



we design and  
create  
professional  
quality software

# Language - Agnostic

## Language-agnostic programming

La programación o secuencias de comandos independientes del lenguaje es un paradigma de desarrollo de software en el que se elige un lenguaje en particular debido a su idoneidad para una tarea en particular, y no simplemente por el conjunto de habilidades disponibles dentro de un equipo de desarrollo.

Modern Portfolio Designed

Te quedo? Listo  
(Rodrigo Bueno)



# Programación Orientada a Objetos

# PYTHON

POO

INTRODUCCION A LAS POO EN PYTHON.

# Programación Orientada a Objetos

 python Repaso

Se trata de un paradigma de programación introducido en los años 1970s, pero que no se hizo popular hasta años más tarde.

Este modo o paradigma de programación nos permite organizar el código de una manera que se asemeja bastante a como pensamos en la vida real, utilizando las famosas **clases**. Estas nos permiten agrupar un conjunto de variables y funciones que veremos a continuación.

Cosas de lo más cotidianas como un perro o un coche pueden ser representadas con clases. Estas clases tienen diferentes características, que en el caso del perro podrían ser la edad, el nombre o la raza. Llamaremos a estas características, atributos.

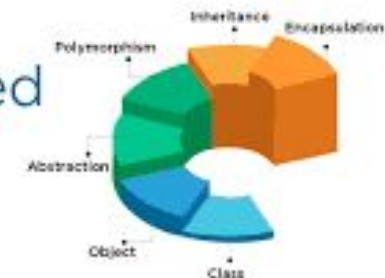
Por otro lado, las clases tienen un conjunto de funcionalidades o cosas que pueden hacer. En el caso del perro podría ser andar o ladrar. Llamaremos a estas funcionalidades métodos.

Por último, pueden existir diferentes tipos de perro. Podemos tener uno que se llama Toby o el del vecino que se llama Laika. Llamaremos a estos diferentes tipos de perro objetos. Es decir, el concepto abstracto de perro es la clase, pero Toby o cualquier otro perro particular será el objeto.

La programación orientada a objetos está basada en 6 principios o pilares básicos:

- Herencia
- Cohesión
- Abstracción
- Polimorfismo
- Acoplamiento
- Encapsulamiento

Object Oriented  
Programming  
with **Python**





# Motivación POO

## Repaso

Una vez repasada la programación orientada a objetos puede parecer bastante lógica, pero no es algo que haya existido siempre, y de hecho hay muchos lenguajes de programación que no la soportan.

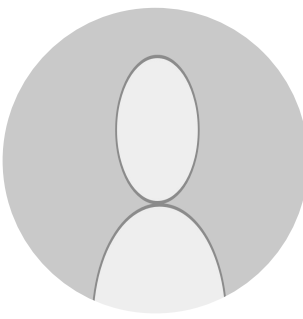
En parte surgió debido a la creciente complejidad a la que los programadores se iban enfrentando conforme pasaban los años. En el mundo de la programación hay gran cantidad de aplicaciones que realizan tareas muy similares y es importante identificar los patrones que nos permiten no reinventar la rueda. La programación orientada a objetos intentaba resolver esto.

Uno de los primeros mecanismos que se crearon fueron las **funciones**, que permiten agrupar bloques de código que hacen una tarea específica bajo un nombre. Algo muy útil ya que permite también reusar esos módulos o funciones sin tener que copiar todo el código, tan solo la llamada.

Las funciones resultaron muy útiles, pero no eran capaces de satisfacer todas las necesidades de los programadores. Uno de los problemas de las funciones es que sólo realizan unas operaciones con unos datos de entrada para entregar una salida, pero no les importa demasiado conservar esos datos o mantener algún tipo de estado. Salvo que se devuelva un valor en la llamada a la función o se usen variables globales, todo lo que pasa dentro de la función queda olvidado, y esto en muchos casos es un problema.

Imaginemos que tenemos un juego con naves espaciales moviéndose por la pantalla. Cada nave tendrá una posición (x,y) y otros parámetros como el tipo de nave, su color o tamaño. Sin hacer uso de clases y POO, tendremos que tener una variable para cada dato que queremos almacenar: coordenadas, color, tamaño, tipo. El problema viene si tenemos 10 naves, ya que nos podríamos juntar con un número muy elevado de variables. Todo un desastre.

En el mundo de la programación existen tareas muy similares al ejemplo con las naves, y en respuesta a ello surgió la programación orientada a objetos. Una herramienta perfecta que permite resolver cierto tipo de problemas de una forma mucho más sencilla, con menos código y más organizado. Agrupa bajo una clase un conjunto de variables y funciones, que pueden ser reutilizadas con características particulares creando objetos.





# PYTHON

POO INTRODUCCION A LAS POO EN PYTHON.

Powered By Chat GPT

Stage 1



# Definiendo clases

## Definiendo Clases

Repasada ya la parte teórica, vamos a ver como podemos hacer uso de la programación orientada a objetos en Python. Lo primero es crear una clase, para ello usaremos el ejemplo de perro.

# Creando una clase vacía

```
class Perro:  
    pass
```

Se trata de una clase vacía y sin mucha utilidad práctica, pero es la mínima clase que podemos crear. Nótese el uso del **pass** que no hace realmente nada, pero daría un error si después de los **:** no tenemos contenido.

Ahora que tenemos la clase, podemos crear un objeto de la misma. **Podemos hacerlo como si de una variable normal se tratase.**

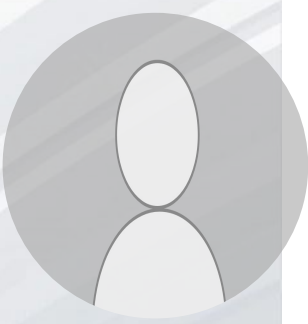
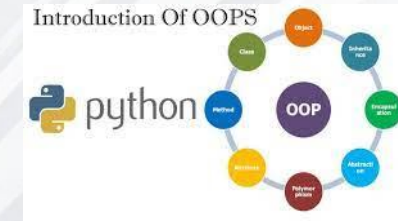
Nombre de la variable igual a la clase con (). Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
# Creamos un objeto de la clase perro  
mi_perro = Perro()
```



Powered By  
ChatGPT & Python

AI ChatGPT Python code



## Definiendo atributos

A continuación vamos a añadir algunos atributos a nuestra clase.

*Antes de nada es importante distinguir que **existen dos tipos de atributos en Python:***

- **Atributos de instancia:** Pertenecen a la instancia de la clase o al objeto. Son atributos particulares de cada instancia (Objeto), en nuestro caso de cada perro.
- **Atributos de clase:** Se trata de atributos que pertenecen a la clase, por lo tanto serán comunes para todos los objetos o Instancias de la Clase.

```
class Perro:
```

```
    # Atributo de clase
```

```
    especie = 'mamífero'
```

```
    # El método __init__ es llamado al crear el objeto (constructor)
```

```
    def __init__(self, nombre, raza):
```

```
        print(f"Creando perro {nombre}, {raza}")
```

```
    # Atributos de instancia
```

```
    self.nombre = nombre
```

```
    self.raza = raza
```



Veámoslo en mas detalle..





# Definiendo atributos

## Atributos de Instancia

Empecemos creando un par de atributos de **instancia** para nuestro perro, el nombre y la raza.

Para ello creamos un método `__init__` que será llamado automáticamente cuando creemos un objeto. Se trata del constructor !!!.

`class Perro:`

`# El método __init__ es llamado al crear el objeto`

`def __init__(self, nombre, raza):`

`print(f"Creando perro {nombre}, {raza}")`

`# Atributos de instancia`

`self.nombre = nombre`

`self.raza = raza`



Powered By  
ChatGPT & Python

**AI GPT:** En Python, los atributos de instancia son variables que pertenecen a una instancia específica de una clase. Cada instancia de la clase puede tener valores diferentes para sus atributos de instancia.

Ahora que hemos definido el método `init` con dos parámetros de entrada, podemos crear el objeto pasando el valor de los atributos. Usando `type()` podemos ver como efectivamente el objeto es de la clase `Perro`.

`mi_perro = Perro("Toby", "Bulldog")`

`print(type(mi_perro))`

`# Creando perro Toby, Bulldog`

`# <class '__main__.Perro'>`

Seguramente te hayas fijado en el **self** que se pasa como parámetro de entrada del método.

*Es una variable que representa la instancia de la clase, y deberá estar siempre ahí.*

El uso de `__init__` y el doble `__` no es una coincidencia. Cuando veas un método con esa forma, significa que está reservado para un uso especial del lenguaje.

En este caso sería lo que se conoce como constructor. Hay gente que llama a estos métodos mágicos.

Objetos que se creen de la clase `perro` compartirán ese atributo de la instancia de la clase, ya que pertenecen a la misma.



# Definiendo atributos

## Atributos de Clase

A diferencia de los atributos de instancia que son visibles en el nivel de objeto, los atributos de clase siguen siendo los mismos para todos los objetos.

Consulte el siguiente ejemplo para demostrar el uso de atributos de nivel de clase.

```
class BookStore:
    instances = 0
    def __init__(self, attrib1, attrib2):
        self.attrib1 = attrib1
        self.attrib2 = attrib2
        BookStore.instances += 1
```

```
b1 = BookStore("", "")
b2 = BookStore("", "")
```

```
print("BookStore.instances:", BookStore.instances)
```

En este ejemplo, "instances" es un atributo de nivel de clase. Puede acceder a él utilizando el nombre de la clase. Tiene el total no. de instancias creadas. Hemos creado dos instancias de la clase <Librería>. Por lo tanto, la ejecución del ejemplo debe imprimir "2" como salida.

```
# output
BookStore.instances: 2
```



Powered By  
ChatGPT & Python



# Demostración de la clase Python

Aquí hay un ejemplo en el que estamos construyendo una clase BookStore y creando instancias de su objeto con diferentes valores.

Crear una clase BookStore en Python

```
class BookStore:
```

```
    noOfBooks = 0
```

```
    def __init__(self, title, author):
```

```
        self.title = title
```

```
        self.author = author
```

```
        BookStore.noOfBooks += 1
```

```
    def bookInfo(self):
```

```
        print("Book title:", self.title)
```

```
        print("Book author:", self.author, "\n")
```

```
# Create a virtual book store
```

```
b1 = BookStore("Great Expectations", "Charles Dickens")
```

```
b2 = BookStore("War and Peace", "Leo Tolstoy")
```

```
b3 = BookStore("Middlemarch", "George Eliot")
```

```
# call member functions for each object
```

```
b1.bookInfo()
```

```
b2.bookInfo()
```

```
b3.bookInfo()
```

```
print("BookStore.noOfBooks:", BookStore.noOfBooks)
```

Puede abrir IDLE o cualquier otro IDE de Python, guardar el código anterior en algún archivo y ejecutar el programa. En este ejemplo, hemos creado tres objetos de la clase BookStore, es decir, b1, b2 y b3. Cada uno de los objetos es una instancia de la clase BookStore.

## Python Class & OOP Fundamentals

**Class**      self Reference to an object  
              \_\_init\_\_ Constructor method  
              class attrib Same for all objects  
              instance attrib Object specific data

```
class BookStore:
    instances = 0
    def __init__(self, attrib1, attrib2):
        self.attrib1 = attrib1
        self.attrib2 = attrib2
        BookStore.instances += 1
```

```
# output
```

```
Book title: Great Expectations
```

```
Book author: Charles Dickens
```

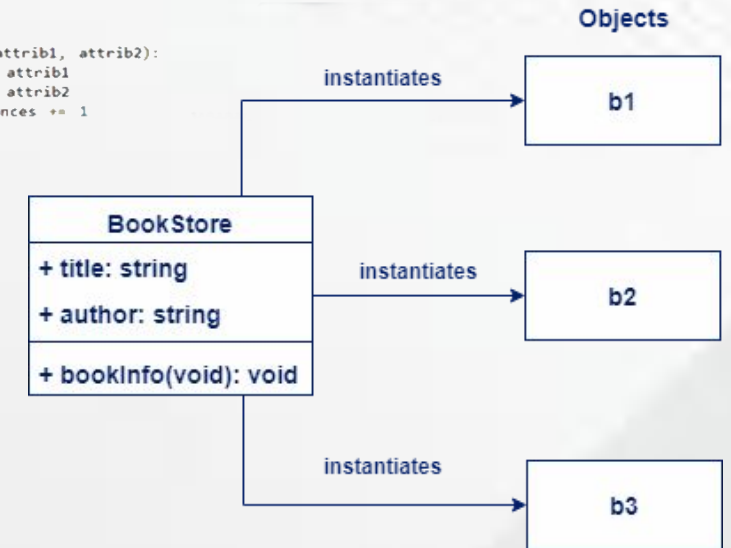
```
Book title: War and Peace
```

```
Book author: Leo Tolstoy
```

```
Book title: Middlemarch
```

```
Book author: George Eliot
```

```
BookStore.noOfBooks: 3
```



Python Class and Object Creation

Diagrama UML de la clase BookStore  
El diagrama UML del código anterior es el siguiente.

Clase y objetos de Python (diagrama UML)  
Después de ejecutar el código del ejemplo, debería ver el siguiente resultado.





# PYTHON

POO

MÉTODOS EN PYTHON



# Definiendo métodos

En realidad cuando usamos `__init__` anteriormente ya estábamos definiendo un método, solo que uno especial. A continuación vamos a ver como definir métodos que le den alguna funcionalidad interesante a nuestra clase, siguiendo con el ejemplo de perro.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

    # Atributos de instancia
    self.nombre = nombre
    self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")
```

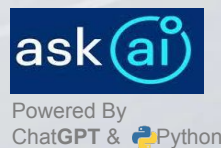


Vamos a codificar dos métodos, ladrar y caminar. El primero no recibirá ningún parámetro y el segundo recibirá el número de pasos que queremos andar. Como hemos indicado anteriormente `self` hace referencia a la instancia de la clase. Se puede definir un método con `def` y el nombre, y entre `()` los parámetros de entrada que recibe, donde siempre tendrá que estar `self` el primero.

Por lo tanto si creamos un objeto `mi_perro`, podremos hacer uso de sus métodos llamándolos con `.` y el nombre del método. Como si de una función se tratase, pueden recibir y devolver argumentos.

