

# Paradigma

Un **paradigma** es una metodología de trabajo. En programación, se trata de un **enfoque concreto de desarrollar y estructurar el desarrollo de programas.**



El lenguaje Python tiene la característica de permitir programar con las siguientes metodologías:

**Programación Lineal:** Es cuando desarrollamos todo el código sin emplear funciones. El código es una secuencia lineal de comando.

**Programación Estructurada:** Es cuando planteamos funciones que agrupan actividades a desarrollar y luego dentro del programa llamamos a dichas funciones que pueden estar dentro del mismo archivo (módulo) o en una librería separada.

**Programación Orientada a Objetos:** Es cuando planteamos clases y definimos objetos de las mismas

# Paradigma Orientado a Objetos

La **clase** es un modelo o prototipo que define las **variables** y **métodos** comunes a todos los objetos de cierta clase.

También se puede decir que una **clase** es una plantilla genérica para un conjunto de **objetos** de similares características.

Por otro lado, una instancia de una **clase** es otra forma de llamar a un **objeto**. En realidad no existe diferencia entre un objeto y una instancia. Sólo que el objeto es un término más general, pero los objetos y las instancias son ambas representación de una **clase**.

Una instancia es un objeto de una clase en particular.

La programación orientada a objetos se basa en la definición de **clases** a diferencia de la programación estructurada, que está centrada en las **funciones**.

Una **clase** es una plantilla (molde), que define **atributos** (lo que conocemos como variables) y **métodos** (lo que conocemos como funciones).

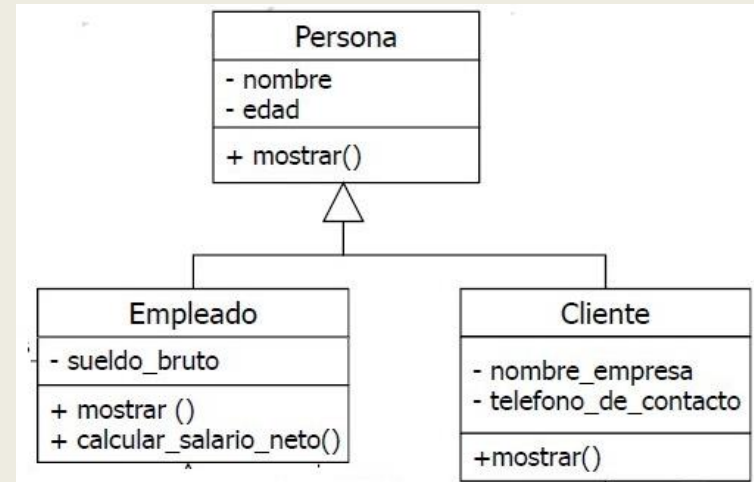
### Clase Vehículo

- Número de Ruedas
- Tipo de Motor
- Numero de Velocidades de la Caja de Cambios
- Color

*funciones arrancar, frenar, acelerar, etc.*

### Objeto Auto BMW

- 4 Ruedas Micheline
- Motor BMW
- Caja de cambios de 7 Velocidades
- Color Azul



Debemos declarar una clase antes de poder crear objetos (instancias) de esa clase. Al crear un objeto de una clase, se dice que se crea una instancia de la clase o un objeto propiamente dicho

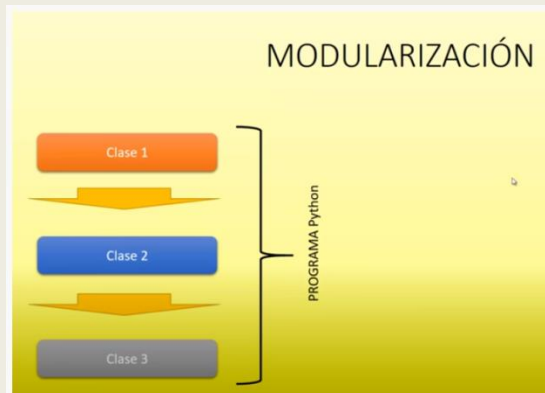


Cada clase internamente está encapsulada. Y se comunican por medio de métodos de acceso.

