Aggregation:

-->Without existing container object if there is a chance of existing contained object then the container and contained objects are weakly associated and that weak association is nothing but Aggregation.
-->Ex: Department contains several Professors. Without existing Department still there may be a chance of existing Professor. Hence Department and Professor objects are weakly associated, which is nothing but Aggregation.



Coding Example:

```
class Student:
        collegeName='DURGASOFT'
        def __init__(self,name):
                self.name=name
print(Student.collegeName)
s=Student('Durga')
print(s.name)
```

Output:
```
DURGASOFT
Durga
```

-->In the above example without existing Student object there is no chance of existing his name. Hence Student Object and his name are strongly associated which is nothing but Composition.

-->But without existing Student object there may be a chance of existing collegeName. Hence Student object and collegeName are weakly associated which is nothing but Aggregation.

Conclusion:
The relation between object and its instance variables is always Composition where as the relation between object and static variables is Aggregation.

Note:
Whenever we are creating child class object then child class constructor will be executed. If the child class does not contain constructor then parent class constructor will be executed, but parent object won't be created.

Ex:

```
class P:
        def __init__(self):
                print(id(self))

class C(P):
        pass
c=C()
print(id(c))
```

```
o/p:D:\pythonclasses>py test.py
3000655032728
3000655032728
```

Eg:

```
class Person:
        def __init__(self,name,age):
                self.name=name
                self.age=age
class Student(Person):
        def __init__(self,name,age,rollno,marks):
                super().__init__(name,age)
                self.rollno=rollno
                self.marks=marks
        def __str__(self):
                return 'Name={}\nAge={}\nRollno={}\nMarks={}'.format(self.name,self.age,self.rollno,self.marks)
s1=Student('Mahesh',48,101,90)
print(s1)
```

o/p:

```
Output:
Name=Mahesh
Age=48
Rollno=101
Marks=90
```

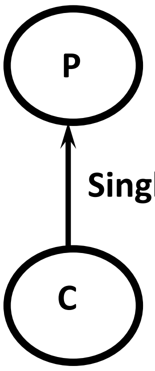
Note: In the above example when ever we are creating child class object both parent and child class constructors got executed to perform initialization of child object

Types of Inheritances:

====================

1).Single Inheritance:

The concept of inheriting the properties from one class to another class is known as single inheritance.



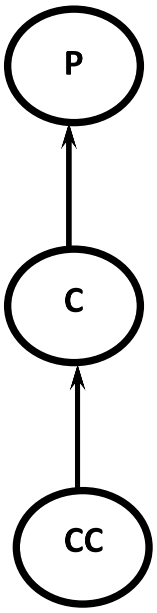
Ex:

```
class P:
        def m1(self):
                print("Parent Method")
class C(P):
        def m2(self):
                print("Child Method")
c=C()
c.m1()
c.m2()
```

```
o/p:D:\pythonclasses>py test.py
Parent Method
Child Method
```

2).Multi Level Inheritance:

The concept of inheriting the properties from multiple classes to single class with the concept of one after another is known as multilevel inheritance.



Multi – Level Inheritance

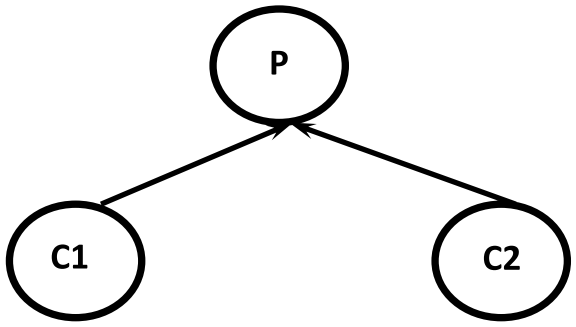
Ex:

```
class P:
        def m1(self):
                print("Parent Method")
class C(P):
        def m2(self):
                print("Child Method")
class CC(C):
        def m3(self):
                print("Sub Child Method")
c=CC()
c.m1()
c.m2()
c.m3()
```

```
o/p:D:\pythonclasses>py test.py
Parent Method
Child Method
Sub Child Method
```

3).Hierarchical Inheritance:

The concept of inheriting properties from one class into multiple classes which are present at same level is known as Hierarchical Inheritance



Hierarchical
Inheritance

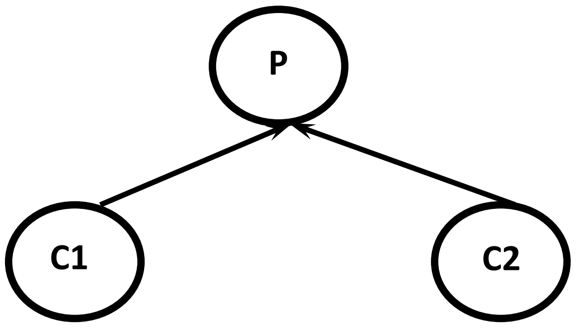
Ex:

```
class P:
        def m1(self):
                print("Parent Method")
class C1(P):
        def m2(self):
                print("Child1 Method")
class C2(P):
        def m3(self):
                print("Child2 Method")
c1=C1()
c1.m1()
c1.m2()
c2=C2()
c2.m1()
c2.m3()
```

```
o/p:D:\pythonclasses>py test.py
Parent Method
Child1 Method
Parent Method
Child2 Method
```

4).Multiple Inheritance:

The concept of inheriting the properties from multiple classes into a single class at a time, is known as multiple inheritance.



Hierarchical Inheritance

Ex:

```
class P1:
        def m1(self):
                print("Parent1 Method")
class P2:
        def m2(self):
                print("Parent2 Method")
class C(P1,P2):
        def m3(self):
                print("Child Method")
c=C()
c.m1()
c.m2()
c.m3()
```

```
o/p:D:\pythonclasses>py test.py
Parent1 Method
Parent2 Method
Child Method
```

-->If the same method is inherited from both parent classes,then Python will always consider the order of Parent classes in the declaration of the child class.

class C(P1,P2): ===>P1 method will be considered
class C(P2,P1): ===>P2 method will be considered

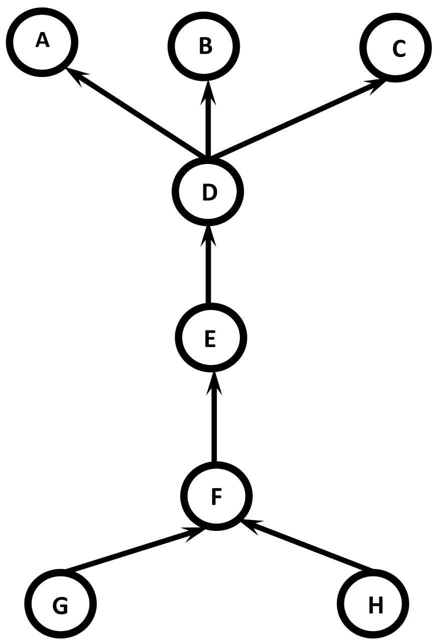
Ex-2:

```
class P1:
        def m(self):
                print("Parent1 Method")
class P2:
        def m(self):
                print("Parent2 Method")
class C(P2,P1):
        def m1(self):
                print("Child Method")
c=C()
c.m()
c.m1()
```

o/p:
Parent2 Method
Child Method

5).Hybrid Inheritance:

Combination of Single, Multi level, multiple and Hierarchical inheritance is known as Hybrid Inheritance.



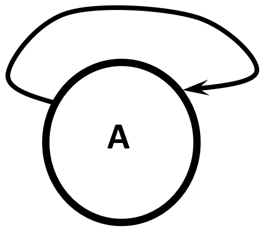
$$\begin{tikzcd} A & B & C \\ & D \arrow[ul] \arrow[u] \arrow[ur] & \\ & E \arrow[u] & \\ & F \arrow[u] & \\ G \arrow[ur] & & H \arrow[ul] \end{tikzcd}$$

6).Cyclic Inheritance:

The concept of inheriting properties from one class to another class in cyclic way, is called Cyclic inheritance.Python won't support for Cyclic Inheritance of course it is really not required.

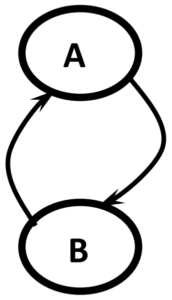
Ex-1:

```
class A(A):pass
NameError: name 'A' is not defined
```



Ex-2:

```
class A(B):
        pass
class B(A):
        pass
NameError: name 'B' is not defined
```



Method Resolution Order (MRO):

-->In Hybrid Inheritance the method resolution order is decided based on MRO algorithm.
-->This algorithm is also known as C3 algorithm.
-->Samuele Pedroni proposed this algorithm.
-->It follows DLR (Depth First Left to Right)
-->i.e Child will get more priority than Parent.
-->Left Parent will get more priority than Right Parent

MRO(X)=X+Merge(MRO(P1),MRO(P2),...,ParentList)

Head Element vs Tail Terminology:

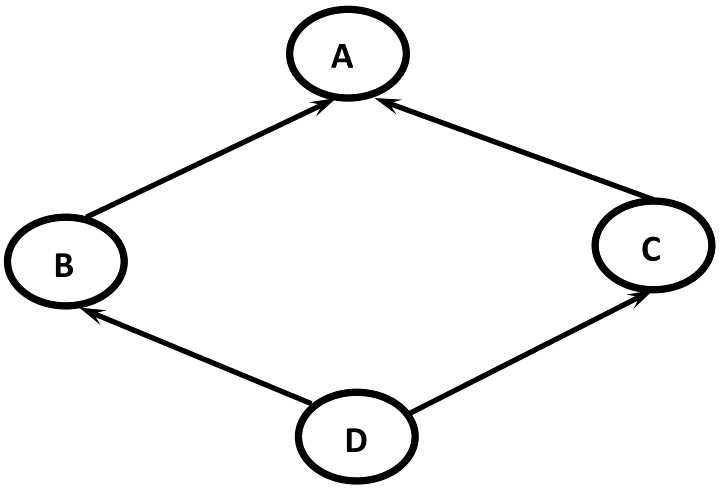
Assume C1,C2,C3,...are classes.
In the list : C1C2C3C4C5....
C1 is considered as Head Element and remaining is considered as Tail.

How to find Merge:

-->Take the head of first list.
-->If the head is not in the tail part of any other list,then add this head to the result and remove it from the lists in the merge.
-->If the head is present in the tail part of any other list,then consider the head element of the next list and continue the same process.

Note: We can find MRO of any class by using mro() function.
Syn:print(ClassName.mro())

Demo Program-1 for Method Resolution Order:
--



```
1) mro(A)=A,object
2) mro(B)=B,A,object
3) mro(C)=C,A,object
4) mro(D)=D,B,C,A,object
```

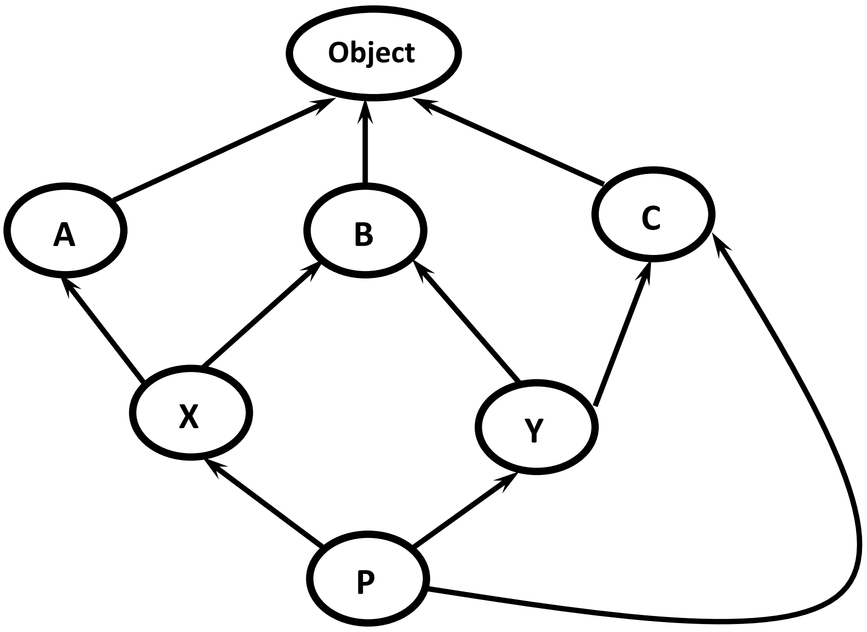
test.py

```
1) class A:pass
2) class B(A):pass
3) class C(A):pass
```

```
4) class D(B,C):pass
5) print(A.mro())
6) print(B.mro())
7) print(C.mro())
8) print(D.mro())
```

o/p:
```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>]
```

Demo Program-2 for Method Resolution Order:



```
1) mro(A)=A,object
2) mro(B)=B,object
3) mro(C)=C,object
4) mro(X)=X,A,B,object
5) mro(Y)=Y,B,C,object
6) mro(P)=P,X,A,Y,B,C,object
```

Finding mro(P) by using C3 algorithm:

Formula: MRO(X)=X+Merge(MRO(P1),MRO(P2),...,ParentList)

```
mro(p)= P+Merge(mro(X),mro(Y),mro(C),XYC)
         = P+Merge(XABO,YBCO,CO,XYC)
         = P+X+Merge(ABO,YBCO,CO,YC)
         = P+X+A+Merge(BO,YBCO,CO,YC)
         = P+X+A+Y+Merge(BO,BCO,CO,C)
         = P+X+A+Y+B+Merge(O,CO,CO,C)
         = P+X+A+Y+B+C+Merge(O,O,O)
         = P+X+A+Y+B+C+O
```

test.py:

```
class A:pass
class B:pass
class C:pass
class X(A,B):pass
class Y(B,C):pass
class P(X,Y,C):pass
print(A.mro())#AO
print(X.mro())#XABO
print(Y.mro())#YBCO
print(P.mro())#PXAYBCO
```

