

METODOLOGÍAS DE PROGRAMACIÓN I

Repaso del paradigma de la programación orientada a objetos implementado en Python

- Objetos
- Clases
- Jerarquías de clases

Clases en Python

Sintaxis de definición de clases

```
class <nombreClase> :  
    <funciones>
```

Clases en Python

```
class Persona:
```

```
    def __init__(self, n, a, e):  
        self.__nombre = n  
        self.__apellido = a  
        self.__edad = e
```

```
    def saludar(self):  
        print("Buen día !!!")
```

```
    def getNombreYApellido(self):  
        return self.__nombre + " " + self.__apellido
```

Clases en Python

```
class Persona:
```

```
    def __init__(self, n, a, e):
```

```
        self.__nombre = n  
        self.__apellido = a  
        self.__edad = e
```

```
    def saludar(self):
```

```
        print("Buen día !!!")
```

```
    def getNombreYApellido(self):
```

```
        return self.__nombre + " " + self.__apellido
```

Estado.
Se definen como
variables

Clases en Python

```
class Persona:
```

```
    def __init__(self, n, a, e):  
        self.__nombre = n  
        self.__apellido = a  
        self.__edad = e
```

```
    def saludar(self):  
        print("Buen día !!!")
```

```
    def getNombreYApellido(self):  
        return self.__nombre + " " + self.__apellido
```

Comportamiento.
Se definen como
funciones

Clases en Python

class Persona:

```
def __init__(self, n, a, e):  
    self.__nombre = n  
    self.__apellido = a  
    self.__edad = e
```

```
def saludar(self):  
    print("Buen día !!!")
```

```
def getNombreYApellido(self):  
    return self.__nombre + " " + self.__apellido
```

Constructor.
Se define con el
nombre especial
__init__

Instanciando objetos

- Para poder usar objetos hay que ***instanciar*** una clase.
- Para crear una instancia se debe especificar la clase del objeto a crear, por ejemplo:
 `per1 = Persona()`
- Se pueden crear tantas instancias de una clase como se necesite.

Clases en Python

```
def main():
```

```
    p = Persona("Ronnie", "Dio", 67)
```