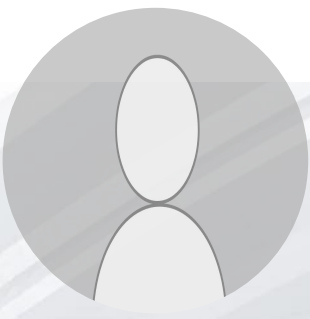
```
mi_perro = Perro("Toby", "Bulldog")
mi_perro.ladra()
mi_perro.camina(10)
```

```
# Creando perro Toby, Bulldog
# Guau
# Caminando 10 pasos
```





# Métodos en Python: instancia, clase y estáticos



En otros apartados hemos visto como se pueden crear métodos con `def` dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia. Pues bien, haciendo uso de los **decoradores**, es posible crear diferentes tipos de métodos:

- Los métodos de **instancia** “normales” que ya hemos visto como `metodo()`
- Métodos de **clase** usando el decorador `@classmethod`
- Métodos **estáticos** usando el decorador `@staticmethod`

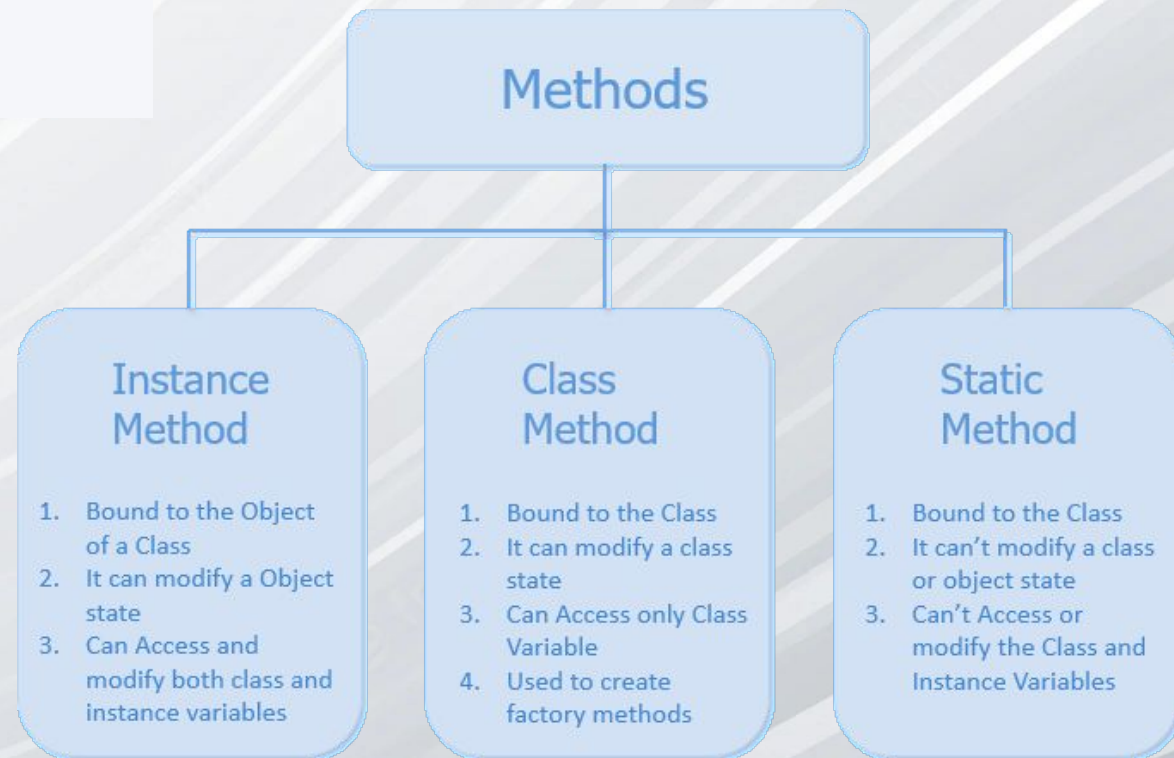
class Clase:

```
def metodo(self):  
    return 'Método normal', self
```

```
@classmethod  
def metododeclase(cls):  
    return 'Método de clase', cls
```

```
@staticmethod  
def metodoestatico():  
    return "Método estático"
```

Veamos su comportamiento en detalle uno por uno.



# Métodos en Python: instancia, clase y estáticos

## Métodos de instancia

Los métodos de instancia son los métodos normales, de toda la vida, que hemos visto anteriormente. Reciben como parámetro de entrada `self` que hace referencia a la instancia que llama al método. También pueden recibir otros argumentos como entrada.

Para saber más: El uso de "`self`" es totalmente arbitrario. Se trata de una convención acordada por los usuarios de Python, usada para referirse a la instancia que llama al método, pero podría ser cualquier otro nombre. Lo mismo ocurre con "`cls`", que veremos a continuación.

```
class Clase:  
    def metodo(self, arg1, arg2):  
        return 'Método normal', self
```

Y como ya sabemos, una vez creado un objeto pueden ser llamados.

```
mi_clase = Clase()  
mi_clase.metodo("a", "b")  
# ('Método normal', <__main__.Clase at 0x10b9daa90>)
```

En vista a esto, los métodos de instancia:

*Pueden acceder y modificar los atributos del objeto.*

*Pueden acceder a otros métodos.*

*Dado que desde el objeto `self` se puede acceder a la clase con ``self.class``, también pueden modificar el estado de la clase*

### Instance Method

1. Bound to the Object of a Class
2. It can modify a Object state
3. Can Access and modify both class and instance variables

# Métodos en Python: instancia, clase y estáticos

## Métodos de clase (classmethod)

A diferencia de los métodos de instancia, los métodos de clase reciben como argumento cls, que hace referencia a la clase. Por lo tanto, pueden acceder a la clase pero no a la instancia.

```
class Clase:  
    @classmethod  
    def metododeclase(cls):  
        return 'Método de clase', cls
```

Se pueden llamar sobre la clase.

```
Clase.metododeclase()  
# ('Método de clase', __main__.Clase)
```

Pero también se pueden llamar **sobre el objeto**.

```
mi_clase = Clase()  
mi_clase.metododeclase()  
# ('Método de clase', __main__.Clase)
```

Por lo tanto, los métodos de clase:

No pueden acceder a los atributos de la instancia.  
Pero si pueden modificar los atributos de la clase.

### Class Method

1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods



# Métodos en Python: instancia, clase y estáticos



## Métodos de clase (classmethod) cont...

**Los métodos de clase son métodos que se usan para realizar operaciones en el ámbito de clase y NO al generar una instancia.** Para crear un método de clase, debemos usar el decorador `@classmethod` y en el método debemos añadir como primer parámetro `cls`.

Al igual que pasaba con `self`, podemos emplear cualquier nombre, aunque por convención se recomienda `cls` que viene de `class`. Este parámetro se emplea para acceder a los atributos de la clase y a otros métodos, siempre y cuando también sean métodos de clase o estáticos.

```
class Perro:
    peso = 30    #Atributo de Clase

    def __init__(self, peso):    #Atributo de Instancia
        self.peso = peso

    @classmethod    #Metodo de Clase
    def get_peso_promedio(cls):
        return cls.peso

labrador = Perro(25)
print(f'El peso de un perro labrador es {labrador.peso} kilos')
# El peso de un perro labrador es 25 kilos
print(f'El peso promedio de un perro es {Perro.get_peso_promedio()} kilos')
# El peso promedio de un perro es 30 kilos
```

### Class Method

1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods

En este ejemplo, se ha creado una clase `Perro` que tiene un atributo llamado `peso` con un valor por defecto, pero que al crear una instancia lo reasignamos con el valor enviado por parámetro.

Después generamos el método de clase `get_peso_promedio` que retornará el atributo `peso`. Pues bien, si creamos una instancia de `Perro` e imprimimos el valor de `peso`, podemos ver que recuperamos el valor al crear la instancia.

Sin embargo, al acceder al método de clase, obtendremos siempre el valor por defecto asignado a `peso`.

Como puedes ver, la diferencia entre uno y otro es que gracias al método de clase podemos acceder al valor inicial del atributo.

# Métodos en Python: instancia, clase y estáticos

## Métodos estáticos (staticmethod)

Por último, los métodos estáticos se pueden definir con el decorador `@staticmethod` y no aceptan como parámetro ni la instancia ni la clase. Es por ello por lo que no pueden modificar el estado ni de la clase ni de la instancia. Pero por supuesto pueden aceptar parámetros de entrada.

```
class Clase:
    @staticmethod
    def metodoestatico():
        return "Método estático"
```

```
mi_clase = Clase()
Clase.metodoestatico()
mi_clase.metodoestatico()
```

```
# 'Método estático'
# 'Método estático'
```

Por lo tanto el uso de los métodos estáticos pueden resultar útil para indicar que un método no modificará el estado de la instancia ni de la clase. Es cierto que se podría hacer lo mismo con un método de instancia por ejemplo, pero a veces resulta importante indicar de alguna manera estas peculiaridades, evitando así futuros problemas y malentendidos.

En otras palabras, los métodos estáticos se podrían ver como funciones normales, con la salvedad de que van ligadas a una clase concreta.

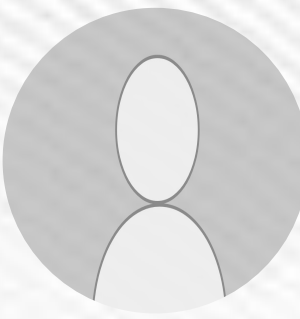


### Static Method

1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables

Veamos un ejemplo...

# Métodos en Python: instancia, clase y estáticos



## Métodos estáticos (staticmethod) cont...

Los métodos estáticos pertenecen a la clase, aunque no dependen ni de la clase ni de una instancia de esta. Por lo tanto, en este caso no necesitaremos ningún parámetro principal como sucede con los métodos normales o los de clase. Este tipo de métodos se utilizan normalmente cuando están ligados de alguna forma a la clase, por lo que no vale la pena tenerlos en un módulo separado de esta.

Para declarar un método estático, solo necesitamos usar el decorador `@staticmethod` y nuestro método estático. Un ejemplo podría ser una clase que se encargase de realizar operaciones matemáticas sencillas:

```
class Math:
```

```
    @staticmethod
    def sumar(num1, num2):
        return num1 + num2
```

```
    @staticmethod
    def restar(num1, num2):
        return num1 - num2
```

```
    @staticmethod
    def multiplicar(num1, num2):
        return num1 * num2
```

```
    @staticmethod
    def dividir(num1, num2):
        return num1 / num2
```

```
print(Math.sumar(5, 7))
print(Math.restar(9, 3))
print(Math.multiplicar(15, 9))
print(Math.dividir(10, 2))
```

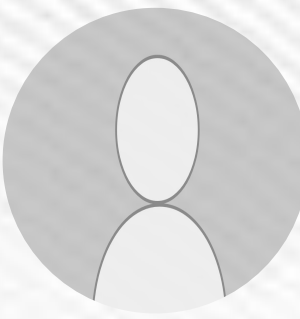
*“Como se puede ver, en ningún caso necesitamos crear una instancia de la clase y si tuviéramos métodos de otro tipo en la clase, podríamos acceder a estos sin problema.”*

### Static Method

1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables



# Métodos en Python: instancia, clase y estáticos



En otros posts hemos visto como se pueden crear métodos con def dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia. Pues bien, haciendo uso de los decoradores, es posible crear diferentes tipos de métodos:

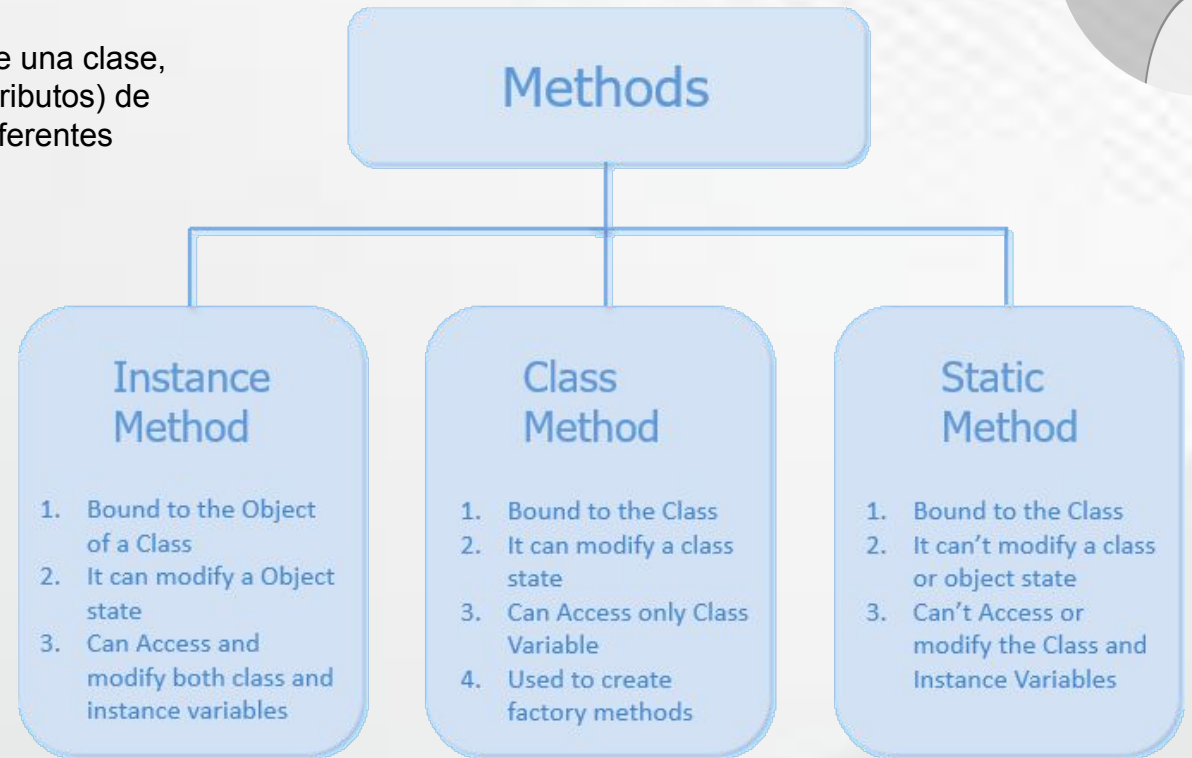
- Los métodos de **instancia** “normales” que ya hemos visto como metodo()
- Métodos de **clase** usando el decorador @classmethod
- Métodos **estáticos** usando el decorador @staticmethod

ass Clase:

```
def metodo(self):  
    return 'Método normal', self
```

```
@classmethod  
def metododeclase(cls):  
    return 'Método de clase', cls
```

```
@staticmethod  
def metodoestatico():  
    return "Método estático"
```



ask ai  
Powered By  
ChatGPT & Python



13 Py





# PYTHON

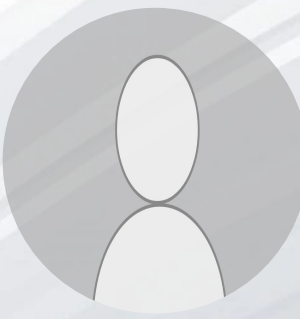
POO



HERENCIA EN PYTHON  
HERENCIA MULTIPLE  
MRO  
super()  
\_\_base\_\_  
\_\_subclasses\_\_

Stage 2

# Herencia en Python



La herencia es un proceso mediante el cual se puede crear una clase hija que hereda de una clase padre, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobrescribir los métodos o atributos, o incluso definir unos nuevos.