Output:

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.X'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
[<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>, <class '__main__.Y'>,
<class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
```

test.py:

```
class A:
    def m1(self):
        print('A class Method')
class B:
    def m1(self):
        print('B class Method')
```

```
class C:
        def m1(self):
                print('C class Method')
class X(A,B):
        def m1(self):
                print('X class Method')
class Y(B,C):
        def m1(self): 1
                print('Y class Method')
class P(X,Y,C):
        def m1(self):
                print('P class Method')
p=P()
p.m1()
```

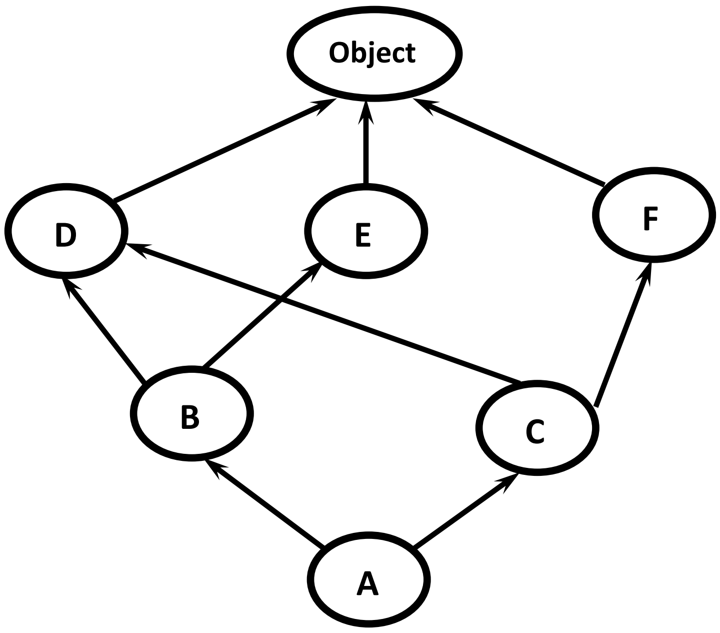
Output:

```
P class Method
```

-->In the above example P class m1() method will be considered.
-->If P class does not contain m1() method then as per MRO, X class method will be considered.
-->If X class does not contain then A class method will be considered and this process will be continued.

-->The method resolution in the following order:PXAYBCO

Demo Program-3 for Method Resolution Order:



```
mro(o)=object
mro(D)=D,object
mro(E)=E,object
mro(F)=F,object
mro(B)=B,D,E,object
mro(C)=C,D,F,object
mro(A)=A+Merge(mro(B),mro(C),BC)
        =A+Merge(BDEO,CDFO,BC)
        =A+B+Merge(DEO,CDFO,C)
        =A+B+C+Merge(DEO,DFO)
        =A+B+C+D+Merge(EO,FO)
        =A+B+C+D+E+Merge(O,FO)
        =A+B+C+D+E+F+Merge(O,O)
        =A+B+C+D+E+F+O
```

test.py:

```
class D:pass
class E:pass
class F:pass
class B(D,E):pass
class C(D,F):pass
class A(B,C):pass
print(D.mro())
print(B.mro())
print(C.mro())
print(A.mro())
```

Output:

```
[<class '__main__.D'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.D'>, <class '__main__.E'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.D'>, <class '__main__.F'>, <class 'object'>]
[<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.D'>,
<class '__main__.E'>, <class '__main__.F'>, <class 'object'>]
```

super() Method:

----

super() is a built-in method which is useful to call the super class constructors,variables and methods from the child class.

Demo Program-1 for super():

```
class Person:
        def __init__(self,name,age):
                self.name=name
                self.age=age 5)
        def display(self):
                print('Name:',self.name)
                print('Age:',self.age)
class Student(Person):
        def __init__(self,name,age,rollno,marks):
                super().__init__(name,age)
                self.rollno=rollno
                self.marks=marks
        def display(self):
                super().display() 17)
                print('Roll No:',self.rollno)
                print('Marks:',self.marks)
s1=Student('Mahesh',22,101,90)
s1.display()
```

Output:

```
Name: Mahesh
Age: 22
Roll No: 101
Marks: 90
```

-->In the above program we are using super() method to call parent class constructor and display() method.

Demo Program-2 for super():

```
class P:
        a=10
        def __init__(self):
                self.b=10
        def m1(self):
```

```
            print('Parent instance method')
    @classmethod
    def m2(cls):
            print('Parent class method')
    @staticmethod
    def m3():
            print('Parent static method')

class C(P):
    a=888
    def __init__(self):
            self.b=999
            super().__init__()
            print(super().a)
            super().m1()
            super().m2()
            super().m3()
c=C()
```

Output:

```
-----------
10
Parent instance method
Parent class method
Parent static method
```

-->In the above example we are using super() to call various members of Parent class.

How to call method of a particular Super class:

-->We can use the following approaches

1).super(D,self).m1():

It will call m1() method of super class of D.

2. A.m1(self):

It will call A class m1() method.

```
class A:
    def m1(self):
            print('A class Method')
class B(A):
```

```
    def m1(self):
        print('B class Method')
class C(B):
    def m1(self):
        print('C class Method')
class D(C): 1
    def m1(self):
        print('D class Method')
class E(D):
    def m1(self):
        A.m1(self)
e=E()
e.m1()
```

Output:

A class Method

Various Important Points about super():

Case-1:

From child class we are not allowed to access parent class instance variables by using super(),Compulsory we should use self only. But we can access parent class static variables by using super().

Ex:

```
class P:
    a=10
    def __init__(self):
        self.b=20
class C(P):
    def m1(self):
        print(super().a)#valid
        print(self.b)#valid
        print(super().b)#invalid
c=C()
c.m1()
```

Output:

```
10
20
AttributeError: 'super' object has no attribute 'b'
```

Case-2:

From child class constructor and instance method, we can access parent class instance method,static method and class method by using super().

```
class P:
    def __init__(self):
        print('Parent Constructor')
    def m1(self):
        print('Parent instance method')
    @classmethod
    def m2(cls):
        print('Parent class method')
    @staticmethod
    def m3():
        print('Parent static method')
class C(P):
    def __init__(self):
        super().__init__()
        super().m1()
        super().m2()
        super().m3()
    def m1(self):
        super().__init__()
        super().m1()
        super().m2()
        super().m3()
c=C()
c.m1()
```

Output:

```
Parent Constructor
Parent instance method
Parent class method
Parent static method
Parent Constructor
Parent instance method
```

Parent class method
Parent static method

Case-3:

From child class, class method we cannot access parent class instance methods and constructors by using super() directly(but indirectly possible). But we can access parent class static and class methods.

```
class P:
        def __init__(self):
                print('Parent Constructor')
        def m1(self):
                print('Parent instance method')
        @classmethod
        def m2(cls):
                print('Parent class method')
        @staticmethod
        def m3():
                print('Parent static method')
class C(P):
        @classmethod
        def m1(cls):
                #super().__init__()--->invalid
                #super().m1()--->invalid
                super().m2()
                super().m3()
C.m1()
```

Output:

Parent class method
Parent static method

From Class Method of Child class,how to call parent class instance methods and constructors:

```
class A:
        def __init__(self):
                print('Parent constructor')
        def m1(self):
                print('Parent instance method')
class B(A):
```

```
    @classmethod
    def m2(cls):
        super(B,cls).__init__(cls)
        super(B,cls).m1(cls)
B.m2()
```

Output:

Parent constructor
Parent instance method

Case-4:

In child class static method we are not allowed to use super() generally (But in special way we can use).

```
class P:
    def __init__(self):
        print('Parent Constructor')
    def m1(self):
        print('Parent instance method')
    @classmethod
    def m2(cls):
        print('Parent class method')
    @staticmethod
    def m3():
        print('Parent static method')

class C(P):
    @staticmethod
    def m1():
        super().m1()-->invalid
        super().m2()--->invalid
        super().m3()--->invalid
C.m1()
```

RuntimeError: super(): no arguments

How to call parent class static method from child class static method by using super():

```
class A:
    @staticmethod
    def m1():
        print('Parent static method')
class B(A):
    @staticmethod
    def m2():
        super(B,B).m1()
B.m2()
```

Output:

```
Parent static method
```

OOP's-PART-3

Polymorphism
=============
-->Poly means many.
-->Morphs means forms.
-->Polymorphism means 'Many Forms'.

Ex-1:
Yourself is best example of polymorphism.In front of Your parents You will have one type of behaviour and with friends another type of behaviour.Same person but different behaviours at different places,which is nothing but polymorphism.

Ex-2:
+ operator acts as concatenation and arithmetic addition

Ex-3:
* operator acts as multiplication and repetition operator

Ex-4:
The Same method with different implementations in Parent class and child classes.(overriding)

Related to polymorphism the following 4 topics are important
1. Duck Typing Philosophy of Python
2. Overloading
    -->Operator Overloading
    -->Method Overloading
    -->Constructor Overloading
3. Overriding
    -->Method overriding
    -->constructor overriding

1. Duck Typing Philosophy of Python:

-->In Python we cannot specify the type explicitly.
-->Based on provided value at runtime the type will be considered automatically.
-->Hence Python is considered as Dynamically Typed Programming Language.

```
def f1(obj):
        obj.talk()
```

What is the type of obj? We cannot decide at the beginning. At runtime we can pass any type.Then how we can decide the type? At runtime if 'it walks like a duck and talks like a duck,it must be duck'. Python follows this principle. This is called Duck Typing Philosophy of Python.

Demo Program:

```
class Duck:
        def talk(self):
                print('Quack.. Quack..')

class Dog:
        def talk(self):
                print('Bow Bow..')

class Cat:
        def talk(self):
                print('Moew Moew ..')

class Goat:
        def talk(self):
                print('Myaah Myaah ..')

def f1(obj):
obj.talk()

l=[Duck(),Cat(),Dog(),Goat()]
for obj in l:
        f1(obj)
```

Output:

```
Quack.. Quack..
Moew Moew ..
Bow Bow..
Myaah Myaah ..
```

-->The problem in this approach is if obj does not contain talk() method then we will get AttributeError.

Ex:

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Dog:
    def bark(self):
        print('Bow Bow..')

def f1(obj):
    obj.talk()
d=Duck()
f1(d)

d=Dog()
f1(d)
```

```
o/p: D:\pythonclasses>py test.py
Quack.. Quack..
Traceback (most recent call last):
File "test.py", line 22, in <module>
  f1(d)
File "test.py", line 13, in f1
  obj.talk()
AttributeError: 'Dog' object has no attribute 'talk'
```

-->But we can solve this problem by using hasattr() function.

-->hasattr(obj,'attributename')
-->attributename can be method name or variable name

Demo Program with hasattr() function:

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Human:
    def talk(self):
        print('Hello Hi...')

```

```
class Dog:
    def bark(self):
        print('Bow Bow..')

def f1(obj):
    if hasattr(obj,'talk'):
        obj.talk()
    elif hasattr(obj,'bark'):
        obj.bark()

d=Duck()
f1(d)

h=Human()
f1(h)

d=Dog()
f1(d)
```

Overloading:
===========
-->We can use same operator or methods for different purposes.

Ex-1:
+ operator can be used for Arithmetic addition and String concatenation
print(10+20)#30 print('durga'+'soft')#durgasoft

Ex-2:
* operator can be used for multiplication and string repetition purposes.
print(10*20)#200 print('mahesh'*3)#maheshmaheshmahesh

Ex-3:
We can use deposit() method to deposit cash or cheque or dd
deposit(cash)
deposit(cheque)
deposit(dd)

There are 3 types of overloading
1. Operator Overloading
2. Method Overloading
3. Constructor Overloading

1. Operator Overloading:

-->We can use the same operator for multiple purposes, which is nothing but operator overloading.

-->Python supports operator overloading.

Ex-1:

+ operator can be used for Arithmetic addition and String concatenation
print(10+20)#30 print('durga'+'soft')#durgasoft

Ex-2:

* operator can be used for multiplication and string repetition purposes.
print(10*20)#200 print('durga'*3)#durgadurgadurga

Demo program to use + operator for our class objects:

```
1) class Book:
2)     def __init__(self,pages):
3)         self.pages=pages
4)
5) b1=Book(100)
6) b2=Book(200)
7) print(b1+b2)
```

```
D:\pythonclasses>py test.py
Traceback (most recent call last):
File "test.py", line 7, in <module>
    print(b1+b2)
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

-->We can overload + operator to work with Book objects also. i.e Python supports Operator Overloading.

-->For every operator Magic Methods are available. To overload any operator we have to override that Method in our class.

-->Internally + operator is implemented by using __add__() method.This method is called magic method for + operator. We have to override this method in our class.

Demo program to overload + operator for our Book class objects:

```
class Book:
        def __init__(self,pages):
                self.pages=pages

        def __add__(self,other):
                return self.pages+other.pages

b1=Book(100)
b2=Book(200)
print('The Total Number of Pages:',b1+b2)
```

Output: The Total Number of Pages: 300

The following is the list of operators and corresponding magic methods.