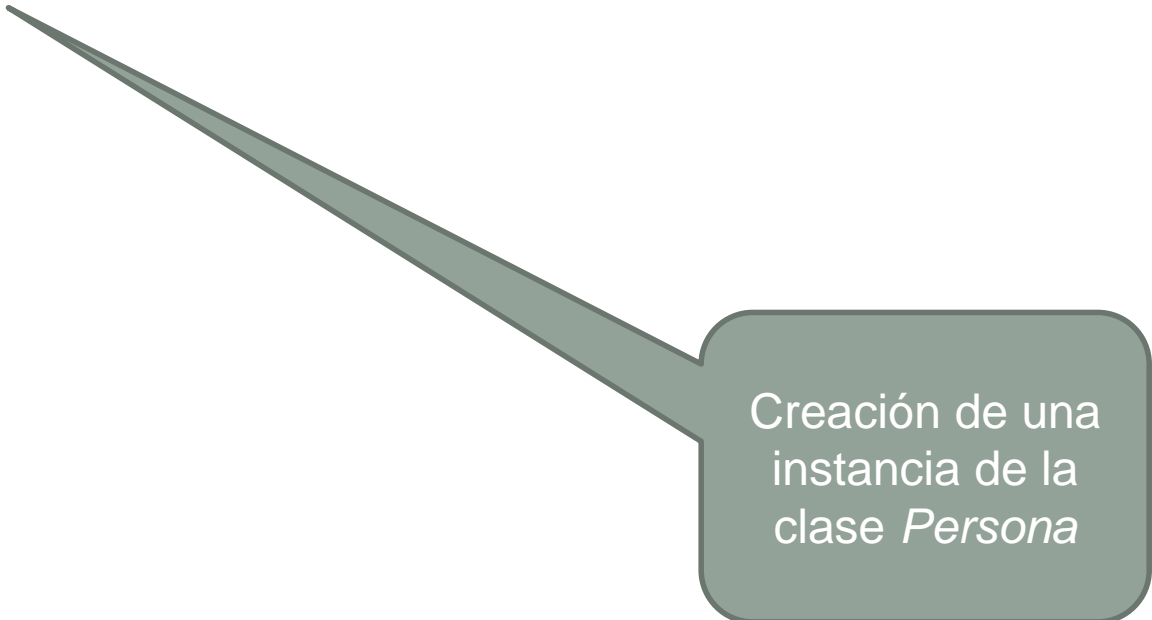
```
    p.saludar()
```


Clases en Python

```
def main():
```

```
    p = Persona("Ronnie", "Dio", 67)
```

```
    p.saludar()
```



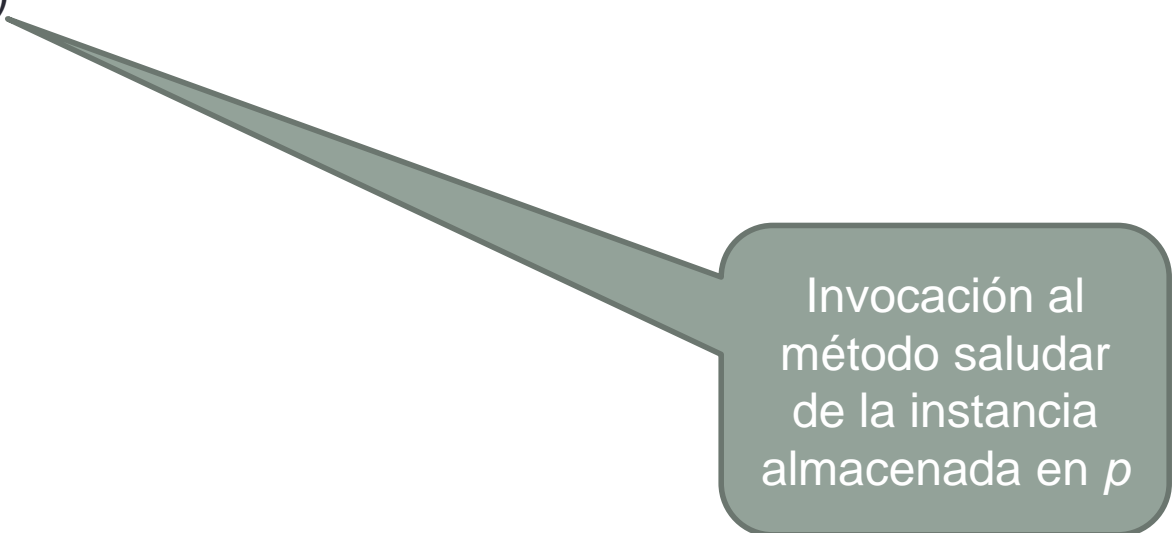
Creación de una
instancia de la
clase *Persona*

Clases en Python

```
def main():
```

```
    p = Persona("Ronnie", "Dio", 67)
```

```
    p.saludar()
```



Invocación al
método saludar
de la instancia
almacenada en *p*

Sobrecarga de métodos

- La sobrecarga de métodos es la característica de que una clase pueda tener más de un método con el mismo nombre siempre que sus firmas sean diferentes.
- La firma de un método consiste en:
 - El nombre
 - El número de parámetros
 - El tipo y el orden de los parámetros
 - Los modificadores de los parámetros
- Python no provee sobrecarga de métodos, pero podemos simular esa característica con la función **isinstance()**

Sobrecarga de métodos

Agreguemos dos "sobrecargas" al método "saludar" de la clase *Persona*.

```
def saludar(self, arg = None):  
    if arg == None:  
        print("Buen día !!!")  
  
    elif isinstance(arg, str):  
        print("Buen día " + arg + " !!!")  
  
    elif isinstance(arg, Persona):  
        print("Buen día " + arg.getNombreYApellido() + " !!!")
```