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar. En el siguiente ejemplo vemos como se puede usar la herencia en Python, con la clase Perro que hereda de Animal. Así de fácil.

```
# Definimos una clase padre
class Animal:
    pass
```

```
# Creamos una clase hija que hereda de la padre
class Perro(Animal):
    pass
```

De hecho podemos ver como efectivamente la clase Perro es la hija de Animal usando `__bases__`.

```
print(Perro.__bases__)
# (<class '__main__.Animal'>,)
```

De manera similar podemos ver que clases descienden de una en concreto con `__subclasses__`.

```
print(Animal.__subclasses__())
# [<class '__main__.Perro'>]
```

¿Y para que queremos la herencia? Dado que una clase hija hereda los atributos y métodos de la padre, nos puede ser muy útil cuando tengamos clases que se parecen entre sí pero tienen ciertas particularidades. En este caso en vez de definir un montón de clases para cada animal, podemos tomar los elementos comunes y crear una clase Animal de la que hereden el resto, respetando por tanto la filosofía DRY. Realizar estas abstracciones y buscar el denominador común para definir una clase de la que hereden las demás, es una tarea de lo más compleja en el mundo de la programación.

*Para saber más: El principio DRY (Don't Repeat Yourself) es muy aplicado en el mundo de la programación y consiste en no repetir código de manera innecesaria. Cuanto más código duplicado exista, más difícil será de modificar y más fácil será crear inconsistencias. Las clases y la herencia a no repetir código.*



Powered By  
ChatGPT & Python



# Extendiendo y modificando métodos

Continuemos con nuestro ejemplo de perros y animales. Vamos a definir una clase padre Animal que tendrá todos los atributos y métodos genéricos que los animales pueden tener. Esta tarea de buscar el denominador común es muy importante en programación. Veamos los atributos:

Tenemos la especie ya que todos los animales pertenecen a una.  
Y la edad, ya que todo ser vivo nace, crece, se reproduce y muere.  
Y los métodos o funcionalidades:

Tendremos el método hablar, que cada animal implementará de una forma. Los perros ladran, las abejas zumban y los caballos relinchan.  
Un método moverse. Unos animales lo harán caminando, otros volando.  
Y por último un método descríbeme que será común.  
Definimos la clase padre, con una serie de atributos comunes para todos los animales como hemos indicado.

class Animal:

```
def __init__(self, especie, edad):  
    self.especie = especie  
    self.edad = edad
```

```
# Método genérico pero con implementación particular  
def hablar(self):  
    # Método vacío  
    pass
```

```
# Método genérico pero con implementación particular  
def moverse(self):  
    # Método vacío  
    pass
```

```
# Método genérico con la misma implementación  
def describeme(self):  
    print("Soy un Animal del tipo", type(self).__name__)
```

Herencia y Extensión de Métodos

```
class Perro(Animal):  
    def hablar(self):  
        print("Guau!")  
    def moverse(self):  
        print("Caminando con 4 patas")
```



# Extendiendo y modificando métodos

Tenemos ya por lo tanto una clase genérica `Animal`, que generaliza las características y funcionalidades que todo animal puede tener. Ahora creamos una clase `Perro` que hereda del `Animal`. Como primer ejemplo vamos a crear una clase vacía, para ver como los métodos y atributos son heredados por defecto.

```
# Perro hereda de Animal
class Perro(Animal):
    pass

mi_perro = Perro('mamífero', 10)
mi_perro.describeme()
# Soy un Animal del tipo Perro
```

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando con 4 patas")
```

```
Class Animal:
    def hablar(self):
    def moverse(self):
    def describirme(self):
```

```
class Vaca(Animal):
    def hablar(self):
        print("Muuu!")
    def moverse(self):
        print("Caminando con 4 patas")
```

```
class Abeja(Animal):
    def hablar(self):
        print("Bzzzz!")
    def moverse(self):
        print("Volando")
```

Con tan solo un par de líneas de código, hemos creado una clase nueva que tiene todo el contenido que la clase padre tiene, pero aquí viene lo que es de verdad interesante. Vamos a crear varios animales concretos y sobrescribir algunos de los métodos que habían sido definidos en la clase `Animal`, como el `hablar` o el `moverse`, ya que cada animal se comporta de una manera distinta. Podemos incluso crear nuevos métodos que se añadirán a los ya heredados, como en el caso de la `Abeja` con `picar()`.

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad

# Método genérico pero con implementación particular
def hablar(self):
    # Método vacío
    pass

# Método genérico pero con implementación particular
def moverse(self):
    # Método vacío
    pass

# Método genérico con la misma implementación
def describeme(self):
    print("Soy un Animal del tipo", type(self).__name__)
```

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando con 4 patas")
```

```
class Vaca(Animal):
    def hablar(self):
        print("Muuu!")
    def moverse(self):
        print("Caminando con 4 patas")
```

```
class Abeja(Animal):
    def hablar(self):
        print("Bzzzz!")
    def moverse(self):
        print("Volando")
```

```
# Nuevo método
def picar(self):
    print("Picar!")
```

# Extendiendo y modificando métodos



Por lo tanto ya podemos crear nuestros objetos de esos animales y hacer uso de sus métodos que podrían clasificarse en tres:

Heredados directamente de la clase padre: describeme()

Heredados de la clase padre pero modificados: hablar() y moverse()

Creados en la clase hija por lo tanto no existentes en la clase padre: picar()

```
mi_perro = Perro('mamífero', 10)
mi_vaca = Vaca('mamífero', 23)
mi_abeja = Abeja('insecto', 1)
```

```
mi_perro.hablar()
mi_vaca.hablar()
# Guau!
# Muuu!
```

```
mi_vaca.describeme()
mi_abeja.describeme()
# Soy un Animal del tipo Vaca
# Soy un Animal del tipo Abeja
```

```
mi_abeja.picar()
# Picar!
```

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando con 4 patas")
```

```
class Vaca(Animal):
    def hablar(self):
        print("Muuu!")
    def moverse(self):
        print("Caminando con 4 patas")
```

```
class Abeja(Animal):
    def hablar(self):
        print("Bzzzz!")
    def moverse(self):
        print("Volando")
```



# Uso de super()

En pocas palabras, la función `super()` nos permite acceder a los métodos de la clase padre desde una de sus hijas. Volvamos al ejemplo de `Animal` y `Perro`.

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad
    def hablar(self):
        pass

    def moverse(self):
        pass

    def describeme(self):
        print("Soy un Animal del tipo", type(self).__name__)
```

Tal vez queramos que nuestro `Perro` tenga un parámetro extra en el constructor, como podría ser el dueño.

Para realizar esto tenemos dos alternativas: Podemos crear un nuevo `__init__` y guardar todas las variables una a una.

O podemos usar `super()` para llamar al `__init__` de la clase padre que ya aceptaba la especie y edad, y sólo asignar la variable nueva manualmente.

```
class Perro(Animal):
    def __init__(self, especie, edad, dueño):
        # Alternativa 1
        # self.especie = especie
        # self.edad = edad
        # self.dueño = dueño
```

```
        # Alternativa 2
        super().__init__(especie, edad)
        self.dueño = dueño
```

```
mi_perro = Perro('mamífero', 7, 'Luis')
mi_perro.especie
mi_perro.edad
mi_perro.dueño
```



# Herencia múltiple

**En Python es posible realizar herencia múltiple.** En otras diapositivas vimos como se podía crear una clase padre que heredaba de una clase hija, pudiendo hacer uso de sus métodos y atributos. La herencia múltiple es similar, pero una clase hereda de varias clases padre en vez de una sola.

Veamos un ejemplo. Por un lado tenemos dos clases Clase1 y Clase2, y por otro tenemos la Clase3 que hereda de las dos anteriores. Por lo tanto, heredará todos los métodos y atributos de ambas.

```
class Clase1:  
    pass  
class Clase2:  
    pass  
class Clase3(Clase1, Clase2):  
    pass
```

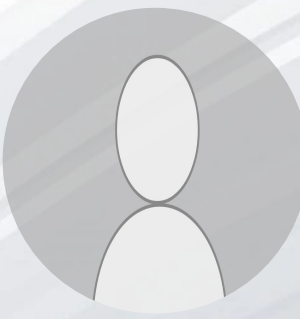
Es posible también que una clase herede de otra clase y a su vez otra clase herede de la anterior.

```
class Clase1:  
    pass  
class Clase2(Clase1):  
    pass  
class Clase3(Clase2):  
    pass
```

Llegados a este punto nos podemos plantear lo siguiente. Las clases hijas heredan los métodos de las clases padre, pero también pueden reimplementarlos de manera distinta. Entonces, si llamo a un método que todas las clases tienen en común ¿a cuál se llama?. Pues bien, existe una forma de saberlo.



# Herencia múltiple - MRO



La forma de saber a que método se llama es consultar el MRO o Method Resolution Order. Esta función nos devuelve una tupla con el orden de búsqueda de los métodos. Como era de esperar se empieza en la propia clase y se va subiendo hasta la clase padre, de izquierda a derecha.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass
```

```
print(Clase3.__mro__)
# (<class '__main__.Clase3'>, <class '__main__.Clase1'>, <class '__main__.Clase2'>, <class 'object'>)
```

Una curiosidad es que al final del todo vemos la clase object. Aunque pueda parecer raro, es correcto ya que en realidad todas las clases en Python heredan de una clase genérica object, aunque no lo especifiquemos explícitamente.

Y como último ejemplo,...el cielo es el límite. Podemos tener una clase heredando de otras tres. Fíjate en que el MRO depende del orden en el que las clases son pasadas: 1, 3, 2.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3:
    pass
class Clase4(Clase1, Clase3, Clase2):
    pass
print(Clase4.__mro__)
# (<class '__main__.Clase4'>, <class '__main__.Clase1'>, <class '__main__.Clase3'>, <class '__main__.Clase2'>, <class 'object'>)
```

Junto con la herencia, la cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento son otros de los conceptos claves para entender la programación orientada a objetos.



14 Py







POO @ DECORADORES EN PYTHON



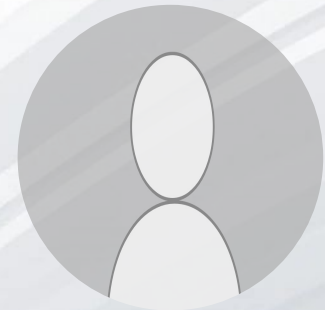
@property

—  
.setter

Stage 3



# Decorador Property ( @property )



Veremos el decorador @property, que viene por defecto con Python, y puede ser usado para modificar un método para que sea un atributo o propiedad. Es importante que conozcan antes la programación orientada a objetos.

*El decorador puede ser usado sobre un método, que hará que actúe como si fuera un atributo.*

```
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo

    @property
    def mi_atributo(self):
        return self.__mi_atributo
```

Como si de un atributo normal se tratase, podemos acceder a el con el objeto . y nombre.

```
mi_clase = Clase("valor_atributo")
mi_clase.mi_atributo
# 'valor_atributo'
```

Muy importante notar que aunque mi\_atributo pueda parecer un método, en realidad no lo es, por lo que no puede ser llamado con ().

```
# mi_clase.mi_atributo()
# Error! Es un atributo, no un método
```

Tal vez te preguntes para que sirve esto, ya que el siguiente código hace exactamente lo mismo sin hacer uso de decoradores.

```
class Clase:
    def __init__(self, mi_atributo):
        self.mi_atributo = mi_atributo
```

```
mi_clase = Clase("valor_atributo")
mi_clase.mi_atributo
# 'valor_atributo'
```



# Decorador Property ( \_\_ )

*Bien, la explicación no es sencilla, pero está relacionada con el concepto de **encapsulación** de la programación orientada a objetos. Este concepto nos indica que en determinadas ocasiones es importante ocultar el estado interno de los objetos al exterior, para evitar que sean modificados de manera incorrecta. Para la gente que venga del mundo de C++ o Java, esto no será nada nuevo, y está muy relacionado con los métodos `set()` y `get()` que veremos a continuación.*

La primera diferencia que vemos entre los códigos anteriores es el uso de `__` antes de `mi_atributo`. Cuando nombramos una variable de esta manera, es una forma de decirle a Python que queremos que se **"oculte"** y que no pueda ser accedida como el resto de atributos.

```
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo
```

```
mi_clase = Clase("valor_atributo")
```

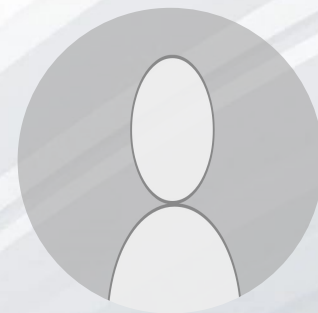
```
# mi_clase.__mi_atributo # Error!
```

Esto puede ser importante con ciertas variables que no queremos que sean accesibles desde el exterior de una manera no controlada. Al definir la propiedad con `@property` el acceso a ese atributo se realiza a través de una función, siendo por lo tanto un acceso controlado.

```
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo

    @property
    def mi_atributo(self):
        # El acceso se realiza a través de este "método" y
        # podría contener código extra y no un simple retorno
        return self.__mi_atributo
```

Otra utilidad podría ser la consulta de un parámetro que requiera de muchos cálculos. Se podría tener un atributo que no estuviera directamente almacenado en la clase, sino que precisara de realizar ciertos cálculos. Para optimizar esto, se podrían hacer los cálculos sólo cuando el atributo es consultado.





# Decorador Property ( setter )



Por último, existen varios añadidos al decorador `@property` como pueden ser el setter. Se trata de otro decorador que permite definir un “método” que modifica el contenido del atributo que se esté usando.

```
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo

    @property
    def mi_atributo(self):
        return self.__mi_atributo

    @mi_atributo.setter
    def mi_atributo(self, valor):
        if valor != "":
            print("Modificando el valor")
            self.__mi_atributo = valor
        else:
            print("Error está vacío")
```

De esta forma podemos añadir código al setter, haciendo que por ejemplo realice comprobaciones antes de modificar el valor. Esto es una cosa que de usar un atributo normal no podríamos hacer, y es muy útil de cara a la encapsulación.

