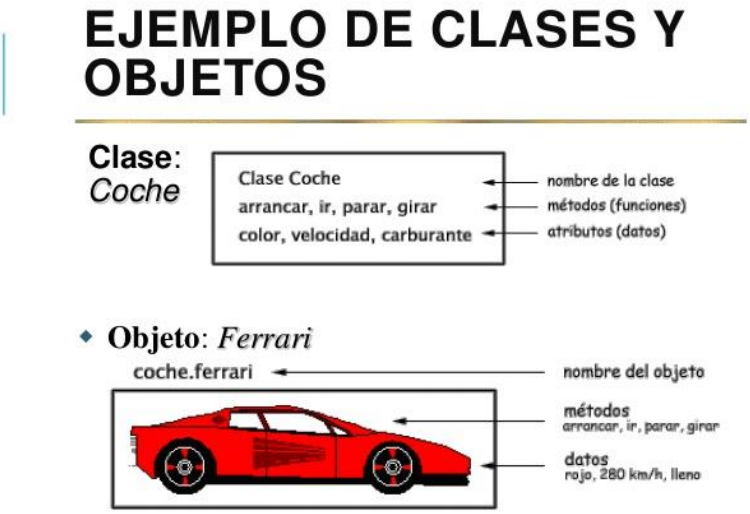
Python
Programación Orientada a Objetos:

La Programación Orientada a Objetos (POO u OOP por sus siglas en inglés), es un paradigma de programación el cual se basa en las interacciones de objetos para resolver las necesidades de un sistema informático.



En Python las clases se definen mediante la palabra clave class seguida del nombre de la clase, dos puntos (:) y a continuación, indentado, el cuerpo de la clase. Como en el caso de las funciones, si la primera línea del cuerpo se trata de una cadena de texto, esta será la cadena de documentación de la clase o docstring.

```
class Coche:
    """Abstraccion de los objetos coche."""
    def __init__(self, gasolina=”0”):
        self.gasolina = gasolina

    def arrancar(self):
        if self.gasolina > 0:
            print "Arranca"
        else:
            print "No arranca..."

    def conducir(self):
        if self.gasolina > 0:
            self.gasolina -= 1
            print "Quedan", self.gasolina, "litros"
        else:
            print "No se mueve..."
```

Lo primero que llama la atención en el ejemplo anterior es el nombre tan curioso que tiene el método **__init__,** este nombre es una convención. El método __init__, con una doble barra baja al principio y final del nombre, se ejecuta justo después de crear un nuevo objeto a partir de la clase, proceso que se conoce con el nombre de instanciación.

Como vemos el primer parámetro de __init__ y del resto de métodos de la clase es siempre self. Esto sirve para referirse al objeto actual, Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto.

Si volvemos al método __init__ de nuestra clase Coche veremos cómo se utiliza self para asignar al atributo gasolina del objeto (self.gasolina) el valor que el programador especificó para el parámetro gasolina. El parámetro gasolina se destruye al final de la función, mientras que el atributo gasolina se conserva (y puede ser accedido) mientras el objeto viva.

Creando un nuevo objeto:

```
ferrari = Coche(100)  //con gasolina igual a 100
honda = Coche()       //con gasolina igual a 0
```

Accediendo a las funciones o metodos:

```
ferrari.conducir()
> Quedan 99 litros // salida
```

Hay tres conceptos que son básicos para cualquier lenguaje de programación orientado a objetos: el encapsulamiento, la herencia y el polimorfismo.

Herencia:
En un lenguaje orientado a objetos cuando hacemos que una clase (subclase) herede de otra clase (superclase) estamos haciendo que la subclase contenga todos los atributos y métodos que tenía la superclase.

```
class Instrumento:
    def __init__(self, precio):
        self.precio = precio

    def tocar(self):
        print "Estamos tocando musica"

    def romper(self):
        print "Eso lo pagas tu"
        print "Son", self.precio, "$$$"

class Bateria(Instrumento): // Hereda de instrumento
    pass

class Guitarra(Instrumento): // Hereda de instrumento
    pass
```

Python a diferencia de otros lenguajes de programación permite la herencia multiple, es decir una clase puede heredar de varias a la misma vez.
En el caso de que alguna de las clases padre tuvieran métodos con el mismo nombre y número de parámetros las clases sobreescribirían la implementación de los métodos de las clases más a su derecha en la definición.

Polimorfismo:
Python, al ser de tipado dinámico no impone restricciones a los tipos que se le pueden pasar a una función. Por ese motivo, a diferencia de lenguajes de tipado estático como Java o C++, el polimorfismo en Python no es de gran importancia.
En ocasiones también se utiliza el término polimorfismo para referirse a la sobrecarga de métodos, término que se define como la capacidad del lenguaje de determinar qué método ejecutar de entre varios métodos con igual nombre según el tipo o número de los parámetros que se le pasa. En Python no existe sobrecarga de métodos.

Encapsulamiento:
La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase.

En Python no existen los modificadores de acceso, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre: si el nombre comienza con dos guiones bajos (y no termina también con dos guiones bajos) se trata de una variable o función privada, en caso contrario es pública.

```
class Ejemplo:
    def publico(self):
        print "Publico"

    def __privado(self):
        print "Privado"

ej = Ejemplo()
ej.publico()
ej.__privado()
```

Al intentar llamar al método __privado Python lanzará una excepción quejándose de que no existe, sin embargo lo que realmente sucede es que cuando los nombres inician con dos guiones bajos se renombra para incluir el nombre de la clase. Esto implica que el método o atributo no es realmente privado.

```
ej._Ejemplo__privado()
```

MÉTODOS ESPECIALES

Ya vimos al principio del artículo el uso del método __init__. Existen otros métodos con significados especiales, cuyos nombres siempre comienzan y terminan con dos guiones bajos. A continuación se listan algunos especialmente útiles.

__init__(self, args)
Método llamado después de crear el objeto para realizar tareas de inicialización.

__new__(cls, args)
Método exclusivo de las clases de nuevo estilo que se ejecuta antes que __init__ y que se encarga de construir y devolver el objeto en sí. Es equivalente a los constructores de C++ o Java. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es self, sino la propia clase: cls.

__del__(self)
Método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.

__str__(self)
Método llamado para crear una cadena de texto que represente a nuestro objeto. Se utiliza cuando usamos print para mostrar nuestro objeto o cuando usamos la función str(obj) para crear una cadena a partir de nuestro objeto.

__cmp__(self, otro)
Método llamado cuando se utilizan los operadores de comparación para comprobar si nuestro objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores <, <=, > o >= se lanzará una excepción. Si se utilizan los operadores == o != para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).

__len__(self)
Método llamado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función len(obj) sobre nuestro objeto. Como es de suponer, el método debe devolver el número la longitud del objeto.

Existen más métodos especiales que brindan funcionalidad extra a nuestros objetos.