+ ---> object.__add__(self,other)
- ---> object.__sub__(self,other)
* ---> object.__mul__(self,other)
/ ---> object.__div__(self,other)
// ---> object.__floordiv__(self,other)
% ---> object.__mod__(self,other)
** ---> object.__pow__(self,other)
+= ---> object.__iadd__(self,other)
-= ---> object.__isub__(self,other)
*= ---> object.__imul__(self,other)
/= ---> object.__idiv__(self,other)
//= ---> object.__ifloordiv__(self,other)
%= ---> object.__imod__(self,other)
**= ---> object.__ipow__(self,other)
< ---> object.__lt__(self,other)
<= ---> object.__le__(self,other)
> ---> object.__gt__(self,other)
>= ---> object.__ge__(self,other)
== ---> object.__eq__(self,other)
!= ---> object.__ne__(self,other)

Ex: How to add multiple objects

```
class Book:
        def __init__(self,pages):
                self.pages=pages

        def __add__(self,other):
                print("add method calling")
                total=self.pages+other.pages
                return Book(total)

        def __str__(self):
                return str(self.pages)

b1=Book(100)
b2=Book(200)
bx=b1+b2+b3
print(bx)
#print(bx.pages)
```

```
o/p:D:\pythonclasses>py test.py
add method calling
add method calling
600
```

Overloading >,>= and <,<= operators for Student class objects:

```
class Student:
        def __init__(self,name,marks):
                self.name=name
                self.marks=marks

        def __gt__(self,other):
                return self.marks>other.marks

        def __le__(self,other):
                return self.marks<=other.marks

print("10>20 =",10>20)
s1=Student("Durga",100)
s2=Student("Mahesh",200)
print("s1>s2=",s1>s2)
```

```
print("s1<s2=",s1<s2)
print("s1<=s2=",s1<=s2)
print("s1>=s2=",s1>=s2)
```

Output:

```
-----------
10>20 = False
s1>s2= False
s1<s2= True
s1<=s2= True
s1>=s2= False
```

Program to overload multiplication operator to work on Employee objects:

```
class Employee:
        def __init__(self,name,salary):
                self.name=name
                self.salary=salary

        def __mul__(self,other):
                return self.salary*other.days

class TimeSheet:
        def __init__(self,name,days):
                self.name=name
                self.days=days

        def __mul__(self,other):
                return self.days*other.salary

e=Employee('Durga',500)
t=TimeSheet('Durga',25)
print('This Month Salary:',e*t)
print('This Month Salary:',t*e)
```

Output: This Month Salary: 12500
Output: This Month Salary: 12500

2. Method Overloading:

-->If 2 methods having same name but different type of arguments then those methods are said to be overloaded methods.

Ex: m1(int a)
m1(double d)

-->But in Python Method overloading is not possible.
-->If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

Demo Program:

```
class Test:
	def m1(self):
		print('no-arg method')
	def m1(self,a):
		print('one-arg method')
	def m1(self,a,b):
		print('two-arg method')
t=Test()
#t.m1()
#t.m1(10)
t.m1(10,20)
```

Output: two-arg method

-->In the above program python will consider only last method.

How we can handle overloaded method requirements in Python:

-->Most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable number of argument methods.

```
class Test:
	def sum(self,a=None,b=None,c=None):
		if a!=None and b!= None and c!= None:
			print('The Sum of 3 Numbers:',a+b+c)
		elif a!=None and b!= None:
			print('The Sum of 2 Numbers:',a+b)
		else: 8) print('Please provide 2 or 3 arguments')
t=Test()
```

Output:

```
The Sum of 2 Numbers: 30
The Sum of 3 Numbers: 60
Please provide 2 or 3 arguments
```

Demo Program with Variable Number of Arguments:

```python
class Test:
        def sum(self,*a):
                total=0
                for x in a:
                        total=total+x
                print('The Sum:',total)
t=Test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10) 13)
t.sum()
```

3. Constructor Overloading:

-->Constructor overloading is not possible in Python.
-->If we define multiple constructors then the last constructor will be considered.

```python
class Test:
        def __init__(self):
                print('No-Arg Constructor')

        def __init__(self,a):
                print('One-Arg constructor')

        def __init__(self,a,b):
                print('Two-Arg constructor')

#t1=Test()
#t1=Test(10)
t1=Test(10,20)
```

Output: Two-Arg constructor
-->In the above program only Two-Arg Constructor is available.
-->But based on our requirement we can declare constructor with default arguments and variable number of arguments.

Constructor with Default Arguments:

```
class Test:
        def __init__(self,a=None,b=None,c=None):
                print('Constructor with 0|1|2|3 number of arguments')

t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
```

Output:

```
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
```

Constructor with Variable Number of Arguments:

```
class Test:
        def __init__(self,*a):
                print('Constructor with variable number of arguments')

t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
t5=Test(10,20,30,40,50,60)
```

Output:

```
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
```

Method overriding:

-->What ever members available in the parent class are bydefault available to the child class through inheritance.
-->If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement.
-->This concept is called overriding.
-->Overriding concept applicable for both methods and constructors.

Demo Program for Method overriding:

```
class P:
        def property(self):
                print('Gold+Land+Cash+Power')
        def marry(self):
                print('Appalamma')

class C(P):
        def marry(self):
                print('Katrina Kaif')

c=C()
c.property()
c.marry()
```

Output:

```
Gold+Land+Cash+Power
Katrina Kaif
```

-->From Overriding method of child class,we can call parent class method also by using super() method.

```
class P:
        def property(self):
                print('Gold+Land+Cash+Power')
        def marry(self):
                print('Appalamma')

class C(P):
        def marry(self):
                super().marry()
                print('Katrina Kaif')
```

```
c=C() 12)
c.property()
c.marry()
```

Output:

```
Gold+Land+Cash+Power
Appalamma
Katrina Kaif
```

Demo Program for Constructor overriding:

```
class P:
    def __init__(self):
        print('Parent Constructor')

class C(P):
    def __init__(self):
        print('Child Constructor')

c=C()
```

Output: Child Constructor

-->In the above example,if child class does not contain constructor then parent class constructor will be executed

-->From child class constuctor we can call parent class constructor by using super() method.

Demo Program to call Parent class constructor by using super():

```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age

class Employee(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal
```

```python

    def display(self):
        print('Employee Name:',self.name)
        print('Employee Age:',self.age)
        print('Employee Number:',self.eno)
        print('Employee Salary:',self.esal)

e1=Employee('Mahesh',48,872425,26000)
e1.display()
e2=Employee('Sunny',39,872426,36000)
e2.display()
```

Output:

```
-----------
Employee Name: Mahesh
Employee Age: 48
Employee Number: 872425
Employee Salary: 26000
Employee Name: Sunny
Employee Age: 39
Employee Number: 872426
Employee Salary: 36000
```

OOP's-PART-4

Agenda:

1. Abstract Method
2. Abstract class
3. Interface
4. Public,Private and Protected Members
5. __str__() method
6. Difference between str() and repr() functions
7. Small Banking Application

Abstract Method:

-->Sometimes we don't know about implementation,still we can declare a method. Such type of methods are called abstract methods.i.e abstract method has only declaration but not implementation.
-->In python we can declare abstract method by using @abstractmethod decorator as follows.

```
@abstractmethod
def m1(self): pass
```

@abstractmethod decorator present in abc module. Hence compulsory we should import abc module,otherwise we will get error.

abc==>abstract base class module

```
class Test:
        @abstractmethod
        def m1(self):
                pass
```

NameError: name 'abstractmethod' is not defined

Ex:

```
from abc import *
class Test:
        @abstractmethod
        def m1(self):
                pass
```

Ex:

```
from abc import *
class Fruit:
        @abstractmethod
        def taste(self):
                pass
```

-->Child classes are responsible to provide implemention for parent class abstract methods.

Abstract class:

-->Some times implementation of a class is not complete,such type of partially implementation classes are called abstract classes.
-->Every abstract class in Python should be derived from ABC class which is present in abc module.

Case-1:

```
from abc import *
class Test:
        pass
t=Test()
```

-->In the above code we can create object for Test class b'z it is concrete class and it does not conatin any abstract method.

Case-2:

```
from abc import *
class Test(ABC):
        pass
t=Test()
```

-->In the above code we can create object,even it is derived from ABC class,b'z it does not contain any abstract method.

Case-3:

```
from abc import *
class Test(ABC):
@abstractmethod
        def m1(self):
                pass
t=Test()
```

TypeError: Can't instantiate abstract class Test with abstract methods m1

Case-4:

```
from abc import *
class Test:
        @abstractmethod
        def m1(self):
                pass
t=Test()
```

-->We can create object even class contains abstract method b'z we are not extending ABC class.

Case-5:

```
from abc import *
class Test:
        @abstractmethod
        def m1(self):
                print('Hello')
t=Test()
t.m1()
```

Output: Hello

Conclusion:

-->If a class contains atleast one abstract method and if we are extending ABC class then instantiation is not possible.

-->"abstract class with abstract method instantiation is not possible"

-->Parent class abstract methods should be implemented in the child classes. otherwise we cannot instantiate child class.If we are not creating child class object then we won't get any error.

Case-1:

```
from abc import *
class Vehicle(ABC):
        @abstractmethod
        def noofwheels(self):
                pass

class Bus(Vehicle):
        pass
```

-->It is valid b'z we are not creating Child class object

Case-2:

```
from abc import *
class Vehicle(ABC):
@abstractmethod
def noofwheels(self):
        pass

class Bus(Vehicle):
        pass

b=Bus()
```

TypeError: Can't instantiate abstract class Bus with abstract methods noofwheels

Note: If we are extending abstract class and does not override its abstract method then child class is also abstract and instantiation is not possible.

Ex:

```
from abc import *
class Vehicle(ABC):
    @abstractmethod
    def noofwheels(self):
        pass

class Bus(Vehicle):
    def noofwheels(self):
        return 7

class Auto(Vehicle):
    def noofwheels(self):
        return 3

b=Bus()
print(b.noofwheels())#7

a=Auto()
print(a.noofwheels())#3
```

Note: Abstract class can contain both abstract and non-abstract methods also.

Interfaces In Python

Concreate class vs Abstract Class vs Inteface:
---
1. If we dont know anything about implementation just we have requirement specification then we should go for interface.

2. If we are talking about implementation but not completely then we should go for abstract class.(partially implemented class).

3. If we are talking about implementation completely and ready to provide service then we should go for concrete class.

Ex:

```
from abc import *
class CollegeAutomation(ABC):
    @abstractmethod
    def m1():pass

    @abstractmethod
    def m2():pass

    @abstractmethod
    def m3():pass

class AbsClass(CollegeAutomation):
    def m1(self):
        print("M1 method implementation")

    def m2(self):
        print("M2 method implementation")

class ConcreteClass(AbsClass):
    def m3(self):
        print("M3 method implementation")

c=ConcreteClass()
c.m1()
c.m2()
c.m3()
```

```
o/p:D:\pythonclasses>py test.py
M1 method implementation
M2 method implementation
M3 method implementation
```

-->In general if an abstract class contains only abstract methods such type of abstract class is considered as interface.

Demo program:

```
from abc import *
class DBinterface(ABC):
        @abstractmethod
        def connect(self):pass

        @abstractmethod
        def disconnect(self):pass

class Oracle(DBinterface):
        def connect(self):
                print("Connecting to Oracle Database......")
        def disconnect(self):
                print("Disconnecting to Oracle Database......")

class Sybase(DBinterface):
        def connect(self):
                print("Connecting to Sybase Database......")
        def disconnect(self):
                print("Disconnecting to Sybase Database......")

'''o=Oracle()
o.connect()
o.disconnect()'''

dname=input("Enter Database:")
print(globals()[dname])
className=globals()[dname]
c=className()
c.connect()
c.disconnect()
```

```
o/p:D:\pythonclasses>py test.py
Enter Database:Oracle
<class '__main__.Oracle'>
Connecting to Oracle Database......
Disconnecting to Oracle Database......

D:\pythonclasses>py test.py
Enter Database:Sybase
<class '__main__.Sybase'>
Connecting to Sybase Database......
Disconnecting to Sybase Database......
```

Note: The inbuilt function globals()[str] converts the string 'str' into a class name and returns the classname.

Demo Program-2:

Reading class name from the file

config.txt:

```
EPSON
```

test.py:

```
from abc import *
class Printer(ABC):
	@abstractmethod
	def printit(self,text):
		pass
	@abstractmethod
	def disconnect(self):
		pass

class EPSON(Printer):
	def printit(self,text):
		print('Printing from EPSON Printer...')
		print(text)
	def disconnect(self):
		print('Printing completed on EPSON Printer...')

class HP(Printer):
	def printit(self,text):
		print('Printing from HP Printer...')
```

```
        print(text)
    def disconnect(self):
        print('Printing completed on HP Printer...')

with open('config.txt','r') as f:
    pname=f.readline()

classname=globals()[pname]
x=classname()
x.printit('This data has to print...')
x.disconnect()
```

Output:

```
Printing from EPSON Printer...
This data has to print...
Printing completed on EPSON Printer...
```

Public, Protected and Private Attributes:

-->By default every attribute is public. We can access from anywhere either within the class or from outside of the class.

Ex: name='Mahesh'

test.py:

```
class Test:
    x=10
    def __init__(self):
        self.y=20
```

test1.py:

```
from test import Test
class Test1:
    t=Test()
    print(t.x)
    print(t.y)
```

o/p:
10
20

-->Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes. We can specify an attribute as protected by prefexing with _ symbol.

syntax: _variablename=value
Ex: _name='Mahesh'

Ex:

```
class Test:
        _x=10
        def __init__(self):
                self._y=20
t=Test()
print(t._x)
print(t._y)
```

o/p:
10
20

-->But is is just convention and in reality does not exists protected attributes.