Python utiliza la llamada ***Nomenclatura del punto*** para hacer referencia a las propiedades y procedimientos del Objeto

*NombreObjeto. Propiedad*

*NombreObjeto. Procedimiento*

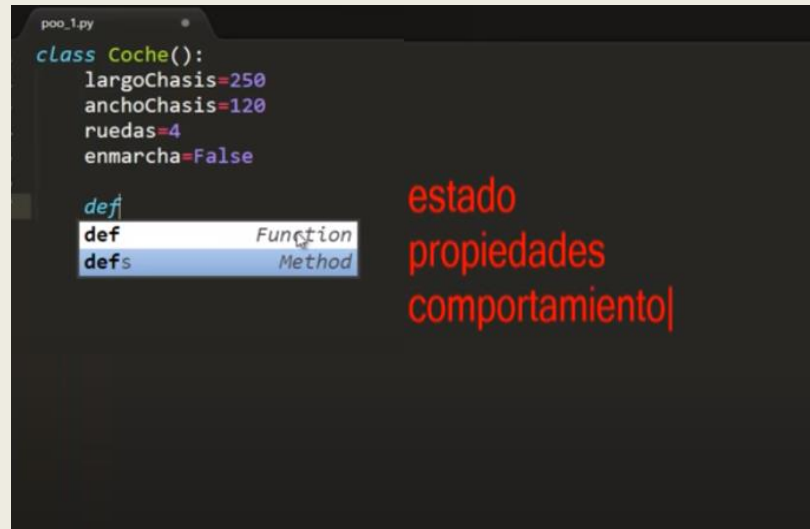


# Class

Para definir una clase vamos a utilizar la palabra clave ***class***

Luego el nombre de la clase seguido de (): y a continuación definimos los atributos de la clase

Para definir el comportamiento(métodos) utilizamos ***def*** en este caso el editor (SublimeText/Visual Code) nos pregunta si el def se refiere a una Función o a un método de una clase.



```
poo_1.py
class Coche():
    largoChasis=250
    anchoChasis=120
    ruedas=4
    enmarcha=False

    def
```

def Function  
defs Method

estado  
propiedades  
comportamiento|

Si elegimos *Method* nos pondrá automáticamente el (*self*)



```
1 class Coche():
2     largoChasis=250
3     anchoChasis=120
4     ruedas=4
5     enmarcha=False
6
7     def function(self):
8         pass
```

objeto perteneciente a la clase

Annotations: Red arrows point from the text 'objeto perteneciente a la clase' to the `self` parameter in the `function` method and to the `Coche` class name in the `class` statement.

*Self* es similar al *this* de C++ y Java pero en estos lenguajes estaba implícito, es decir, no hacía falta ponerlo, en cambio en Python, **hay que incluirlo**



```
1 class Coche():
2     largoChasis=250
3     anchoChasis=120
4     ruedas=4
5     enmarcha=False
6
7     def arrancar(self):
8         pass
9
10
11 miCoche=Coche()
```

instanciar una clase

~~new~~

Annotations: A red arrow points from the text 'instanciar una clase' to the `Coche()` call in line 11. The word 'new' is written in red and crossed out with a red 'X' below the code.

Aquí estaríamos definiendo una objeto o instancia de una clase.

