Unit 8 Introduction to Object Oriented Programming and Exception Handling

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual objects. Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line a system component processes some material, ultimately transforming raw material into a finished product.

An object contains data, like the raw or preprocessed materials at each step on an assembly line, and behavior, like the action each assembly line component performs.

What Is Object-Oriented Programming in Python?

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

For instance, an object could represent a person with properties like a name, age, and address and behaviors such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

Another common programming paradigm is procedural programming, which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, that flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

What is Class?

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let’s consider an example, let’s say you wanted to track the number of dogs that may have different attributes like breed, age. If a list is used, the

first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:
Classes are created by keyword class. Attributes are the variables that belong to a class. Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute Class Definition Syntax:

class ClassName:

Statement-1 . . . Statement-N

In [1]:
```
#Example Creating an empty Class in Python
# Python3 program to
# demonstrate defining
# a class

class Dog:
    pass
```

In [4]:
```
#Example 2 for creating an class in python
class MyClass:
    x = 5
print(MyClass)
```

```
<class '__main__.MyClass'>
```

What is Object?

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of :

State: It is represented by the attributes of an object. It also reflects the properties of an object. Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects. Identity: It gives a unique name to an object and enables one object to interact with other objects. To understand the state, behavior, and identity let us take the example of the class dog (explained above).

The identity can be considered as the name of the dog. State or Attributes can be considered as the breed, age, or color of the dog. The behavior can be considered as to whether the dog is eating or sleeping.

In [5]:
```
##Example of creating the objects for Class MyClass():

obj = MyClass()
```

What is the self?

The self
Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it If we have a method that takes no arguments, then we still have to have one argument. When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

Syntax for the self: def function(self, name):

The **init**Function():

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in **init**() function.

All classes have a function called **init**(), which is always executed when the class is being initiated.

Use the **init**() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

In [6]:
```
#Example to Create a class named Person, use the __init__() function to assign values

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

p1 = Person("John", 36)

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

```
John
36
```

Attributes in Python:

In the body of **init**(), we are using self variables 2 times, for the following: self.name='name' creates an attribute called name and assigns to it the value of the name parameter. self.age=age attribute is created and assigned to the value of age parameter passed.

There are two types of attributes in Python:

1. Class Attribute: -variables are same for all the instances of the class. -no new values for each new instances created.

In [7]:
```
#Example of Class Attribute:
class Human:
    #class attribute
    species="Homo Sapiens"
#species will have a fixed value for any object we create.
```

1. Instance Attribute: -are the variables which are defined inside of any function in class. -have different values for every instance of the class. -depends upon the value we pass while creating the instance.

In [8]:
```
#Example of instance attribute:
class Human:
    #class attribute
    species="Homo Sapiens"
    def __init__(self,name,age,gender):
        self.name=name
        self.age=age
        self.gender=gender
#Here name,age and gender are the instance attributes.They will have a different value
#For properties having a similar values per instance-class attribute
#For properties having different instance,-instance attribute
```

Instance Methods

An instance method is just like function which is defined within a class that can be called whenever required for as many as times required. Like **init**, an instance also takes it's first parameter as self.

In [1]:
```
#Example of creating methods:
class Human:
    #class attribute
    species="Homo Sapiens"
    def __init__(self,name,age,gender):
        self.name=name
        self.age=age
        self.gender=gender
    #instance method
    def speak(self):
        return
```

Constructors in Python

Constructor is a special method used to create and initialize an object of a class. On the other hand, a destructor is used to destroy the object. In object-oriented programming, A constructor

is a special method used to create and initialize an object of a class. This method is defined in the class.

The constructor is executed automatically at the time of object creation. The primary use of a constructor is to declare and initialize data member/ instance variables of a class. The constructor contains a collection of statements (i.e., instructions) that executes at the time of object creation to initialize the attributes of an object.

In Python, Object creation is divided into two parts in Object Creation and Object initialization

Internally, the **new** is the method that creates the object And, using the **init**() method we can implement constructor to initialize the object.

Syntax of a constructor

def **init**(self):