```
mi_clase = Clase("valor_atributo")
mi_clase.mi_atributo
# 'valor_atributo'
```

```
mi_clase.mi_atributo = "nuevo_valor"
mi_clase.mi_atributo
# 'nuevo_valor'
```

```
mi_clase.mi_atributo = ""
# Error está vacío
```

Resulta lógico pensar que si un determinado atributo pertenece a una clase, si queremos modificarlo debería de tener la “aprobación” de la clase, para asegurarse que ninguna entidad externa está “haciendo cosas raras”.





POO

- ABSTRACCION
- ACOPLAMIENTO
- ENCAPSULAMIENTO
- COHESION
- EN PYTON



Stage 4



# Abstracción en programación

La **abstracción** es un termino que hace referencia a la ocultación de la complejidad intrínseca de una aplicación al exterior, centrándose sólo en como puede ser usada, lo que se conoce como **interface**.

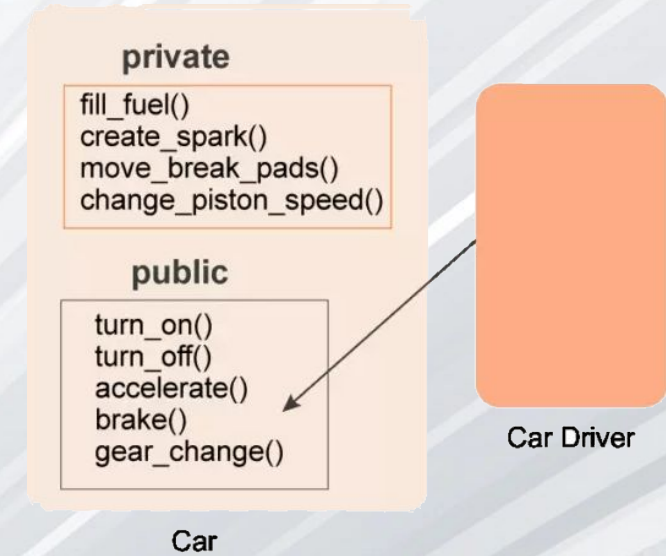
El enfoque caja negra, es un concepto muy relacionado. Dicho en otras palabras, la abstracción consiste en ocultar toda la complejidad que algo puede tener por dentro, ofreciéndonos unas funciones de alto nivel, por lo general sencillas de usar, que pueden ser usadas para interactuar con la aplicación sin tener conocimiento de lo que hay dentro.

Una analogía del mundo real podría ser la televisión. Se trata de un dispositivo muy complejo donde han trabajado gran cantidad de ingenieros de diversas disciplinas como telecomunicaciones o electrónica. ¿Os imagináis que para cambiar de canal tuviéramos que saber todos los entresijos del aparato?. Pues bien, se nos ofrece una abstracción de la televisión, un mando a distancia. El mando nos abstrae por completo de la complejidad de los circuitos y señales, y nos da una interface sencilla que con unos pocos botones podemos usar.

Algo muy parecido sucede en la programación orientada a objetos. Si tuviéramos una clase Televisor, en su interior podría haber líneas y líneas de código super complejas, pero una buena abstracción sería la que simplemente ofreciera los métodos encender(), apagar() y cambiar\_canal() al exterior.

Un concepto relacionado con la abstracción, serían las clases abstractas o más bien los métodos abstractos. Se define como clase abstracta la que contiene métodos abstractos, y se define como método abstracto a un método que ha sido declarado pero no implementado. Es decir, que no tiene código.

*Es posible crear métodos abstractos en Python con decoradores como `@abstractmethod`.*



## Types of Abstraction

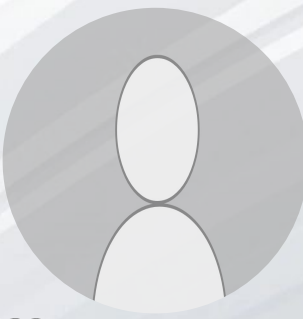
Data  
Abstraction

Process  
Abstraction

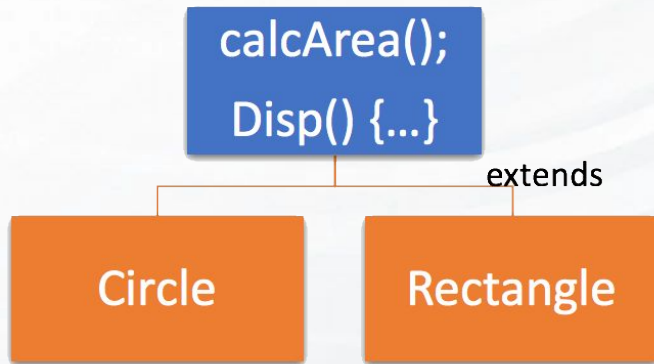


# Tipos de Abstracción en POO

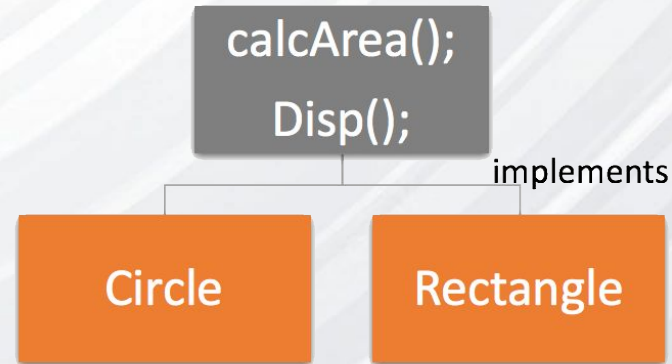
La **abstracción** es un método para **ocultar los detalles de implementación** y **mostrar solo la funcionalidad**, básicamente Proporciona pautas para construir un producto estándar.



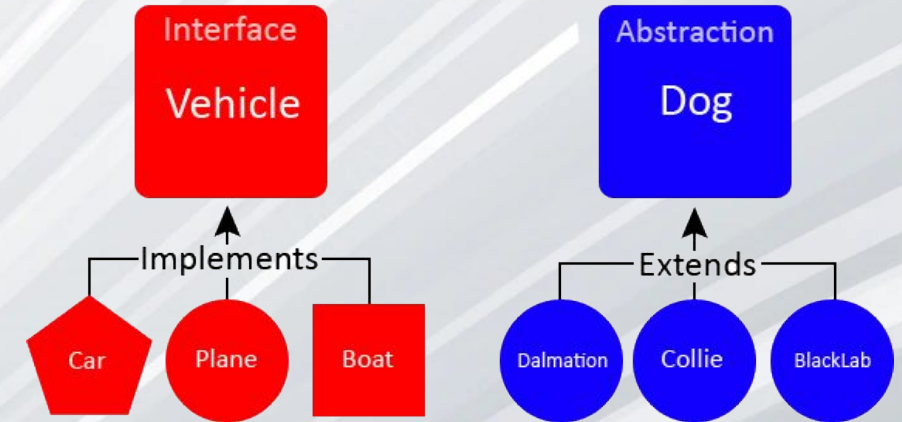
## Abstract



## Interface



## Interfaces vs. Abstract Classes



### Abstract class

- La clase abstracta tendrá una combinación de métodos implementados y no implementados
- Sintaxis – Depende del Lenguaje (conceptualice, modele e implemente)
- Una clase abstracta puede o no contener métodos abstractos
- Si al menos un método es abstracto, entonces debe declararse como abstracta
- No se puede crear un objeto de una clase abstracta
- La clase secundaria que hereda la clase abstracta con métodos abstractos, debe implementar todos los métodos abstractos, caso contrario, será Abstracta.
- La clase abstracta puede tener constructores

### Interface

- La interfaz debe tener todos los métodos no implementados.
- Sintaxis – (Java interfaz InterfaceName)
- Todos los métodos en la interfaz son abstractos
- No se puede crear un objeto de una interfaz
- Todos los campos de una interfaz deben declararse (como finales estáticos)
- Una interfaz es implementada por una clase
- Una interfaz se extiende por múltiples interfaces

# Interfaces & Abstract Classes



Las interfaces me permiten a mí (Cliente), un consumidor externo, interactuar con el sistema detrás de él.  
De esta forma:

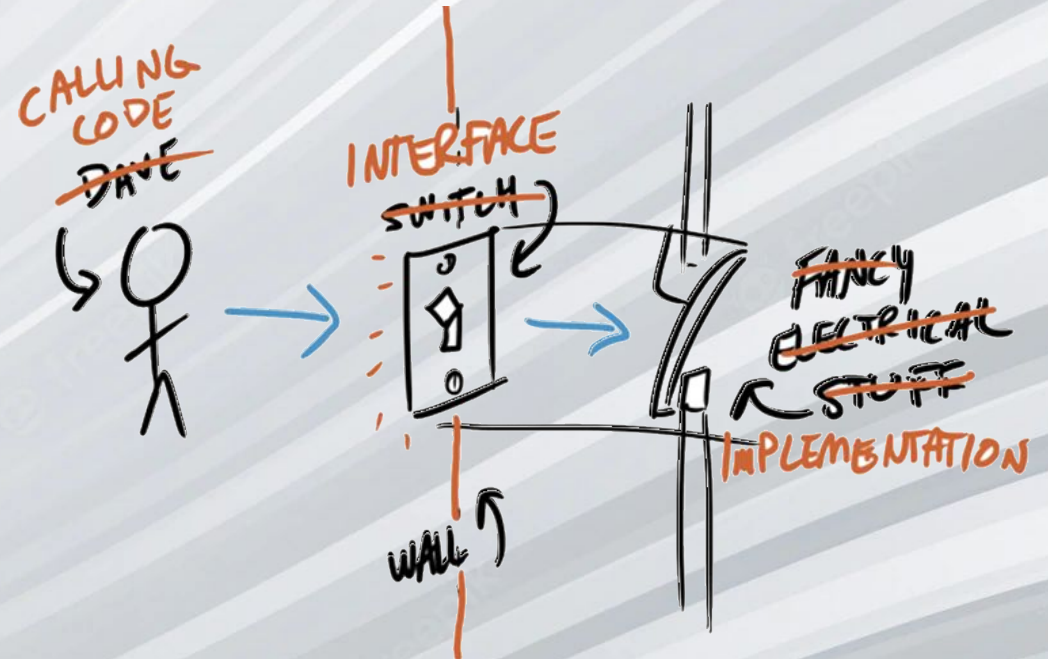
- El interruptor de la luz es la interfaz que me permite encender o apagar una luz.
- El cambio de marchas en mi coche es la interfaz que me permite interactuar con la transmisión. (¡Sí, soy un tipo de transmisión manual!)
- El ojo de la cerradura es mi interfaz con el sistema de bloqueo de mi puerta. Es con lo que interactúo para cerrar o desbloquear mi puerta.

Es importante notar que, en estos tres casos, no tuve que abrir nada para lograr mi objetivo. Por ejemplo, no tuve que desenroscar la perilla de la puerta y hurgar dentro para activar la cerradura. En su lugar, solo uso la interfaz: el ojo de la cerradura.

En cada ejemplo anterior, el objeto tiene dos lados.

1. La parte con la que interactúo, por ejemplo, el interruptor de la luz
2. La parte que hace lo que estoy tratando de lograr, por ejemplo, la mecánica de completar o romper el circuito eléctrico.

En términos generales, una interfaz consiste en cualquier método y variable a la que se puede acceder fuera de la propia clase. Usando una comprensión bastante ingenua de la ingeniería eléctrica, echemos un vistazo...





# Interfaces Classes

Un método consta de dos partes: operación más implementación.

**Una interfaz es “tan solo” una colección de operaciones.**

Las interfaces pueden ampliar otras interfaces. En este caso, La interfaz de la extensión hereda todas las operaciones de la interfaz extendida. Por otro lado, una clase C implementa una interfaz I si C implementa todos los las operaciones en I.

**Por lo tanto, podemos hacer una interfaz como la especificación de una o más clases.**

En este ejemplo, la clase TapePlayer debe implementar las operaciones en la interfaz Recorder. Esto incluye la operación record(), así como las cuatro operaciones heredadas.

## Java Notation

Java provides special notation for declaring and implementing interfaces:

```
interface Player {  
    void play();  
    void stop();  
    void pause();  
    void reverse();  
}
```

```
interface Recorder extends Player {  
    void record();  
}
```

```
class TapePlayer implements Recorder {  
    public void play() { ... }  
    public void stop() { ... }  
    public void pause() { ... }  
    public void reverse() { ... }  
    public void record() { ... }  
}
```

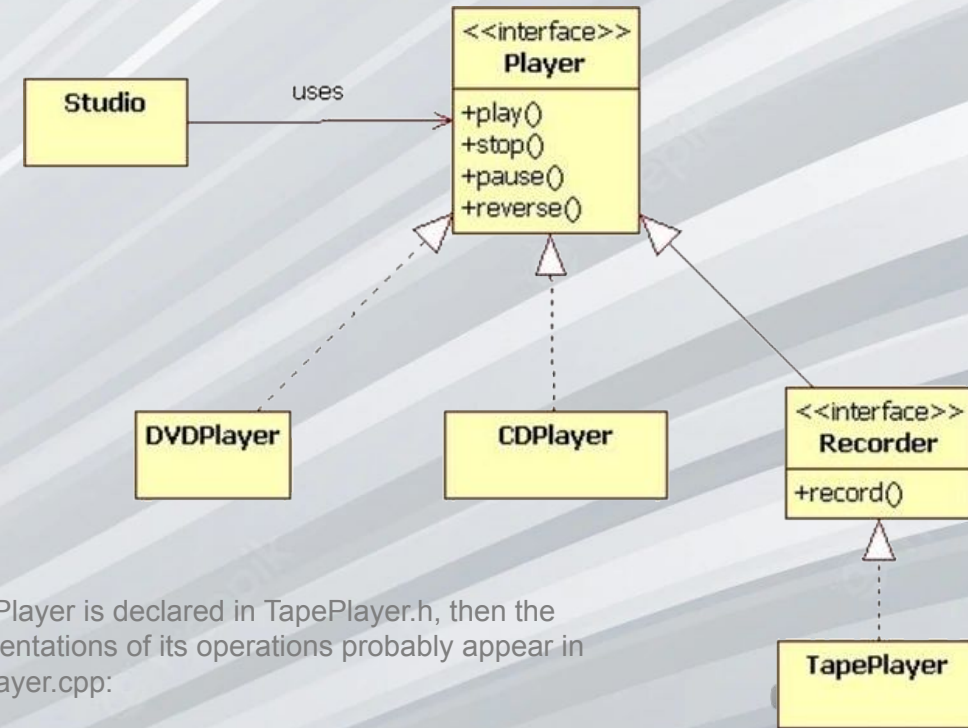
## C++ Notation

In C++ an interface is a class containing public pure virtual functions:

```
class Player {  
public:  
    virtual void play() = 0;  
    virtual void stop() = 0;  
    virtual void pause() = 0;  
    virtual void reverse() = 0;  
};
```

```
class Recorder: public Player {  
public:  
    virtual void record() = 0;  
};
```

```
class TapePlayer: public Recorder {  
public:  
    void play();  
    void stop();  
    void pause();  
    void reverse();  
    void record();  
};
```



If TapePlayer is declared in TapePlayer.h, then the implementations of its operations probably appear in TapePlayer.cpp:



# Abstract Classes

Una interfaz no contiene variables ni métodos. Las variables y los métodos deben declararse mediante la implementación de clases. Esto puede llevar a Replicación de código cuando una variable o método es común a todas las implementaciones Clases. Por ejemplo, supongamos que todos los jugadores tienen un estado que algunos de Los métodos deben establecer (además de otras tareas posibles):

En cada una de las tres clases de implementación necesitaríamos Introduzca una variable de estado y un código para establecer la variable.

En cada una de las tres clases de implementación necesitaríamos introducir una variable de estado y un código para establecer la variable.:

Here's the Java declaration:

```
abstract class Player {  
    static enum State {STOPPED, PLAYING, PAUSED}  
    private State state = STOPPED;  
    public void play() { state = PLAYING; }  
    public void pause() { state = PAUSED; }  
    public void stop() { state = STOPPED; }  
    public abstract reverse();  
}
```

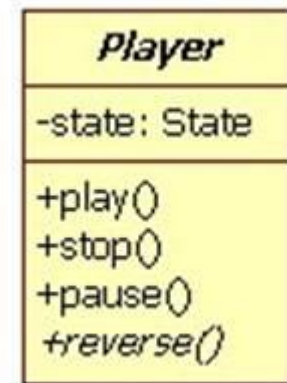
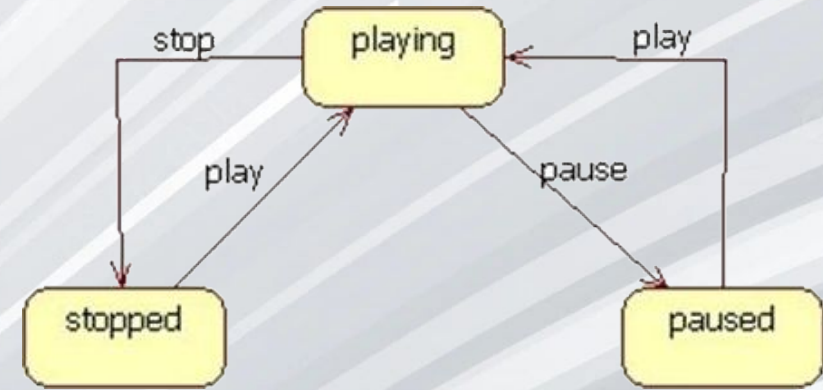
Here's the C++ declaration:

```
class Player {  
private:  
    enum State {STOPPED, PLAYING, PAUSED};  
    State state;  
public:  
    virtual void play() = 0;  
    virtual void stop() = 0;  
    virtual void pause() = 0;  
    virtual void reverse() = 0;  
};
```

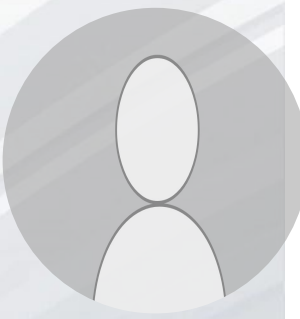
A pesar de que play, stop y pause se declaran funciones virtuales puras, todavía podemos proporcionar implementaciones en Player.cpp:

```
void Player::play() { state = PLAYING; }  
void Player::stop() { state = STOPPED; }  
void Player::pause() { state = PAUSED; }
```

In UML abstract classes and methods are italicized:



# Abstract Classes



```
abstract class Player {  
    static enum State {STOPPED, PLAYING, PAUSED}  
    private State state = STOPPED;  
    public void play() { state = PLAYING; }  
    public void pause() { state = PAUSED; }  
    public void stop() { state = STOPPED; }  
    public abstract reverse();  
}
```

Here's the C++ declaration:

```
class Player {  
private:  
    enum State {STOPPED, PLAYING, PAUSED};  
    State state;  
public:  
    virtual void play() = 0;  
    virtual void stop() = 0;  
    virtual void pause() = 0;  
    virtual void reverse() = 0;  
};
```

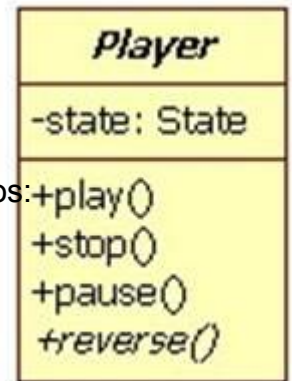
A pesar de que play, stop y pause se declaran funciones virtuales puras, todavía podemos proporcionar implementaciones en Player.cpp:

```
void Player::play() { state = PLAYING; }  
void Player::stop() { state = STOPPED; }  
void Player::pause() { state = PAUSED; }
```

In UML abstract classes and methods are italicized:

Las clases de implementación pueden redefinir los métodos heredados y llamarlos:

```
class DVDPlayer extends Player {  
    public void play() {  
        super.play(); // change state  
        // DVD-specific play code goes here  
    }  
    // etc.  
}
```

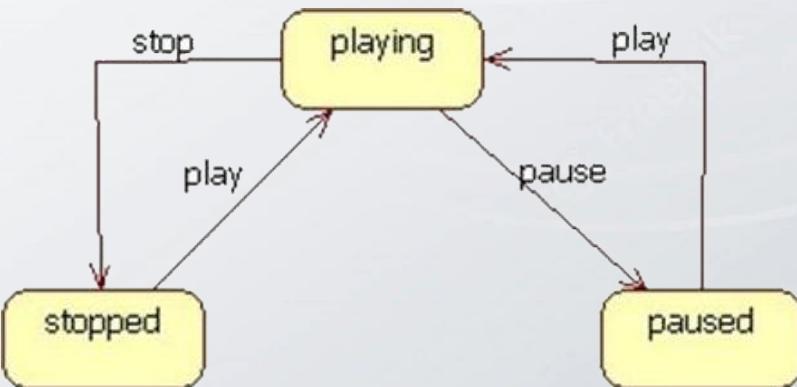


No se pueden crear instancias de clases abstractas:

```
Player p = new Player(); // error
```

Una clase es abstracta si contiene un método abstracto. Esto puede suceder si una clase declara un método abstracto o hereda uno y no lo implementa.

Supongamos que el implementador de DVDPlayer olvida implementar el método abstracto inverso. Entonces DVDPlayer es una clase abstracta. En Java la compilación se queja de que DVDPlayer debe ser declarado abstracto. En C++, el compilador se queja cuando ve código que intenta crear instancias de DVDPlayer.





# Acoplamiento en programación

El **acoplamiento** en programación (denominado *coupling* en Inglés) es un concepto que mide la dependencia entre dos módulos distintos de software, como pueden ser por ejemplo las clases.

El acoplamiento puede ser de dos tipos:

**Acoplamiento débil**, que indica que no existe dependencia de un módulo con otros. Esto debería ser la meta de nuestro software.

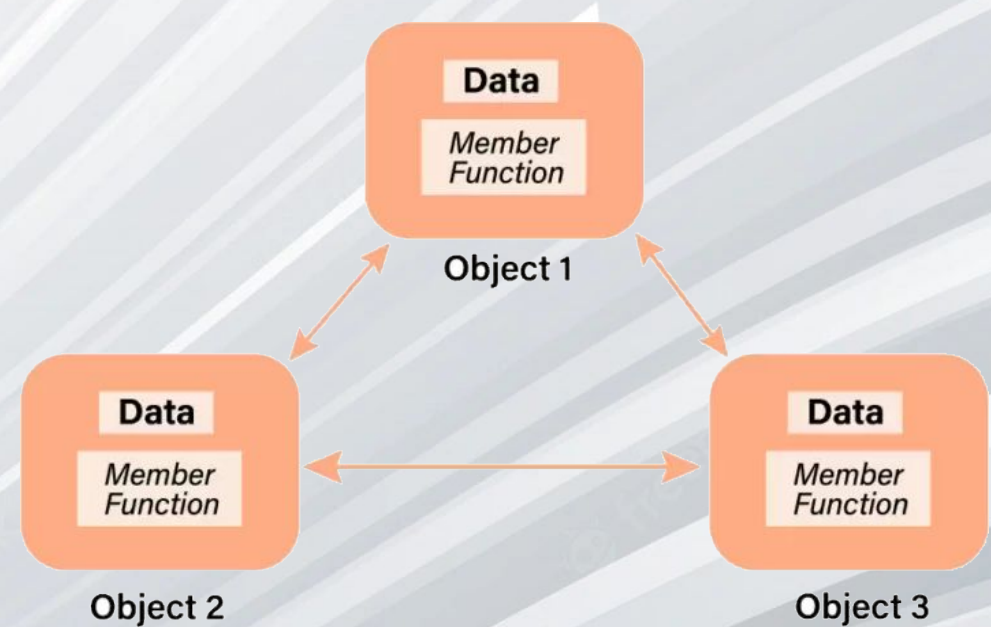
**Acoplamiento fuerte**, que por lo contrario indica que un módulo tiene dependencias internas con otros.

El término acoplamiento está muy relacionado con la cohesión, ya que acoplamiento débil suele ir ligado a cohesión fuerte. En general lo que buscamos en nuestro código es que tenga **acoplamiento débil y cohesión fuerte**, es decir, que no tenga dependencias con otros módulos y que las tareas que realiza estén relacionadas entre sí. Un código así es fácil de leer, de reusar, mantener y tiene que ser nuestra meta. Nótese que se suele emplear alta y baja para designar fuerza y débil respectivamente.

Si aún no te hemos convencido de porque buscamos código débilmente acoplado, veamos lo que pasaría con un código fuertemente acoplado:

Debido a las dependencias con otros módulo, un cambio en un modulo ajeno al nuestro podría tener un “efecto mariposa” en nuestro código, aún sin haber modificado directamente nuestro módulo.

Si un módulo tiene dependencias con otros, reduce la reusabilidad, ya que para reusarlo deberíamos copiar también las dependencias.





# Acoplamiento en programación

Veamos un ejemplo usando clases y objetos en Python. Tenemos una Clase1 que define un atributo de clase x. Por otro lado la Clase2 basa el comportamiento del método mi\_metodo() en el valor de x de la Clase1. En este ejemplo existe acoplamiento fuerte, ya que existe una dependencia con una variable de otro módulo.

```
class Clase1:
    x = True
    pass

class Clase2:
    def mi_metodo(self, valor):
        if Clase1.x:
            self.valor = valor

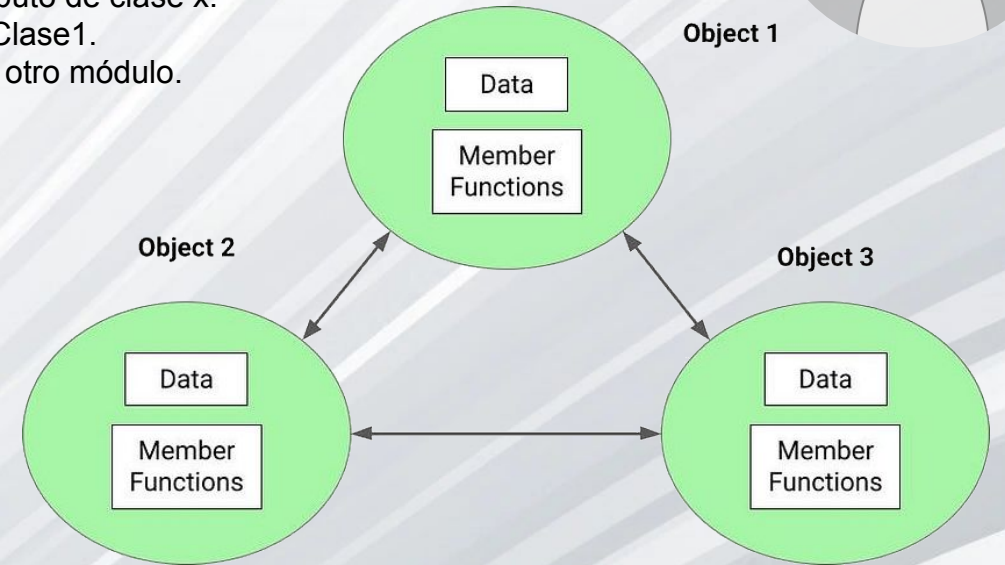
mi_clase = Clase2()
mi_clase.mi_metodo("Hola")
mi_clase.valor
```

Puede parecer un ejemplo trivial, pero cuando el software se va complicando, no es nada raro acabar haciendo cosas de este tipo casi sin darnos cuenta. Hay veces que dependencias externas pueden estar justificadas, pero hay que estar muy seguro de lo que se hace.

Este tipo de dependencias también puede hacer el código muy difícil de depurar. Imaginemos que nuestro código de la Clase2 funciona perfectamente, pero de repente alguien hace un cambio en la Clase1. Un cambio tan sencillo como el siguiente.

**Clase1.x = False**

Este cambio estaría modificando el comportamiento de nuestra clase y nos preguntaríamos ¿porqué ha dejado de funcionar mi código si no he tocado nada? A veces atribuimos estos comportamientos a la magia o radiación cósmica, pero simplemente tenemos código con acoplamiento fuerte.



# Cohesión en Programación

La **cohesión** hace referencia al grado de relación entre los elementos de un módulo.

En el diseño de una función, es importante pensar bien la tarea que va a realizar, intentando que sea única y bien definida. Cuantas más cosas diferentes haga una función sin relación entre sí, más complicado será el código de entender. Existen por lo tanto dos tipos de cohesión:

Por un lado tenemos la cohesión **débil** que indica que la relación entre los elementos es baja. Es decir, no pertenecen a una única funcionalidad. Por otro la cohesión **fuerte (Alta)**, que debe ser nuestro objetivo al diseñar programas.

***La cohesión fuerte indica que existe una alta relación entre los elementos existentes dentro del módulo.***

Veámoslo con un ejemplo. Tenemos una función suma() que suma dos números. El problema es que además de sumar dos números, los convierte a float() y además pide al usuario que introduzca por pantalla el número. Podría parecer que esas otras dos funcionalidades no son para tanto, pero si por ejemplo una persona quiere usar nuestra función suma() pero ya tiene los números y no quiere pedirlos por pantalla, no le serviría nuestra función.

# Mal. Cohesión débil

```
def suma():  
    num1 = float(input("Dame primer número"))  
    num2 = float(input("Dame segundo número"))  
    suma = num1 + num2  
    print(suma)
```

suma()

Para que la función tuviese una cohesión fuerte, sería conveniente que la suma realizara una única tarea bien definida, que es sumar.

# Bien. Cohesión fuerte

```
def suma(numeros):  
    total = 0  
    for i in numeros:  
        total = total + i  
    return total
```

Evidentemente un ejemplo tan sencillo como el explicado no tiene implicaciones demasiado graves, pero es importante buscar que las funciones realicen una única tarea (o conjunto) pero relacionadas entre sí. Diseñar código con cohesión fuerte nos permite: Reducir la complejidad del módulo, ya que tendrá un menor número de operaciones.

Se podrá reutilizar los módulos más fácilmente  
El sistema será más fácilmente mantenible.

Los Invito a la Próxima Unidad de buenas Practicas de Diseño Utilizando **S.O.L.I.D y Grasp**.



# Encapsulamiento en programación



El **encapsulamiento** o encapsulación en programación es un concepto relacionado con la programación orientada a objetos, y hace referencia al ocultamiento de los estado internos de una clase al exterior.

Dicho de otra manera, encapsular consiste en hacer que los atributos o métodos internos a una clase no se puedan acceder ni modificar desde fuera, sino que tan solo el propio objeto pueda acceder a ellos.

Para la gente que conozca C++ o Java, le resultará un termino muy familiar, pero en Python es algo distinto. Digamos que Python por defecto no oculta los atributos y métodos de una clase al exterior. Veamos un ejemplo con el lenguaje Python.