-->private attributes can be accessed only within the class.i.e from outside of the class we cannot access. We can declare a variable as private explicitly by prefexing with 2 underscore symbols.

syntax: __variablename=value
Ex: __name='Mahesh'

Demo Program:

```
class Test:
        x=10
        _y=20
        __z=30
        def m1(self):
                print(Test.x)
                print(Test._y)
                print(Test.__z)
```

```
9)  t=Test()
10) t.m1()
11) print(Test.x)
12) print(Test._y)
13) print(Test.__z)
```

Output:

```
----------
10
20
30
10
20
Traceback (most recent call last):
File "test.py", line 14, in <module>
        print(Test.__z)
AttributeError: type object 'Test' has no attribute '__z'
```

How to access private variables from outside of the class:

--

-->We cannot access private variables directly from outside of the class.
-->But we can access indirectly as follows

objectreference._classname__variablename

Ex:

```
class Test:
        __x=10
        def __init__(self):
                self.__y=20
t=Test()
print(t.__dict__)#{'_Test__y': 20}
print(t._Test__y)#20
print(Test._Test__x)#10
```

__str__() method:

===============

-->Whenever we are printing any object reference internally __str__() method will be called which is returns string in the following format

<__main__.classname object at 0x022144B0>

-->To return meaningful string representation we have to override __str__() method.

Demo Program:

```
class Student:
        def __init__(self,name,rollno):
                self.name=name
                self.rollno=rollno
        def __str__(self):
                return 'This is Student with Name:{} and Rollno:{}'.format(self.name,self.rollno)

s1=Student('Mahesh',101)
s2=Student('Durga',102)
print(s1)
print(s2)
```

```
o/p:D:\pythonclasses>py test.py
This is student name:Mahesh and Roll no:101
This is student name:Durga and Roll no:102
```

output without overriding __str__():

```
<__main__.Student object at 0x022144B0>
<__main__.Student object at 0x022144D0>
```

output with overriding __str__():

```
This is Student with Name:Durga and Rollno:101
This is Student with Name:Mahesh and Rollno:102
```

Difference between str() and repr() (OR) Difference between __str__() and __repr__():

-->str() internally calls __str__() function and hence functionality of both is same.

-->Similarly,repr() internally calls __repr__() function and hence functionality of both is same.

-->str() returns a string containing a nicely printable representation object.

-->The main purpose of str() is for readability.It may not possible to convert result string to original object.

Ex:

```
import datetime
today=datetime.datetime.now()
s=str(today)#converting datetime object to str
print(s)
d=eval(s)#converting str object to datetime
print(d)
```

```
o/p:py test.py
2018-11-30 09:06:50.207142
D:\pythonclasses>py test.py
2018-11-30 09:07:40.313067
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    d=eval(s)
  File "<string>", line 1
    2018-11-30 09:07:40.313067
             ^
SyntaxError: invalid token
```

-->But repr() returns a string containing a printable representation of object.
-->The main goal of repr() is unambigouous.
-->We can convert result string to original object by using eval() function,which may not possible in str() function.

Ex:

```
import datetime
today=datetime.datetime.now()
s=repr(today)#converting datetime object to str
print(s)
d=eval(s)#converting str object to datetime
print(d)
print(type(d))
```

```
o/p:D:\pythonclasses>py test.py
datetime.datetime(2018, 11, 30, 9, 9, 15, 322157)
2018-11-30 09:09:15.322157
<class 'datetime.datetime'>
```

Note: It is recommended to use repr() instead of str()

Mini Project: Banking Application

```
class Account:
        def __init__(self,name,balance,min_balance):
                self.name=name
                self.balance=balance
                self.min_balance=min_balance
        def deposit(self,amount):
                self.balance +=amount
        def withdraw(self,amount):
                if self.balance-amount >= self.min_balance:
                        self.balance -=amount
                else:
                        print("Sorry, Insufficient Funds")
        def printStatement(self):
                print("Account Balance:",self.balance)

class Current(Account):
        def __init__(self,name,balance):
                super().__init__(name,balance,min_balance=-1000)
                def __str__(self):
       return "{}'s Current Account with Balance :{}".format(self.name,self.balance)

class Savings(Account):
        def __init__(self,name,balance):
                super().__init__(name,balance,min_balance=0)
                def __str__(self):
       return "{}'s Savings Account with Balance :{}".format(self.name,self.balance)

c=Savings("Mahesh",10000)
print(c)
c.deposit(5000)
c.printStatement()
c.withdraw(16000)
c.withdraw(15000)
print(c)

c2=Current('Durga',20000)
c2.deposit(6000)
print(c2)
c2.withdraw(27000)
print(c2)
```

Multi Threading

Multi Tasking:

-->Executing several tasks simultaneously is the concept of multitasking.

There are 2 types of Multi Tasking:
---

1. **Process based Multi Tasking**
2. **Thread based Multi Tasking**

1).Process based Multi Tasking:
--
-->Executing several tasks simmultaneously where each task is a seperate independent process is called process based multi tasking.

Ex: while typing python program in the editor we can listen mp3 audio songs from the same system. At the same time we can download a file from the internet. All these taks are executing simultaneously and independent of each other. Hence it is process based multi tasking.

-->This type of multi tasking is best suitable at operating system level.

2).Thread based MultiTasking:

-->Executing several tasks simultaneously where each task is a seperate independent part of the same program, is called Thread based multi tasking, and each independent part is called a Thread.

-->This type of multi tasking is best suitable at programmatic level.

Note: Whether it is process based or thread based, the main advantage of multi tasking is to improve performance of the system by reducing response time.

-->The main important application areas of multi threading are:

1. **To implement Multimedia graphics**
2. **To develop animations**
3. **To develop video games**
4. **To develop web and application servers etc...**

Note: Where ever a group of independent jobs are available, then it is highly recommended to execute simultaneously instead of executing one by one.For such type of cases we should go for Multi Threading.

-->Python provides one inbuilt module "threading" to provide support for developing threads. Hence developing multi threaded Programs is very easy in python.

-->Every Python Program by default contains one thread which is nothing but MainThread.

Q.Program to print name of current executing thread:

```
import threading
print("Current Executing Thread:",threading.current_thread().getName())
```

o/p: Current Executing Thread: MainThread

Note: threading module contains function current_thread() which returns the current executing Thread object. On this object if we call getName() method then we will get current executing thread name.

The ways of Creating Thread in Python:

We can create a thread in Python by using 3 ways

1. **Creating a Thread without using any class**
2. **Creating a Thread by extending Thread class**
3. **Creating a Thread without extending Thread class**

1).Creating a Thread without using any class:

```
from threading import *
def display():
	print("This code(Display function) is executed by thread:",current_thread().getName())
t=Thread(target=display)#Main thread created child thread
t.start()#MainThread started child thread
print("This code executed by Thread:",current_thread().getName())
```

```
o/p:D:\pythonclasses>py test.py
This code(Display function) is executed by thread: Thread-1
This code executed by Thread: MainThread
```

Creating multiple threads:

```
from threading import *
def display():
        for i in range(10):
                print("Child Thread")

t=Thread(target=display)#Main thread created child thread
t.start()# started child thread
for i in range(10):
        print("Main Thread")
```

-->If multiple threads present in our program, then we cannot expect execution order and hence we cannot expect exact output for the multi threaded programs. B'z of this we cannot provide exact output for the above program.It is varied from machine to machine and run to run.

Note:

Thread is a pre-defined class present in threading module which can be used to create our own Threads.

2. Creating a Thread by extending Thread class:

We have to create child class for Thread class. In that child class we have to override run() method with our required job. Whenever we call start() method then automatically run() method will be executed and performs our job.

```
from threading import *
class MyThread(Thread):
        def run(self):
                for i in range(10):
                        print("Child Thread-1")
t=MyThread()
t.start()
for i in range(10):
        print("Main Thread-1")
```

3. Creating a Thread without extending Thread class:

```
from threading import *
class Test:
        def display(self):
```

```
        for i in range(10):
            print("child thread")
obj=Test()
t=Thread(target=obj.display)
t.start()
for i in range(10):
    print("Main Thread")
```

Without multithreading:

```
import time
def doubles(numbers):
    for n in numbers:
        time.sleep(1)
        print("Double is:",2*n)
def squares(numbers):
    for n in numbers:
        time.sleep(1)
        print("Squares is:",n*n)

numbers=[1,2,3,4,5,6]
begintime=time.time()
doubles(numbers)
squares(numbers)
endtime=time.time()
print("The total time taken:",endtime-begintime)
```

```
o/p:D:\pythonclasses>py test.py
Double is: 2
Double is: 4
Double is: 6
Double is: 8
Double is: 10
Double is: 12
Squares is: 1
Squares is: 4
Squares is: 9
Squares is: 16
Squares is: 25
Squares is: 36
The total time taken: 12.186678886413574
```

With multithreading:

```
from threading import *
import time
def doubles(numbers):
        for n in numbers:
                time.sleep(1)
                print("Double is:",2*n)
def squares(numbers):
        for n in numbers:
                time.sleep(1)
                print("Squares is:",n*n)

numbers=[1,2,3,4,5,6]
begintime=time.time()
t1=Thread(target=doubles,args=(numbers,))
t2=Thread(target=squares,args=(numbers,))
t1.start()
t2.start()
t1.join()
t2.join()
endtime=time.time()
print("The total time taken:",endtime-begintime)
```

```
o/p:D:\pythonclasses>py test.py
Squares is: 1
Double is: 2
Double is: 4
Squares is: 4
Squares is: 9
Double is: 6
Double is: 8
Squares is: 16
Squares is: 25
Double is: 10
Double is: 12
Squares is: 36
The total time taken: 6.09333062171936
```

Setting and Getting Name of a Thread:

-->Every thread in python has name. It may be default name generated by Python or Customized Name provided by programmer.

-->We can get and set name of thread by using the following Thread class methods.

-->t.getName() ==> Returns Name of Thread
-->t.setName(newName) ==>To set our own name

Note: Every Thread has implicit variable "name" to represent name of Thread.

Setting and getting name of thread:

```
from threading import *
print(current_thread().getName())
current_thread().setName("Sunny")
print(current_thread().getName())
print(current_thread().name)
```

```
o/p:D:\pythonclasses>py test.py
MainThread
Sunny
Sunny
```

Thread Identification Number (ident):

For every thread internally a unique identification number is available. We can access this id by using implicit variable "ident".

```
from threading import *
def test():
        print("Child Thread")
        print("Child Thread identification number:",current_thread().ident)
t=Thread(target=test)
t.start()
print("Main Thread identification number:",current_thread().ident)
print("Child Thread identification number:",t.ident)
```

```
o/p:D:\pythonclasses>py test.py
Child Thread
Main Thread identification number: 10348
Child Thread identification number: 3556
Child Thread identification number: 3556
```

active_count():

-->This function returns the number of active threads currently running.

```python
from threading import *
import time
def display():
        print(current_thread().getName(),".....started")
        time.sleep(3)
        print(current_thread().getName(),".....ended")
print("The number of active threads:",active_count())
t1=Thread(target=display,name="ChildThread1")
t2=Thread(target=display,name="ChildThread2")
t3=Thread(target=display,name="ChildThread3")
t1.start()
t2.start()
t3.start()
print("The number of active threads:",active_count())
time.sleep(5)
print("The number of active threads:",active_count())
```

```
o/p:D:\pythonclasses>py test.py
The number of active threads: 1
ChildThread1 .....started
ChildThread2 .....started
ChildThread3 .....started
The number of active threads: 4
ChildThread1 .....ended
ChildThread2 .....ended
ChildThread3 .....ended
The number of active threads: 1
```

enumerate() function:

-->This function returns a list of all active threads currently running.

```python
from threading import *
import time
def display():
        print(current_thread().getName(),".....started")
        time.sleep(3)
        print(current_thread().getName(),".....ended")
print("The number of active threads:",active_count())
t1=Thread(target=display,name="ChildThread1")
```

```
t2=Thread(target=display,name="ChildThread2")
t3=Thread(target=display,name="ChildThread3")
t1.start()
t2.start()
t3.start()
l=enumerate()
for t in l:
    print("Thread Name:",t.name)
time.sleep(5)
l=enumerate()
for t in l:
    print("Thread Name:",t.name)
```

```
o/p:D:\pythonclasses>py test.py
The number of active threads: 1
ChildThread1 .....started
ChildThread2 .....started
ChildThread3 .....started
Thread Name: MainThread
Thread Name: ChildThread1
Thread Name: ChildThread2
Thread Name: ChildThread3
ChildThread1 .....ended
ChildThread3 .....ended
ChildThread2 .....ended
Thread Name: MainThread
```

isAlive():

-->isAlive() method checks whether a thread is still executing or not.

```
from threading import *
import time
def display():
    print(current_thread().getName(),".....started")
    time.sleep(3)
    print(current_thread().getName(),".....ended")

t1=Thread(target=display,name="ChildThread1")
t2=Thread(target=display,name="ChildThread2")
t1.start()
t2.start()

print(t1.name,"is Alive:",t1.isAlive())
```

```
print(t2.name,"is Alive:",t2.isAlive())
time.sleep(5)
print(t1.name,"is Alive:",t1.isAlive())
print(t2.name,"is Alive:",t2.isAlive())
```

```
o/p:D:\pythonclasses>py test.py
ChildThread1 .....started
ChildThread2 .....started
ChildThread1 is Alive: True
ChildThread2 is Alive: True
ChildThread2 .....ended
ChildThread1 .....ended
ChildThread1 is Alive: False
ChildThread2 is Alive: False
```

join() method:

-->If a thread wants to wait until completing some other thread then we should go for join() method.

```
from threading import *
import time
def display():
    for i in range(10):
        print("Seetha Thread")
        time.sleep(2)
t=Thread(target=display)
t.start()
t.join()#This line executed by main thread
for i in range(10):
    print("Rama Thread")
```

-->In the above example Main Thread waited until completing child thread. In this case output is:

```
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
```

```
Seetha Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
```

Note: We can call join() method with time period also.

t.join(seconds)

-->In this case thread will wait only specified amount of time.

