

Programación Orientada a Objetos - Parte II

1. Funciones para atributos: `getattr()`, `hasattr()`, `setattr()` y `delattr()`

1.1. `getattr()`

La función `getattr()` se utiliza para acceder al valor del atributo de un objeto. Si un atributo no existe retorna el valor del tercer argumento (es opcional).

Ejercicio de Inducción: Pruebe las siguientes líneas de código y observe los resultados:

```
nota_alumno2 = getattr(alumno2, 'nota', 0)
edad_alumno2 = getattr(alumno2, 'edad', 0)
suma_notas = getattr(Alumno, 'sumanotas')
print(nota_alumno2) # 6
print(edad_alumno2) # 0
print(suma_notas) # 15
```

1.2. `hasattr()`

La función `hasattr()` devuelve `True` o `False` dependiendo si existe o no el atributo indicado.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y observe los resultados:

```
if not hasattr(alumno2, 'edad'):
    print("El atributo 'edad' no existe")
```

1.3. `setattr()`

Se utiliza para asignar un valor a un atributo. Si el atributo no existe entonces será creado.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y observe los resultados:

```
setattr(alumno2, 'edad', 18)
print(alumno2.edad)
# 18
```

1.4. `delattr()`

La función `delattr()` es para borrar el atributo de un objeto. Si el atributo no existe se producirá una excepción del tipo `AttributeError`.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y observe los resultados:

```
delattr(alumno2, 'edad')
delattr(alumno2, 'curso')
```

2. Atributos de clase (Built-In)

Todas las clases Python incorporan los siguientes atributos especiales:

`__dict__` Devuelve un diccionario que contiene el espacio de nombres de la clase (o de un objeto instanciado).

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
print(Alumno.__dict__)

{ 'mostrarSumaNotas': function 0xb7071854="" alumno.mostrarsumanotas="" at="", '
  mostrarNotaMedia': function 0xb707189c="" alumno.mostrarnotamedia="" at="", '
  __doc__': 'Clase para alumnos', '__init__': function 0xb70717c4="" alumno.
  __init__="" at="", '__weakref__': attribute lumno="" objects="" of="",
  weakref__=""', '__dict__': attribute dict__="" lumno="" objects="" of="", '
  mostrarNumAlumnos': function 0xb7071734="" alumno.mostrarnumalumnos="" at="",
  'sumanotas': 14, '__module__': '__main__', 'numalumnos': 2, '
  mostrarNombreNota': function 0xb707177c="" alumno.mostrarnombrenota="" at="" }

print(alumno1.__dict__)

# { 'nombre': 'Carmen', 'nota': 8 }
```

`__name__` Devuelve el nombre de la clase.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
print(Alumno.__name__)

# Alumno
```

`__doc__` Devuelve la cadena de documentación de la clase o None si no se ha definido.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
print(Alumno.__doc__)

# Clase para alumnos
```

`__module__` Devuelve el nombre del módulo donde se encuentra la clase definida.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
print(Alumno.__module__)
# __main__

from datetime import date
print(date.__module__)

# datetime
```

`__bases__` Devuelve una tupla (posiblemente vacía) que contiene las clases base, en orden de aparición.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
print(Alumno.__bases__)  
  
# (class object = "",)
```

A continuación, se define la clase **Becario** a partir de la clase **Alumno**. La clase **Becario** hereda los atributos y métodos de la clase **Alumno**. Al acceder al atributo `__bases__` de la nueva clase retorna la referencia a la clase padre **Alumno**.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
class Becario(Alumno):  
    pass  
  
print(Becario.__bases__)  
  
# (class main__.alumno = "",)
```

3. Destrucción de objetos (Recolección de basura)

Python elimina objetos que no son necesarios (del tipo Built-In o instancias de clase) automáticamente para liberar espacio de memoria. El proceso por el cual Python reclama periódicamente bloques de memoria que ya no están en uso se denomina Recolección de Basura.

La Recolección de Basura (*Garbage Collection*) de Python se ejecuta durante la ejecución de un programa y se activa cuando el contador de referencia de un objeto llega al valor cero. Este contador va cambiando su valor en función del uso que se haga del objeto. Si se asigna un nuevo nombre o se utiliza en una lista, tupla o diccionario el contador irá en aumento y si se borra el objeto con `del`, su referencia es reasignada o bien se queda fuera de ámbito, va disminuyendo. En el momento que se alcanza el valor cero el recolector comienza con la tarea de "rescate".

Ejemplo:

```
pizza1 = "Margarita" # Crear objeto margarita  
pizza2 = pizza1 # Incrementa el contador de referencia de margarita  
  
cena['lunes'] = pizza2 # Incrementa contador de referencia margarita  
  
del pizza1 # Decrementa el contador de referencia de margarita  
pizza2 = "Cuatro Estaciones" # Decrementa contador ref. margarita  
cena['lunes'] = pizza2 # Decrementa contador referencia margarita
```

El trabajo del recolector pasa totalmente desapercibido para un usuario. Actuará automáticamente cuando detecte instancias huérfanas, recuperando su espacio de memoria.

No obstante, una clase puede incluir el método `__del__()`, llamado destructor, que se invoca cuando una instancia está a punto de ser destruida con `del`, o si no ha sido destruida con esta sentencia, cuando el programa finalice su ejecución.

Este método especial puede ser utilizado para limpiar aquellos recursos de memoria no usados por una instancia.