```
class Clase:  
    atributo_clase = "Hola"  
    def __init__(self, atributo_instancia):  
        self.atributo_instancia = atributo_instancia
```

```
mi_clase = Clase("Que tal")  
mi_clase.atributo_clase  
mi_clase.atributo_instancia
```

```
# 'Hola'  
# 'Que tal'
```



*Como lo solucionamos...*



# Encapsulamiento en programación ( doble \_\_ )



Ambos atributos son perfectamente accesibles desde el exterior. Sin embargo esto es algo que tal vez no queramos. Hay ciertos métodos o atributos que queremos que pertenezcan sólo a la clase o al objeto, y que sólo puedan ser accedidos por los mismos.

Para ello podemos **usar la doble \_\_ para nombrar a un atributo o método**.

Esto hará que Python los interprete como **“privados”**, de manera que no podrán ser accedidos desde el exterior.

class Clase:

```
    atributo_clase = "Hola" # Accesible desde el exterior
```

```
    __atributo_clase = "Hola" # No accesible
```

```
    # No accesible desde el exterior
```

```
    def __mi_metodo(self):
        print("Haz algo")
        self.__variable = 0
```

```
    # Accesible desde el exterior
```

```
    def metodo_normal(self):
        # El método si es accesible desde el interior
        self.__mi_metodo()
```

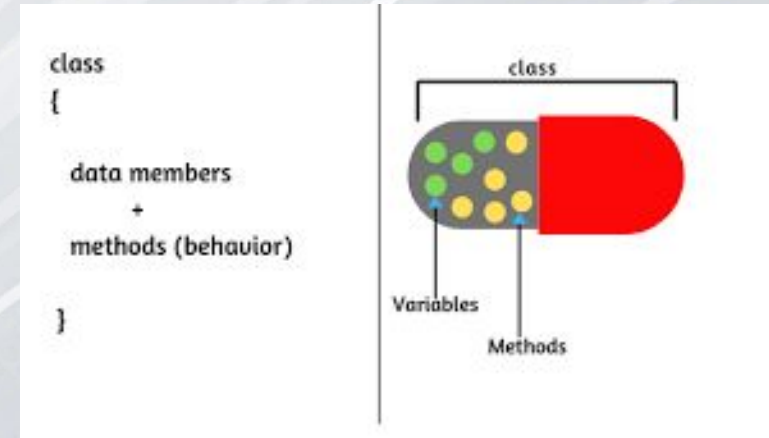
```
mi_clase = Clase()
```

```
#mi_clase.__atributo_clase # Error! El atributo no es accesible
```

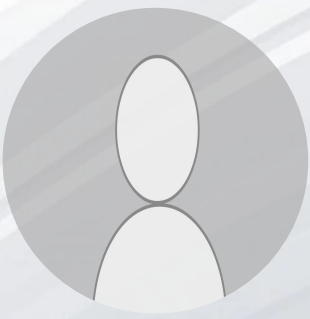
```
#mi_clase.__mi_metodo()    # Error! El método no es accesible
```

```
mi_clase.atributo_clase    # Ok!
```

```
mi_clase.metodo_normal()   # Ok!
```



# Encapsulamiento en programación ( dir )

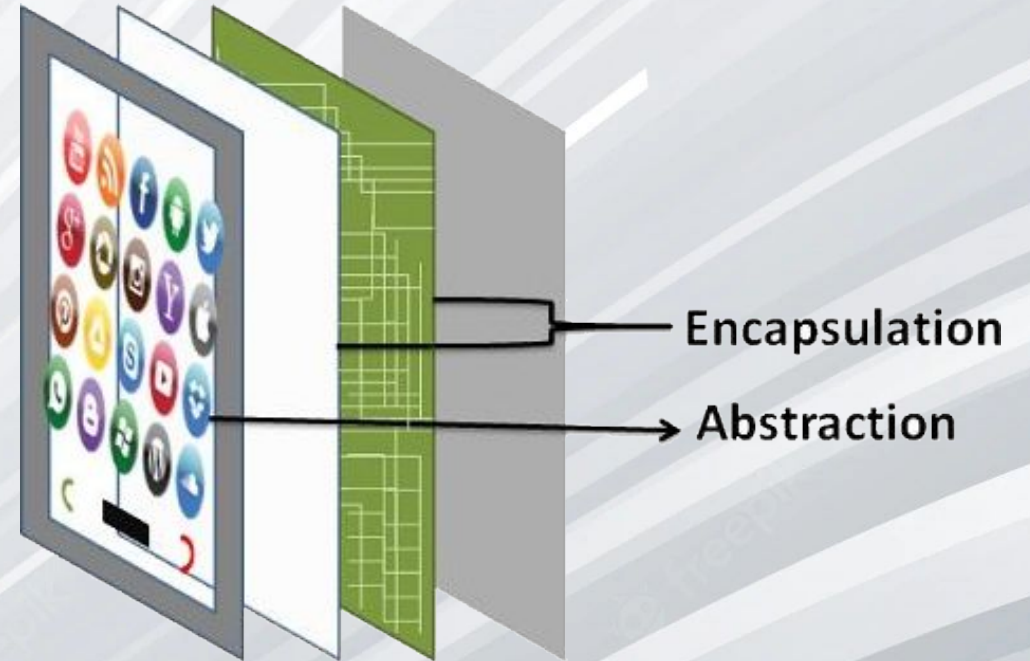


Y como curiosidad, podemos hacer uso de `dir` para ver el listado de métodos y atributos de nuestra clase. Podemos ver claramente como tenemos el `metodo_normal` y el atributo de clase, pero no podemos encontrar `__mi_metodo` ni `__atributo_clase`.

```
print(dir(mi_clase))
#['_Class__atributo_clase', '_Class__mi_metodo', '_Class__variable',
# '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
# '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',
# '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
# '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
# '__str__', '__subclasshook__', '__weakref__', 'atributo_clase', 'metodo_normal']
```

Pues bien, en realidad si que podríamos acceder a `__atributo_clase` y a `__mi_metodo` haciendo un poco de trampa. Aunque no se vea a simple vista, si que están pero con un nombre distinto, para de alguna manera ocultarlos y evitar su uso. Pero podemos llamarlos de la siguiente manera, pero por lo general no es una buena idea.

```
mi_clase._Class__atributo_clase
# 'Hola'
mi_clase._Class__mi_metodo()
# 'Haz algo'
```





POO POLIMORFISMO EN PYTHON



Stage 5





# Polimorfismo en programación

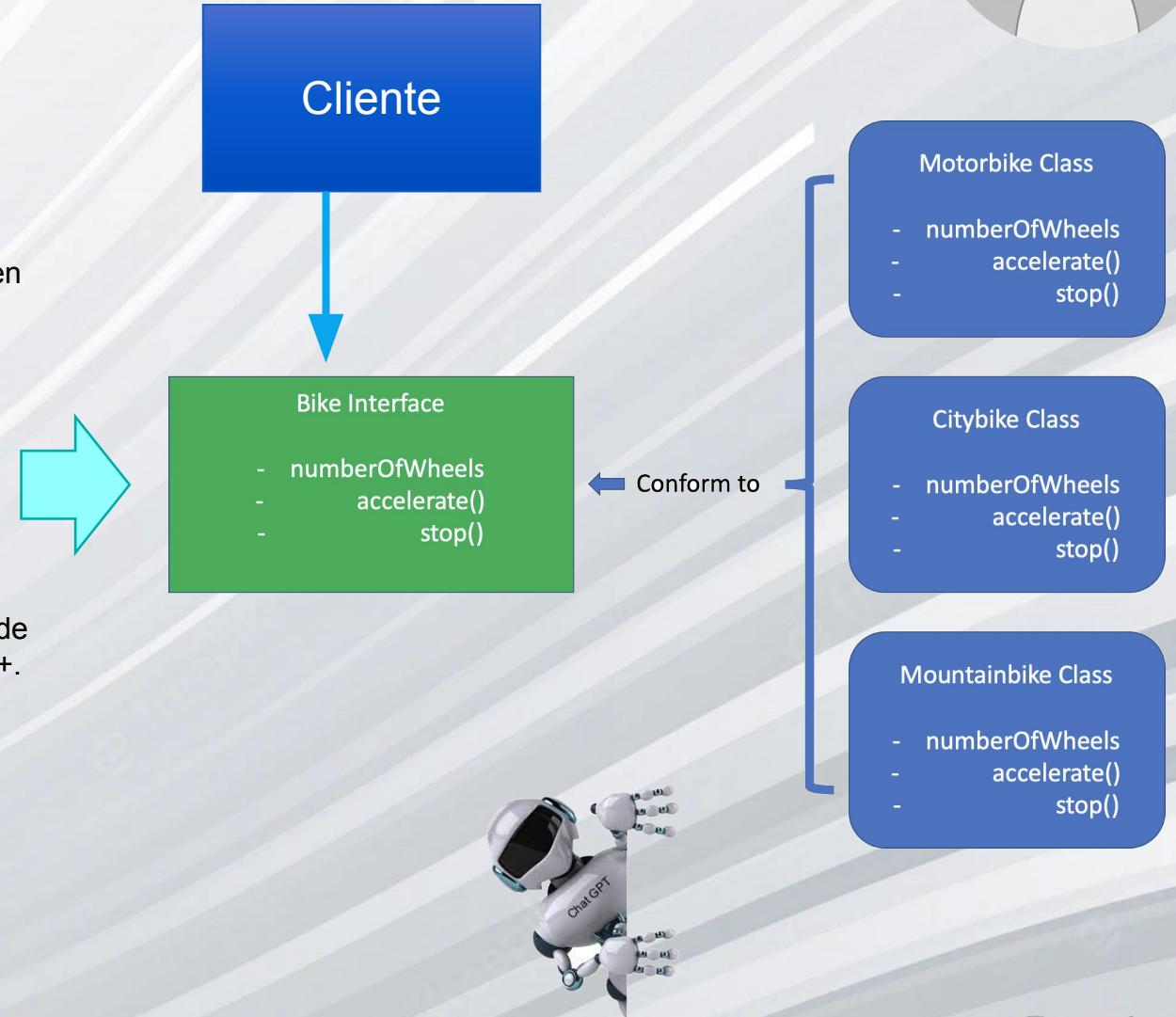
El polimorfismo es uno de los pilares básicos en la programación orientada a objetos, por lo que para entenderlo es importante tener las bases de la POO y la herencia bien asentadas.

El término polimorfismo tiene origen en las palabras poly (muchos) y morfo (formas), y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas. ¿Pero qué significa esto?

*Pues bien, significa que objetos de diferentes clases pueden ser accedidos utilizando el mismo interface, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos.*

Sin embargo, para entender bien este concepto, es conveniente explicarlo desde el punto de vista de un lenguaje de programación con tipado estático como C++. Vamos a por ello.

\*En lenguajes de programación como Python, que tiene tipado dinámico, el polimorfismo va muy relacionado con el **duck typing**.



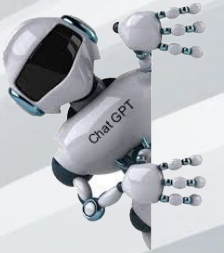
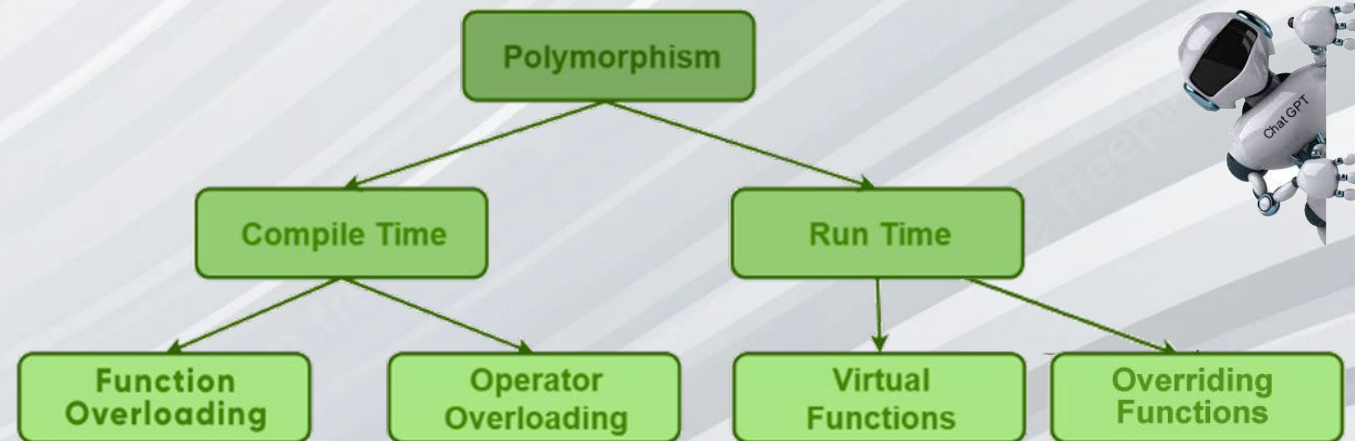


# Polimorfismo en C++

La palabra "polimorfismo" significa tener muchas formas. En palabras simples, podemos definir el polimorfismo como la capacidad de un mensaje para mostrarse en más de una forma. Un ejemplo real de polimorfismo es una persona que al mismo tiempo puede tener características diferentes. Un hombre al mismo tiempo es un padre, un esposo y un empleado. Así que la misma persona exhibe un comportamiento diferente en diferentes situaciones. Esto se llama polimorfismo. El polimorfismo se considera una de las características importantes de la programación orientada a objetos.

## Tipos de polimorfismo

- Polimorfismo en tiempo de compilación
  - Sobrecarga de Funciones (Function Overloading)
  - Sobrecarga de Operadores (Operator Overloading)
- Polimorfismo en tiempo de ejecución
  - Funciones Anuladas (Function Overriding)
  - Funciones Virtuales (Virtual Función))



# Polimorfismo en C++

## 1. Polimorfismo en tiempo de compilación

Este tipo de polimorfismo se logra mediante sobrecarga de funciones o sobrecarga de operadores.

### 1.A. Sobrecarga de funciones (Function Overloading)

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Geeks {
public:
    // Function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // Function with same name and
    // 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y
            << endl;
    }
};
```

Cuando hay varias funciones con el mismo nombre pero diferentes parámetros, entonces se dice que las funciones están sobrecargadas, por lo tanto, esto se conoce como sobrecarga de funciones. Las funciones se pueden sobrecargar cambiando el número de argumentos y/o cambiando el tipo de argumentos. En términos simples, es una característica de la programación orientada a objetos que proporciona muchas funciones que tienen el mismo nombre pero parámetros distintos cuando numerosas tareas se enumeran bajo un nombre de función. Hay ciertas reglas de sobrecarga de funciones que deben seguirse al sobrecargar una función.