```
    # body of the constructor
```

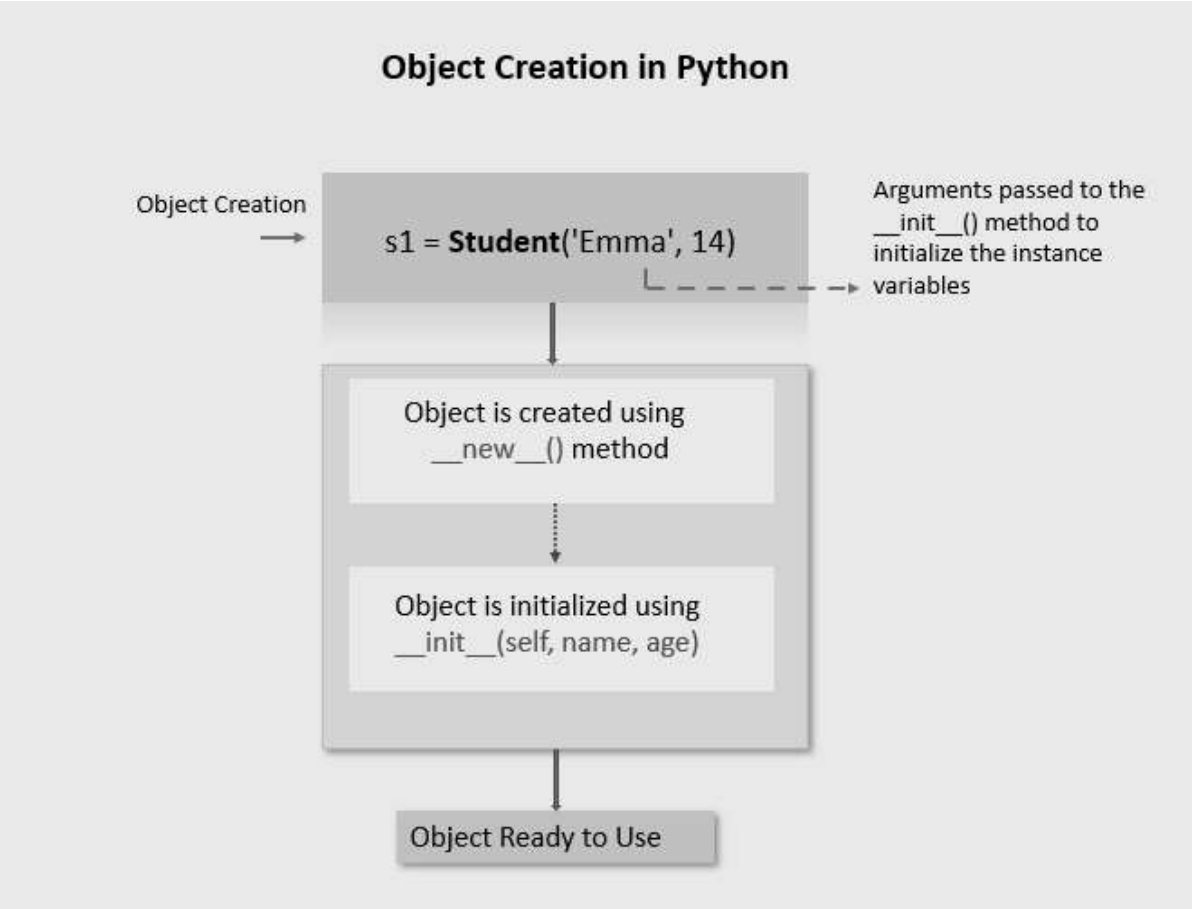
Where,

def: The keyword is used to define function. **init**() Method: It is a reserved method. This method gets called as soon as an object of a class is instantiated. self: The first argument self refers to the current object. It binds the instance to the **init**() method. It's usually named self to follow the naming convention. Note: The **init**() method arguments are optional. We can define a constructor with any number of arguments.

In [2]:
```
##Example: Create a Constructor in Python.
class Student:

    # constructor
    # initialize instance variable
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('All variables initialized')