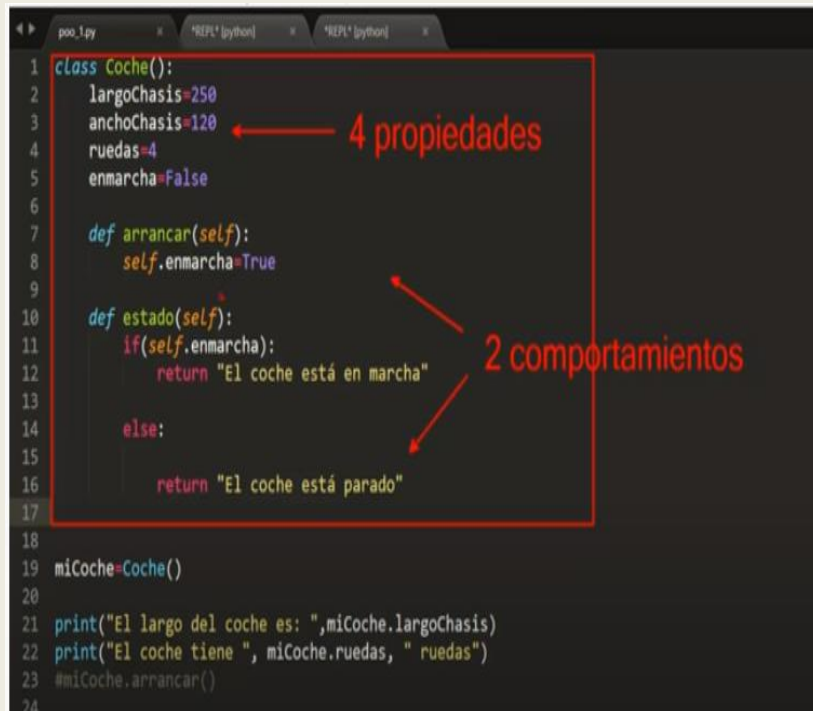
**No es necesario poner new como en otros lenguajes.**

En este ejemplo vemos como utilizamos la nomenclatura del punto para hacer referencia a los atributos y asignar valores. También para el caso de una condición *if*

```
poo_1.py
1 class Coche():
2     largoChasis=250
3     anchoChasis=120
4     ruedas=4
5     enmarcha=False
6
7     def arrancar(self):
8         pass
9
10
11 miCoche=Coche()
12
13 print(miCoche.largoChasis)
```

```
poo_1.py
1 class Coche():
2     largoChasis=250
3     anchoChasis=120
4     ruedas=4
5     enmarcha=False
6
7     def arrancar(self):
8         self.enmarcha=True
9
10    def estado(self):
11        if(self.enmarcha):
12            return "El coche está en marcha"
13
14        else:
15
16            return "El coche está parado"
17
18
19 miCoche=Coche()
20
21 print("El largo del coche es: ",miCoche.largoChasis)
22 print("El coche tiene ", miCoche.ruedas, " ruedas")
23 miCoche.arrancar()
24
25 print(miCoche.estado())
```

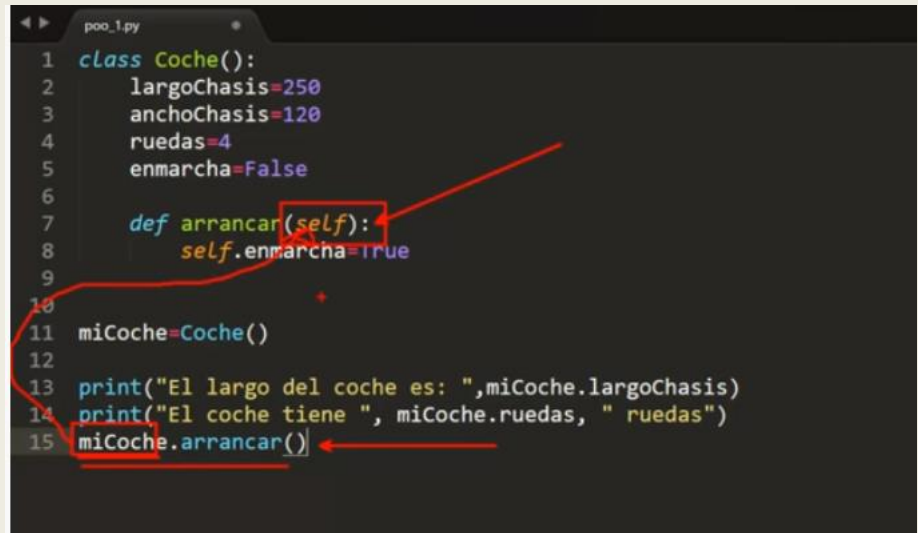
El self funciona tomando el valor del objeto (instancia de la clase).  
Y con ese objeto llamamos al procedimiento o método



```
1 class Coche():
2     largoChasis=250
3     anchoChasis=120
4     ruedas=4
5     enmarcha=False
6
7     def arrancar(self):
8         self.enmarcha=True
9
10    def estado(self):
11        if(self.enmarcha):
12            return "El coche está en marcha"
13
14        else:
15            return "El coche está parado"
16
17
18 miCoche=Coche()
19
20
21 print("El largo del coche es: ",miCoche.largoChasis)
22 print("El coche tiene ", miCoche.ruedas, " ruedas")
23 #miCoche.arrancar()
24
```