Clases en Python

```
def main():
```

```
    p1 = Persona("Ronnie", "Dio", 67)  
    p2 = Persona("Tony", "Iommi", 69)
```

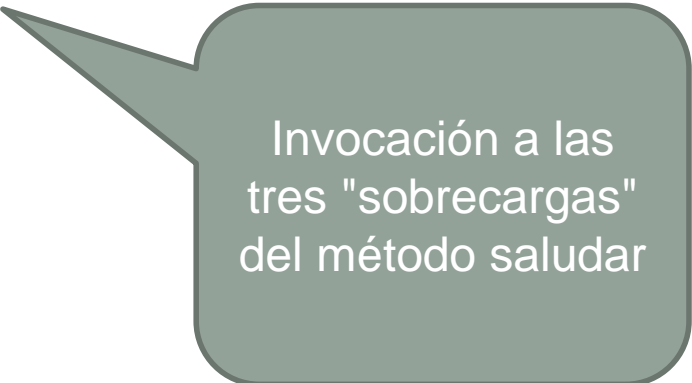
```
    p1.saludar()  
    p1.saludar("Vinny Appice")  
    p1.saludar(p2);
```

Clases en Python

```
def main():
```

```
    p1 = Persona("Ronnie", "Dio", 67)  
    p2 = Persona("Tony", "Iommi", 69)
```

```
    p1.saludar()  
    p1.saludar("Vinny Appice")  
    p1.saludar(p2);
```



Invocación a las
tres "sobrecargas"
del método saludar

Constructor por defecto

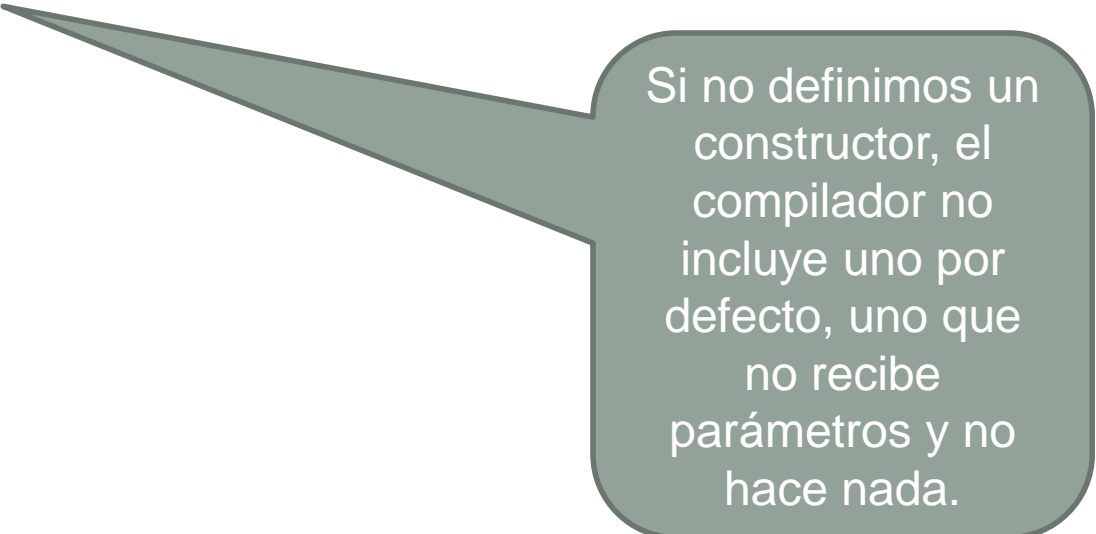
En caso de no definir un constructor para la clase el compilador creará uno por defecto:

```
def __init__(self):  
    pass
```

Constructores

```
class Persona:  
    def saludar(self):  
        print("Buen día !!!")
```

```
def main():  
    p = Persona()  
    p.saludar()
```



Si no definimos un constructor, el compilador no incluye uno por defecto, uno que no recibe parámetros y no hace nada.

