GUIA 4. Python Orientado a Objetos

Introducción

Python ha sido un lenguaje orientado a objetos desde que existe, es por esto que usar objetos y clases es muy fácil. Veremos a continuación veremos una introducción de programación orientada a objetos inicialmente:

Términos relevantes

Clase: Plantilla o estructura que sirve para definir los atributos (métodos y variables) que caracterizan cualquier objeto de la clase. Otras palabras, una clase es el molde con el que los objetos son creados, y todos los objetos tienen los mismos atributos o características por ser creados de dicho molde.

Variable de clase: Variables definidas dentro de la clase pero fuera de los métodos y es compartida por todas las instancias u objetos de la clase.

Método: Son un tipo de función, se diferencia de una función común es que el método es llamado desde un objeto o instancia y en que el método puede operar datos que se encuentran contenidos en la clase.

Sobrecarga de función: Asignar varios comportamientos a un método dependiendo de los tipos de objetos u argumentos usados.

Variable de instancia: Variables usadas dentro de un método y pertenece sólo a la instancia actual u objeto.

Herencia: Creación de una clase que herede las características de una clase padre.

Instancia: Un objeto individual creado de una clase.

Instanciación: Creación de una instancia de una clase.

Objeto: única instancia con la estructura definida por su clase. Un objeto tiene variables de clase, variables de instancia y métodos.

Nota: Si algún concepto no es claro debe profundizar el concepto en algún libro de programación orientado a objetos.

Creación de clase

Para crear una clase usamos la sentencia ***class***, luego el nombre de la clase como se describe:

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

La clase tiene un String para documentarla que puede ser accedida por NombreClase.__doc__

Creemos la clase Aprendiz en el archivo aprendiz.py:

```
class Aprendiz:
   'Clase de aprendices Sena'
   total = 0

   def __init__(self, name, lastname):
      self.nombre = name
      self.apellido = lastname
      Aprendiz.total += 1

   def mostrarTotal(self):
     print("Total %d", Aprendiz.total)

   def mostrarAprendiz(self):
      print("Name : ", self.nombre,  ", LastName: ", self.apellido)
```

La variable ***total***, es una variable de clase cuyo valor es compartido por todas las instancias de clase. Puede ser accedida por Aprendiz.total.

El método **__init__()** es un método especial, llamado constructor o método de inicialización y se llama cuando se crea el objeto.

Se declaran los otros métodos como las funciones, con la excepción que el primer argumento es **self**. Este argumento lo agrega Python no es necesario incluirlo cuando invoquemos los métodos.

Creación de objetos

Para crear instancias de una clase, se llama la clase usando el nombre de la clase y se pasan los argumentos definidos en el método __init()__.

Creamos un objeto de la clase Aprendiz, lo podemos hacer en otro archivo principal.py importando la clase:

```
aprendiz.py | principal.py | moduloprincipal.py | modulo1.py
1  import aprendiz
2
3  aprendiz1 =aprendiz.Aprendiz("carlos","chaguendo")
4  aprendiz2 =aprendiz.Aprendiz("sara","lopez")
```

Accediendo a Atributos

Se puede acceder a los atributos usando el . dentro del objeto:

```
aprendiz.py | principal.py | moduloprincipal.py | modulo1.py
1  import aprendiz
2
3  aprendiz1 =aprendiz.Aprendiz("carlos","chaguendo")
4  aprendiz2 =aprendiz.Aprendiz("sara","lopez")
5
6  aprendiz1.mostrarAprendiz()
7  aprendiz2.mostrarAprendiz()
8  print("Total  %d" , aprendiz.Aprendiz.total)
9
```

Pruebe el código y verifique el resultado.