Ejemplo:

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
class Robot:
    'Clase Robot'
    def __init__(self, nombre):
        self.nombre = nombre

    def saludo(self):
        print("Hola, mi nombre es", self.nombre)

    def __del__(self):
        print("Se han terminado mis baterias. Me voy a dormir.")

robot1 = Robot("C7PQ")
robot1.saludo() # Hola, mi nombre es C7PQ
print(id(robot1)) # Muestra el ID del objeto, Ejem.: 3070861004
del robot1 # Se han terminado mis baterias. Me voy a dormir.
```

Si intentamos mostrar de nuevo el ID del objeto, llamar al método saludo() o acceder al atributo nombre se producirá el siguiente error:

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
print(id(robot1))
robot1.saludo()
print(robot1.nombre)

NameError: name 'robot1' is not defined
```

Es recomendable definir las clases en un archivo separado. Más adelante, cuando se deseen utilizar las clases en un programa se importará el archivo con la sentencia import.

4. Herencia

La herencia es una de las características más importantes de la Programación Orientada a Objetos. Consiste en la posibilidad de crear una nueva clase a partir de una o más clases existentes, heredando de ellas sus atributos y métodos que podrán utilizarse como si estuvieran definidos en la clase hija.

Las clases derivadas se declaran como cualquier clase con la diferencia de incluir después de su nombre el nombre de la clase superior (entre paréntesis) de la que heredará sus características:

```
class NombreSubClase (NombreClaseSuperior):
    'Cadena de documentación opcional'
    Declaración de atributos y métodos...
```

Ejemplo:

Ejercicio de Inducción: Pruebe las siguientes líneas de código y observe los resultados:

```
class volumen:
    'Clase para controlar volumen de un reproductor multimedia'
    def __init__(self): # método constructor de objeto. Activa volumen
        self.nivel = 3 # sitúa el nivel de volumen en 3
```

```

    print('nivel', self.__class__.__name__, self.nivel)

def subir(self): # método para subir el nivel de volumen de 1 en 1
    self.nivel += 1
    if self.nivel > 10: # al intentar sobrepasar el nivel 10
        self.nivel = 10 # el nivel permanece en 10

    print('nivel', self.__class__.__name__, self.nivel)

def bajar(self): # método para bajar el nivel de 1 en 1
    self.nivel -= 1
    if self.nivel < 0: # al intentar bajar por debajo del nivel 0
        self.nivel = 0 # el nivel permanece en 0

    print('nivel', self.__class__.__name__, self.nivel)

def silenciar(self): # método para silenciar
    self.nivel = 0 # el nivel se sitúa en el 0
    print('nivel', self.__class__.__name__, self.nivel)

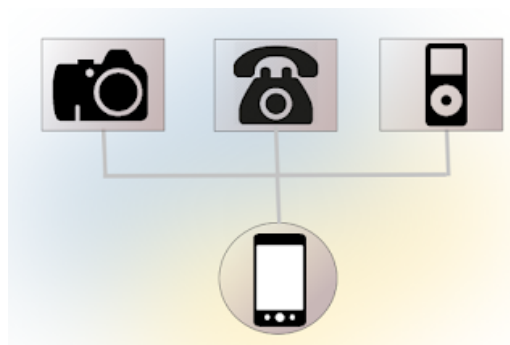
class graves(volumen): # crea clase graves a partir de clase volumen
    pass

ControlVolumen = volumen() # crea objeto y activa el volumen en 3
ControlVolumen.subir() # sube el volumen del nivel 3 al 4
ControlVolumen.bajar() # baja el volumen del nivel 4 al 3
ControlVolumen.silenciar() # silencia volumen bajando del 3 al 0
ControlGraves = graves() # crea objeto control graves, activa nivel 3
ControlGraves.subir() # sube el nivel de graves del 3 nivel al 4
del ControlVolumen # borra el objeto
del ControlGraves # borra el objeto

```

5. Herencia múltiple

La herencia múltiple se refiere a la posibilidad de crear una clase a partir de múltiples clases superiores. Es importante nombrar adecuadamente los atributos y los métodos en cada clase para no crear conflictos.



Ejemplo:

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```

class Telefono:
    "Clase teléfono"
    def __init__(self):

```

```

    pass
def telefonar(self):
    print('llamando')
def colgar(self):
    print('colgando')

class Camara:
    "Clase camara fotografica"
    def __init__(self):
        pass
    def fotografiar(self):
        print('fotografiando')

class Reproductor:
    "Clase Reproductor Mp3"
    def __init__(self):
        pass
    def reproducirmp3(self):
        print('reproduciendo mp3')
    def reproducirvideo(self):
        print('reproduciendo video')

class Movil(Telefono, Camara, Reproductor):
    def __del__(self):
        print('Móvil apagado')

movil1 = Movil()
print(dir(movil1))
movil1.reproducirmp3()
movil1.telefonar()
movil1.fotografiar()
del movil1

Salida:

['__class__', '__del__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'colgar', 'fotografiar', 'reproducirmp3', 'reproducirvideo', 'telefonar']

reproduciendo mp3
llamando
fotografiando
Móvil apagado

```

6. Funciones `issubclass()` y `isinstance()`

La función `issubclass(SubClase, ClaseSup)` se utiliza para comprobar si una clase (SubClase) es hija de otra superior (ClaseSup), devolviendo **True** o **False** según sea el caso.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
print(issubclass(Movil, Telefono)) # True
```

```
print(issubclass(Movil, Reproductor)) # True
```

La función booleana `isinstance(Objeto, Clase)` se utiliza para comprobar si un objeto pertenece a una clase o clase superior.

Ejercicio de Inducción: Pruebe las siguientes líneas de código y verifique los resultados presentados:

```
movil2 = Movil()
print(isinstance(movil2, Movil)) # True
print(isinstance(movil2, Camara)) # True
```
