

# Programación orientada a Objetos

## 1. Lógica de clases

## 2. Encapsulación

## 3. Relaciones entre objetos

### 3.1. Polimorfismo

### 3.2. Herencia

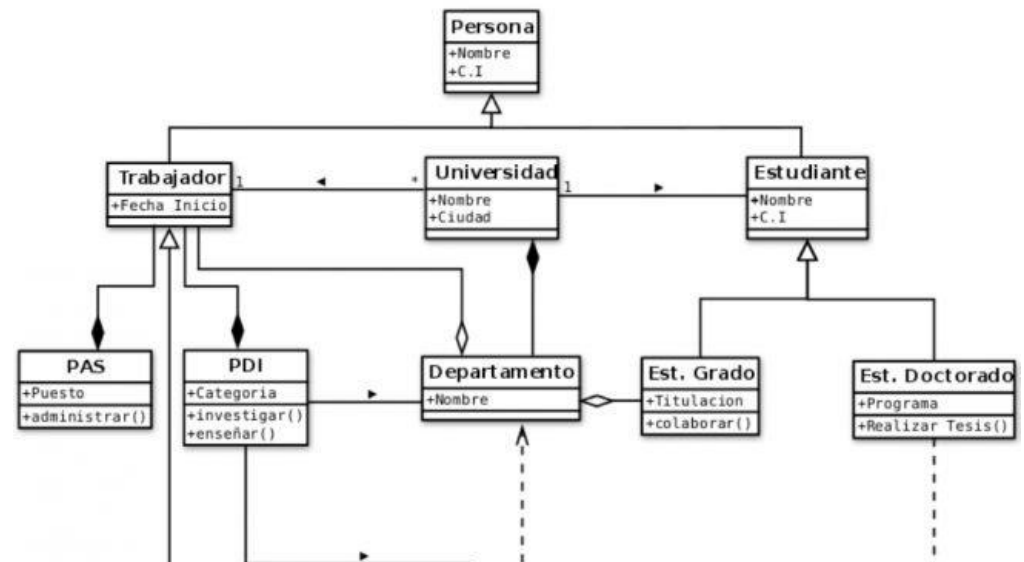
### 3.3. Dependencia

### 3.4. Asociación

Programació  
amb Python



Unió Europea  
Fons social europeu



# 1. Lógica de clases

## Pasando de lógica de diccionarios a lógica de clases

Partimos de un problema planteado sin objetos pasamos a lógica de objetos definiendo que clases necesitamos y qué métodos tendrá cada una de ellas.



# Cómo trataríamos una lista de clientes con programación estructurada ?

## 1. Organización de los datos

```
In [2]: """ Ejemplo de implementación con Programación Estructurada """

clientes= [
    {'Nombre': 'Hector', 'Apellidos': 'Costa Guzman', 'dni': '11111111A'},
    {'Nombre': 'Juan', 'Apellidos': 'González Márquez', 'dni': '22222222B'}
]
```

```
In [3]: clientes
```

```
Out[3]: [{'Apellidos': 'Costa Guzman', 'Nombre': 'Hector', 'dni': '11111111A'},
          {'Apellidos': 'González Márquez', 'Nombre': 'Juan', 'dni': '22222222B'}]
```

## 2. Función para mostrar los clientes con un dni determinado

```
def mostrar_cliente(clientes, dni):
    for c in clientes:
        if (dni == c['dni']):
            print('{} {}'.format(c['Nombre'], c['Apellidos']))
            return

    print('Cliente no encontrado')
```

```
mostrar_cliente(clientes, '11111111A')
```

Hector Costa Guzman

Una función para eliminar un cliente del diccionario.

```
[3]: def borrar_cliente(clientes, dni):  
    for i,c in enumerate(clientes):  
        if (dni == c['dni']):  
            del( clientes[i] )  
            print(str(c), "> BORRADO")  
            return  
  
    print('Cliente no encontrado')
```