    # instance Method
    def show(self):
        print('Hello, my name is', self.name)


# create object using constructor
s1 = Student('Emma')
s1.show()
```

```
Inside Constructor
All variables initialized
Hello, my name is Emma
```

Chart for creating a constructor in Python



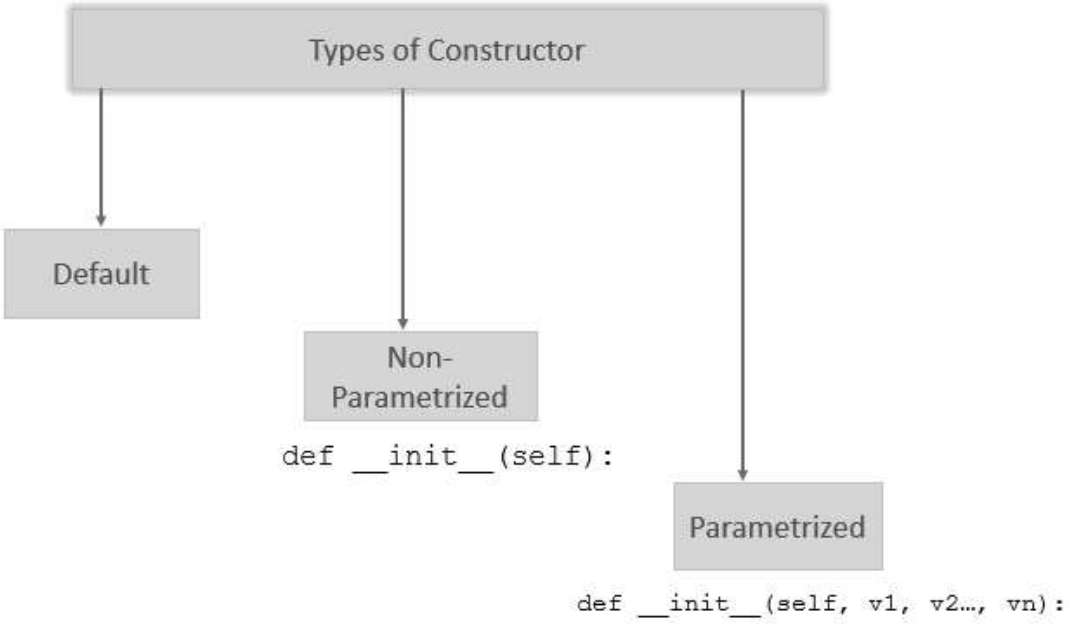
For every object, the constructor will be executed only once. For example, if we create four objects, the constructor is called four times. In Python, every class has a constructor, but it's not required to define it explicitly. Defining constructors in class is optional. Python will provide a default constructor if no constructor is defined.

Types of Constructors:

In Python, we have the following three types of constructors.

1. Default Constructor
2. Non-parametrized constructor

3. Parameterized constructor



1. Default Constructor

Python will provide a default constructor if no constructor is defined. Python adds a default constructor when we do not include the constructor in the class or forget to declare it. It does not perform any task but initializes the objects. It is an empty constructor without a body.

In [3]:
```
#Example of Default Constructor
class Employee:

    def display(self):
        print('Inside Display')

emp = Employee()
emp.display()
```

```
Inside Display
```

As you can see in the example, we do not have a constructor, but we can still create an object for the class because Python added the default constructor during a program compilation.

2. Non-Parametrized Constructor

A constructor without any arguments is called a non-parameterized constructor. This type of constructor is used to initialize each object with default values.

This constructor doesn't accept the arguments during object creation. Instead, it initializes every object with the same set of values.

In [4]:
```
#Example of Non-Parametrized Constructor
class Company:

    # no-argument constructor
    def __init__(self):
        self.name = "Python"
        self.address = "ABC Street"

    # a method for printing data members
    def show(self):
        print('Name:', self.name, 'Address:', self.address)

# creating object of the class
cmp = Company()

# calling the instance method using the object
cmp.show()
```

```
Name: Python Address: ABC Street
```

As you can see in the example, we do not send any argument to a constructor while creating an object.

3. Parameterized Constructor

A constructor with defined parameters or arguments is called a parameterized constructor. We can pass different values to each object at the time of creation using a parameterized constructor.

The first parameter to constructor is self that is a reference to the being constructed, and the rest of the arguments are provided by the programmer. A parameterized constructor can have any number of arguments.

For example, consider a company that contains thousands of employees. In this case, while creating each employee object, we need to pass a different name, age, and salary. In such cases, use the parameterized constructor.

In [5]:
```
#Example of Parameterized Constructor
class Employee:
    # parameterized constructor
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    # display object
    def show(self):
        print(self.name, self.age, self.salary)

# creating object of the Employee class
emma = Employee('Emma', 23, 7500)
emma.show()

kelly = Employee('Kelly', 25, 8500)
```

```
kelly.show()
#In the above example, we define a parameterized constructor which takes three paramet
```

```
Emma 23 7500
Kelly 25 8500
```

Python Generators

Python's generator functions are used to create iterators (which can be transversed liek list, tuple) and return a traversal object. It helps to transverse all the items one at a time present in the iterator.

Generator functions are defined as the normal function, but to identify the difference between the normal function and generator function is that in the normal function, we use the return keyword to return the values, and in the generator function, instead of using the return, we use yield to execute our iterator.

In [6]:
```
#Example of using Python Generators:
def gen_fun():
    yield 10
    yield 20
    yield 30
for i in gen_fun():
    print(i)
```