4 propiedades

2 comportamientos



```
1 class Coche():
2     largoChasis=250
3     anchoChasis=120
4     ruedas=4
5     enmarcha=False
6
7     def arrancar(self):
8         self.enmarcha=True
9
10
11 miCoche=Coche()
12
13 print("El largo del coche es: ",miCoche.largoChasis)
14 print("El coche tiene ", miCoche.ruedas, " ruedas")
15 miCoche.arrancar()
```

Tenemos así en este ejemplo 4 propiedades y 2 comportamientos



```
class Nueva_clase(object):  
    def metodo1(self, [parametros]):  
        codigo_metodo1
```

donde self

- Es el primer parámetro de cualquier método.
- Hace referencia a la propia clase (y a su contenido).
- Nunca se pasa como parámetro cuando se llama a un método. Es un *parámetro implícito*.

```
>>> class Saludo (object):  
    nombre = 'Jose Luis'  
    apellidos = 'Montero Fuentes'  
    def saludar (self):  
        print 'Hola. Soy %s %s' % (self.nombre, self.apellidos)
```

```
>>> objeto = Saludo ()
```

Creación de objeto (instanciación)

```
>>> objeto.saludar ()
```

Ejecución de método

```
Hola. Soy Jose Luis Montero Fuentes
```

```
>>>
```

Implementaremos una clase llamada Persona que tendrá como atributo (variable) su nombre y dos métodos (funciones), uno de dichos métodos inicializará el atributo nombre y el siguiente método mostrará en la pantalla el contenido del mismo.

### ***Definir dos objetos de la clase Persona***

```
class Persona:

    def inicializar(self,nom):
        self.nombre=nom

    def imprimir(self):
        print("Nombre",self.nombre)

# bloque principal
personal=Persona()
personal.inicializar("Pedro")
personal.imprimir()

persona2=Persona()
persona2.inicializar("Carla")
persona2.imprimir()
```

Las clases de Python tienen algo similar a un constructor: el método `__init__`. `__init__` se llama inmediatamente tras crear una instancia de la clase



En Python cuando crear una función `def __init__(self):`

Dentro del constructor los atributos deben estar precedidos por `self`

El método `__init__` es un método especial de una clase en Python. El objetivo fundamental del método `__init__` es inicializar los atributos del objeto que creamos.

Básicamente el método `__init__` reemplaza al método inicializar que habíamos hecho en el concepto anterior.

```
1 class Coche():
2
3     def __init__(self):
4
5         self.largoChasis=250  ← estado inicial
6         self.anchoChasis=120
7         self.ruedas=4
8         self.enmarcha=False   ← constructor
9
10    def arrancar(self,arrancamos):
11        self.enmarcha=arrancamos
12
13        if(self.enmarcha):
14            return "El coche está en marcha"
15
16        else:
17
18            return "El coche está parado"
19
20
21    def estado(self):
22        print("El coche tiene ", self.ruedas, " ruedas. Un ancho de ", self.anchoChasis, " y un largo de ",
23              self.largoChasis)
24
```

Las **ventajas** de implementar el método `__init__` en lugar del método inicializar son:

El método `__init__` es el primer método que se ejecuta cuando se crea un objeto.