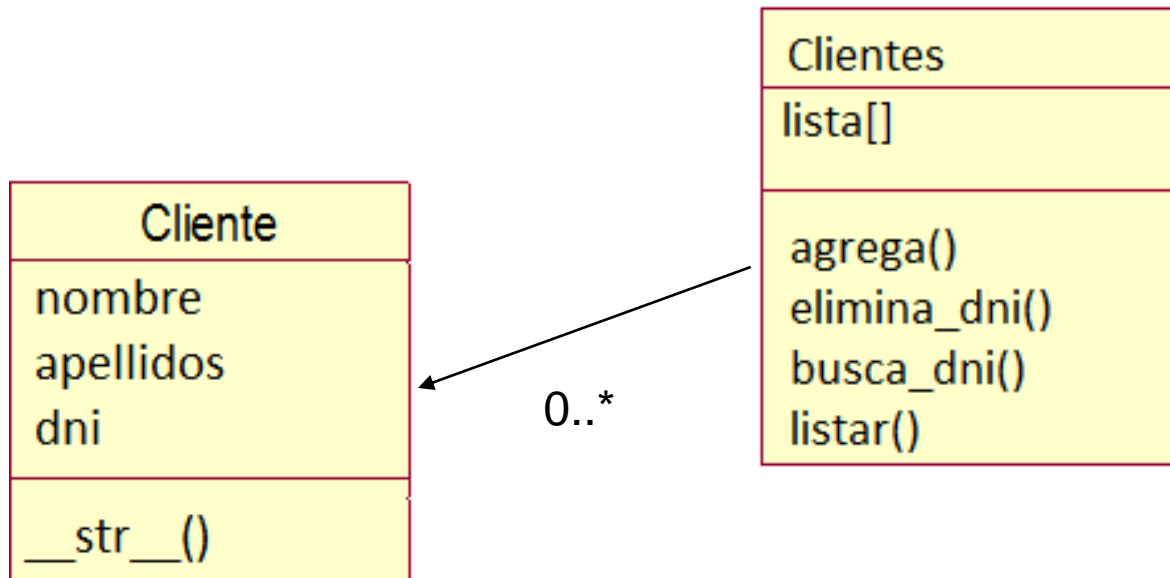
```
borrar_cliente(clientes, '22222222B')
```

```
{'dni': '22222222B', 'Nombre': 'Juan', 'Apellidos': 'González Márquez'} > BORRADO
```

## Creando clases para trabajar con estos clientes

```
clientes
```

```
[{'Apellidos': 'Costa Guzman', 'Nombre': 'Hector', 'dni': '11111111A'},  
 {'Apellidos': 'González Márquez', 'Nombre': 'Juan', 'dni': '22222222B'}]
```



Representación gráfica de una  
clase

# Elementos de los diagramas de clases

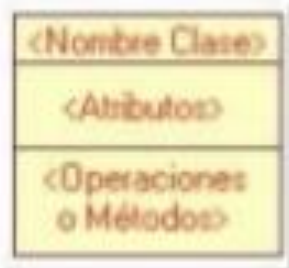
## ► Clase

Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

- En UML, una clase es representada por un rectángulo que posee tres divisiones:

En donde:

- **Superior:** Contiene el nombre de la Clase
- **Intermedio:** Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).
- **Inferior:** Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).



# Esquema de la clase : cliente

Cliente
nombre apellidos dni
<code>__str__()</code>

#----- Definición de clases

```
class Cliente:
```

```
    def __init__(self, dni, nombre, apellidos):  
        self.dni = dni  
        self.nombre = nombre  
        self.apellidos = apellidos
```

```
    def __str__(self):  
        return '{} {}'.format(self.nombre, self.apellidos)
```

#----- Programa principal : main

```
cli1 = Cliente("22222222B", "Juan", "Gonzalez Marquez")  
cli2 = Cliente("11111111A", "Hector", "Costa Guzman")  
print(cli1)  
print(cli2)
```

# Práctica P01

1- Crea la clase **Cientes**, que incluya una lista de clientes.