A continuación se muestra el programa C++ para mostrar la sobrecarga de funciones o el polimorfismo en tiempo de compilación:

```
// Driver code
int main()
{
    Geeks obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

Salida  
value of x is 7  
value of x is 9.132  
value of x and y is 85, 64

Explicación: En el ejemplo anterior, una sola función llamada function func() actúa de manera diferente en tres situaciones diferentes, que es una propiedad del polimorfismo.



# Polimorfismo en C++

## 1. Polimorfismo en tiempo de compilación

Este tipo de polimorfismo se logra mediante sobrecarga de funciones o sobrecarga de operadores.

### 1.B. Sobrecarga del operador (Operator Overloading)


C++ tiene la capacidad de proporcionar a los operadores un significado especial para un tipo de datos, esta capacidad se conoce como sobrecarga del operador.

Por ejemplo, podemos hacer uso del operador de adición (+) para la clase string para concatenar dos cadenas. Sabemos que la tarea de este operador es agregar dos operandos. Así que un solo operador '+', cuando se coloca entre operandos enteros, los suma y cuando se coloca entre operandos de cadena, los concatena.

A continuación se muestra el programa C++ para demostrar la sobrecarga del operador:

Salida  
12 + i9

Explicación: En el ejemplo anterior, el operador '+' está sobrecargado. Por lo general, este operador se usa para sumar dos números (números enteros o números de coma flotante), pero aquí el operador está hecho para realizar la suma de dos números imaginarios o complejos.



```
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    // This is automatically called
    // when '+' is used with between
    // two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() { cout << real << " + i" << imag << endl; }
};

// Driver code
int main()
{
    Complex c1(10, 5), c2(2, 4);

    // An example call to "operator+"
    Complex c3 = c1 + c2;
    c3.print();
}
```

# Polimorfismo en C++

## 2. Polimorfismo en tiempo de ejecución

Este tipo de polimorfismo se logra mediante la anulación de funciones. **El enlace tardío y el polimorfismo dinámico son otros nombres para el polimorfismo en tiempo de ejecución.** La llamada a la función se resuelve en tiempo de ejecución en el polimorfismo en tiempo de ejecución. Por el contrario, con el polimorfismo en tiempo de compilación, el compilador determina qué llamada a la función enlazar al objeto después de deducirlo en tiempo de ejecución.

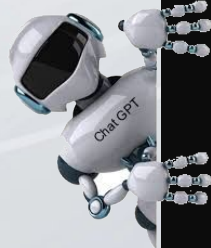
### A. Anulación de funciones (Function Overriding)

La anulación de funciones se produce cuando una clase derivada tiene una definición para una de las funciones miembro de la clase base. Se dice que esa función base está anulada.

```
class Parent
{
public:
    void GeeksforGeeks()
    {
        statements;
    }
};

class Child: public Parent
{
public:
    void GeeksforGeeks()
    {
        Statements;
    }
};

int main()
{
    Child Child_Derived;
    Child_Derived.GeeksforGeeks();
    return 0;
}
```



```
// C++ program for function overriding
#include <bits/stdc++.h>
using namespace std;

class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }

    void show() { cout << "show base class" << endl; }
};

class derived : public base {
public:
    // print () is already virtual function in
    // derived class, we could also declared as
    // virtual void print () explicitly
    void print() { cout << "print derived class" << endl; }

    void show() { cout << "show derived class" << endl; }
};

// Driver code
int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // Virtual function, binded at
    // runtime (Runtime polymorphism)
    bptr->print();

    // Non-virtual function, binded
    // at compile time
    bptr->show();

    return 0;
}
```

**Output**  
print derived class  
show base class



# Polimorfismo en C++

## 2. Polimorfismo en tiempo de ejecución

Este tipo de polimorfismo se logra mediante la anulación de funciones. El enlace tardío y el polimorfismo dinámico son otros nombres para el polimorfismo en tiempo de ejecución. La llamada a la función se resuelve en tiempo de ejecución en el polimorfismo en tiempo de ejecución. Por el contrario, con el polimorfismo en tiempo de compilación, el compilador determina qué llamada a la función enlazar al objeto después de deducirlo en tiempo de ejecución.

### Virtual Function

Una función virtual es una función miembro que se declara en la clase base utilizando la palabra clave virtual y se vuelve a definir (Overrideden) en la clase derivada.

Algunos puntos clave sobre las funciones virtuales:

Las funciones virtuales son de naturaleza dinámica.

Se definen insertando la palabra clave "virtual" dentro de una clase base y siempre se declaran con una clase base y se anulan en una clase secundaria.

Se llama a una función virtual durante el tiempo de ejecución

A continuación se muestra el programa C++ para demostrar la función virtual:

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
    // virtual function
    virtual void display()
    {
        cout << "Called virtual Base Class function"
              << "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Base print function"
              << "\n\n";
    }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {

public:
    void display()
    {
        cout << "Called GFG_Child Display Function"
              << "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Child print Function"
              << "\n\n";
    }
};
```

```
// Driver code
int main()
{
    // Create a reference of class GFG_Base
    /*
    GFG_Base* base;
    GFG_Child child;
    base = &child;
    */
    GFG_Base * base = new GFG_Child();

    // This will call the virtual function
    base->GFG_Base::display();
    base->display();

    // this will call the non-virtual function
    base->print();
}
```

Salida

Called virtual Base Class function

Called GFG\_Child Display Function

Called GFG\_Base print function





# Polimorfismo en Python

Al ser un lenguaje con **tipado dinámico y permitir duck typing**, en Python no es necesario que los objetos compartan un interface, simplemente basta con que tengan los métodos que se quieren llamar.

Podemos recrear el ejemplo de la siguiente manera. Supongamos que tenemos un clase Animal con un método hablar().

```
class Animal:
    def hablar(self):
        pass
```

Por otro lado tenemos otras dos clases, Perro, Gato que heredan de la anterior. Además, implementan el método hablar() de una forma distinta.

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")
```

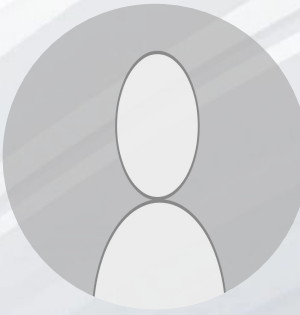
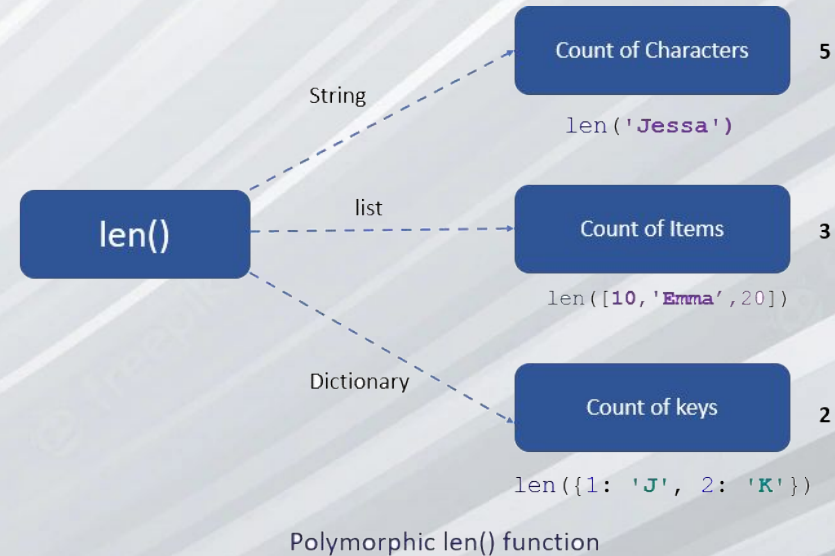
```
class Gato(Animal):
    def hablar(self):
        print("Miau!")
```

A continuación creamos un objeto de cada clase y llamamos al método hablar(). Podemos observar que cada animal se comporta de manera distinta al usar hablar().

```
for animal in Perro(), Gato():
    animal.hablar()
```

```
# Guau!
# Miau!
```

En el caso anterior, la variable animal ha ido “tomando las formas” de Perro y Gato. Sin embargo, nótese que al tener tipado dinámico este ejemplo hubiera funcionado igual sin que existiera herencia entre Perro y Gato, pero esta explicación la dejamos para el capítulo de duck typing (Coming soon)



18 Py



# Polimorfismo en Python

El polimorfismo está exhibiendo un comportamiento diferente en diferentes condiciones. El polimorfismo viene en dos sabores.

## Método de sobrecarga Método Anulando

### Método de sobrecarga

Python no admite la sobrecarga de métodos como otros idiomas. Simplemente reemplazará la última función definida como la última definición. Sin embargo, podemos intentar lograr un resultado similar a la sobrecarga utilizando \* args o usando argumentos opcionales.

```
class OptionalArgDemo:
    def addNums(self, i, j, k=0):
        return i + j + k

o = OptionalArgDemo()
print(o.addNums(2,3))
print(o.addNums(2,3,7))
```

Resultado:

5  
12

### Método Anulando

El polimorfismo en tiempo de ejecución no es más que una anulación del método. Funciona en conjunto con la herencia.

La anulación del método es un concepto en el que, aunque el nombre del método y los parámetros pasados son similares, el comportamiento es diferente según el tipo de objeto.

```
class Animal:
    def makeNoise(self):
        raise NotImplementedError

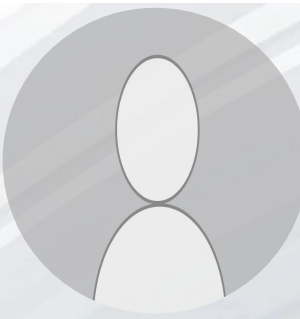
class Cat(Animal):
    def makeNoise(self):
        print("Meooooowwww")

class Dog(Animal):
    def makeNoise(self):
        print("Woooooof")
```

```
a = Cat()
a.makeNoise()
#Prints Meeeeowwwwww

a = Dog()
a.makeNoise()
#Prints Woooooof
```

En el ejemplo anterior creamos la clase Cat and Dog, que se hereda de Animal. Luego implementamos el método makeNoise en las clases Cat y Dog. Como puede ver en el ejemplo anterior, makeNoise imprime resultados diferentes en la misma referencia de Animal. En el tiempo de ejecución, decide el tipo de objeto y llama al método correspondiente.





POO CLASES ABSTRACTAS  
INTERFACES  
FORMALES  
INFORMALES  
CLASES VIRTUALES



Stage 6





# Interfaces y Abstract Base Class (ABC)

En la programación orientada a objetos, un **interface** define al conjunto de métodos que tiene que tener un objeto para que pueda cumplir una determinada función en nuestro sistema. Dicho de otra manera, un interface define como se comporta un objeto y lo que se puede hacer con el.

Piensa en el mando a distancia del televisor. Todos los mandos nos ofrecen el mismo interface con las mismas funcionalidades o métodos. En pseudocódigo se podría escribir su interface como:

```
# Pseudocódigo
interface Mando{
    def siguiente_canal():
    def canal_anterior():
    def subir_volumen():
    def bajar_volumen():
}
```

Es importante notar que los interfaces no poseen una implementación per se, es decir, no llevan código asociado. “La **interface se centra en el qué y no en el cómo...**”

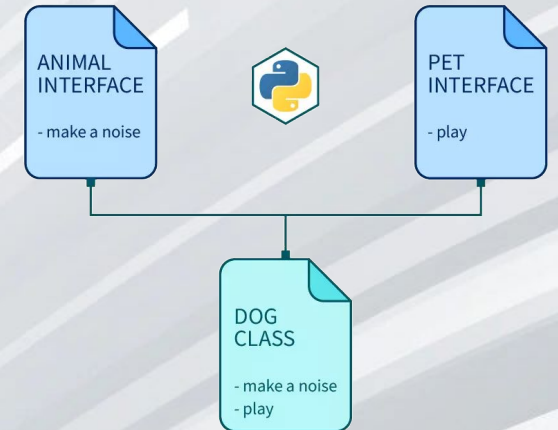
Se dice entonces que una determinada clase implementa una interface, cuando añade código a los métodos que no lo tenían (denominados abstractos, en su clase Base). Es decir, implementar un interface consiste en pasar del **qué** (clase abstracta o interface) se hace al **cómo** se hace (clase concreta que Implementa la Interface. Esto permite que el cliente se conecte con la Interface e implemente una clase concreta.

Podríamos decir entonces que los mandos de Samsung y LG implementan nuestro interface Mando, ya que ambos tienen los métodos definidos, pero con implementaciones diferentes. Esto es debido a que cada empresa resuelve el mismo problema con un enfoque diferente, pero lo que se ofrece visto desde el exterior es lo mismo.

Aunque lo veremos más adelante, ya podemos adelantar que Python no posee la **keyword interface** como otros lenguajes de programación (Java). A pesar de esto, existen dos formas de definir interfaces en Python:

- Interfaces informales
- Interfaces formales

*Dependiendo de la magnitud y tipo del proyecto en el que trabajemos, es posible que los interfaces informales sean suficientes. Sin embargo, a veces no bastan, y es donde entran los interfaces formales y las metaclasses, ambos conceptos bastante avanzados pero que la mayoría de programadores tal vez pueda ignorar.*



# Interfaces y Abstract Base Class (ABC)

## Interfaces Informales

Los interfaces informales pueden ser definidos con una simple clase que no implementa los métodos. Volviendo al ejemplo de nuestro interface mando a distancia, lo podríamos escribir en Python como:

```
class Mando:
    def siguiente_canal(self):
        pass
    def canal_anterior(self):
        pass
    def subir_volumen(self):
        pass
    def bajar_volumen(self):
        pass
```

Una vez definido nuestro interface informal, podemos usarlo mediante herencia. Las clases MandoSamsung y MandoLG implementan el interface Mando con código particular en los métodos. Recuerda, pasamos del qué hace al cómo se hace.