El método `__init__` se llama automáticamente. Es decir es imposible de olvidarse de llamarlo ya que se llamará automáticamente

Otras **características** del método `__init__` son:

Se ejecuta inmediatamente luego de crear un objeto

El método `__init__` no puede retornar dato

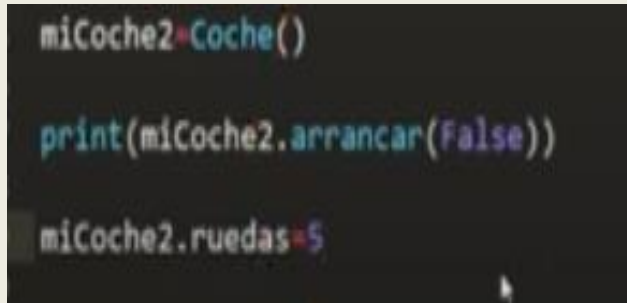
el método `__init__` puede recibir parámetros que se utilizan normalmente para inicializar atributos.

El método `__init__` es un método opcional, de todos modos es muy común declararlo.

Veamos la sintaxis del constructor:

```
def __init__([parámetros]):  
    [algoritmo]
```

Al definir un método llamado `__init__` utilizamos dos caracteres de subrayado, la palabra `init` y seguidamente otros dos caracteres de subrayado).



```
miCoche2=Coche()  
  
print(miCoche2.arrancar(False))  
  
miCoche2.ruedas=5
```

Si queremos cambiar el valor de un atributo en una instancia usamos la nomenclatura del punto

Estamos asignando 5 al atributo rueda

Pero como este valor no debería permitirse para la cantidad de ruedas de un auto, entonces como podemos hacer validar que se asignen determinados valores? Mediante la encapsulación podemos hacer que los valores no se modifiquen desde afuera de la clase.

# Encapsulamiento

**Encapsulamiento:** Limitar el acceso o dar acceso restringido de una propiedad a los elementos

```
class Coche():  
  
    def __init__(self):  
  
        self.largoChasis=250  
        self.anchoChasis=120  
        self.__ruedas=4  
        self.enmarcha=False
```

Podemos encapsular una atributo de la clase si al definir la clase ponemos antes del atributo rueda ponemos \_\_ (dos guiones bajos) estamos encapsulando ese atributo para que no pueda modificarse desde afuera de la definición de la clase.

```
def estado(self):  
    print("El coche tiene " , self.__ruedas, " ruedas. Un ancho de ", self.anchoChasis, " y un largo de ",  
          self.largoChasis)
```

Si quiero modificar el valor de algún atributo encapsulado debo hacerlo desde un método

**¡OJO!** Después al referenciar esta variable lo tengo que hacer con \_\_ sino se referirá a otra variable.

```
class Coche():  
  
    def __init__(self):  
  
        self.__largoChasis=250  
        self.__anchoChasis=120  
        self.__ruedas=4  
        self.__enmarcha=False  
  
    def arrancar(self,arrancamos):  
        self.__enmarcha=arrancamos  
  
        if(self.__enmarcha):  
            return "El coche está en marcha"  
  
        else:  
  
            return "El coche está parado"
```

```
def __chequeo_interno(self):  
    print("realizando chequeo interno")  
  
    self.gasolina="ok"  
    self.aceite="ok"  
    self.puertas="cerradas"
```

También podemos encapsular un Método, es decir definir el método en la clase y sólo acceder a él desde ahí y no desde afuera.

Al nombre del método le ponemos \_\_ . De esta forma no lo podremos llamar desde afuera de la Clase, o sea desde un objeto. Sí lo podremos llamar desde otro método.

```
def arrancar(self, arrancamos):  
    self.__enmarcha=arrancamos  
  
    if(self.__enmarcha):  
        chequeo=self.__chequeo_interno()
```

También al referenciarlo debemos hacerlo con \_\_

Encapsular o no una variable o método dependerá de la lógica de cada caso y de lo que queremos realizar