Implementa las funciones para **agregar** un cliente nuevo,

Para **buscar** y **eliminar** por dni

Para **listar** todos los clientes

Tienes ayuda en la diapositiva de la página siguiente

Cientes
lista[]
agrega ( cliente ) elimina_dni ( dni ) busca_dni (dni) listar ()



# Uso de la clase Clientes

```
#----- PROCESO PRINCIPAL
clientes= Clientes()

clientes.agrega(Cliente( '11111111A' , 'Hector','Costa Guzman' ))
clientes.agrega(Cliente( '11111111Z' , 'Maria','Rosca Rus' ))
clientes.agrega(Cliente( '22222222V' , 'Laura','Pinillos Rodriguez' ))

print("__ LISTADO DE CLIENTES __")
clientes.listar()

print("\n__ BUSCAR POR DNI __")
clientes.busca_dni ( '11111111A' )
clientes.busca_dni ( '11111111Z' )

print("\n__ BORRAR POR DNI __")
clientes.elimina_dni( '22222222V' )
clientes.elimina_dni( '22222222B' )
```

## 2. Encapsulación

La encapsulación consiste en denegar el acceso a los atributos y métodos internos de la clase desde el exterior.

En Python no existe, pero se puede simular precediendo atributos y métodos con dos barras bajas `__` como indicando que son "especiales"

```
In [1]: class Ejemplo:
        __atributo_privado = "Soy un atributo inalcanzable desde fuera"

        def __metodo_privado(self):
            print("Soy un método inalcanzable desde fuera")
```

```
In [2]: e = Ejemplo()
```

```
In [3]: e.__atributo_privado
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-3-73328dd71c23> in <module>()
----> 1 e.__atributo_privado

AttributeError: 'Ejemplo' object has no attribute '__atributo_privado'
```

```
In [4]: e.__metodo_privado()
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-4-164c67db4a9b> in <module>()  
----> 1 e.__metodo_privado()  
  
AttributeError: 'Ejemplo' object has no attribute '__metodo_privado'
```

Para acceder a esos datos se deberían crear métodos públicos que hagan de interfaz. En otros lenguajes les llamaríamos *getters* y *setters* y es lo que da lugar a las *propiedades*, que no son más que atributos protegidos con interfaces de acceso:

```
In [10]: class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera"

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera")

    def atributo_publico(self):
        return self.__atributo_privado

    def metodo_publico(self):
        return self.__metodo_privado()
```



```
In [11]: e = Ejemplo()
```

```
In [7]: e.atributo_publico()
```

```
Out[7]: 'Soy un atributo inalcanzable desde fuera'
```

```
In [12]: e.metodo_publico()
```

```
Soy un método inalcanzable desde fuera
```

# Práctica P02

Estudia este ejemplo que construye una clase punto para poder calcular distancias en ejes de coordenadas y dados cuatro puntos calcula el área del rectángulo.

No es necesario que hagas la programación, pero imagina como lo resolverías y posteriormente visualiza la solución propuesta.

## **Ejercicios « Programación Orientada a Objetos**

<https://docs.hektorprofe.net/python/programacion-orientada-a-objetos/ejercicios/>

La clase creada, tiene atributos o métodos privados ?

# 3. Relaciones entre objetos

## 3.1.Polimorfismo

La palabra polimorfismo significa tener muchas formas.  
En programación, polimorfismo significa que el mismo nombre de función (pero diferentes firmas) se usa para diferentes tipos.

```
# A simple Python function to demonstrate  
# Polymorphism
```

```
def add(x, y, z = 0):  
    return x+y+z
```

```
# Driver code  
print(add(2, 3))  
print(add(2, 3, 4))
```

Otra variedad , el mismo método para varios objetos.

```
#----- Definición de clases
class Perro:
    def sonido(self):
        print('Guauuuuu!!!')
class Gato:
    def sonido(self):
        print('Miaaaauuuu!!!')
class Vaca:
    def sonido(self):
        print('Múuuuuuuuu!!!')

#----- Definición de funciones
def a_cantar(animales):
    for animal in animales:
        animal.sonido()

#----- Programa principal : main
perro1 = Perro()
perro2 = Perro()
gato1 = Gato()
gato2 = Gato()
vaca = Vaca()

granja = [perro1, gato1, vaca, gato2, perro2]
a_cantar(granja)
```