Se pueden agregar, modificar o remover atributos de las clases y objetos en cualquier momento:

```
aprendiz1.edad = 12  # Agregar edad a aprendiz1
aprendiz1.edad = 13  # Modificar edad a aprendiz1
del aprendiz.edad # Borra el atributo edad
```

Existen funciones para acceder a atributos que podemos usar:

getattr: Acceder al atributo del objeto

hasattr: verifica si el atributo existe o no

setattr: agrega un atributo, si no existe lo crea

delattr borra un atributo

Pruebe el siguiente código:

```
aprendiz1.edad = 12  # Agregar edad a aprendiz1
aprendiz1.edad = 13  # Modificar edad a aprendiz1
#del aprendiz.edad # Borra el atributo edad

print(hasattr(aprendiz1, 'edad') )   # Returns true if 'edad' attribute exists
print(getattr(aprendiz1, 'edad') )   # Returns value of 'edad' attribute
print(setattr(aprendiz1, 'edad', 8) ) # Set attribute 'edad' at 8
print(aprendiz1.edad)
print(delattr(aprendiz1, 'edad') )   # Delete attribute 'edad'
print(hasattr(aprendiz1, 'edad') )
```

Destruyendo Objetos

Python borra objetos innecesarios automáticamente para liberar espacio en memoria. El proceso usado por Python para optimizar la memoria se llama Garbage Collection. Este garbage collector se ejecuta durante la ejecución del programa y se dispara cuando el número de referencias del objeto es cero.

El número de referencias del objeto se vuelve cero cuando se disminuye ese número, lo que se da cuando se llama la función del, o cuando su referencia queda fuera del contexto.

```
a = 40      # Create object <40>
del a       # Decrease ref. count  of <40>
```

Podemos implementar el método __del__(), llamado destructor que se llama cuando el objeto es destruido. Este método puede ser usado para limpiar recursos que no están en memoria (archivos, bases de datos, etc) o mejorar el rendimiento en general.

Pruebe el siguiente código y analice el resultado, puede crearlo en el archivo punto.py, observe cómo se llama el método destructor:

aprendiz.py | principal.py | punto.py | moduloprincipal.py | modulo1.py

```python
class Punto:
    def __init( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        nombre_clase = "Punto"
        print(nombre_clase, "destroyed")

pt1 = Punto()
pt2 = pt1
pt3 = pt1
print(id(pt1), id(pt2), id(pt3)) # prints the ids of the obejcts
del pt1
del pt2
del pt3
```

Herencia

En lugar de crear clases desde cero, podemos crearlas a partir de clases preexistentes o clases padres que permitan transmitir la estructura (Métodos y Variables) a la clase hija. De esta forma la clase hija hereda los atributos del padre y puede usarlos como si fueran definidos en la clase hija.

La estructura para declarar las clases hijas se define:

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

Hagamos un ejemplo:

```python
class Padre:
    parentAttr = 100
    def __init__(self):
        print("Constructor Padre")

    def parentMethod(self):
        print('Llamando método del padre')
    def setAttr(self, attr):
        Padre.parentAttr = attr
    def getAttr(self):
        print("Atributo padre :", Padre.parentAttr)

class Hija(Padre):
    def __init__(self):
        print("Constructor Hija")
    def childMethod(self):
        print('Llamando método del hijo')

c = Hija()          # instancia del hijo
c.childMethod()      # hijo llama  método propio
c.parentMethod()     # hijo llama método del padre
c.setAttr(200)       # hijo llama método del padre
c.getAttr()          # hijo llama método del padre
```

Analice cómo se pueden usar los métodos del padre en la clase hija y los valores que se llaman en los métodos.

Sobrescritura de métodos

Puedes sobreescribir el comportamiento de los métodos de la clase padre, observemos el siguiente ejemplo:

```python
class Padre1:
    def myMethod(self):
        print('Calling parent method')

class Hijo1(Padre1):
    def myMethod(self):
        print('Calling child method')

c = Hijo1()
c.myMethod()          # clase hija llama el método sobreecrito
```

Observe la sobreescritura del método myMethod.

Visibilidad de Atributos

Un atributo de un objeto puede o no ser visible fuera de la definición de la clase. Si nombramos los atributos con dos guiones bajos al comienzo, entonces no serán visibles por fuera de la clase.

Analice el siguiente ejemplo y observe el resultado, en este caso el atributo __secretCount no es visible por fuera de la clase:

```python
class JustCounter:
    __secretCount = 0

    def count(self):
        self.__secretCount += 1
        print(self.__secretCount)

counter = JustCounter()
counter.count()
counter.count()
print(counter.__secretCount)
```