Constructores

```
class Persona:
```

```
    def __init__(self, n, a, e):  
        self.__nombre = n  
        self.__apellido = a  
        self.__edad = e  
    def saludar(self):  
        print("Buen día !!!")
```

Al definir un constructor no-nulo, el compilador no incluye el compilador por defecto

```
def main():
```

```
    p = Persona()  
    p.saludar()
```

Esta línea ahora da error

Constructores

```
class Persona:
```

```
    def __init__(self, n = None, a = None, e = None):
```

```
        self.__nombre = n
```

```
        self.__apellido = a
```

```
        self.__edad = e
```

```
    def saludar(self):
```

```
        print("Buen día !!!")
```

```
def main():
```

```
    p = Persona()
```

```
    p.saludar()
```

Si queremos seguir construyendo instancias sin pasar parámetros, tenemos que declarar un valor por defecto a cada parámetro que recibe el constructor

Esta línea ahora no da error

Constructores

- Al igual que hicimos con el método saludar, podemos darle distintos comportamientos al constructor usando la función **isinstance** para determinar el tipo de datos que recibimos por parámetro

Miembros

- En POO hay dos tipos de miembros (variables y métodos)
 - De instancia
 - De clase
- En Python si queremos declarar miembros de clase usamos la directiva *@staticmethod*.

@staticmethod

```
def UnMetodoDeClase():
```

...

- Si queremos que los miembros sean de instancia no ponemos nada (y recibimos siempre como primer parámetro a *self*).

```
def UnMetodoDeInstancia(self):
```

...

Miembros de instancia

- Los miembros de instancia, ya sean variables o métodos son utilizados cuando se trabaja con instancias

```
p1.saludar()
```

```
p1.saludar("Vinny Appice")
```

```
p1.saludar(p2)
```

```
print(p2.getNombreYApellido())
```

Miembros de clase

- Los miembros de clase no pertenecen a ninguna instancia, pertenecen a la clase
- Se declaran con la directiva **@staticmethod**
- Todas las instancias comparten los mismos miembros de clase
- La referencia a un miembro de clase se hace mediante el nombre de clase

`<clase> . <miembro>`

Modificadores de acceso

- En POO se definen modificadores de acceso a miembros para asegurar el encapsulamiento:
 - Públicos
 - Privados
 - Protegidos
- Python **NO** provee mecanismos para asegurar el encapsulamiento (todos los miembros son públicos)
- Por convención:
 - Los nombres de los miembros privados comienzan con "__" (doble guión bajo)
 - Los nombres de los miembros protegidos comienzan con "_" (un guión bajo)

Modificadores de acceso

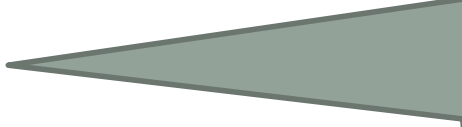
**** *Por convención* ****

- Miembros públicos
def metodo()
- Miembros privados
def __metodo()
- Miembros protegidos
def _metodo()

Modificadores de acceso

**** *Por convención* ****

- Miembros públicos
def metodo()
- Miembros privados
def __metodo()
- Miembros protegidos
def _metodo()

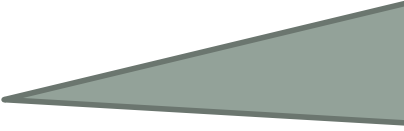


Los miembros públicos
pueden ser accedidos
desde cualquier clase
externa a quien declara
el miembro

Modificadores de acceso

**** Por convención ****

- Miembros públicos
def metodo()
- Miembros privados
def __metodo()
- Miembros protegidos
def _metodo()