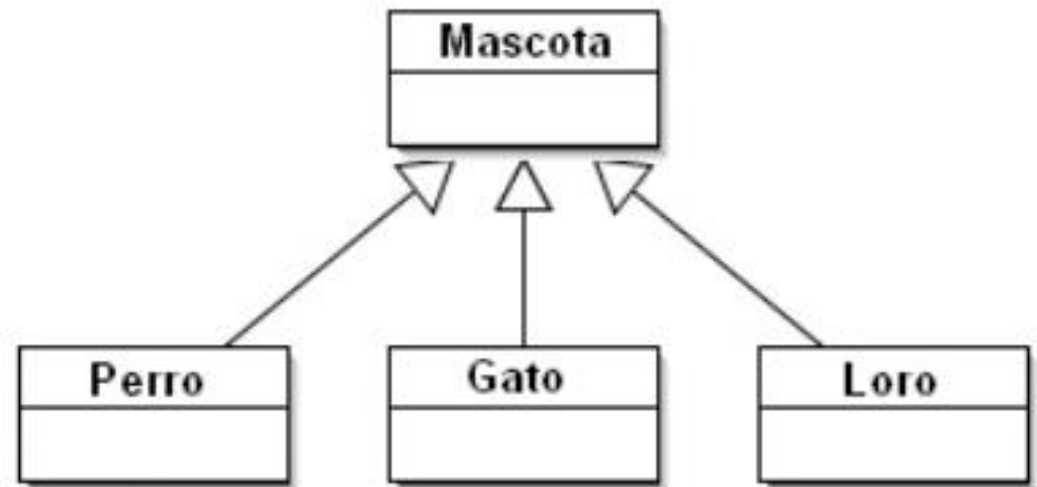
# 3. Relaciones entre objetos

## 3.2. Herencia

Una clase nueva se crea a partir de una clase existente.

Se establece una relación jerárquica entre la clase y su derivada o subclase.

De este modo puede aprovechar atributos y métodos de la clase padre y incluir otros nuevos.





# Ejemplo 1 de Herencia : Mascota

```
class Mascota :
    def __init__(self, tipo_animal, nombre, edad ) :
        self.tipo_animal = tipo_animal
        self.nombre = nombre
        self.edad = edad
    def __str__(self):
        return ( "tipo {} nombre {} edad {} años.format(
                    self.tipo_animal, self.nombre, self.edad))

class Perro(Mascota):
    def __init__(self, nombre, edad, raza, peso ) :
        self.tipo_animal = "perro"
        self.nombre = nombre
        self.edad = edad
        self.raza = raza
        self.peso = peso

    def __str__(self):
        return ( "PERRO: {} {} años {} {} kgs".format(self.nombre,
                    self.edad, self.raza, self.peso))
```

```
class Loro (Mascota):
    def __init__ (self, nombre, edad, color):
        self.tipo_animal = "loro"
        self.nombre = nombre
        self.edad = edad
        self.color = color
    def __str__(self):
        return ( "LORO: {} {} años, color {}".format(
            self.nombre, self.edad, self.color))

p = Perro("bobby", 6, "foxterrier", 3)
l = Loro ("juanjo", 3, "verde")
g = Mascota ("gusano","gusanito", "0.6")
print(p)
print(l)
print(g)
```

## Ejemplo 2 de Herencia : Qué sabe ?

```
class A :  
    def sabe_nadar (self):  
        print ("-- mira como nado --")  
  
class B :  
    def sabe_volar (self):  
        print ("-- mira como vuelo --")  
  
class C (A) :  
    def sabe_patinar (self):  
        print ("-- mira como patino --")  
  
a = C()
```

En el editor teclear a. y tab y verificar los métodos disponibles para la clase.

# Ejemplo 3 de Herencia : Productos

## Una tienda sin Herencia

Hace muchos años me vi en la necesidad de diseñar una estructura para una tienda que vendía tres tipos de productos: adornos, alimentos y libros.

Todos los productos de la tienda tenían una serie de atributos comunes, como la referencia, el nombre, el pvp... pero algunos eran específicos de cada tipo.

