Garbage Collection
Python is dynamic language. In python memory management is done implicitly.
Memory management consist of two things

1. Allocation of memory
2. De-Allocation of memory

In python programmer create object in main memory but programmer never delete objects from main memory. These deleting of objects are done by Python Virtual machine service called garbage collector.

Garbage collector is a service provided by PVM (Python Virtual Machine). Garbage collector will not delete all the objects. Garbage collector deletes or removes only unreferenced objects.

What is unreferenced object?
An object which does not bind with variable is called unreferenced object or unused object. An object whose reference count is 0 is called unreferenced object.

Every object is having a property called refcount. This count is incremented when object bind with variable and decremented when object is unbind with variable.

sys.getrefcount(object)
Return the reference count of the object.

```
>>> import sys
>>> class Employee:
...     def __init__(self):
...         self.empno=None
...         self.ename=None
...
...
>>> emp1=Employee()
>>> emp2=emp1
>>> sys.getrefcount(emp1)
3
```

Destructor

Destructor is a special function which is executed before object is destroyed by garbage collector.

Constructor is executed automatically whenever we create object.
Constructor is for allocating resources and destructor is de-allocating resources allocated within constructor.
Block of code which has to execute automatically before object is destroyed.

```
def __del__(self):
        statement-1
        statement-2
```

Example:

```
class A:
   def __init__(self):
      print("inside constructor")
   def __del__(self):
      print("inside destructor")

def main():
   obj1=A()
main()
```

Output:

```
inside constructor
inside destructor
```

gc module

gc is a predefined module in python. This module comes with python software. using this gc module, python program can communicate garbage collector.

Advantage of automatic memory management

1. Avoid wastage of memory
2. Avoids memory leak problem

gc.enable()

Enable automatic garbage collection.

gc.disable()

Disable automatic garbage collection.

gc.isenabled()

Return True if automatic collection is enabled.

Example:

```
>>> import gc
>>> gc.isenabled()
True
>>> gc.disable()
>>> gc.isenabled()
False
>>> gc.enable()
>>> gc.isenabled()
True
```

Once the garbage collector is disabled, during the execution of garbage collection is not done. After execution of program, all the objects are garbage collected.

Example:

```
import gc
class A:
    def __init__(self):
        print("inside constructor")
    def __del__(self):
        print("inside destructor")

def main():
    obj1=A()
    gc.disable()
    obj1=None
    input()
main()
```

Output:

```
inside constructor
inside destructor
```

Python Logging

Logging is a very useful tool in a programmer's toolbox. It can help you develop a better understanding of the flow of a program and discover scenarios that you might not even have thought of while developing.

Logs provide developers with an extra set of eyes that are constantly looking at the flow that an application is going through. They can store information, like which user or IP accessed the application. If an error occurs, then they can provide more insights than a stack trace by telling you what the state of the program was before it arrived at the line of code where the error occurred.

The Logging Module

The logging module in Python is a ready-to-use and powerful module that is designed to meet the needs of beginners as well as enterprise teams. It is used by most of the third-party Python libraries, so you can integrate your log messages with the ones from those libraries to produce a homogeneous log for your application.

Adding logging to your Python program is as easy as this:

```
import logging
```

With the logging module imported, you can use something called a “logger” to log messages that you want to see. By default, there are 5 standard levels indicating the severity of events. Each has a corresponding method that can be used to log events at that level of severity. The defined levels, in order of increasing severity, are the following:

- **DEBUG**
- **INFO**
- **WARNING**
- **ERROR**
- **CRITICAL**

The logging module provides you with a default logger that allows you to get started without needing to do much configuration. The corresponding methods for each level can be called as shown in the following 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')
```

The output of the above program would look like this:

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

The output shows the severity level before each message along with root, which is the name the logging module gives to its default logger. (Loggers are discussed in detail in later sections.) This format, which shows the level, name, and message separated by a colon (:), is the default output format that can be configured to include things like timestamp, line number, and other details.

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.
By using the level parameter, you can set what level of log messages you want to record. This can be done by passing one of the constants available in the class, and this would enable all logging calls at or above that level to be logged. Here's an example:
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('This will get logged')

DEBUG:root:This will get logged
All events at or above DEBUG level will now get logged.

The following example shows the usage of all three:

```
import logging
logging.basicConfig(filename='app.log', filemode='w', format='%(name)s -
%(levelname)s - %(message)s')
logging.warning('This will get logged to a file')
```

```
root - ERROR - This will get logged to a file
```

The message will look like this but will be written to a file
named app.log instead of the console. The filemode is set to w, which means the log file is opened in “write mode” each time basicConfig() is called, and each run of the program will rewrite the file. The default configuration for filemode is a, which is append.

Add Timestamp to logs with log message
You can also add date and time info(timestamp) to your logs along with the log message. The code snippet for the same is given below:

```
import logging

logging.basicConfig(format='%(asctime)s - %(message)s',
level=logging.INFO)
logging.info('This message is to indicate that Admin has just logged in')
```

```
2020-06-19 11:43:59,887 - This message is to indicate that Admin has just
logged in
```

Python Logging - Store Logs in a File
There are some basic steps and these are given below:

First of all, simply import the logging module just by writing import logging.

The second step is to create and configure the logger. To configure logger to store logs in a file, it is mandatory to pass the name of the file in which you want to record the events.

In the third step, the format of the logger can also be set. Note that by default, the file works in append mode but we can change that to write mode if required.

You can also set the level of the logger.

So let's move on to the code now:

```
#importing the module
import logging

#now we will Create and configure logger
logging.basicConfig(filename="std.log",
                                    format='%(asctime)s %(message)s',
                                    filemode='w')

#Let us Create an object
logger=logging.getLogger()

#Now we are going to Set the threshold of logger to DEBUG
logger.setLevel(logging.DEBUG)

#some messages to test
logger.debug("This is just a harmless debug message")
logger.info("This is just an information for you")
logger.warning("OOPS!!!Its a Warning")
logger.error("Have you try to divide a number by zero")
logger.critical("The Internet is not working....")
```

The above code will write some messages to file named std.log. If we will open the file then the messages will be written as follows:

```
2020-06-19 12:48:00,449 - This is just harmless debug message
2020-06-19 12:48:00,449 - This is just an information for you
2020-06-19 12:48:00,449 - OOPS!!!Its a Warning
2020-06-19 12:48:00,449 - Have you try to divide a number by zero
2020-06-19 12:48:00,449 - The Internet is not working...
```

You can change the format of logs, log level or any other attribute of the LogRecord along with setting the filename to store logs in a file along with the mode.

Turtle module

Turtle graphics is a popular way for introducing programming to kids. It is used to drawing graphics or painting.

https://realpython.com/beginners-guide-python-turtle/