#join() method():with time

```python
from threading import *
import time
def display():
    for i in range(10):
        print("Seetha Thread")
        time.sleep(2)
t=Thread(target=display)
t.start()
t.join(5)
for i in range(10):
    print("Rama Thread")
```

-->In this case Main Thread waited only 5 seconds.

Output:

```
Seetha Thread
Seetha Thread
Seetha Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
```

```
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Rama Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
Seetha Thread
```

Summary of all methods related to threading module and Thread

Daemon Threads:

-->The threads which are running in the background are called Daemon Threads.

-->The main objective of Daemon Threads is to provide support for Non Daemon Threads(like main thread).

Eg: Garbage Collector

-->Whenever Main Thread runs with low memory, immediately PVM runs Garbage Collector to destroy useless objects and to provide free memory,so that Main Thread can continue it's execution without having any memory problems.

-->We can check whether thread is Daemon or not by using t.isDaemon() method of Thread class or by using daemon property.

Ex:

```
from threading import *
print(current_thread().isDaemon()) #False
print(current_thread().daemon) #False
```

-->We can change Daemon nature by using setDaemon() method of Thread class.

```
t.setDaemon(True)
```

-->But we can use this method before starting of Thread.i.e once thread started,we cannot change its Daemon nature,otherwise we will get

RuntimeException:cannot set daemon status of active thread

Ex:

```
from threading import *
print(current_thread().isDaemon())
current_thread().setDaemon(True)
```

RuntimeError: cannot set daemon status of active thread

Default Nature:

-->By default Main Thread is always non-daemon.But for the remaining threads Daemon nature will be inherited from parent to child.i.e if the Parent Thread is Daemon then child thread is also Daemon and if the Parent Thread is Non Daemon then ChildThread is also Non Daemon.

Ex:

```
from threading import *
def job():
        print("Child Thread")
t=Thread(target=job)
print(t.isDaemon())#False
t.setDaemon(True)
print(t.isDaemon()) #True
```

Note: Main Thread is always Non-Daemon and we cannot change its Daemon Nature b'z it is already started at the beginning only.

-->Whenever the last Non-Daemon Thread terminates automatically all Daemon Threads will be terminated.

Ex:

```
from threading import *
import time
def job():
        for i in range(10):
                print("Lazy Thread")
                time.sleep(2)
t=Thread(target=job)
t.setDaemon(True)===>Line-1
```

```
9)  t.start()
10) time.sleep(5)
11) print("End Of Main Thread")
```

-->In the above program if we comment Line-1 then both Main Thread and Child Threads are Non Daemon and hence both will be executed until their completion.

In this case output is:

```
-------------------------------
Lazy Thread
Lazy Thread
Lazy Thread
End Of Main Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
```

-->If we are not commenting Line-1 then Main Thread is Non-Daemon and Child Thread is Daemon.

-->Hence whenever MainThread terminates automatically child thread will be terminated.

In this case output is:

```
-------------------------------
Lazy Thread
Lazy Thread
Lazy Thread
End of Main Thread
```

-->MainThread is only Non-Daemon, but not all child threads. Daemon nature will iherited from the parent to child.
-->If parent is Non-Daemon child is also Non-Daemon.
-->If parent is Daemon child is also Daemon.

Ex:

```
1)  from threading import *
2)  import time
3)  def job1():
```

```
        print("Job1 execution.....")
        print(current_thread().name,"is daemon",current_thread().daemon)
        ct=Thread(target=job2,name="Child Thread2")
        print("ct is daemon:",ct.daemon)
def job2():
        print("Job2 execution.....")

t=Thread(target=job1,name="Child Thread")
t.setDaemon(True)
t.start()
time.sleep(10)
```

```
o/p:D:\pythonclasses>py test.py
Job1 execution.....
Child Thread is daemon True
ct is daemon: True
```

Synchronization:
================
-->If multiple threads are executing simultaneously then there may be a chance of data inconsistency problems.

Ex:

```
from threading import *
import time
def wish(name):
        for i in range(10):
        print("Good Evening:",end='')
        time.sleep(2)
        print(name)
t1=Thread(target=wish,args=("Mahesh",))
t2=Thread(target=wish,args=("Durga",))
t1.start()
t2.start()
```

Output:

```
Good Evening:Good Evening:Durga
Mahesh
Good Evening:Good Evening:Durga
Mahesh
....
```

-->We are getting irregular output b'z both threads are executing simultaneously wish() function.

-->To overcome this problem we should go for synchronization.

-->In synchronization the threads will be executed one by one so that we can overcome data inconsistency problems.

-->Synchronization means at a time only one Thread.

-->The main application areas of synchronization are

1. Online Reservation system
2. Funds Transfer from joint accounts

etc

-->In Python, we can implement synchronization by using the following

1. Lock
2. RLock
3. Semaphore

Synchronization By using Lock concept:

-->Locks are the most fundamental synchronization mechanism provided by threading module.

-->We can create Lock object as follows

```
l=Lock()
```

-->The Lock object can be hold by only one thread at a time.If any other thread required the same lock then it will wait until thread releases lock.(similar to common wash rooms,public telephone booth etc).

-->A Thread can acquire the lock by using acquire() method.

```
l.acquire()
```

-->A Thread can release the lock by using release() method.

```
l.release()
```

Note:

To call release() method compulsory thread should be owner of that lock.i.e thread should has the lock already,otherwise we will get Runtime Exception saying

RuntimeError: release unlocked lock

Ex:

```
1) from threading import *
2) l=Lock()
3) #l.acquire() ==>1
4) l.release()
```

-->If we are commenting line-1 then we will get
RuntimeError: release unlocked lock

```
from threading import *
import time
l=Lock()
def wish(name):
        l.acquire()
        for i in range(10):
        print("Good Morning:",end='')
        time.sleep(2)
        print(name)
l.release()
t1=Thread(target=wish,args=("Mahesh",))
t2=Thread(target=wish,args=("Durga",))
t3=Thread(target=wish,args=("Sunny",))
t1.start()
t2.start()
t3.start()
```

-->In the above program at a time only one thread is allowed to execute wish() method and hence we will get regular output.

Problem with Simple Lock:

-->The standard Lock object does not care which thread is currently holding that lock. If the lock is held and any thread attempts to acquire lock, then it will be blocked,even the same thread is already holding that lock.

Ex:

```
from threading import *
l=Lock()
print("Main Thread trying to acquire Lock")
l.acquire()
```

```
print("Main Thread trying to acquire Lock Again")
l.acquire()
```

Output:D:\pythonclasses>py test.py

```
Main Thread trying to acquire Lock
Main Thread trying to acquire Lock Again
--
```

-->In the above Program main thread will be blocked b'z it is trying to acquire the lock second time.

Note:

To kill the blocking thread from windows command prompt we have to use ctrl+break. Here ctrl+C won't work.

-->If the Thread calls recursive functions or nested access to resources,then the thread may trying to acquire the same lock again and again,which may block our thread.

-->Hence Traditional Locking mechanism won't work for executing recursive functions.

-->To overcome this problem, we should go for RLock(Reentrant Lock). Reentrant means the thread can acquire the same lock again and again.If the lock is held by other threads then only the thread will be blocked.

-->Reentrant facility is available only for owner thread but not for other threads.

Ex:

```
from threading import *
l=RLock()
print("Main Thread trying to acquire Lock")
l.acquire()
print("Main Thread trying to acquire Lock Again")
l.acquire()
```

-->In this case Main Thread won't be Locked b'z thread can acquire the lock any number of times.

-->This RLock keeps track of recursion level and hence for every acquire() call compulsory release() call should be available. i.e the number of acquire() calls and release() calls should be matched then only lock will be released.

Ex:

```
l=RLock()
l.acquire()
l.acquire()
l.release()
l.release()
```

-->After 2 release() calls only the Lock will be released.

Note:
----
1. Only owner thread can acquire the lock multiple times
2. The number of acquire() calls and release() calls should be matched.

Demo Program for synchronization by using RLock:

```
from threading import *
import time
l=RLock()
def factorial(n):
    l.acquire()
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    l.release()
    return result

def results(n):
    print("The Factorial of",n,"is:",factorial(n))

t1=Thread(target=results,args=(5,))
t2=Thread(target=results,args=(9,))
t1.start()
t2.start()
```

Output:
```
The Factorial of 5 is: 120
The Factorial of 9 is: 362880
```

-->In the above program instead of RLock if we use normal Lock then the thread will be blocked.

Difference between Lock and RLock:

Lock	RLock
1. Lock object can be acquired by only one thread at a time.Even owner thread also cannot acquire multiple times.	1. RLock object can be acquired by only one thread at a time, but owner thread can acquire same lock object multiple times.
2. Not suitable to execute recursive functions and nested access calls.	2. Best suitable to execute recursive functions and nested access calls.
3. In this case Lock object will takes care only Locked or unlocked and it never takes care about owner thread and recursion level.	3. In this case RLock object will takes care whether Locked or unlocked and owner thread information, recursiion level.

Synchronization by using Semaphore:

-->In the case of Lock and RLock, at a time only one thread is allowed to execute.

-->Sometimes our requirement is at a time a particular number of threads are allowed to access(like at a time 10 memebers are allowed to access database server,4 members are allowed to access Network connection etc).To handle this requirement we cannot use Lock and RLock concepts and we should go for Semaphore concept.

-->Semaphore can be used to limit the access to the shared resources with limited capacity.

-->Semaphore is advanced Synchronization Mechanism.

-->We can create Semaphore object as follows.

```
                    s=Semaphore(counter)
```

-->Here counter represents the maximum number of threads are allowed to access simultaneously.

-->The default value of counter is 1.

-->Whenever thread executes acquire() method,then the counter value will be decremented by 1 and if thread executes release() method then the counter value will be incremented by 1.

-->i.e for every acquire() call counter value will be decremented and for every release() call counter value will be incremented.

Case-1:

s=Semaphore()

In this case counter value is 1 and at a time only one thread is allowed to access. It is exactly same as Lock concept.

Case-2:

s=Semaphore(3)

In this case Semaphore object can be accessed by 3 threads at a time.The remaining threads have to wait until releasing the semaphore.

Ex:

```
from threading import *
import time
s=Semaphore(2)
def wish(name):
    s.acquire()
    for i in range(10):
        print("Good Morning!!!!!!!",end='')
        time.sleep(2)
        print(name)
    s.release()
t1=Thread(target=wish,args=("Mahesh",))
t2=Thread(target=wish,args=("Durga",))
t3=Thread(target=wish,args=("Sunny",))
t4=Thread(target=wish,args=("Bunny",))
t1.start()
t2.start()
t3.start()
t4.start()
```

-->In the above program at a time 2 threads are allowed to access semaphore and hence 2 threads are allowed to execute wish() function.

BoundedSemaphore:

-->Normal Semaphore is an unlimited semaphore which allows us to call release() method any number of times to increment counter.The number of release() calls can exceed the number of acquire() calls also.

Ex:

```
from threading import *
s=Semaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
s.release()
print("End")
```

-->It is valid because in normal semaphore we can call release() any number of times.

-->BounedSemaphore is exactly same as Semaphore except that the number of release() calls should not exceed the number of acquire() calls,otherwise we will get

ValueError: Semaphore released too many times

Ex:

```
from threading import *
s=BoundedSemaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
s.release()
print("End")
```

ValueError: Semaphore released too many times
It is invalid b'z the number of release() calls should not exceed the number of acquire() calls in BoundedSemaphore.

Note: To prevent simple programming mistakes, it is recommended to use BoundedSemaphore over normal Semaphore.

Difference between Lock and Semaphore:

-->At a time Lock object can be acquired by only one thread, but Semaphore object can be acquired by fixed number of threads specified by counter value.

Conclusion:

The main advantage of synchronization is we can overcome data inconsistency problems.But the main disadvantage of synchronization is it increases waiting time of threads and creates performance problems. Hence if there is no specific requirement then it is not recommended to use synchronization.

Inter Thread Communication:

-->Some times as the part of programming requirement, threads are required to communicate with each other. This concept is nothing but interthread communication.

Ex: After producing items Producer thread has to communicate with Consumer thread to notify about new item.Then consumer thread can consume that new item.
In Python, we can implement interthread communication by using the following ways

1. Event
2. Condition
3. Queue

etc

Interthread communication by using Event Objects:

-->Event object is the simplest communication mechanism between the threads. One thread signals an event and other thereds wait for it.

-->We can create Event object as follows...

```
event = threading.Event()
```

-->Event manages an internal flag that can set() or clear()

-->Threads can wait until event set.

Methods of Event class:

1.set():
internal flag value will become True and it represents GREEN signal for all waiting threads.

2. clear():
inernal flag value will become False and it represents RED signal for all waiting threads.

