Programación orientada a objetos

Hay tres paradigmas de programación ampliamente utilizados. Programación por procedimientos, programación funcional y programación orientada a objetos. Python admite tanto la programación procedural y la orientada a objetos y algo con algo limitado la programación funcional también.

La programación orientada a objetos (POO) es un paradigma de programación que utiliza objetos y sus interacciones para diseñar aplicaciones y programas.

Hay unos conceptos básicos en P00:

- Abstracción
- Polimorfismo
- Encapsulación
- Herencia

La abstracción es la simplificación de la compleja realidad modelando clases apropiadas para un problema. El polimorfismo es el proceso de utilización de un operador o funciones de diferentes maneras para diferentes entrada de datos. La encapsulación oculta los detalles de implementación de una clase de otros objetos. La herencia es una manera de formar nuevas clases utilizando las clases que ya han sido definidas.

1. Objetos

Todo en Python es un objeto. Los objetos son elementos básicos de un programa POO de Python.

```
#!/usr/bin/python

import sys

def function(): pass

print type(1)
print type("")
print type([])
print type({})
print type(())
print type(object)
print type(function)
print type(sys)
```

En este ejemplo se muestra que todas estas entidades son objetos de datos. La función `type()` devuelve el tipo de objeto especificado.

```
%edit class1.py
Editing... done. Executing edited code...
<type 'int'>
<type 'str'>
<type 'list'>
```

```
<type 'dict'>
<type 'tuple'>
<type 'type'>
<type 'function'>
<type 'module'>
```

2. Clases

El usuario, puede crear sus propios objetos usando la palabra reservada class. La clase es un modelo que define la naturaleza de un objeto. Desde las clases construimos instancias. Una instancia es un objeto específico creado a partir de una clase particular.

```
#!/usr/bin/python

class A1:
   pass

a1 = A1()

print type(a1)
print type(A1)
```

Esta es nuestra primera clase. El cuerpo de la clase se deja en blanco por ahora. Es una convención dar a las clases un nombre que comienza con una letra mayúscula.

```
a1 = A1()
```

Aquí hemos creamos una nueva instancia de la clase `A1`. O en otras palabras, hemos instanciado la clase `A1`. `a1` es una referencia a nuestro nuevo objeto.

```
%edit class2.py
Editing... done. Executing edited code...
<type 'instance'>
<type 'classobj'>
```

Aquí vemos que `a1` es una instancia de objeto y `A1` es un objeto clase.

Dentro de una clase, podemos definir atributos y métodos. Un atributo es una característica de un objeto. Esto puede ser, por ejemplo, un salario de un empleado o la edad de una persona. Un método define las operaciones que podemos realizar con nuestros objetos. Un mtodo podra definir una cancelación de una cuenta. Técnicamente, los atributos son variables y los métodos son funciones definidas dentro de una clase.

2. Atributos

Los atributos son características de un objeto. Un especial método llamado __init__() es usado para inicializar los atributos de un objeto.

```
#!/usr/bin/python

class Cat:
```

```
   def __init__(self, name):
      self.name = name

cat1 = Cat('Sazy')
cat2 = Cat('Kaperuz')

print cat1.name
print cat2.name
```

En este ejemplo tenemos una clase Cat. El método especial __init__() se llama de forma autom"atica justo después de que el objeto ha sido creado.

```
def __init__(self, name):
```

Cada método en una definición de clase comienza con una referencia a la instancia objeto. Es por convención llamada **self**. Name es el argumento. El valor se transmite a través de la creación de instancias de clase.

```
self.name
```

Aquí pasamos un atributo a la instancia objeto.

```
cat1  = Cat('Sazy')
cat2 = Cat ('Kaperuz')
```

Aquí hemos creamos una instancia dos objetos: Mis gatos Sazy y Kaperuz. El número de argumentos debe coincidir con el método __init__() de la definición de clase. 'Sazy' y 'Kaperuz' se convierten en el parámetro `name` del método __init__().

```
print cat1.name
print cat2.name
```

Aquí imprimimos las variables de instancia de dos objetos Cat. Cada instancia de una clase puede tener sus propios atributos.

```
%edit class3.py
Editing... done. Executing edited code...
Sazy
Kaperuz
```

Los atributos se pueden asignar de forma dinámica, no sólo durante la inicialización. Esto lo demuestra el siguiente ejemplo.

```
#!/usr/bin/python

class D:
   pass

d = D()
d.name = "D"
print d.name
```

Definimos y creamos una clase D vacía.

```
d.name = "D"
```

Esta línea de código crea un nuevo atributo name.

```
%edit class4.py
Editing... done. Executing edited code...
D
```