Si partimos de una clase que contenga todos los atributos, quedaría más o menos así:

```
class Producto:
    def __init__(self, referencia, tipo, nombre,
                  pvp, descripcion, productor=None,
                  distribuidor=None, isbn=None, autor=None):
        self.referencia = referencia
        self.tipo = tipo
        self.nombre = nombre
        self.pvp = pvp
        self.descripcion = descripcion
        self.productor = productor
        self.distribuidor = distribuidor
        self.isbn = isbn
        self.autor = autor
```

```
adorno = Producto('000A', 'ADORNO', 'Vaso Adornado', 15,  
                  'Vaso de porcelana con dibujos')  
  
print(adorno)  
print(adorno.tipo)
```

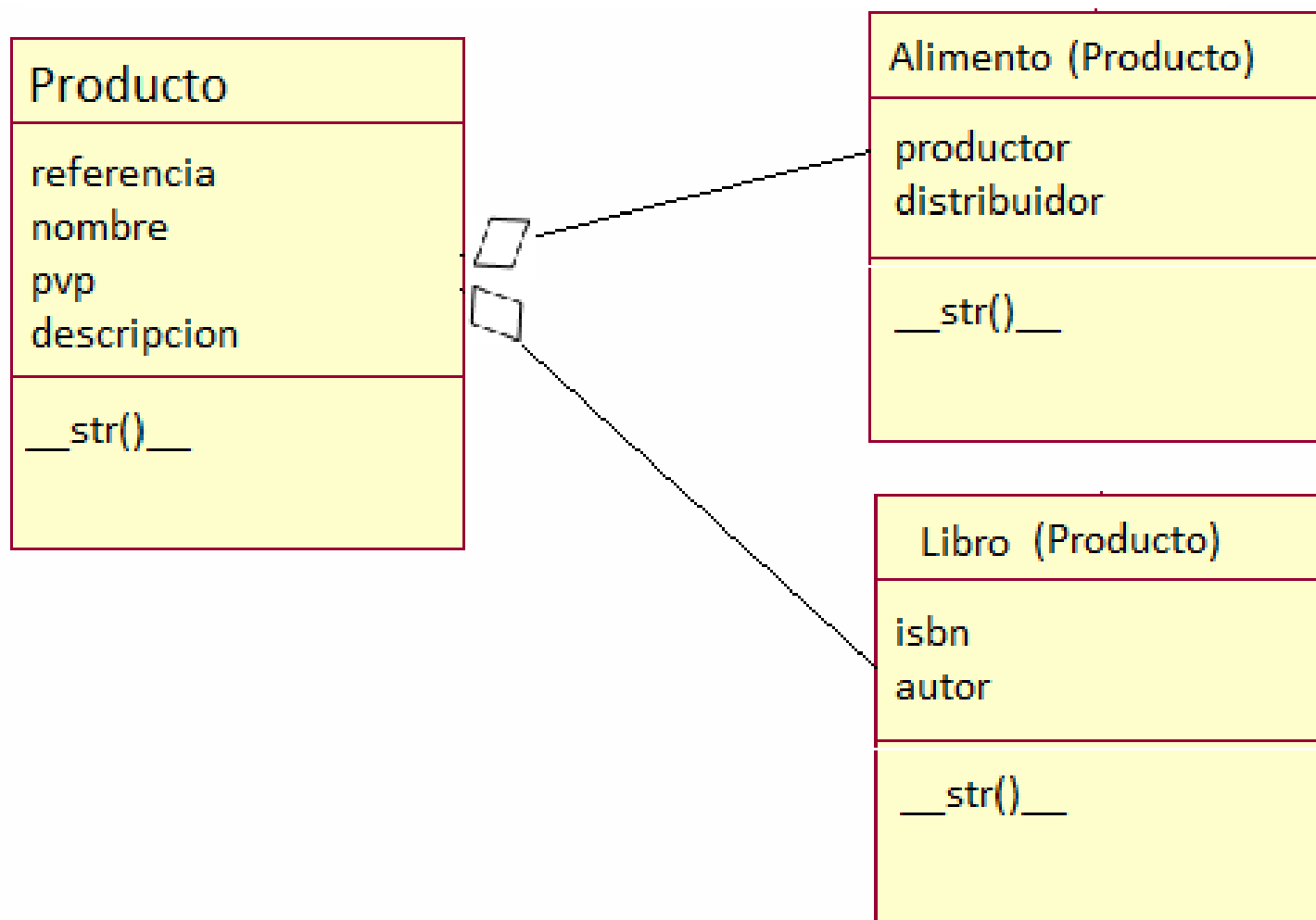
## Aplicando la Herencia

```
class Producto:  
    def __init__(self, referencia, nombre, pvp, descripcion):  
        self.referencia = referencia  
        self.nombre = nombre  
        self.pvp = pvp  
        self.descripcion = descripcion  
  
    def __str__(self):  
        return f"REFERENCIA\t {self.referencia}\n" \  
               f"NOMBRE\t\t {self.nombre}\n" \  
               f"PVP\t\t {self.pvp}\n" \  
               f"DESCRIPCIÓN\t {self.descripcion}\n"
```