3. isSet():
This method can be used whether the event is set or not

4. wait()|wait(seconds):Thread can wait until event is set

Pseudo Code:

```
                              e=Event()

#consumer thread has to wait until event is set
e.wait()

#producer thread can set or clear event
e.set()
e.clear()
```

Ex:

```
from threading import *
import time
e=Event()
def consumer():
        print("Consumer Thread is waiting for updation")
        e.wait()
        print("Consumer thread got notification and conusming items")

def producer():
        time.sleep(5)
        print("Producer thread produce items")
        print("Producer thread giving notification by setting event")
        e.set()
t1=Thread(target=producer)
t2=Thread(target=consumer)
t1.start()
t2.start()
```

```
D:\pythonclasses>py test.py
Consumer Thread is waiting for updation
Producer thread produce items
Producer thread giving notification by setting event
Consumer thread got notification and conusming items
```

Ex-2:

```
from threading import *
import time
e=Event()
def trafficpolice():
    while True:
        time.sleep(10)
        print("Traffic Police Giving GREEN Signal")
        e.set()
        time.sleep(10)
        print("Traffic Police Giving RED Signal")
        e.clear()
def driver():
    num=0
    while True:
        print("Driver is waiting for GREEN Signal")
        e.wait()
        while e.isSet():
            num=num+1
            print("Vehicle No:",num,"crossing signal")
            time.sleep(2)
        print("Traffic Signal is RED.......Driver Have to wait")

t1=Thread(target=trafficpolice)
t2=Thread(target=driver)
t1.start()
t2.start()
```

-->In the above program driver thread has to wait until Trafficpolice thread sets event.ie until giving GREEN signal.Once Traffic police thread sets event(giving GREEN signal),vehicles can cross the signal.Once traffic police thread clears event (giving RED Signal)then the driver thread has to wait.

Interthread communication by using Condition Object:

-->Condition is the more advanced version of Event object for interthread communication. A condition represents some kind of state change in the application like producing item or consuming item. Threads can wait for that condition and threads can be notified once condition happend.i.e Condition object allows one or more threads to wait until notified by another thread.

-->Condition is always associated with a lock (ReentrantLock).

-->A condition has acquire() and release() methods that call the corresponding methods of the associated lock.

-->We can create Condition object as follows

```
condition = threading.Condition()
```

Methods of Condition:
----

1. acquire():
   To acquire Condition object before producing or consuming items.i.e thread acquiring internal lock.

2. release():
   To release Condition object after producing or consuming items. i.e thread releases internal lock.

3. wait()|wait(time):
   To wait until getting Notification or time expired

4. notify():
   To give notification for one waiting thread

5. notifyAll():
   To give notification for all waiting threads

Case Study:

The producing thread needs to acquire the Condition before producing item to the resource and notifying the consumers.

```
#Producer Thread
...generate item..
condition.acquire()
...add item to the resource...
condition.notify()#signal that a new item is available(notifyAll())
condition.release()
```

The Consumer must acquire the Condition and then it can consume items from the resource

```
#Consumer Thread
condition.acquire()
condition.wait()
consume item
condition.release()
```

Demo Program-1:

```
from threading import *
def consume(c):
        c.acquire()
        print("Consumer waiting for updation")
        c.wait()
        print("Consumer got notification & consuming the item")
        c.release()

def produce(c):
        c.acquire()
        print("Producer Producing Items")
        print("Producer giving Notification")
        c.notify()
        c.release()
        c=Condition()

t1=Thread(target=consume,args=(c,))
t2=Thread(target=produce,args=(c,))
t1.start()
t2.start()
```

Output:

```
Consumer waiting for updation
Producer Producing Items
Producer giving Notification
Consumer got notification & consuming the item
```

Demo Program-2:

```
from threading import *
import time
import random
items=[]
def produce(c):
        while True:
                c.acquire()
                item=random.randint(1,100)
                print("Producer Producing Item:",item)
                items.append(item)
```

```
            print("Producer giving Notification")
            c.notify()
            c.release()
            time.sleep(5)

def consume(c):
    while True:
            c.acquire()
            print("Consumer waiting for updation")
            c.wait()
            print("Consumer consumed the item",items.pop())
            c.release()
            time.sleep(5)

c=Condition()
t1=Thread(target=consume,args=(c,))
t2=Thread(target=produce,args=(c,))
t1.start()
t2.start()
```

Output:

```
----------
Consumer waiting for updation
Producer Producing Item: 49
Producer giving Notification
Consumer consumed the item 49
.....
```

-->In the above program Consumer thread expecting updation and hence it is responsible to call wait() method on Condition object.

-->Producer thread performing updation and hence it is responsible to call notify() or notifyAll() on Condition object.

Interthread communication by using Queue:

-->Queues Concept is the most enhanced Mechanism for interthread communication and to share data between threads.

-->Queue internally has Condition and that Condition has Lock.Hence whenever we are using Queue we are not required to worry about Synchronization.

-->If we want to use Queues first we should import queue module.

```
                                        import queue
```

We can create Queue object as follows

```
q = queue.Queue()
```

Important Methods of Queue:
----
1. put(): Put an item into the queue.
2. get(): Remove and return an item from the queue.

-->Producer Thread uses put() method to insert data in the queue. Internally this method has logic to acquire the lock before inserting data into queue. After inserting data lock will be released automatically.

-->put() method also checks whether the queue is full or not and if queue is full then the Producer thread will entered in to waiting state by calling wait() method internally.

-->Consumer Thread uses get() method to remove and get data from the queue. Internally this method has logic to acquire the lock before removing data from the queue. Once removal completed then the lock will be released automatically.

-->If the queue is empty then consumer thread will entered into waiting state by calling wait() method internally.Once queue updated with data then the thread will be notified automatically.

Note:

The queue module takes care of locking for us which is a great advantage.

Ex:

```
from threading import *
import time
import random
import queue
def produce(q):
while True:
        item=random.randint(1,100)
        print("Producer Producing Item:",item)
        q.put(item)
        print("Producer giving Notification")
        time.sleep(5)
def consume(q):
        while True:
                print("Consumer waiting for updation")
```

```
            print("Consumer consumed the item:",q.get())
            time.sleep(5)

q=queue.Queue()
t1=Thread(target=consume,args=(q,))
t2=Thread(target=produce,args=(q,))
t1.start()
t2.start()
```

Output:

```
Consumer waiting for updation
Producer Producing Item: 58
Producer giving Notification
Consumer consumed the item: 58
```

Types of Queues:

-->Python Supports 3 Types of Queues.

1).FIFO Queue:

```
q = queue.Queue()
```
This is Default Behaviour. In which order we put items in the queue, in the same order the items will come out (FIFO-First In First Out).

Ex:

```
import queue
q=queue.Queue()
q.put(10)
q.put(5)
q.put(20)
q.put(15)
while not q.empty():
print(q.get(),end=' ')
```

Output: 10 5 20 15

2. LIFO Queue:

-->The removal will be happend in the reverse order of insertion(Last In First Out)

Ex:

```
import queue
q=queue.LifoQueue()
q.put(10)
q.put(5)
q.put(20)
q.put(15)
while not q.empty():
        print(q.get(),end=' ')
```

Output: 15 20 5 10

3).Priority Queue:

-->The elements will be inserted based on some priority order.

```
import queue
q=queue.PriorityQueue()
q.put(10)
q.put(5)
q.put(20)
q.put(15)
while not q.empty():
        print(q.get(),end=' ')
```

Output: 5 10 15 20

Ex-2: If the data is non-numeric, then we have to provide our data in the form of tuple.

(x,y)

x is priority
y is our element

```
import queue
q=queue.PriorityQueue()
q.put((1,"AAA"))
q.put((3,"CCC"))
q.put((2,"BBB"))
q.put((4,"DDD"))
while not q.empty():
        print(q.get()[1],end=' ')
```

Output: AAA BBB CCC DDD

Good Programming Practices with usage of Locks:

--

Case-1:

-->It is highly recommended to write code of releasing locks inside finally block.The advantage is lock will be released always whether exception raised or not raised and whether handled or not handled.

```
l=threading.Lock()
l.acquire()

try:
        perform required safe operations
finally:
        l.release()
```

Demo Program:

```
from threading import *
import time
l=Lock()
def wish(name):
        l.acquire()
        try:
                for i in range(10):
                        print("Good Morning:",end='')
                        time.sleep(2)
                        print(name)
        finally:
                l.release()

t1=Thread(target=wish,args=("Mahesh",))
t2=Thread(target=wish,args=("Durga",))
t3=Thread(target=wish,args=("Sunny",))
t1.start()
t2.start()
t3.start()
```

Case-2:

-->It is highly recommended to acquire lock by using with statement. The main advantage of with statement is the lock will be released automatically once control reaches end of with block and we are not required to release explicitly.

-->This is exactly same as usage of with statement for files.

Example for File:

```
with open('demo.txt','w') as f:
        f.write("Hello...")
```

Example for Lock:

```
lock=threading.Lock()
with lock:
        perform required safe operations
        lock will be released automatically
```

Demo Program:

```
from threading import *
import time
lock=Lock()
def wish(name):
        with lock:
                for i in range(10):
                        print("Good Morning:",end='')
                        time.sleep(2)
                        print(name)

t1=Thread(target=wish,args=("Mahesh",))
t2=Thread(target=wish,args=("Durga",))
t3=Thread(target=wish,args=("Sunny",))
t1.start()
t2.start()
t3.start()
```

Q. What is the advantage of using with statement to acquire a lock in threading?

-->Lock will be released automatically once control reaches end of with block and We are not required to release explicitly.

Note:
We can use with statement in multithreading for the following cases:

1. Lock
2. RLock
3. Semaphore
4. Condition

Regular Expressions

-->If we want to represent a group of Strings according to a particular format/pattern then we should go for Regular Expressions.

-->i.e Regualr Expressions is a declarative mechanism to represent a group of Strings accroding to particular format/pattern.

Ex 1: We can write a regular expression to represent all mobile numbers
Ex 2: We can write a regular expression to represent all mail ids.

The main important application areas of Regular Expressions are

1. To develop validation frameworks/validation logic
2. To develop Pattern matching applications (ctrl-f in windows, grep in UNIX etc)
3. To develop Translators like compilers, interpreters etc
4. To develop digital circuits
5. To develop communication protocols like TCP/IP, UDP etc.

We can develop Regular Expression Based applications by using python module: re
This module contains several inbuilt functions to use Regular Expressions very easily in our applications.

1. compile():
----
re module contains compile() function to compile a pattern into RegexObject.

pattern = re.compile("ab")

2. finditer():

Returns an Iterator object which yields Match object for every Match

matcher = pattern.finditer("abaababa")

On Match object we can call the following methods.
1. start():Returns start index of the match
2. end():Returns end+1 index of the match
3. group():Returns the matched string

Ex:

```
import re count=0
pattern=re.compile("ab")
matcher=pattern.finditer("abaababa")
for match in matcher:
```

```
    count+=1
    print(match.start(),"...",match.end(),"...",match.group())
print("The number of occurrences: ",count)
```

Output:

```
0 ... 2 ... ab
3 ... 5 ... ab
5 ... 7 ... ab
The number of occurrences: 3
```

Note: We can pass pattern directly as argument to finditer() function.

Ex:

```
import re
count=0
matcher=re.finditer("ab","abaababa")
for match in matcher:
    count+=1
    print(match.start(),"...",match.end(),"...",match.group())
print("The number of occurrences: ",count)
```

Output:

```
0 ... 2 ... ab
3 ... 5 ... ab
5 ... 7 ... ab
The number of occurrences: 3
```

Character classes:

We can use character classes to search a group of characters
1. [abc]===>Either a or b or c
2. [^abc] ===>Except a and b and c
3. [a-z]==>Any Lower case alphabet symbol
4. [A-Z]===>Any upper case alphabet symbol
5. [a-zA-Z]==>Any alphabet symbol
6. [0-9] Any digit from 0 to 9
7. [a-zA-Z0-9]==>Any alphanumeric character
8. [^a-zA-Z0-9]==>Except alphanumeric characters(Special Characters)

Ex:

```
import re
matcher=re.finditer("x","a7b@k9z")
for match in matcher:
```

```
4)        print(match.start(),"......",match.group())
```

```
x = [abc]
-----------
0 ...... a
2 ...... b

x = [^abc]
------------
1 ...... 7
3 ...... @
4 ...... k
5 ...... 9
6 ...... z

x = [a-z]
----------
0 ...... a
2 ...... b
4 ...... k
6 ...... z

x = [0-9]
----------
1 ...... 7
5 ...... 9

x = [a-zA-Z0-9]
-------------------
0 ...... a
1 ...... 7
2 ...... b
4 ...... k
5 ...... 9
6 ...... z

x = [^a-zA-Z0-9]
--------------------
3 ...... @
```

Pre-defined Character classes:

\s==>Space character
\S==>Any character except space character
\d==>Any digit from 0 to 9
\D==>Any character except digit
\w==>Any word character [a-zA-Z0-9]
\W==>Any character except word character (Special Characters)

.==>Any character including special characters

Ex:

```
import re
matcher=re.finditer("x","a7b k@9z")
for match in matcher:
        print(match.start(),"......",match.group())
```

x = \s:

```
3 ......
```

x = \S:

```
0 ...... a
1 ...... 7
2 ...... b
4 ...... k
5 ...... @
6 ...... 9
7 ...... z
```

x = \d:

```
1 ...... 7
6 ...... 9
```

x = \D:

```
0 ...... a
2 ...... b
3 ......
4 ...... k
5 ...... @
7 ...... z
```

x = \w:

```
0 ...... a
1 ...... 7
2 ...... b
4 ...... k
6 ...... 9
7 ...... z
```

x = \W:

```
3 ......
5 ...... @
```
x = .
```
0 ...... a
1 ...... 7
2 ...... b
3 ......
4 ...... k
5 ...... @
6 ...... 9
7 ...... z
```

Qunatifiers:

We can use quantifiers to specify the number of occurrences to match.

a==>Exactly one 'a'
a+==>Atleast one 'a'
a*==>Any number of a's including zero number
a?==>Atmost one 'a' ie either zero number or one number
a{m}==>Exactly m number of a's
a{m,n}==>Minimum m number of a's and Maximum n number of a's

Ex:

```
import re
matcher=re.finditer("x","abaabaaab")
for match in matcher:
        print(match.start(),"......",match.group())
```

x = a:

```
0 ...... a
2 ...... a
3 ...... a
5 ...... a
6 ...... a
7 ...... a
```

x = a+:

```
0 ...... a
2 ...... aa
5 ...... aaa
```

x = a*:
----
0 ...... a
1 ......
2 ...... aa
4 ......
5 ...... aaa
8 ......
9 ......

x = a?:
----
0 ...... a
1 ......
2 ...... a
3 ...... a
4 ......
5 ...... a
6 ...... a
7 ...... a
8 ......
9 ......

x = a{3}:
----
5 ...... aaa
x = a{2,4}:
2 ...... aa
5 ...... aaa

Note:
----
^x==>It will check whether target string starts with x or not
x$==>It will check whether target string ends with x or not

Important functions of re module:
----
1. match()
2. fullmatch()
3. search()
4.findall()
5.finditer()
6. sub()
7.subn()
8. split()
9. compile()

1. match():

We can use match function to check the given pattern at beginning of target string.
If the match is available then we will get Match object, otherwise we will get None.

