Logging Module

It is a default module which comes with python software.
This module defines functions and classes which implement a flexible event logging system for applications and libraries.

Types Messages

1. Information message
2. Debugging message
3. Warning message
4. Error message
5. Critical message

All these messages are printed or displayed using print function. It is complex to identify the type of message. Print function cannot manage by programmer. This can be avoided by display messages logging module. Logging module provides different methods of display messages (File, Database, Console, Network,…)

1. log()
2. error()
3. debug()
4. warning()
5. critical()

Example:

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

Output:

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

Basic Configurations
You can use the basicConfig(***kwargs*) method to configure the logging:

Some of the commonly used parameters for basicConfig() are the following:

level: The root logger will be set to the specified severity level.
filename: This specifies the file.
filemode: If filename is given, the file is opened in this mode. The default is a, which means append.
format: This is the format of the log message.

Example:

```
import logging

logging.basicConfig(level=logging.CRITICAL)
logging.debug("This is debug message")
logging.critical("This is Cirtical message")
logging.warning("This is warning message")
logging.info("This is info message")
logging.error("This error message")
```

Output:

```
CRITICAL:root:This is Cirtical message
```

Example:

```
import logging

logging.basicConfig(level=logging.WARNING)
logging.debug("This is debug message")
logging.critical("This is Cirtical message")
logging.warning("This is warning message")
logging.info("This is info message")
logging.error("This error message")

# CRITICAL
# ERROR
# WARNING
# INFO
```

DEBUG

Output:
CRITICAL:root:This is Cirtical message
WARNING:root:This is warning message
ERROR:root:This error message

Example:
```
import logging

logging.basicConfig(level=logging.ERROR)
logging.info("Number of must be integer type")
try:
    num1=int(input("Enter first number "))
    num2=int(input("Enter second number "))
    res=num1/num2
    print(f'{num1}/{num2} is {res}')
except ZeroDivisionError:
    logging.error("Cannot divide number with zero")
```

Output:
Enter first number 4
Enter second number 0
ERROR:root:Cannot divide number with zero

Example:
```
import logging

logging.basicConfig(level=logging.DEBUG,filename="e:\\error.log",filemode
="w")

logging.debug("This is debug message")
logging.info("This is info message")
```

Output:
The output is saved inside error.log file

Example:

```
import logging


a=int(input("Enter first number "))
b=int(input("Enter second number "))
if a>b:
    logging.debug("inside if block")
    print(f'{a} is max')
else:
    logging.debug("else block")
    print(f'{b} is max')
```

Output:

```
Enter first number 4
Enter second number 2
4 is max
```

Garbage Collection (gc module)

In python memory management is done automatically.
Memory management consist of two things,

1. Allocation of Memory (Reserving)
2. De-Allocation of Memory (Un-Reserving)

Python programmer always reserve memory by creating variables or objects.
All the objects are not deleted by garbage collector (service) provided by PVM (Python Virtual Machine).
It is delete only objects which are not in use.

How to Garbage collector identity which objects is not in use?
Garbage collector identify object using object's reference count.
If object reference count is zero, it is elg for garbage collection.

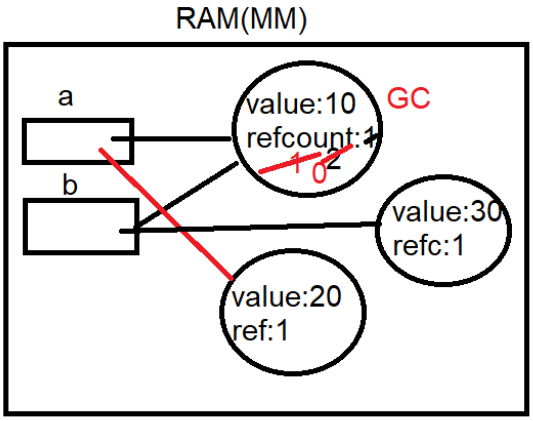
The object which is not binding with any variable whose reference count will be zero.

a=10

b=a

a=20

b=30



Example:

```
import sys
class Employee:
    def __init__(self,e,n):
        self.__empno=e
        self.__ename=n
    def __str__(self):
        return f'{self.__empno},{self.__ename}'



e1=Employee(1,"naresh")
print(e1)
e2=Employee(2,"suresh")
print(e2)
e3=e1
e4=e1
c1=sys.getrefcount(e1)
c2=sys.getrefcount(e2)
print(c1)
```

```
print(c2)
del e3
del e4
c1=sys.getrefcount(e1)
print(c1)
```

Output:

```
1,naresh
2,suresh
4
2
2
```

Destructor

Destructor is a special function or magic function/method.
This method is executed by PVM before object is garbage collected or deleted from main memory.

Name of the destructor is __del__(self)
Name of the constructor is __init__(self)

Block of code which has to executed on deletion of object is written inside destructor.

Example:

```
import sys
class Employee:
    def __init__(self,e,n):
        self.__empno=e
        self.__ename=n
        print("Employee is created...")
    def __str__(self):
        return f'{self.__empno},{self.__ename}'
    def __del__(self):
        print("Employee object is deleted...")

def main():
    e1=Employee(1,"naresh")
    print(e1)
```

```
    e2=Employee(2,"suresh")
    print(e2)

main()
e3=Employee(3,"kishore")
print(e3)
del e3
```

Output:

```
Employee is created...
1,naresh
Employee is created...
2,suresh
Employee object is deleted...
Employee object is deleted...
Employee is created...
3,kishore
Employee object is deleted...
```