<https://docs.hektorprofe.net/python/herencia-en-la-poo/herencia/>

# Superclases

Así pues la idea de la herencia es identificar una clase base (la superclase) con los atributos comunes y luego crear las demás clases heredando de ella (las subclasses) extendiendo sus campos específicos. En nuestro caso esa clase sería el Producto en sí mismo:



```

class Alimento(Producto):
    productor = ""
    distribuidor = ""

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n" \
               f"PRODUCTOR\t\t {self.productor}\n" \
               f"DISTRIBUIDOR\t\t {self.distribuidor}\n"

```

Alimento (Producto)
productor distribuidor
__str().__

```

class Libro(Producto):
    isbn = ""
    autor = ""

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n" \
               f"ISBN\t\t {self.isbn}\n" \
               f"AUTOR\t\t {self.autor}\n"

```

Libro (Producto)
isbn autor
__str().__

```
alimento = Alimento(2035, "Botella de Aceite de Oliva", 5, "250 ML")
alimento.productor = "La Aceitera"
alimento.distribuidor = "Distribuciones SA"
print(alimento)

libro = Libro(2036, "Cocina Mediterránea", 9, "Recetas sanas y buenas")
libro.isbn = "0-123456-78-9"
libro.autor = "Doña Juana"
print(libro)
```



Luego en los ejercicios os mostraré como podemos sobrescribir el constructor de una forma eficiente para inicializar campos extra, por ahora veamos como trabajar con estos objetos de distintas clases de forma común.

Passar a la web ...

<https://docs.hektorprofe.net/python/herencia-en-la-poo/herencia/>



## Ejemplo 4 de Herencia : Vehículos

```
class Vehiculo():  
  
    def __init__(self, color, ruedas):  
        self.color = color  
        self.ruedas = ruedas  
  
    def __str__(self):  
        return "Color {}, {} ruedas".format( self.color, self.ruedas )  
  
class Coche(Vehiculo):  
  
    def __init__(self, color, ruedas, velocidad, cilindrada):  
         Vehiculo.__init__(self, color, ruedas)  
        self.velocidad = velocidad  
        self.cilindrada = cilindrada  
  
    def __str__(self):  
        return Vehiculo.__str__(self) + ", {} km/h, {} cc".format(self.veloc:  
          
  
c = Coche("azul", 4, 150, 1200)  
print(c)
```

# Práctica P03

Estudia esta clase.

Crea una clase heredada que se llame Textil y con atributos : color y material.

Crea otra clase heredada Brico que con atributos : medida y peso.

Pruébalas.

```
#----- Programa principal : main
class Producto:
    def __init__(self, referencia, nombre, pvp, descripcion):
        self.referencia = referencia
        self.nombre = nombre
        self.pvp = pvp
        self.descripcion = descripcion

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n"

#
#----- Programa principal : main
p=Producto("J72w", "Tornillo y tuerca 4mm", 2.5,
           "blister de 20 unidades")
print(p)
```

# Práctica P04 (Avanzado)

2.1. Crea un programa que se llame `clase_carrito.py`.

2.2. Incorpora las clases de Productos y adáptalos al tu gusto.

2.3. Usando las clases anteriores crea una nueva clase para `carro de la Compra`, te propongo unos métodos :

- `agregar_al_carro (producto)`
- `eliminar_del_carro (producto)`
- `mostrar_carro()`
- `total_compra()`

