Python Orientado a Objetos

Mario González
Facultad de Ingeniería y Ciencias Aplicadas
Universidad de las Américas, Quito, Ecuador

Agosto, 2021

Terminología de la POO I

A continuación una pequeña introducción a la terminología de la programación orientada a objetos (POO):

- **Clase:** Un prototipo definido por el usuario para un objeto, que define un conjunto de atributos que caracterizan a cualquier objeto de la clase.
- Los atributos son miembros de datos (variables de clase y variables de instancia) y métodos, a los cuales se accede a través de la notación de punto.
- **Variable de clase:** Una variable que es compartida por todas las instancias de una clase. Las variables de clase se definen dentro de una clase pero fuera de cualquiera de los métodos de la clase. Las variables de clase no se utilizan con tanta frecuencia como las variables de instancia.
- **Miembro de Datos** (Data member): Una variable variable o instancia de clase que posee los datos asociados a una clase y sus objetos.

Terminología de la POO II

- **Sobrecarga de funciones:** La asignación de más de un comportamiento a una función particular. La operación realizada varía según los tipos de objetos o argumentos involucrados.
- **Variable de instancia:** Una variable que se define dentro de un método y pertenece sólo a la instancia actual de una clase.
- **Herencia:** La transferencia de las características de una clase a otras clases que se derivan de ella.
- **Instancia:** Un objeto individual de una determinada clase. Por ejemplo, un objeto **obj** que pertenece a la clase **Circle**, es una instancia de la clase **Circle**.
- **Instanciación (Instantiation):** La creación de una instancia de una clase.

Terminología de la POO III

- **Método:** Un tipo especial de función que se define en una clase.
- **Objeto:** Una instancia única de una estructura de datos que se define por su clase. Un objeto comprende dos miembros de datos (variables de clase y variables de instancia) y métodos.
- **Sobrecarga de operadores:** La asignación de más de una función a un operador particular.

Creando clases

- La sentencia **`class`** crea una nueva definición de clase. El nombre de la clase sigue inmediatamente a la palabra clave **`class`** seguida de dos puntos, según se muestra a continuación:

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

- La clase tiene una cadena de documentación, que se puede acceder a través de `NombreClase.__doc__`.
- La suite de clase (`class_suite`) se compone de todas las sentencias de los componentes que definen miembros de la clase, atributos de datos y funciones.

Ejemplo I

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)
```

Ejemplo II

- La variable `empCount` es una variable de clase cuyo valor es compartido entre todas las instancias de esta clase. Es posible acceder a esta interna o externamente usando `Employee.empCount`.
- El primer método `__init__()` es un método especial, que se llama **constructor** de la clase o método de inicialización que Python llama cuando se crea una nueva instancia de esta clase.
- Otros métodos de la clase son declarados como funciones normales con la excepción de que el primer argumento de cada método es `self`. Python añade automáticamente el argumento `self`, no es necesario incluirlo cuando se llama a los métodos.

Creación de objetos de instancia

- Para crear instancias de una clase, se llama el nombre de la clase clase y se pasan los argumentos que el método `__init__()` acepta.

```
"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
```

Accediendo a los atributos de un objeto

- Para acceder a los atributos de un objeto utilizamos el operador punto junto con el objeto a cuyo atributo queremos acceder.
- Se accede a una variable de clase utilizando el nombre de clase de la siguiente manera:

```
emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

Ejemplo juntando los conceptos anteriores

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

Modificando atributos I

- Podemos agregar, eliminar o modificar los atributos de clases y objetos en cualquier momento:

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Modificando atributos II

- En lugar de utilizar las declaraciones normales para acceder a los atributos, podemos utilizar las siguientes funciones:
  - **`getattr(obj, name[, default])`**: para acceder al atributo del objeto.
  - **`hasattr(obj, name)`**: para verificar si existe un atributo o no.
  - **`setattr (obj, name, value)`**: para establecer el valor de un atributo. Si el atributo no existe, entonces se crearía.
  - **`El delattr (obj, name)`**: para eliminar un atributo.

```
hasattr(emp1, 'age')     # Returns true if 'age' attribute exis
getattr(emp1, 'age')     # Returns value of 'age' attribute
setattr(emp1, 'age', 8)  # Set attribute 'age' at 8
delattr(empl, 'age')     # Delete attribute 'age'
```

Built-In Class Attributes I

- Cada clase de Python contiend los siguientes atributos predefinidos a los que se puede acceder mediante operador punto como cualquier otro atributo:
- `__dict__`: Diccionario que contiene el espacio de nombres de la clase.
- `__doc__`: cadena de documentación de clase o None, si indefinido.
- `__name__`: Nombre de clase.
- `__module__`: Nombre del módulo en el que se define la clase. Este atributo es `__main__` en modo interactivo.
- `__bases__`: Una tupla posiblemente vacía que contiene las clases base, en el orden de su aparición en la lista de la clases base.