Ex:

```
import re
s=input("Enter pattern to check: ")
m=re.match(s,"abcabdefg")
if m!= None:
        print("Match is available at the beginning of the String")
        print("Start Index:",m.start(), "and End Index:",m.end())
else:
        print("Match is not available at the beginning of the String")
```

Output:

```
D:\pythonclasses>py test.py
Enter pattern to check: abc
Match is available at the beginning of the String
Start Index: 0 and End Index: 3

D:\pythonclasses>py test.py
Enter pattern to check: bde
Match is not available at the beginning of the String
```

2. fullmatch():

-->We can use fullmatch() function to match a pattern to all of target string. i.e complete string should be matched according to given pattern.
-->If complete string matched then this function returns Match object otherwise it returns None.

Ex:

```
import re
s=input("Enter pattern to check: ")
m=re.fullmatch(s,"ababab")
if m!= None:
        print("Full String Matched")
else:
        print("Full String not Matched")
```

Output:

```
D:\pythonclasses>py test.py
Enter pattern to check: ab
Full String not Matched

D:\pythonclasses>py test.py
Enter pattern to check: ababab
Full String Matched
```

3. search():

-->We can use search() function to search the given pattern in the target string.
If the match is available then it returns the Match object which represents first occurrence of the match.
-->If the match is not available then it returns None.

Ex:

```
import re
s=input("Enter pattern to check: ")
m=re.search(s,"abaaaba")
if m!= None:
        print("Match is available")
        print("First Occurrence of match with start index:",m.start(),"and end index:",m.end())
else:
        print("Match is not available")
```

Output:

```
D:\pythonclasses>py test.py
Enter pattern to check: aaa
Match is available
First Occurrence of match with start index: 2 and end index: 5

D:\pythonclasses>py test.py
Enter pattern to check: bbb
Match is not available
```

4. findall():

To find all occurrences of the match.
This function returns a list object which contains all occurrences.

Ex:

```
import re
l=re.findall("[0-9]","a7b9c5kz")
print(l)
```

Output: ['7', '9', '5']

5. finditer():

Returns the iterator yielding a match object for each match.
On each match object we can call start(), end() and group() functions.

Ex:

```
import re
itr=re.finditer("[a-z]","a7b9c5k8z")
for m in itr:
        print(m.start(),"...",m.end(),"...",m.group())
```

Output:

```
D:\pythonclasses>py test.py
0 ... 1 ... a
2 ... 3 ... b
4 ... 5 ... c
6 ... 7 ... k
8 ... 9 ... z
```

6. sub():
----
sub means substitution or replacement
re.sub(regex,replacement,targetstring)
In the target string every matched pattern will be replaced with provided replacement.

Ex:

```
import re
s=re.sub("[a-z]","#","a7b9c5k8z")
print(s)
```

Output: #7#9#5#8#

Every alphabet symbol is replaced with # symbol

7. subn():

It is exactly same as sub except it can also returns the number of replacements.
This function returns a tuple where first element is result string and second element is number of replacements.

(resultstring, number of replacements)

Ex:

```
import re
t=re.subn("[a-z]","#","a7b9c5k8z")
print(t)
print("The Result String:",t[0])
print("The number of replacements:",t[1])
```

Output:

```
D:\pythonclasses>py test.py
('#7#9#5#8#', 5)
The Result String: #7#9#5#8#
The number of replacements: 5
```

8. split():

-->If we want to split the given target string according to a particular pattern then we should go for split() function.
-->This function returns list of all tokens.

Ex:

```
import re
l=re.split(",","sunny,bunny,chinny,vinny,pinny")
print(l)
for t in l:
	print(t)
```

Output:

```
D:\pythonclasses>py test.py
['sunny', 'bunny', 'chinny', 'vinny', 'pinny']
sunny
bunny
chinny
vinny
pinny
```

Ex:

```
import re
l=re.split("\.","www.durgasoft.com")
for t in l:
        print(t)
```

Output:
```
D:\pythonclasses>py test.py
www
durgasoft
com
```

^symbol:

We can use ^ symbol to check whether the given target string starts with our provided pattern or not.

Ex:
res=re.search("^Learn",s)
if the target string starts with Learn then it will return Match object,otherwise returns None.

test.py:

```
import re
s="Learning Python is Very Easy"
res=re.search("^Learn",s)
if res != None:
        print("Target String starts with Learn")
else:
        print("Target String Not starts with Learn")
```

Output: Target String starts with Learn

$ symbol:

We can use $ symbol to check whether the given target string ends with our provided pattern or not.

Ex: res=re.search("Easy$",s)

If the target string ends with Easy then it will return Match object,otherwise returns None.

test.py:

```
import re
s="Learning Python is Very Easy"
res=re.search("Easy$",s)
if res != None:
        print("Target String ends with Easy")
else:
        print("Target String Not ends with Easy")
```

Output: Target String ends with Easy

Note: If we want to ignore case then we have to pass 3rd argument re.IGNORECASE for search() function.

Ex: res = re.search("easy$",s,re.IGNORECASE)

test.py:

```
import re
s="Learning Python is Very Easy"
res=re.search("easy$",s,re.IGNORECASE)
if res != None:
        print("Target String ends with Easy by ignoring case")
else:
        print("Target String Not ends with Easy by ignoring case")
```

Output: Target String ends with Easy by ignoring case

App1: Write a Regular Expression to represent all Yava language identifiers
----

Rules:
----
1. The allowed characters are a-z,A-Z,0-9,#
2. The first character should be a lower case alphabet symbol from a to k
3. The second character should be a digit divisible by 3
4. The length of identifier should be atleast 2.

[a-k][0369][a-zA-Z0-9#]*

App2: Write a python program to check whether the given string is Yava language identifier or not?

```
import re
s=input("Enter Identifier:")
m=re.fullmatch("[a-k][0369][a-zA-Z0-9#]*",s)
if m!= None:
        print(s,"is valid Yava Identifier")
else:
        print(s,"is invalid Yava Identifier")
```

Output:

```
D:\pythonclasses>py test.py
Enter Identifier:a6kk9z##
a6kk9z## is valid Yava Identifier

D:\pythonclasses>py test.py
Enter Identifier:k9b876
k9b876 is valid Yava Identifier

D:\pythonclasses>py test.py
Enter Identifier:k7b9
k7b9 is invalid Yava Identifier
```

App3: Write a Regular Expression to represent all 10 digit mobile numbers.

Rules:

----

1. Every number should contains exactly 10 digits
2. The first digit should be 7 or 8 or 9

[7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]

or

[7-9][0-9]{9}

or

[7-9]\d{9}

App4: Write a Python Program to check whether the given number is valid mobile number or not?

```
import re
n=input("Enter number:")
m=re.fullmatch("[7-9]\d{9}",n)
if m!= None:
        print("Valid Mobile Number")
else:
```

```
    print("Invalid Mobile Number")
```

Output:
```
D:\pythonclasses>py test.py
Enter number:9898989898
Valid Mobile Number

D:\pythonclasses>py test.py
Enter number:6786786787
Invalid Mobile Number

D:\pythonclasses>py test.py
Enter number:898989
Invalid Mobile Number
```

App5: Write a python program to extract all mobile numbers present in input.txt where numbers are mixed with normal text data
---

```
import re
f1=open("input.txt","r")
f2=open("output.txt","w")
for line in f1:
    list=re.findall("[7-9]\d{9}",line)
    for n in list:
        f2.write(n+"\n")
print("Extracted all Mobile Numbers into output.txt")
f1.close()
f2.close()
```

Web Scraping by using Regular Expressions:
--
The process of collecting information from web pages is called web scraping. In web scraping to match our required patterns like mail ids, mobile numbers we can use regular expressions.

Ex:

```
import re,urllib
import urllib.request
sites="google rediff".split()
print(sites)
for s in sites:
    print("Searching...",s)
    u=urllib.request.urlopen("http://"+s+".com")
    text=u.read()
```

```
        title=re.findall("<title>.*</title>",str(text),re.I)
        print(title[0])
```

Ex: Program to get all phone numbers of redbus.in by using web scraping and regular expressions
----

```
import re,urllib
import urllib.request
u=urllib.request.urlopen("https://www.redbus.in/info/contactus")
text=u.read()
numbers=re.findall("[0-9-]{7}[0-9-]+",str(text),re.I)
for n in numbers:
        print(n)
```

Q. Write a Python Program to check whether the given mail id is valid gmail id or not?

```
import re
s=input("Enter Mail id:")
m=re.fullmatch("\w[a-zA-Z0-9_.]*@gmail[.]com",s)
if m!=None:
        print("Valid Mail Id");
else:
        print("Invalid Mail id")
```

Output:

```
D:\pythonclasses>py test.py
Enter Mail id:pythonbymahesh@gmail.com
Valid Mail Id

D:\pythonclasses>py test.py
Enter Mail id:pythonbymahesh
Invalid Mail id

D:\pythonclasses>py test.py
Enter Mail id:pythonbymahesh@yahoo.co.in
Invalid Mail id
```

Q. Write a python program to check whether given car registration number is valid Telangana State Registration number or not?

```
import re
s=input("Enter Vehicle Registration Number:")
m=re.fullmatch("TS[012][0-9][A-Z]{2}\d{4}",s)
```

```
if m!=None:
        print("Valid Vehicle Registration Number");
else:
        print("Invalid Vehicle Registration Number")
```

Output:

```
D:\pythonclasses>py test.py
Enter Vehicle Registration Number:TS07EA7777
Valid Vehicle Registration Number

D:\pythonclasses>py test.py
Enter Vehicle Registration Number:TS07KF0786
Valid Vehicle Registration Number

D:\pythonclasses>py test.py
Enter Vehicle Registration Number:AP07EA7898
Invalid Vehicle Registration Number
```

Q. Python Program to check whether the given mobile number is valid OR not (10 digit OR 11 digit OR 12 digit)

```
import re
s=input("Enter Mobile Number:")
m=re.fullmatch("(0|91)?[7-9][0-9]{9}",s)
if m!=None:
        print("Valid Mobile Number");
else:
        print("Invalid Mobile Number")
```

PYTHON DATABASE PROGRAMMING

Storage Areas:

-->As the Part of our Applications, we required to store our Data like Customers Information, Billing Information, Calls Information etc..

-->To store this Data, we required Storage Areas. There are 2 types of Storage Areas.
1) Temporary Storage Areas
2) Permanent Storage Areas

1.Temporary Storage Areas:

-->These are the Memory Areas where Data will be stored temporarily.
Ex: Python objects like List, Tuple, Dictionary.

-->Once Python program completes its execution then these objects will be destroyed automatically and data will be lost.

2. Permanent Storage Areas:

-->Also known as Persistent Storage Areas. Here we can store Data permanently.
Ex: File Systems, Databases, Data warehouses, Big Data Technologies etc

File Systems:

-->File Systems can be provided by Local operating System. File Systems are best suitable to store very less Amount of Information.

Limitations:
----
1) We cannot store huge Amount of Information.
2) There is no Query Language support and hence operations will become very complex.
3) There is no Security for Data.
4) There is no Mechanism to prevent duplicate Data. Hence there may be a chance of Data Inconsistency Problems.

-->To overcome the above Problems of File Systems, we should go for Databases.

Databases:
----
1) We can store Huge Amount of Information in the Databases.
2) Query Language Support is available for every Database and hence we can perform Database Operations very easily.
3) To access Data present in the Database, compulsory username and pwd must be required. Hence Data is secured.
4) Inside Database Data will be stored in the form of Tables. While developing Database Table Schemas, Database Admin follow various Normalization Techniques and can implement various Constraints like Unique Key Constrains, Primary Key Constraints etc which prevent Data Duplication. Hence there is no chance of Data Inconsistency Problems.

Limitations of Databases:
----
1) Database cannot hold very Huge Amount of Information like Terabytes of Data.
2) Database can provide support only for Structured Data (Tabular Data OR Relational Data) and cannot provide support for Semi Structured Data (like XML Files) and Unstructured Data (like Video Files, Audio Files, Images etc)

-->To overcome these Problems we should go for more Advanced Storage Areas like Big Data Technologies, Data warehouses etc.

Python Database Programming:

-->Sometimes as the part of Programming requirement we have to connect to the database and we have to perform several operations like creating tables, inserting data,updating data,deleting data,selecting data etc.