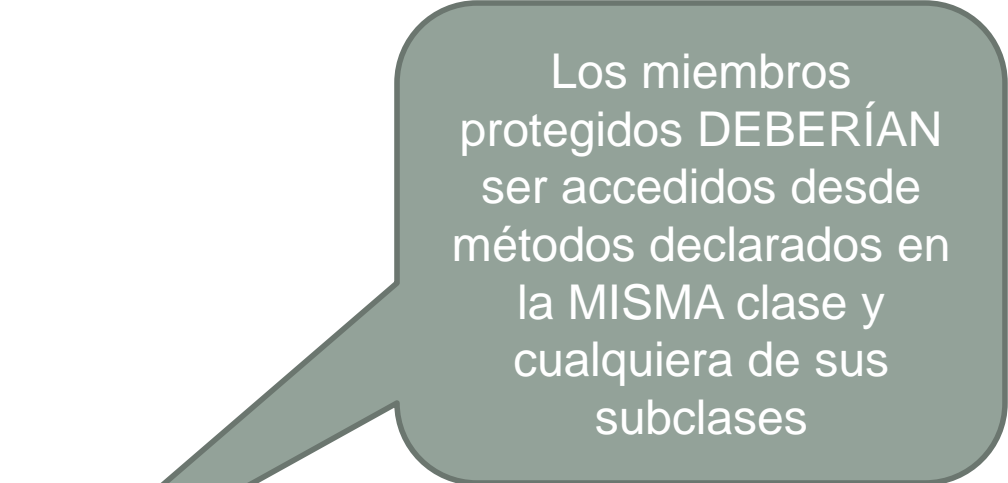


Los miembros privados
DEBERÍAN ser
accedidos
ÚNICAMENTE desde
métodos declarados en
la MISMA clase donde
está el miembro
privado.

Modificadores de acceso

**** *Por convención* ****

- Miembros públicos
def metodo()
- Miembros privados
def __metodo()
- Miembros protegidos
def _metodo()



Los miembros protegidos DEBERÍAN ser accedidos desde métodos declarados en la MISMA clase y cualquiera de sus subclases

Modificadores de acceso

El estado siempre lo vamos a declarar como privado o protegido.

```
class Persona:
```

```
    def __init__(self, n, a, e):  
        self.__nombre = n  
        self.__apellido = a  
        self.__edad = e
```

```
    def saludar(self):  
        print("Buen día !!!")
```

```
    def getNombreYApellido(self):  
        return self.__nombre + " " + self.__apellido
```


Modificadores de acceso

```
class Persona:
```

```
    def __init__(self, n, a, e):  
        self.__nombre = n  
        self.__apellido = a  
        self.__edad = e
```

```
    def saludar(self):  
        print("Buen día !!!")
```

```
    def getNombreYApellido(self):  
        return self.__nombre + " " + self.__apellido
```



En el caso de los métodos
los declararemos privados,
protegidos o públicos según
su función

Herencia de clases

- Las subclases se declaran indicando la superclase entre paréntesis

```
class MiPrimerSubClase (MiPrimeraClase):
```

Clases abstractas

```
from abc import ABCMeta, abstractmethod
```

```
class Vehiculo (metaclass = ABCMeta):
```

```
    @abstractmethod
```

```
    def acelerar(self, vel):  
        pass
```

```
    @abstractmethod
```

```
    def frenar(self):  
        pass
```

```
    def cantidadDePasajeros(self):  
        return self.__cant
```

Clases abstractas

```
from abc import ABCMeta, abstractmethod
```

```
class Vehiculo(metaclass = ABCMeta):
```

```
    @abstractmethod  
    def acelerar(self, vel):  
        pass
```

```
    @abstractmethod  
    def frenar(self):  
        pass
```

```
    def cantidadDePasajeros(self):  
        return self.__cant
```

Para que una clase se comporte como clase abstracta, se debe declarar la propiedad metaclass con la clase ABCMeta, la cual debe ser importada