Built-In Class Attributes II

```
print("Employee.__doc__:", Employee.__doc__)
print("Employee.__name__:", Employee.__name__)
print("Employee.__module__:", Employee.__module__)
print("Employee.__bases__:", Employee.__bases__)
print("Employee.__dict__:", Employee.__dict__)
```

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

Destruyendo objetos (Garbage collection) I

- Python elimina objetos cuando ya no son necesarios (built-in types o instancias de clase) automáticamente para liberar espacio en memoria.
- El proceso por el cual Python reclama periódicamente bloques de memoria que ya no están en uso se denomina Recolección de Basura (Garbage collection).
- El Recolector de basura de Python corre durante la ejecución del programa y se activa cuando el contador de referencia de un objeto llega a cero.
- El contador de referencias de un objeto cambia a medida que el número de alias que apuntan a este cambie.
- El contador de referencias a un objeto aumenta cuando se le asigna un nuevo nombre o es colocado en un recipiente (lista, tupla o diccionario).

Destruyendo objetos (Garbage collection) II

- El contador de referencias del objeto disminuye cuando se elimina con `del`, su referencia es reasignada, o su referencia sale de ámbito (scope).
- Cuando el contador de referencia de un objeto llega a cero, Python lo recolecta automáticamente.

```
a = 40      # Create object <40>
b = a       # Increase ref. count  of <40>
c = [b]     # Increase ref. count  of <40>

del a       # Decrease ref. count  of <40>
b = 100     # Decrease ref. count  of <40>
c[0] = -1   # Decrease ref. count  of <40>
```

- Normalmente no se dará cuenta de que el recolector de basura destruye una instancia huérfana y reclama su espacio.
- Sin embargo, una clase puede implementar el método especial `__del__()`, llamado destructor, que se invoca cuando la instancia está a punto de ser destruida.

Destruyendo objetos (Garbage collection) III

- Este método puede ser utilizado para limpiar los recursos no de memoria utilizados por una instancia.
- Ejemplo:

```
class Point:
   def __init( self, x=0, y=0):
      self.x = x
      self.y = y
   def __del__(self):
      class_name = self.__class__.__name__
      print(class_name, "destroyed")

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

Herencia de clases I

- En lugar de comenzar desde cero, podemos crear una clase derivándola de una clase preexistente listando la clase padre entre paréntesis después del nuevo nombre de la clase.
- La clase hijo hereda los atributos de su clase padre, y puede utilizar esos atributos como si se hubieran definido en la clase hijo.
- Una clase de niño también puede sobre-escribir los miembros de datos y métodos de los padres.
- Las clases derivadas se declaran muy parecido a su clase padre; sin embargo, se da una lista de clases base a heredar de después de el nombre de la clase:

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

- Ejemplo:

Herencia de clases II

```
class Parent:          # define parent class
   parentAttr = 100
   def __init__(self):
      print("Calling parent constructor")

   def parentMethod(self):
      print('Calling parent method')

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print("Parent attribute :", Parent.parentAttr)

class Child(Parent): # define child class
   def __init__(self):
      print("Calling child constructor")

   def childMethod(self):
      print('Calling child method')
```

Herencia de clases III

```
c = Child()             # instance of child
c.childMethod()         # child calls its method
c.parentMethod()        # calls parent's method
c.setAttr(200)          # again call parent's method
c.getAttr()             # again call parent's method

Resultado:

Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Herencia de clases IV

- De manera similar, se puede derivar una clase de múltiples clases padres de la siguiente manera:

```
class A:          # define your class A
.....

class B:           # define your calss B
.....

class C(A, B):    # subclass of A and B
.....
```

Herencia de clases V

- Podemos utilizar las funciones `issubclass()` o `isinstance()` para comprobar las relaciones entre clases e instancias.
- La función booleana `issubclass(sub, sup)` devuelve `True` si la subclase `sub` es una subclase de la superclase `sup`.
- `isinstance(obj, Class)` es una función booleana que devuelve True si `obj` es una instancia de la clase `Class` o es una instancia de una subclase de `Class`.

Sobre-escritura de métodos

- Siempre se puede sobre-escribir los métodos de la clase padre. Una de las razones para sobre-escribir los métodos padres se debe a que es posible que se desee una funcionalidad especial o diferente en una subclase:

```
class Parent:          # define parent class
   def myMethod(self):
      print('Calling parent method')

class Child(Parent): # define child class
   def myMethod(self):
      print('Calling child method')

c = Child()            # instance of child
c.myMethod()           # child calls overridden method

Resultado:

Calling child method
```

Sobre-escritura de métodos base

- La siguiente tabla enumera algunas funcionalidades genéricas que se pueden sobre-escribir en las clases propias

1	__init__(self [,args...]) Constructor (con argumentos opcionales) Ejemplo: obj = className(args)
2	__del__(self) Destructor, elimina (del) un objeto Ejemplo: del obj
3	__repr__(self) Representación string evaluable Ejemplo: repr(obj)
4	__str__(self) Representación string para impresión Ejemplo: str(obj)
5	__cmp__(self, x) Comparación de objetos Ejemplo: cmp(obj, x)

Sobrecarga de operadores I

- Supongamos que hemos creado una clase Vector para representar vectores en dos dimensiones, lo que sucede cuando se utiliza el operador más para sumarlos, lo más probable es que Python devuelva un error.
- Podemos definir el método `__add__` en nuestra clase para llevar a cabo la suma de vectores y luego el operador más le comportará como se espera.