```
10
20
30
```

In the above example, gen_fun() is a generator function. This function uses the yield keyword instead of return, and it will return a value whenever it is called.

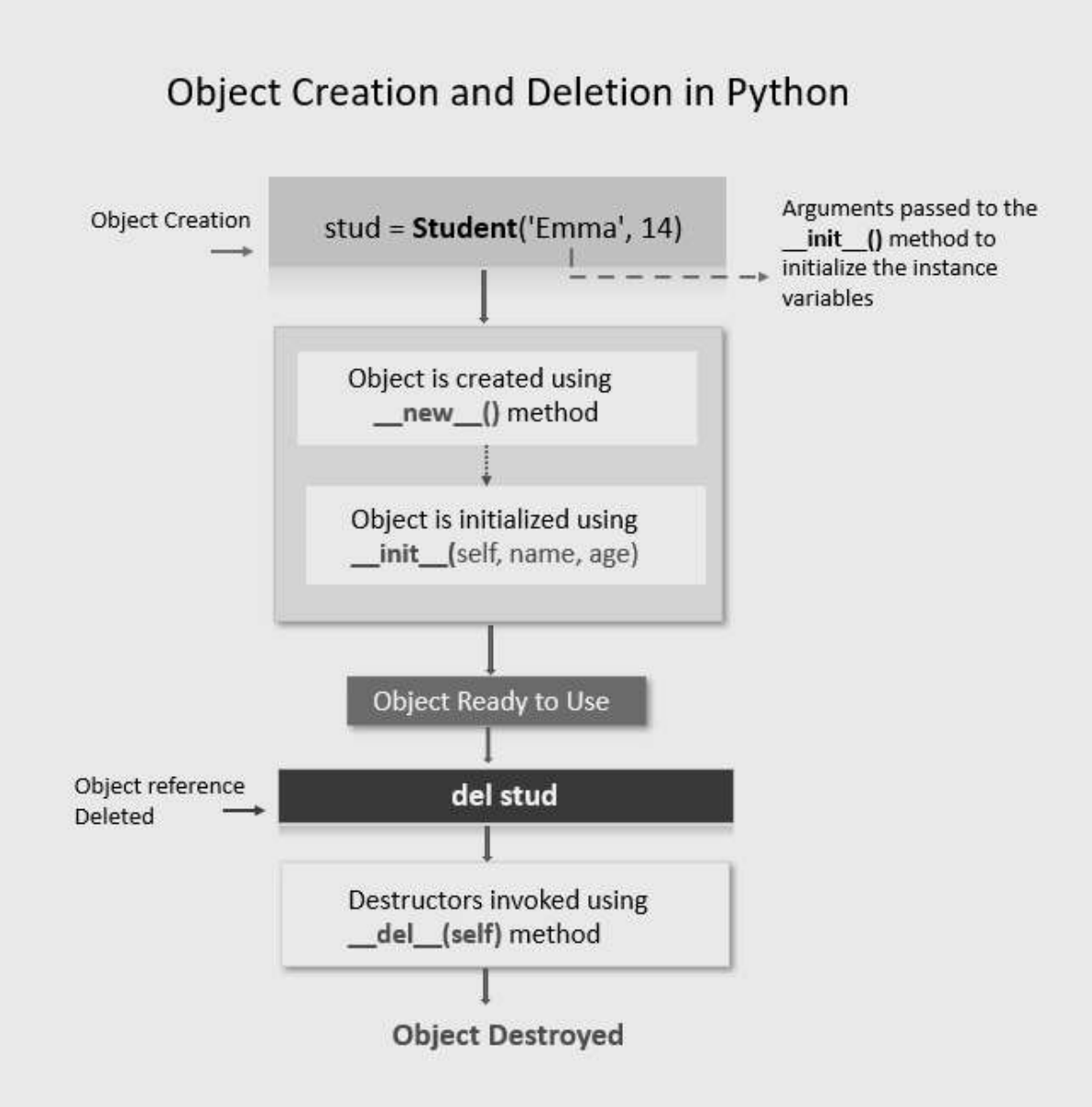
Yield It is used in generator functions. It is responsible for controlling the flow of the generator function. After returning the value from yield, it pauses the execution by saving the states. Return It is used in normal functions. Return statement returns the value and terminates the function.

Difference Between Generator Function & Normal Function • In generator functions, there are one or more yield functions, whereas, in Normal functions, there is only one function • When the generator function is called, the normal function pauses its execution, and the call is transferred to the generator function. • Local variables and their states are remembered between successive calls.

Destructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.

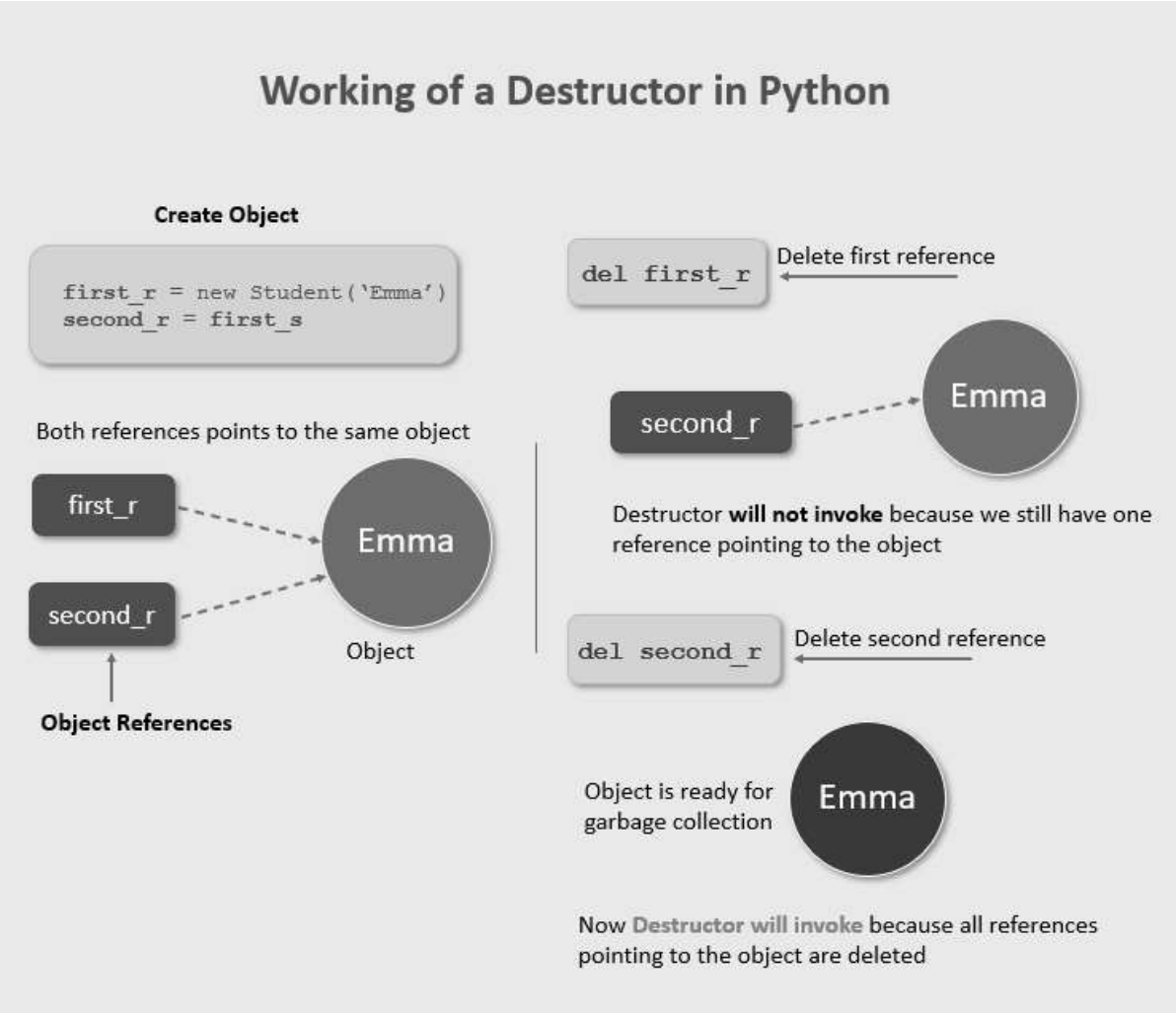
The **del**() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.



Syntax of destructor declaration :

def **del**(self):

body of destructor



In [7]:
```
# Python program to illustrate destructor
class Employee:

    # Initializing
    def __init__(self):
        print('Employee created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

```
Employee created.
Destructor called, Employee deleted.
```

In [9]:
```
# Python program to illustrate destructor
class Student:

    # constructor
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')

    def show(self):
        print('Hello, my name is', self.name)
```

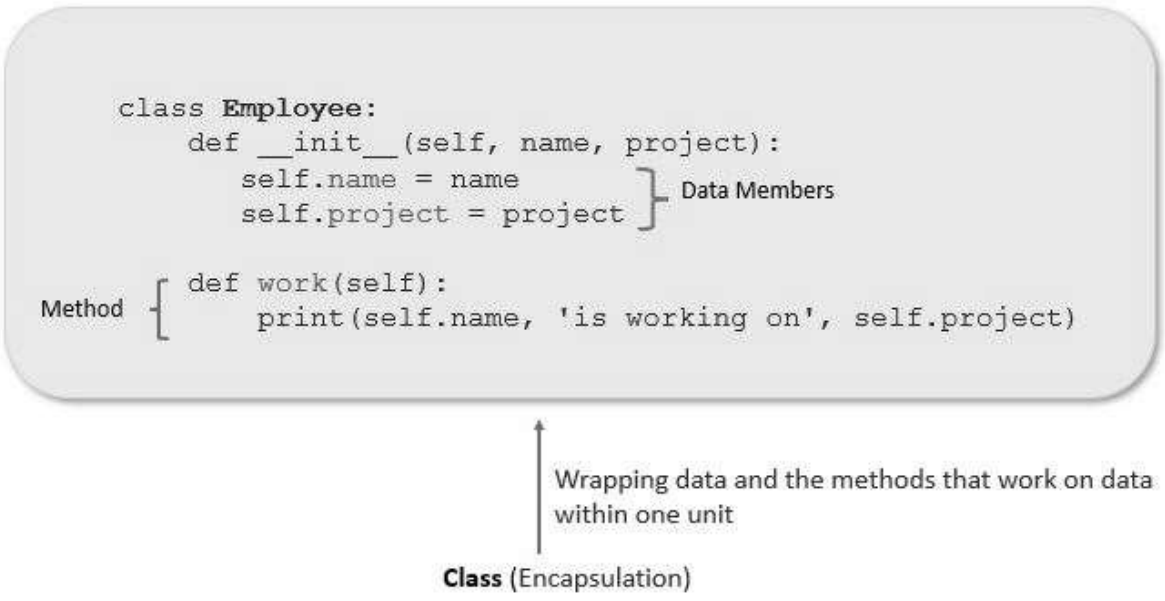
```
    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Student('Emma')
s1.show()

# delete object
del s1
```

```
Inside Constructor
Object initialized
Hello, my name is Emma
Inside destructor
Object destroyed
```

Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP), including abstraction, inheritance, and polymorphism. Encapsulation in Python describes the concept of bundling data and methods within a single unit. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.

Implementation of Encapsulation in Python



In [10]:
```
#Example of Encapsulation
#we create an Employee class by defining employee attributes such as name and salary a
#instance variable and implementing behavior using work() and show() instance methods.
class Employee:
    # constructor
    def __init__(self, name, salary, project):
        # data members
        self.name = name
```

```
        self.salary = salary
        self.project = project

    # method
    # to display employee's details
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

    # method
    def work(self):
        print(self.name, 'is working on', self.project)

# creating object of a class
emp = Employee('Jessa', 8000, 'NLP')

# calling public method of the class
emp.show()
emp.work()
```

```
Name:  Jessa Salary: 8000
Jessa is working on NLP
```

Getter and Setter method in Python for implementing Encapsulation

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.

A getter retrieves an object's current attribute value, whereas a setter changes an object's attribute value.

In Python, private variables are not hidden fields like in other programming languages.

What is Getter in Python?

Getters are the methods that are used in Object-Oriented Programming (OOPS) to access a class's private attributes. The setattr() function in Python corresponds to the getattr() function in Python. It alters an object's attribute values.

What is Setter in Python?

The setter is a method that is used to set the property's value. It is very useful in object-oriented programming to set the value of private attributes in a class.

Generally, getters and setters are mainly used to ensure the data encapsulation in OOPs.

In [11]:
```
#Example of getter and setter method for implementing encapsulation in python.
class Student:
    def __init__(self, name, age):
```

```
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())

# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

```
Name: Jessa 14
Name: Jessa 16
```

Advantages of Encapsulation

Security: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification. Data Hiding: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them. Simplicity: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other. Aesthetics: Bundling data and methods within a class makes code more readable and maintainable

Exception Handling in Python

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax Here is simple syntax of try....except...else blocks –

try: You do your operations here; ..... except ExceptionI: If there is ExceptionI, then execute this block. except ExceptionII: If there is ExceptionII, then execute this block. ..... else: If there is no exception then execute this block.

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

In [13]:
```
#Example of Error occured i.e. Exception
# initialize the amount variable
marks = 10000

# perform division with 0
a = marks / 0
print(a)
#In the above example raised the ZeroDivisionError as we are trying to divide a number
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In [13], line 6
      3 marks = 10000
      5 # perform division with 0
----> 6 a = marks / 0
      7 print(a)

ZeroDivisionError: division by zero
```

List of Standard Exceptions:

Sr.No. Exception Name & Description 1
Exception

Base class for all exceptions

2
StopIteration

Raised when the next() method of an iterator does not point to any object.

3
SystemExit

Raised by the sys.exit() function.

4
StandardError

Base class for all built-in exceptions except StopIteration and SystemExit.

5
ArithmeticError

Base class for all errors that occur for numeric calculation.

6
OverflowError

Raised when a calculation exceeds maximum limit for a numeric type.

7
FloatingPointError

Raised when a floating point calculation fails.

8
ZeroDivisionError

Raised when division or modulo by zero takes place for all numeric types.

9
AssertionError

Raised in case of failure of the Assert statement.

10
AttributeError

Raised in case of failure of attribute reference or assignment.

11
EOFError

Raised when there is no input from either the raw_input() or input() function and the end of file is reached.

12
ImportError

Raised when an import statement fails.

13
KeyboardInterrupt

Raised when the user interrupts program execution, usually by pressing Ctrl+c.

14
LookupError

Base class for all lookup errors.

15
IndexError

Raised when an index is not found in a sequence.

16
KeyError

Raised when the specified key is not found in the dictionary.

17
NameError

Raised when an identifier is not found in the local or global namespace.

18
UnboundLocalError

Raised when trying to access a local variable in a function or method but no value has been assigned to it.

19
EnvironmentError

Base class for all exceptions that occur outside the Python environment.

20
IOError

Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.

21
IOError

Raised for operating system-related errors.

22
SyntaxError

Raised when there is an error in Python syntax.

23
IndentationError

Raised when indentation is not specified properly.

24
SystemError

Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.

25
SystemExit

Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

26
TypeError

Raised when an operation or function is attempted that is invalid for the specified data type.

27
ValueError

Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

28
RuntimeError

Raised when a generated error does not fall into any category.

29
NotImplementedError

Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

In [14]:
```
# Python program to handle simple runtime error
a = [1, 2, 3]
try:
    print ("Second element = %d" %(a[1]))