Clases abstractas

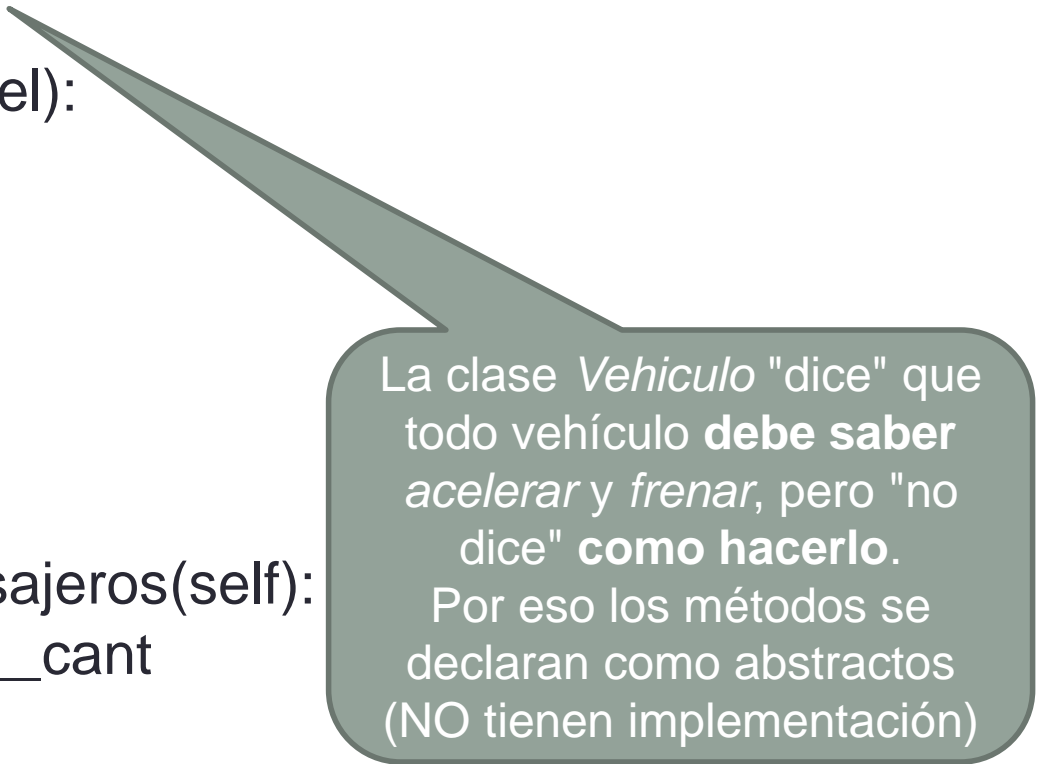
```
from abc import ABCMeta, abstractmethod
```

```
class Vehiculo (metaclass = ABCMeta):
```

```
    @abstractmethod  
    def acelerar(self, vel):  
        pass
```

```
    @abstractmethod  
    def frenar(self):  
        pass
```

```
    def cantidadDePasajeros(self):  
        return self.__cant
```



La clase *Vehiculo* "dice" que todo vehículo **debe saber acelerar y frenar**, pero "no dice" **como hacerlo**.

Por eso los métodos se declaran como abstractos (NO tienen implementación)

Clases abstractas

```
from abc import ABCMeta, abstractmethod
```

```
class Vehiculo (metaclass = ABCMeta):
```

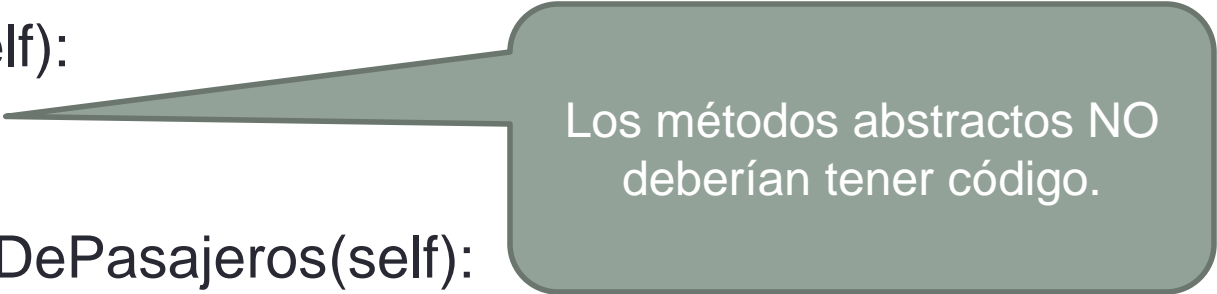
```
    @abstractmethod
```

```
    def acelerar(self, vel):  
        pass
```

```
    @abstractmethod
```

```
    def frenar(self):  
        pass
```

```
    def cantidadDePasajeros(self):  
        return self.__cant
```