Hasta ahora, hemos estado hablando acerca de los atributos de instancia. En Python también hay atributos llamados **objetos de clase** . Los atributos de los objetos de clase son los mismos para todas las instancias de una clase.

```
#!/usr/bin/python

class Cat:
   species = 'Miau'

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

cat1 = Cat('Sazy', 3)
cat2 = Cat('Kaperuz', 5)

print cat1.name, cat1.age
print cat1.name, cat2.age

print Cat.specie
print cat1.__class__.specie
print cat2.__class__.specie
```

Aquí tenemos dos gatitos con atributos name y age específicos. Ambos gatitos comparten algunas características. Sazy y Kaperuz son ambos de la especie Miau. Esto se refleja en el atributos de nivel clase specie. El atributo se define fuera de cualquier método en el cuerpo de una clase.

```
print Cat.specie
print cat1.__class__.specie
```

Hay dos maneras, como podemos acceder a los atributos de los objetos de la clase. Ya sea a través del nombre de la clase Cat, o con la ayuda del atributo especial __class__().

```
 %edit class5.py
Editing... done. Executing edited code...
Sazy 3
Kaperuz 5
Miau
Miau
Miau
```

3. Métodos

Los métodos son funciones definidas en el interior del cuerpo de una clase. Se utilizan para realizar operaciones con los atributos de nuestros objetos. Los métodos son esenciales en el concepto de **encapsulación** para POO.

Por ejemplo, podríamos tener un método connect () en nuestra clase AccessDatabase. Nosotros no tenemos que estar informados, cómo exactamente el método se conecta a la base de datos. Lo único que sabemos, es que se utiliza para conectarse a una base de datos. Esto es esencial en la división de responsabilidades en la programación. Especialmente en aplicaciones grandes.

```
#!/usr/bin/python

class Circle:
   pi = 3.141592

   def __init__(self, radius=1):
      self.radius = radius

   def area(self):
      return self.radius * self.radius * Circle.pi

   def sRadius(self, radius):
      self.radius = radius

   def gRadius(self):
      return self.radius

c = Circle()

c.sRadius(5)
print c.gRadius()
print c.area()
```

En el ejemplo anterior tenemos una clase Circle. Hemos definido tres métodos

```
def area(self):
    return self.radius * self.radius * Circle.pi
```

El método area() retorna el área de un círculo

```
def sRadius(self, radius):
    self.radius = radius
```

El método sRadius coloca un nuevo valor al atributo radius.

```
 def gRadius(self):
    return self.radius
```

El método gRadius retorna el actual radius.

```
c.setRadius(5)
```

El método es llamado sobre una instancia de objeto. El objeto c está emparejado con el parámetro self de la definición de clase. El número 5 está emparejado con el parámetro de radius.

```
%edit class6.py
Editing... done. Executing edited code...
5
78.5398
```

En Python, podemos llamar a los métodos de dos maneras.

```
#!/usr/bin/python

class M:
   def __init__(self):
      self.name = 'M'

   def gName(self):
      return self.name

m = M()

print m.gName()
print M.gName(m)
```

En este ejemplo, demostramos ambas llamadas a métodos.

```
print m.gName()
```

Este es el llamado método acotado (bounded). El intérprete de Python automáticamente empareja la instancia m con el parámetro self.

```
print M.gName(m)
```

Y esta es la llamada al método no acotado (unbounded). El objeto de instancia se da de forma explícita al método gName().

```
%edit class7.py
Editing... done. Executing edited code...
M
M
```

4. Herencia

La herencia es una manera de formar nuevas clases utilizando clases que ya se han definido. Las clases recién formadas se llaman **clases derivadas** , las clases de donde se deriva se llaman **clases base**. Beneficios importantes de la herencia son la reutilización del cdigo y la reducción de la complejidad de un programa. Las clases derivadas (descendientes) anulan o amplían la funcionalidad de las clases base (ancestros).

```
#!/usr/bin/python

class Animal:
   def __init__(self):
      print "Animalito creado"

   def whoAmI(self):
      print "Animalito!"

   def comer(self):
      print "Comer"


class Perro(Animal):
```

```
    def __init__(self):
        Animal.__init__(self)
        print "Perrito creado"

    def whoAmI(self):
        print "Perro"

    def ladrido(self):
        print "Woof!"


d = Perro()
d.whoAmI()
d.comer()
d.ladrido()
```

En este ejemplo, tenemos dos clases. Animal y Perro. Animal es la clase base, Perro es la clase derivada. La clase derivada hereda la funcionalidad de la clase base. Esto se demuestra por el método comer(). La clase derivada modifica el comportamiento existente de la clase base, como se muestra por el método whoAmI(). La clase derivada amplía la funcionalidad de la clase base, mediante la definición de un nuevo método ladrido() .

```
class Perro(Animal):
    def __init__(self):
        Animal.__init__(self)
        print "Perrito creado"
```