Sobrecarga de operadores II

- Ejemplo:

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self,other):
        return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print(v1 + v2)
```

```
Resultado:

Vector(7,8)
```

Sobrecarga de operadores III

- Operadores que se pueden sobrecargar

Operator	Method	Expression
+ Addition	`__add__(self, other)`	`a1 + a2`
- Subtraction	`__sub__(self, other)`	`a1 - a2`
* Multiplication	`__mul__(self, other)`	`a1 * a2`
@ Matrix Multiplication	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
/ Division	`__div__(self, other)`	`a1 / a2` (*Python 2 only*)
/ Division	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
// Floor Division	`__floordiv__(self, other)`	`a1 // a2`
% Modulo/Remainder	`__mod__(self, other)`	`a1 % a2`
** Power	`__pow__(self, other[, modulo])`	`a1 ** a2`
<< Bitwise Left Shift	`__lshift__(self, other)`	`a1 << a2`
>> Bitwise Right Shift	`__rshift__(self, other)`	`a1 >> a2`
& Bitwise AND	`__and__(self, other)`	`a1 & a2`
^ Bitwise XOR	`__xor__(self, other)`	`a1 ^ a2`
| (Bitwise OR)	`__or__(self, other)`	`a1 | a2`
- Negation (Arithmetic)	`__neg__(self)`	`-a1`
+ Positive	`__pos__(self)`	`+a1`
~ Bitwise NOT	`__invert__(self)`	`~a1`
< Less than	`__lt__(self, other)`	`a1 < a2`
<= Less than or Equal to	`__le__(self, other)`	`a1 <= a2`
== Equal to	`__eq__(self, other)`	`a1 == a2`
!= Not Equal to	`__ne__(self, other)`	`a1 != a2`
> Greater than	`__gt__(self, other)`	`a1 > a2`
>= Greater than or Equal to	`__ge__(self, other)`	`a1 >= a2`
`[index]` Index operator	`__getitem__(self, index)`	`a1[index]`
in In operator	`__contains__(self, other)`	`a2 in a1`
`(*args, ...)` Calling	`__call__(self, *args, **kwargs)`	`a1(*args, **kwargs)`

Sobrecarga de operadores IV

- Funciones que se pueden sobrecargar

Function	Method	Expression
Casting to `int`	`__int__(self)`	`int(a1)`
Absolute function	`__abs__(self)`	`abs(a1)`
Casting to `str`	`__str__(self)`	`str(a1)`
Casting to `unicode`	`__unicode__(self)`	`unicode(a1)` (Python 2 only)
String representation	`__repr__(self)`	`repr(a1)`
Casting to `bool`	`__nonzero__(self)`	`bool(a1)`
String formatting	`__format__(self, formatstr)`	`"Hi {:abc}".format(a1)`
Hashing	`__hash__(self)`	`hash(a1)`
Length	`__len__(self)`	`len(a1)`
Reversed	`__reversed__(self)`	`reversed(a1)`
Floor	`__floor__(self)`	`math.floor(a1)`
Ceiling	`__ceil__(self)`	`math.ceil(a1)`

There are also the special methods `__enter__` and `__exit__` for context managers, and many more.

Ocultación de datos I

- Los atributos de un objeto pueden o no ser visibles fuera de la definición de clase. Es necesario nombrar los atributos con un prefijo doble subrayado, y estos atributos no serán directamente visibles fuera de la clase.
- Ejemplo:

```
class JustCounter:
    __secretCount = 0

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

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

Ocultación de datos II

- Cuando el código anterior es ejecutado, produce el siguiente resultado:

```
1
2
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print(counter.__secretCount)
AttributeError: JustCounter instance has no attribute '__secretCount'
```

- Python protege los miembros cambiando internamente el nombre de modo que debe incluir el nombre de la clase.

  Se puede acceder a este tipo de atributos de la manera siguiente `object._className__attrName`.

Ocultación de datos III

- Si reemplazamos la última línea de la siguiente manera, entonces el código funcionaría:

```
.........................
print(counter._JustCounter__secretCount)

1
2
2
```

Polimorfismo I

- El operador suma es polimórfico:

```
num1 = 1
num2 = 2
print(num1+num2)

str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

- La funcion `len()` es polimórfica:

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Polimorfismo II

- Polimorfismo es la capacidad de una entidad de referenciar en tiempo de ejecución a instancias de diferentes clases.

```
class Perro:
    def sonido(self):
        print('Guauuuuu!!!')

class Gato:
    def sonido(self):
        print('Miaaauuuu!!!')

class Vaca:
    def sonido(self):
        print('Muuuuuuuu!!!')

def a_cantar(animales):
    for animal in animales:
        animal.sonido()

perro = Perro(); gato = Gato(); gato_2 = Gato(); vaca = Vaca(); perro_
granja = [perro, gato, vaca, gato_2, perro_2]
a_cantar(granja)
```

Polimorfismo III

Ver ejemplos adicionales.