Los métodos abstractos NO deberían tener código.

Clases abstractas

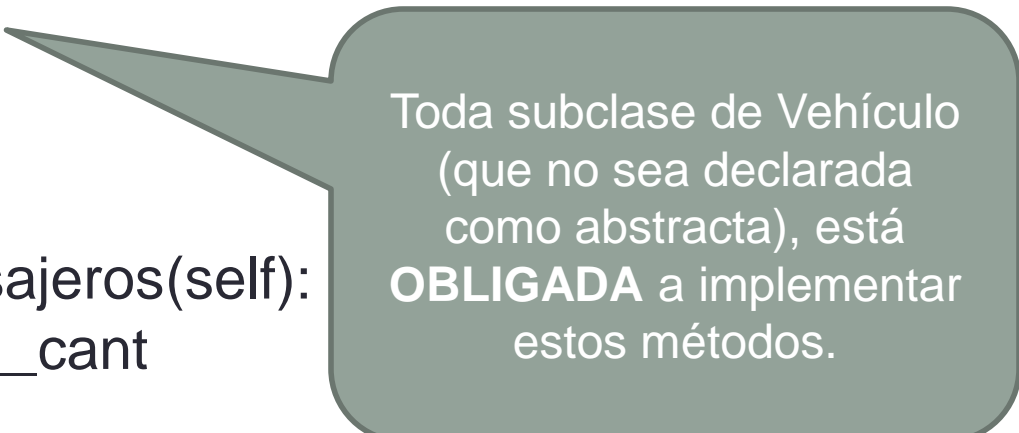
```
from abc import ABCMeta, abstractmethod
```

```
class Vehiculo (metaclass = ABCMeta):
```

```
    @abstractmethod  
    def acelerar(self, vel):  
        pass
```

```
    @abstractmethod  
    def frenar(self):  
        pass
```

```
    def cantidadDePasajeros(self):  
        return self.__cant
```



Toda subclase de Vehículo (que no sea declarada como abstracta), está **OBLIGADA** a implementar estos métodos.

Sobre-escritura de métodos

```
class Auto (Vehiculo):
```

```
    def acelerar(self, vel):
```

```
        . . .
```

```
    def frenar(self):
```

```
        . . .
```

La clase *Auto* DEBE
implementar los métodos
abstractos de *Vehiculo*

Sobre-escritura de métodos

```
class Taxi (Auto):
```

```
    def girar(self, angulo):
```

```
        super(Taxi, self).girar(angulo)
```

```
        print("Soy taxista, no necesito usar la  
            luz de giro")
```

En algunos casos, la sobre-escritura de un método necesita ejecutar el método de la superclase. Para ello se utiliza la función `super`

Sobre-escritura de método

```
class Taxi (Auto):
```

```
    def girar(self, angulo):
```

```
        print("Soy taxista, no necesito usar la  
            luz de giro")
```

```
        super(Taxi, self).girar(angulo)
```

El método *girar* de la superclase es un método más. Puede ser llamado en cualquier momento, incluso puede ser llamado más de una vez.