Ponemos la clase base entre paréntesis después del nombre de la clase derivada. Si la clase derivada proporciona su propio método __init__(), debemos llamar explícitamente al método __init__() de la clase base.

```
 %edit class8.py
Editing... done. Executing edited code...
Animalito creado
Perrito creado
Perro
Comer
Woof
```

5. Polimorfismo

El polimorfismo es el proceso de utilización de un operador o función de diferentes maneras para diferentes de entrada de datos. El polimorfismo significa que si la clase B hereda de la clase A, esta no tiene que heredar todo lo relacionado con la clase A, sino que puede hacer algunas de las cosas que la clase A hace de manera diferente.

```
#!/usr/bin/python

a = "python"
b = ( 2, 4, 6)
c = ['j', 'a', 'v',  'a']

print a[2]
print b[1]
print c[1]
```

Python usa mucho el polimorfismo en tipos integrados. Aquí estamos usamos el mismo operador de indexado para tres diferentes tipos de datos.

```
%edit class9.py
Editing... done. Executing edited code...
t
4
a
```

Polimorfismo es usado cuando tratamos con herencia

```
#!/usr/bin/python

class Animal:
   def __init__(self, name=''):
      self.name = name

   def hablar(self):
      pass


class Cat(Animal):
   def hablar(self):
      print "Meow"


class Perro(Animal):
   def hablar(self):
      print "Woof"


a = Animal()
a.hablar()

c = Cat("Sazy")
c.hablar()

d = Perro("R")
d.hablar()
```

Aquí tenemos dos especies Un perro y un gato. Ambos son animales. La clase Perro y la clase Cat heredan de la clase Animal. Ellos tienen el método hablar(), el cúal produce diferentes salidas.

```
%edit class10.py
Editing... done. Executing edited code...
Meow
Woof
```

6. Métodos especiales

Las clases en Python puede ejecutar determinadas operaciones con nombres especiales de método. Estos métodos no son llamados directamente, sino mediante una sintaxis específica. Esto es similar a lo que se conoce como la sobrecarga de operadores en C++ o en Ruby.

```
#!/usr/bin/python

class Libro:
   def __init__(self, titulo, autor, paginas):
```

```
        print "Un libro va ser creado"
        self.titulo = titulo
        self.autor = autor
        self.paginas = pagina

    def __str__(self):
        return "Titulo: %s ,autor:%s, paginas:%s " %\
                (self.titulo, self.autor, self.paginas)

    def __len__(self):
        return self.paginas

    def __del__(self):
        print "Un libro sera vendido"


libro = Libro("Introduction Process Stochastic", "Fima Klebaner", 400)

print libro
print len(libro)
del libro
```

En nuestro ejemplo, tenemos una clase Libro. Aquí hemos introducido cuatro especiales métodos. Los métodos __init__(), __str__(), __len__() y __del__().

```
libro = Libro("Introduction Process Stochastic", "Fima Klebaner", 400)
```

Aquí hemos llamado al método __init__(). El método crea una nueva instancia de la clase Libro.

```
print libro
```

La palabra integrada print llama al método __str__(). Este método debería retornar una informal cadena de representación del objeto.

```
print len(libro)
```

La función len(), invoca al método __len__(). En nuestro caso imprimimos el número de hojas del libro.

```
del libro
```

La palabra clave 'elimina' un objeto. Llama al método __del__().

Veamos como podemos implementar una clase Vector y las operaciones de adición y substracción.

```
#!/usr/bin/python

class Vector:

  def __init__(self, data):
    self.data = data

  def __str__(self):
    return repr(self.data)
```

```
  def __add__(self, otro):
    data = []
    for j in range(len(self.data)):
      data.append(self.data[j] + otro.data[j])
    return Vector(data)

  def __sub__(self, otro):
    data = []
    for j in range(len(self.data)):
      data.append(self.data[j] - otro.data[j])
    return Vector(data)


x = Vector([1, 4, 8])
y = Vector([3, 0, 2])
print x + y
print y - x
```

```
def __add__(self, otro):
    data = []
    for j in range(len(self.data)):
      data.append(self.data[j] + otro.data[j])
    return Vector(data)
```

Aquí implementamos la adición de vectores. El método __add__() es llamado, cuando agregamos dos vectores con el operador +. Agregamos cada miembro de los respectivos vectores.

```
%edit class12.py
Editing... done. Executing edited code...
[4, 4, 10]
[2, -4, -6]
```