# Práctica P05

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

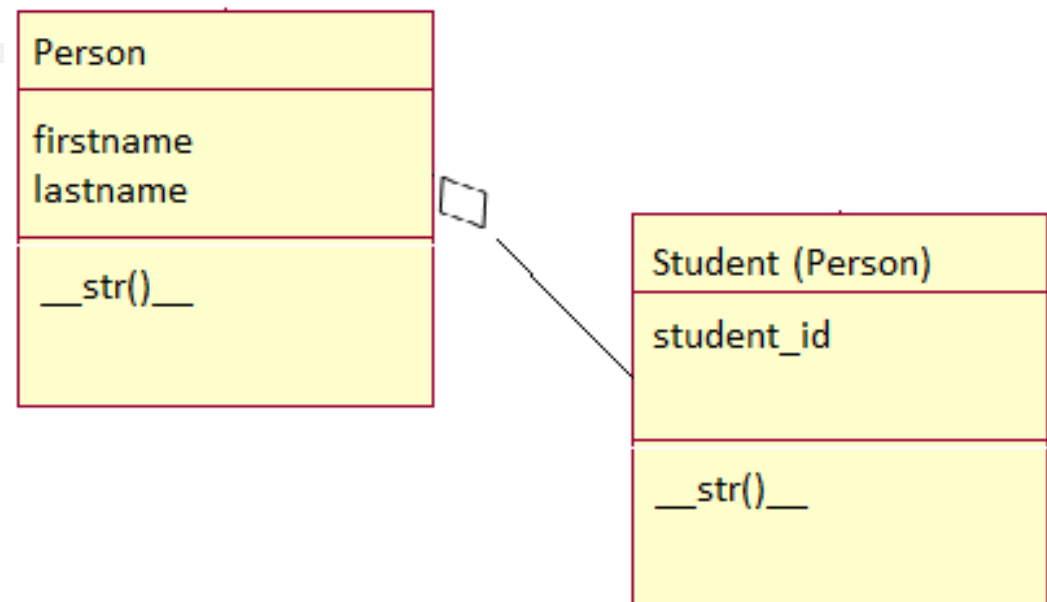
    def printname(self):
        print(self.firstname, self.lastname)

class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
```

---

Observa estas clases y  
piensa en la classe  
Estudiante y Profesor.

Qué atributos tendrá la clase  
Estudiante que no tenga la  
clase Profesor ?



# 3. Relaciones entre objetos

## 3.3. Dependencia

- **Dependencia o Instanciación (uso):**

Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.

El uso más particular de este tipo de relación es para denotar la dependencia que tiene una clase de otra, como por ejemplo una aplicación grafica que instancia una ventana (la creación del Objeto Ventana esta condicionado a la instanciación proveniente desde el objeto Aplicación):



- Cabe destacar que el objeto creado (en este caso la Ventana gráfica) no se almacena dentro del objeto que lo crea (en este caso la Aplicación).

## SIMBOLOGIA:

### CLASE



### HERENCIA



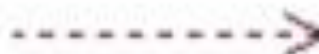
### AGREGACION



### ASOCIACION



### DEPENDENCIA O INSTANCIACION (USO)



# 3. Relaciones entre objetos

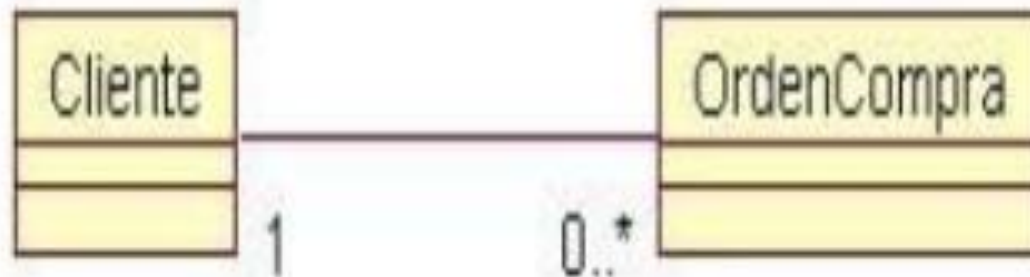
## 3.4. Asociación

- **Asociación:**



La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

- Ejemplo:



Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.

# Practica P06 Avanzada

Observa este diagrama que corresponde a una versión del juego de busca minas. Está adaptado a otro lenguaje distinto de python.

Puedes definir las clases en python, siguiendo este diagrama?

Ampliado en diapo siguiente