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

```
Second element = 2
An error occurred
```

Catching Specific Exception

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

```
    # statement(s)
```

except IndexError:

```
    # statement(s)
```

except ValueError:

```
    # statement(s)
```

In [18]:
```
# Program to handle multiple errors with one
# except statement
def fun(a):
    if a < 4:

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

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

try:
    fun(5)

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

```
NameError Occurred and Handled
```

In [19]:
```
# Program to handle multiple errors with one
# except statement
def fun(a):
    if a < 4:

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

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

try:
    fun(3)

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

```
ZeroDivisionError Occurred and Handled
```

Try with Else Clause

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

In [20]:
```
#Program to depict else clause with try-except

# Function which returns a/b
def AbyB(a,b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)
# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

```
-5.0
a/b result in 0
```

Finally Keyword in Python

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

Syntax:

try:

```
    # Some Code....
```

except:

```
    # optional block
    # Handling of exception (if required)
```

else:

```
    # execute if no exception
```

finally:

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

In [21]:
```
# Python program to demonstrate finally

# No exception Exception raised in try block
```

```
try:
    k = 5//0  # raises divide by zero exception.
    print(k)
# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")
finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

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

Raising Exception

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

In [22]:
```
# Program to depict Raising Exception

try:
    raise NameError("Hi there")  # Raise Error
except NameError:
    print ("An exception")
    raise  # To determine whether the exception was raised or not
```

```
An exception
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [22], line 4
      1 # Program to depict Raising Exception
      3 try:
----> 4     raise NameError("Hi there")  # Raise Error
      5 except NameError:
      6     print ("An exception")

NameError: Hi there
```

In [1]:
```
#Another example of raising custom exception
def isStringEmpty(a):
    if(type(a)!=str):
        raise TypeError('a has to be string')
    if (not a):
        raise ValueError("a cannot be null")
    a.strip()
    if(a==""):
        return True
    return False
try:
    a=123
    print('isStringEmpty',isStringEmpty(a))
except ValueError as e:
    print('ValueError raised:',e)
except TypeError as e:
    print('TypeError raised:',e)
```

```
TypeError raised: a has to be string
```

In [2]:
```
#Another example of raising custom exception
def isStringEmpty(a):
    if(type(a)!=str):
        raise TypeError('a has to be string')
    if (not a):
        raise ValueError("a cannot be null")
    a.strip()
    if(a==""):
        return True
    return False
try:
    a=''
    print('isStringEmpty',isStringEmpty(a))
except ValueError as e:
    print('ValueError raised:',e)
except TypeError as e:
    print('TypeError raised:',e)
```

```
ValueError raised: a cannot be null
```

In [3]:
```
#Another example of raising custom exception
def isStringEmpty(a):
    if(type(a)!=str):
        raise TypeError('a has to be string')
    if (not a):
        raise ValueError("a cannot be null")
    a.strip()
    if(a==""):
        return True
    return False
try:
    a='a'
    print('isStringEmpty',isStringEmpty(a))
except ValueError as e:
    print('ValueError raised:',e)
except TypeError as e:
    print('TypeError raised:',e)
```

```
isStringEmpty False
```

In [ ]:
```

```