Constructores en jerarquías

```
class Musico (Persona):  
    pass
```

```
def main():
```

```
    p = Musico("Frank", "Zappa", 35)  
    p.saludar()
```

Los constructores se heredan de manera directa. Como en la clase *Persona* tenemos un constructor que recibe un nombre, un apellido y una edad, podemos instanciar la clase *Musico* usando ese constructor

Constructores en jerarquías

```
class Musico (Persona):  
    def __init__(self, n, a, e, i):  
        super(Musico, self).__init__(n, a, e)  
        self.__instrumento = i
```

```
def main():  
    p = Musico("Frank", "Zappa", 35, "Guitarra")  
    p.saludar()
```

En algunos casos necesitamos que las subclases sean instanciadas con atributos específicos.

Constructores en jerarquías

```
class Musico (Persona):
```

```
    def __init__(self, n, a, e):  
        super(Musico, self).__init__(n, a, e)  
        self.__instrumento = i
```

```
def main():
```

```
    p = Musico("Frank", "Zappa", 35, "Guitarra")  
    p.saludar()
```

Podemos hacer un constructor que reciba los cuatro argumentos. Con los primeros tres se invoca al super-constructor y el cuarto argumento lo almacenamos en una variable privada

Interfaces

- Una interface es un "tipo de dato". Es como declarar una clase abstracta pero sin ser superclase de una jerarquía.
- Python **NO** posee el concepto de interface.
- Existen varios métodos para emular una interface. En este documento declararemos las interfaces como clases abstractas, haciendo uso de la herencia múltiple que provee Python.

Interfaces

- Todo método declarado en una interface es abstracto y público.
- La interface dice que métodos se deben implementar.

```
from abc import ABCMeta, abstractmethod
```

```
class Amigable (metaclass = ABCMeta)  
    @abstractmethod  
    def jugar(self):  
        pass
```

Interfaces

- Las clases puede implementar cero, una o más interfaces.

```
class Persona (Amigable):  
    pass
```

```
class Perro (Amigable):  
    pass
```

Interfaces

- Las clases puede implementar cero, una o más interfaces.

```
class Persona (Amigable):  
    pass
```

```
class Perro (Amigable):  
    pass
```

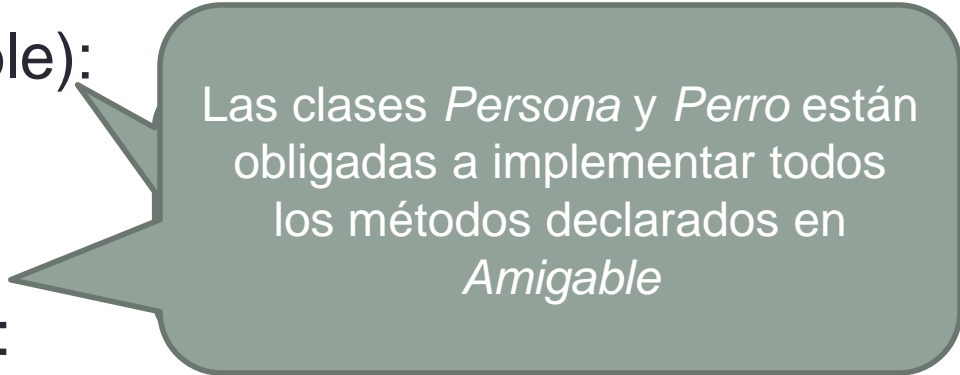
Como las interfaces las implementamos con clases abstractas, para decir que una clase debe implementar una interface, basta con decir que es subclase de la clase-interface

Interfaces

- Las clases puede implementar cero, una o más interfaces.

```
class Persona (Amigable):  
    pass
```

```
class Perro (Amigable):  
    pass
```



Las clases *Persona* y *Perro* están obligadas a implementar todos los métodos declarados en *Amigable*

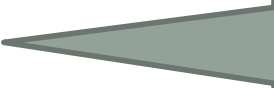
Interfaces

```
def metodo(self):
```

```
    c1 = Musico()
```

```
    c2 = Perro()
```

```
    c3 = Amigable()
```



Las interfaces (por ser
clases abstractas) NO se
pueden instanciar