-->We can use SQL Language to talk to the database and we can use Python to send those SQL commands to the database.
-->Python provides inbuilt support for several databases like Oracle, MySql, SqlServer, GadFly, sqlite, etc.

-->Python has seperate module for each database.

Ex: cx_Oralce module for communicating with Oracle database
pymssql module for communicating with Microsoft Sql Server
pymysql module for communicating with MySql

Standard Steps for Python database Programming:
----

1. Import database specific module
Ex: import cx_Oracle

2. Establish Connection between Python Program and database.
We can create this Connection object by using connect() function of the module.
con = cx_Oracle.connect(datbase information)
Ex: con=cx_Oracle.connect('scott/tiger@localhost')

3. To execute our sql queries and to hold results some special object is required, which is nothing but Cursor object. We can create Cursor object by using cursor() method.
cursor=con.cursor()

4. Execute SQL Queries By using Cursor object. For this we can use the following methods
i) execute(sqlquery):
To execute a single sql query

ii) executescript(sqlqueries):
To execute a string of sql queries seperated by semi-colon ';'

iii) executemany():
To execute a Parameterized query
Ex: cursor.execute("select * from employees")

5. commit or rollback changes based on our requirement in the case of DML Queries(insert|update|delete)
commit():Saves the changes to the database
rollback():rolls all temporary changes back

6. Fetch the result from the Cursor object in the case of select queries
fetchone():To fetch only one row
fetchall():To fetch all rows and it returns a list of rows
fecthmany(n):To fetch first n rows

Ex 1: data =cursor.fetchone()
print(data)

Ex 2: data=cursor.fetchall()
for row in data:
print(row)

7. close the resources:
After completing our operations it is highly recommended to close the resources in the reverse order of their opening by using close() methods.

```
cursor.close()
con.close()
```

Note: The following is the list of all important methods which can be used for python database programming.

```
connect()
cursor()
execute()
executescript()
executemany()
commit()
rollback()
fetchone()
fetchall()
fetchmany(n)
fetch
close()
```

These methods won't be changed from database to database and same for all databases.

Working with Oracle Database:

From Python Program if we want to communicate with any database,some translator must be required to translate Python calls into Database specific calls and Database specific calls into Python calls.This translator is nothing but Driver/Connector.

Diagram

For Oracle database the name of driver needed is cx_Oracle.
cx_Oracle is a Python extension module that enables access to Oracle Database.It can be used for both Python2 and Python3. It can work with any version of Oracle database like 9,10,11 and 12.

Installing cx_Oracle:

From Normal Command Prompt (But not from Python console)execute the following command

```
D:\pythonclasses>pip install cx_Oracle
Collecting cx_Oracle
Installing collected packages: cx-Oracle
Successfully installed cx-Oracle-7.0.0
```

How to Test Installation:

From python console execute the following command:

```
>>> help("modules")
```

In the output we can see cx_Oracle

```
....
_multiprocessing          crypt        ntpath       timeit
_opcode                   csv          nturl2path   tkinter
_operator                 csvr         numbers      token
_osx_support              csvw         opcode       tokenize
_overlapped               ctypes       operator     trace
_pickle                   curses       optparse     traceback
_pydecimal                custexcept   os           tracemalloc
_pyio                     cx_Oracle    parser       try
_random                   data         pathlib      tty
_sha1                     datetime     pdb          turtle
```

Note: grant all privileges to scott;

App1: Program to connect with Oracle database and print its version.

```
import cx_Oracle
con=cx_Oracle.connect('scott/tiger@localhost')
if con!=None:
	print("Connection established successfully....")
	print("Version",con.version)
else:
	print("Connection not established")
```

```
o/p:D:\pythonclasses>py test.py
Connection established successfully....
Version 11.2.0.2.0
```

App2: Write a Program to create employees table in the oracle database :

```
employees(eno,ename,esal,eaddr)
import cx_Oracle
try:
	query="create table employees(eno number,ename varchar2(10),esal number(10,2),eaddr varchar2(10))"
	con=cx_Oracle.connect('scott/tiger@localhost')
	cursor=con.cursor()
```

```
        cursor.execute(query)
        print("Table is craeted successfully.........")
except cx_Oracle.DatabaseError as e:
        if con:
                con.rollback()
                print("There is a problem in sql",e)
finally:
        if cursor:
                cursor.close()
        if con:
                con.close()
```

App3: Write a program to drop employees table from oracle database?

```
import cx_Oracle
try:
        query="drop table employees"
        con=cx_Oracle.connect('scott/tiger@localhost')
        cursor=con.cursor()
        cursor.execute(query)
        print("Table is dropped successfully.........")
except cx_Oracle.DatabaseError as e:
        if con:
                con.rollback()
                print("There is a problem in sql",e)
finally:
        if cursor:
                cursor.close()
        if con:
                con.close()
```

App4: Write a program to insert a single row in the employees table.

```
import cx_Oracle
try:
        query="insert into employees values(100, 'Mahesh',1000,'Hyd')"
        con=cx_Oracle.connect('scott/tiger@localhost')
        cursor=con.cursor()
        cursor.execute(query)
        con.commit()
        print("Record inserted successfully.........")
except cx_Oracle.DatabaseError as e:
```

```
	if con:
		con.rollback()
		print("There is a problem in sql",e)
finally:
	if cursor:
		cursor.close()
	if con:
		con.close()
```

Note: While performing DML Operations (insert|update|delte), compulsory we have to use commit() method,then only the results will be reflected in the database.

App5: Write a program to insert multiple rows in the employees table by using executemany() method.

```
import cx_Oracle
try:
	query="insert into employees values(:eno,:ename,:esal,:eaddr)"
	records=[(200,'Durga',2000,'Hyd'),(300,'Sunny',3000,'Mumbai'),(400,'Bunny',4000,'Hyd')]
	con=cx_Oracle.connect('scott/tiger@localhost')
	cursor=con.cursor()
	cursor.executemany(query,records)
	con.commit()
	print("Records are inserted successfully.........")
except cx_Oracle.DatabaseError as e:
	if con:
		con.rollback()
		print("There is a problem in sql",e)
finally:
	if cursor:
		cursor.close()
	if con:
		con.close()
```

App6: Write a program to insert multiple rows in the employees table with dynamic input from the keyboard?

```
import cx_Oracle
try:
	con=cx_Oracle.connect('scott/tiger@localhost')
	cursor=con.cursor()
```

```
    while True:
        eno=int(input("Enter Employee Number:"))
        ename=input("Enter Employee Name:")
        esal=float(input("Enter Employee Salary:"))
        eaddr=input("Enter Employee Address:")
        sql="insert into employees values(%d,'%s',%f,'%s')"
        cursor.execute(sql%(eno,ename,esal,eaddr))
        print("Record inserted successfully.......")
        option=input("Do you want to insert one more record[Yes|No]:")
        if option=="No":
            con.commit()
            break
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem in sql",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

App7: Write a program to update employee salaries with increment for the certain range with dynamic input.

Ex: Increment all employee salaries by 500 whose salary < 5000

```
import cx_Oracle
try:
    con=cx_Oracle.connect('scott/tiger@localhost')
    cursor=con.cursor()
    increment=float(input("Enter Increment salary:"))
    salrange=float(input("Enter Salary Range:"))
    sql="update employees set esal=esal+%f where esal<%f"
    cursor.execute(sql%(increment,salrange))
    print("Records are updated successfully....")
    con.commit()
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem in sql",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

App8: Write a program to delete employees whose salary greater provided salary as dynamic input?

Ex: delete all employees whose salary > 4500

```
import cx_Oracle
try:
    con=cx_Oracle.connect('scott/tiger@localhost')
    cursor=con.cursor()
    cutoffsal=float(input("Enter Cutoff Salary:"))
    sql="delete from employees where esal>%f"
    cursor.execute(sql%(cutoffsal))
    print("Records are deleted successfully....")
    con.commit()
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem in sql",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

App9: Write a program to select all employees info by using fetchone() method?

```
import cx_Oracle
try:
    con=cx_Oracle.connect('scott/tiger@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    row=cursor.fetchone()
    while row is not None:
        print(row)
        row=cursor.fetchone()
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql :",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

App10: Write a program to select all employees info by using fetchall() method?

```
import cx_Oracle
try:
    con=cx_Oracle.connect('scott/tiger@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    data=cursor.fetchall()
    for row in data:
        print("Employee Number:",row[0])
        print("Employee Name:",row[1])
        print("Employee Salary:",row[2])
        print("Employee Address:",row[3])
        print()
        print()
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql :",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

App11: Write a program to select employees info by using fetchmany() method and the required number of rows will be provided as dynamic input?

```
import cx_Oracle
try:
    con=cx_Oracle.connect('scott/tiger@localhost')
    cursor=con.cursor()
    cursor.execute("select * from employees")
    n=int(input("Enter the number of required rows:"))
    data=cursor.fetchmany(n)
    for row in data:
    print(row)
```

```
except cx_Oracle.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql :",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

Working with Mysql database:
===========================
Current version: 5.7.19
Vendor: SUN Micro Systems/Oracle Corporation
Open Source and Freeware
Default Port: 3306
Default user: root

Note: In MySQL, everything we have to work with our own databases, which are also known as logical Databases.

The following are 4 default databases available in mysql.
1. information_schema
2. mysql
3. performance_schema
4. test

Diagram

In the above diagram only one physical database is available and 4 logical databases are available.

Commonly used commands in MySql:
--
1. To know available databases:
```
mysql> show databases;
```

2. To create our own logical database
```
mysql> create database maheshdb;
```

3. To drop our own database:
```
mysql> drop database maheshdb;
```

4. To use a particular logical database

```
mysql> use maheshdb; OR mysql> connect maheshdb;
```

5. To create a table:

```
create table employees(eno int(5) primary key,ename varchar(10),esal double(10,2),eaddr varchar(10));
```

6. To insert data:

```
insert into employees values(100,'Mahesh',1000,'Hyd');
insert into employees values(200,'Durga',2000,'Mumbai');
```

-->In MySQL instead of single quotes we can use double quotes also.

Driver/Connector Information:

From Python program if we want to communicates with MySql database,compulsory some translator is required to convert python specific calls into mysql database specific calls and mysql database specific calls into python specific calls. This translator is nothing but Driver or Connector.

Diagram

-->We have to download connector seperately from mysql database.

-->https://dev.mysql.com/downloads/connector/python/2.1.html

How to check installation:

From python console we have to use
help("modules")

In the list of modules,compulsory mysql should be there.

Note: In the case of Python3.4 we have to set PATH and PYTHONPATH explicitly

```
PATH=C:\Python34
PYTHONPATH=C:\Python34\Lib\site-packages
```

Q:write a program to create Table, insert data and display data by using MySQL Database.

```
import pymysql
try:
        con=pymysql.connect(host='localhost',database='maheshdb',user='root',password='root')
```

```
    cursor=con.cursor()
    cursor.execute("create table employees(eno int(5) primary key, ename varchar(10), esal double(10,2),eaddr varchar(10))")
    print("Table is created.......")
    query="insert into employees (eno,ename,esal,eaddr) values (%s,%s,%s,%s)"
    records=[(111,'Katrina',1000,'mumbai'),(222,'Kareena',2000,"mumbai"),(333,'Deepika',3000,'Mumbai')]
    cursor.executemany(query,records)
    con.commit()
    print("Records are inserted successfully")
    cursor.execute("select * from employees")
    data=cursor.fetchall()
    print(type(data))
    for row in data:
        print("Employee Number:",row[0])
        print("Employee Name:",row[1])
        print("Employee Salary:",row[2])
        print("Employee Address:",row[3])
        print()
except pymysql.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem with sql :",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```

```
o/p:D:\pythonclasses>py test.py
Table is created.......
Records are inserted successfully
<class 'tuple'>
Employee Number: 111
Employee Name: Katrina
Employee Salary: 1000.0
Employee Address: mumbai

Employee Number: 222
Employee Name: Kareena
Employee Salary: 2000.0
Employee Address: mumbai
```

```
Employee Number: 333
Employee Name: Deepika
Employee Salary: 3000.0
Employee Address: Mumbai
```

Q).write a program to copy data present in employees table of MySQL data base into Oracle Database.

```
import cx_Oracle
import pymysql
try:
	con=pymysql.connect(host='localhost',database='maheshdb',user='root',password='root')
	cursor=con.cursor()
	cursor.execute("select * from employees")
	data=cursor.fetchall()
	print(type(data))
	list=list(data)
	print(type(list))
	print(list)
except pymysql.DatabaseError as e:
	if con:
		con.rollback()
		print("There is a problem with sql :",e)
finally:
	if cursor:
		cursor.close()
	if con:
		con.close()
try:
	con=cx_Oracle.connect('scott/tiger@localhost')
	cursor=con.cursor()
	query="insert into employees values(:eno,:ename,:esal,:eaddr)"
	cursor.executemany(query,list)
	con.commit()
	print("Records copied from Mysql to Oracle database successfully.....")
except cx_Oracle.DatabaseError as e:
	if con:
		con.rollback()
		print("There is a problem with sql :",e)
finally:
	if cursor:
		cursor.close()
```

```
    if con:
        con.close()
```

```
o/p:D:\pythonclasses>py test.py
<class 'tuple'>
<class 'list'>
[(111, 'Katrina', 1000.0, 'mumbai'), (222, 'Kareena', 2000.0, 'mumbai'), (333, 'Deepika',
3000.0, 'Mumbai')]
Records copied from Mysql to Oracle database successfully.....
```