```
class MandoSamsung(Mando):
    def siguiente_canal(self):
        print("Samsung->Siguiente")
    def canal_anterior(self):
        print("Samsung->Anterior")
    def subir_volumen(self):
        print("Samsung->Subir")
    def bajar_volumen(self):
        print("Samsung->Bajar")
```

```
class MandoLG(Mando):
    def siguiente_canal(self):
        print("LG->Siguiente")
    def canal_anterior(self):
        print("LG->Anterior")
    def subir_volumen(self):
        print("LG->Subir")
    def bajar_volumen(self):
        print("LG->Bajar")
```





# Interfaces y Abstract Base Class (ABC)

## Interfaces Informales

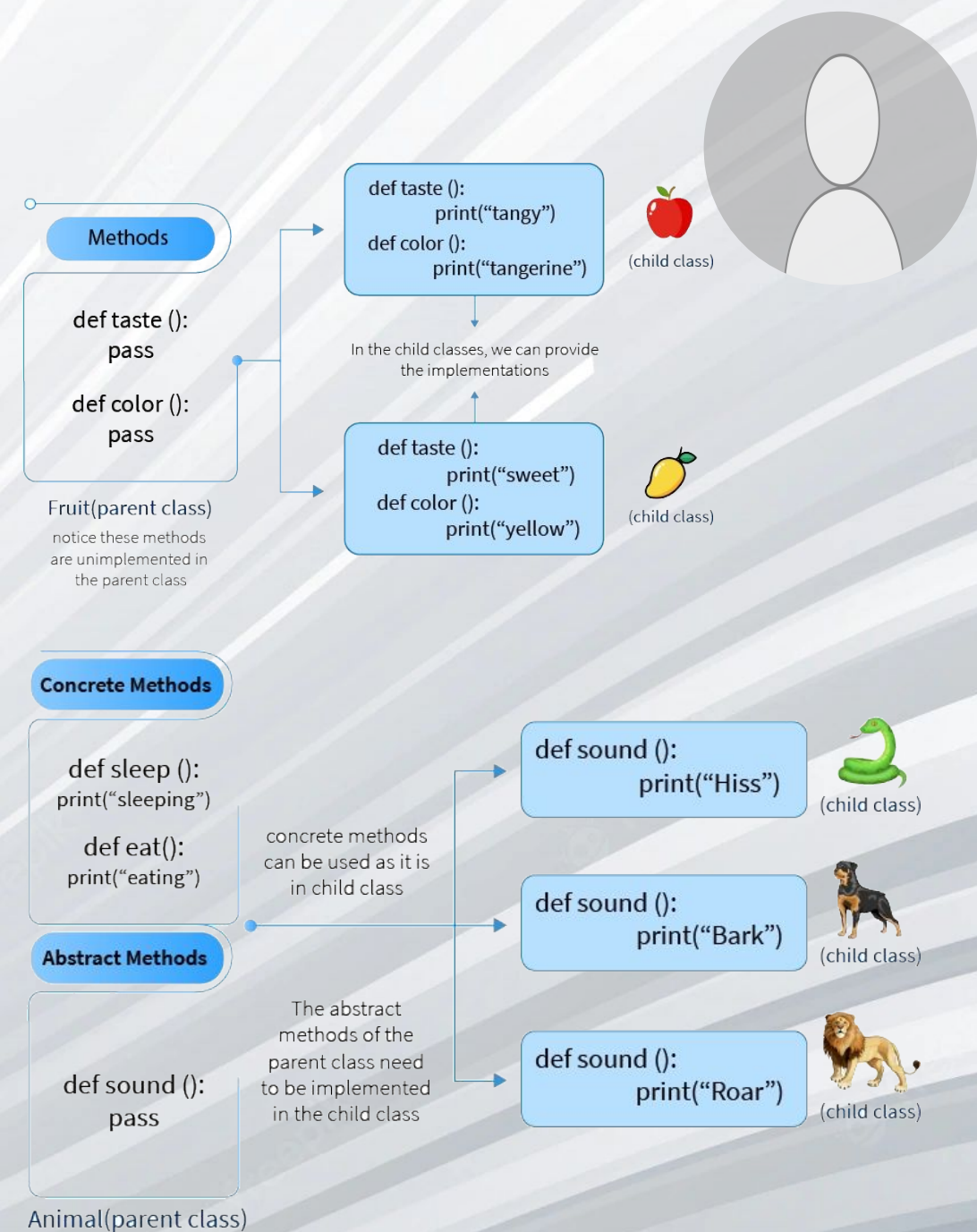
Como hemos dicho, esto es una solución perfectamente “válida” en la mayoría de los casos, pero existe un problema con el que entenderás perfectamente porqué lo llamamos interface informal.

*...Al heredar de la clase Mando, no se obliga a MandoSamsung o MandoLG a implementar todos los métodos. Es decir, ambas clases podrían no tener código para todos los métodos, y esto es algo que puede causar problemas.*

El razonamiento es el siguiente. Si Mando es un interface que como tal no implementa ningún método (tan sólo define los métodos), ¿no sería acaso importante asegurarse de que las clases que usan dicho interface implementan los métodos?

- Si un método queda sin implementar, podríamos tener problemas en el futuro, ya que al llamar a dicho método no tendríamos código que ejecutar. Es cierto que se podría resolver cambiando pass por raise NotImplementedError(), pero el error lo obtendríamos en tiempo de ejecución.

Hasta aquí los interfaces informales. Nótese que este tipo de interfaces es posible en Python debido a una de sus características estrella, el duck typing, abordaremos este concepto tan importante en Python luego...





# Interfaces y Abstract Base Class (ABC)

## Interfaces Formales

Una vez tenemos el contexto de lo que son los interfaces informales, ya estamos en condiciones de entender los interfaces formales.

Los interfaces formales pueden ser definidos en Python utilizando el módulo por defecto llamado **ABC** (Abstract Base Classes). Los abc fueron añadidos a Python en la PEP3119.

Simplemente definen una forma de crear interfaces (a través de metaclasses) en los que se definen unos métodos (pero no se implementan) y donde se fuerza a las clases que usan ese interface a implementar los métodos. Veamos unos ejemplos.

El interface más sencillo que podemos crear es de la siguiente manera, heredando de abc.ABC.

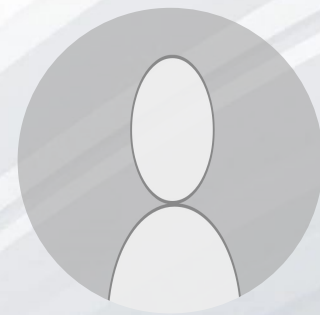
```
from abc import ABC
class Mando(ABC):
    pass
```



Interface en Python

La siguiente sintaxis es también válida, y aunque se sale del contenido de este capítulo, es importante que asocies el módulo abc con las metaclasses.

```
from abc import ABCMeta
class Mando(metaclass=ABCMeta):
    pass
```



# Interfaces y Abstract Base Class (ABC) `@abstractmethod`

## Interfaces Formales

Pero veamos un ejemplo concreto continuando con nuestro ejemplo del mando a distancia. Podemos observar como se usa el decorador `@abstractmethod`. Un método abstracto es un método que no tiene una implementación, es decir, que no lleva código. Un método definido con este decorador, forzará a las clases que implementen dicho interface a codificarlo.

Veamos como queda nuestro interface formal Mando.

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

```
class Mando(metaclass=ABCMeta):
    @abstractmethod
    def siguiente_canal(self):
        pass

    @abstractmethod
    def canal_anterior(self):
        pass

    @abstractmethod
    def subir_volumen(self):
        pass

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

Sin embargo si que podemos heredar de Mando para crear una clase MandoSamsung. Es muy importante que implementemos todos los métodos, o de lo contrario tendremos un error. Esta es una de las diferencias con respecto a los interfaces informales.



Lo primero a tener en cuenta es que no se puede crear un objeto de una clase interface, ya que sus métodos no están implementados.

```
mando = Mando()
# TypeError: Can't instantiate abstract class Mando with abstract methods bajar_volumen, canal_anterior, siguiente_canal, subir_volumen
```

# Interfaces y Abstract Base Class (ABC)

## Interfaces Formales

```
class MandoSamsung(Mando):
    def siguiente_canal(self):
        print("Samsung->Siguiente")
    def canal_anterior(self):
        print("Samsung->Anterior")
    def subir_volumen(self):
        print("Samsung->Subir")
    def bajar_volumen(self):
        print("Samsung->Bajar")
```

Y como de costumbre podemos crear un objeto y llamar a sus métodos.

```
mando_samsung = MandoSamsung()
mando_samsung.bajar_volumen()
# Samsung->Bajar
```

Siguiendo con el ejemplo podemos definir la clase MandoLG.

**Por un lado tenemos nuestro interface Mando.** Se trata de una clase que define el comportamiento de un mando genérico, pero sin centrarse en los detalles de cómo funciona. Se centra en el qué.

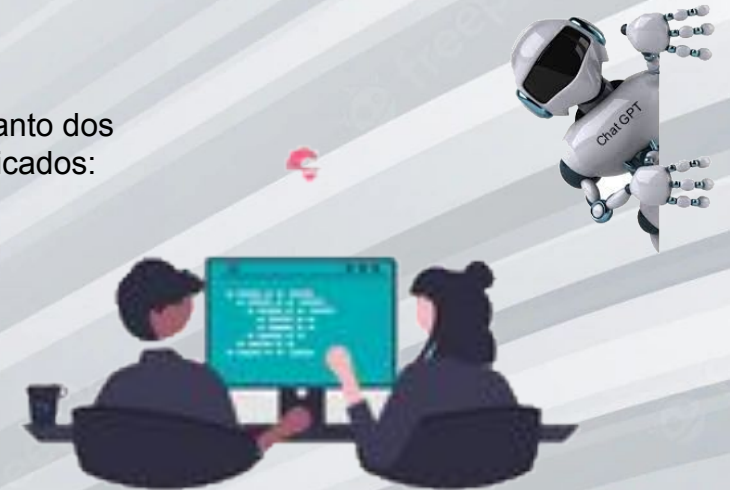
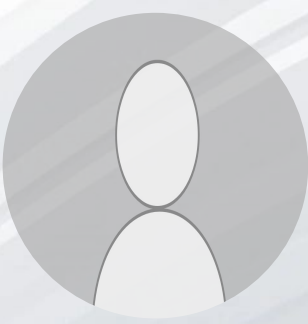
**Por otro lado tenemos las implementaciones concretas,** dos clases MandoSamsung y MandoLG que implementan/heredan el interface anterior, añadiendo un código concreto y diferente para cada mando. Ambas clases representan el cómo.

Hasta aquí hemos visto como crear un interface formal sencilla usando abc con métodos abstractos, pero existen más funcionalidades que merece la pena ver. Vamos a por ello.

```
class MandoLG(Mando):
    def siguiente_canal(self):
        print("LG->Siguiente")
    def canal_anterior(self):
        print("LG->Anterior")
    def subir_volumen(self):
        print("LG->Subir")
    def bajar_volumen(self):
        print("LG->Bajar")
Y creamos un objeto de MandoLG.
```

```
mando_lg = MandoLG()
mando_lg.bajar_volumen()
# LG->Bajar
```

Llegados a este punto tenemos por lo tanto dos conceptos diferentes claramente identificados:





# Clases virtuales

Como ya sabemos, se considera que una clase es subclase o subclass de otra si hereda de la misma, como podemos ver en el siguiente ejemplo.

```
class ClaseA:
```

```
    pass
```

```
class ClaseB(ClaseA):
```

```
    pass
```

```
print(issubclass(ClaseB, ClaseA))
```

```
# True
```

**Pero, ¿y si queremos que se considere a una clase la padre cuando no existe herencia entre ellas?**

Es aquí donde entran las clases virtuales. Usando register() podemos registrar a una ABC como clase padre de otra. En el siguiente ejemplo FloatABC se registra como clase virtual padre de float.

```
from abc import ABCMeta
```

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

```
    pass
```

```
FloatABC.register(float)
```

Y esto implica que el comportamiento de issubclass se ve modificado.

```
print(issubclass(float, FloatABC))
```

```
# True
```

Análogamente podemos realizar lo mismo con una clase definida por nosotros.

```
@FloatABC.register
```

```
class MiFloat():
```

```
    pass
```

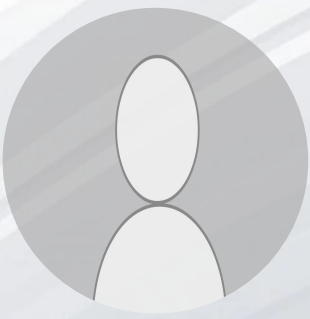
```
x = MiFloat()
```

```
print(issubclass(MiFloat, FloatABC))
```

```
# True
```



# Métodos abstractos



Como ya hemos visto los métodos abstractos son aquellos que son declarados pero no tienen una implementación. También hemos visto como Python nos obliga a implementarlos en las clases que heredan de nuestro interface. Esto es posible gracias al decorador `@abstractmethod`.

```
from abc import ABC, abstractmethod
class Clase(metaclass=ABCMeta):
    @abstractmethod
    def metodo_abstracto(self):
        pass
```

Sin embargo, también es posible combinar el decorador `@abstractmethod` con los decoradores `@classmethod` y `@staticmethod` que ya vimos anteriormente. Nótese que `@abstractmethod` debe ser usado siempre justo antes del método.

Como recordatorio, un método de clase es llamado sobre la clase y no sobre el objeto, pudiendo modificar la clase pero no el objeto.

```
class Clase(ABC):
    @classmethod
    @abstractmethod
    def metodo_abstracto_de_clase(cls):
        pass
```

Análogamente podemos definir un `@staticmethod`. Se trata de un método que no permite realizar modificaciones ni de la clase ni del objeto, ya que no lo recibe como parámetro.

```
class Clase(ABC):
    @staticmethod
    @abstractmethod
    def metodo_abstracto_estatico():
        pass
```

Y por último también podemos combinarlo con el decorador `property`.

```
class Clase(ABC):
    @property
    @abstractmethod
    def metodo_abstracto_propiedad(self):
        pass
```

# Abstract Base Classes y colecciones

Python nos ofrece un conjunto de **Abstract Base Classes** que podemos usar para crear nuestras propias clases, denominado **collections.abc**. Es por tanto importante echarles un vistazo, ya que tal vez exista ya la que necesitemos.

Podemos por ejemplo crear una clase MiSet que use abc.Set, pero que tenga un comportamiento ligeramente distinto. En este caso, deberemos implementar los métodos mágicos `__iter__`, `__contains__` y `__len__`, ya que son definidos como abstractos en el abc.

```
from collections import abc
class MiSet(abc.Set):
    def __init__(self, iterable):
        self.elements = []
        for value in iterable:
            if value not in self.elements:
                self.elements.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

    def __str__(self):
        return "".join(str(i) for i in self.elements)
```

Como podemos ver, heredamos ciertas funcionalidades como los operadores `&` y `|` que pueden ser usados sobre nuestra nueva clase.

```
s1 = MiSet("abcdefg")
s2 = MiSet("efghij")

print(s1 & s2)
print(s1 | s2)
# efg
# abcdefghij
```



19 20 21 Py







# DUCK TYPING

POO

INTRODUCCION A LAS POO EN PYTHON.



# Duck Typing en Python

El duck typing o tipado de pato es un concepto relacionado con la programación que aplica a ciertos lenguajes orientados a objetos, y que tiene origen en la siguiente frase:

**If it walks like a duck and it quacks like a duck, then it must be a duck**

Lo que se podría traducir al español como. Si camina como un pato y habla como un pato, entonces tiene que ser un pato.

¿Y qué relación tienen los patos con la programación? Pues bien, se trata de un símil en el que los patos son objetos y hablar/andar métodos. Es decir, que si un determinado objeto tiene los métodos que nos interesan, nos basta, siendo su tipo irrelevante.

Dicho de otra manera, no mires si es un pato. Fíjate si habla como un pato, camina como un pato, etc. Si cumple con todas estas características, ¿no podríamos acaso decir que se trata de un pato?

**Don't check whether it is-a duck: check whether it quacks-like-a duck, walks-like-a duck, etc, etc, depending on exactly what subset of duck-like behavior you need to play your language-games with.** (comp.lang.python, Jul. 26, 2000) — Alex Martelli

El concepto de **duck typing** se fundamenta en el razonamiento inductivo, donde una serie de premisas apoyan la conclusión, pero no la garantizan. Si vemos a un animal que parece un pato, habla como tal y anda como tal, sería razonable pensar que se trata de un pato, pero sin un test de ADN nunca estaríamos al cien por cien seguros.





# Duck Typing en Python

Una vez entendido el origen del concepto, veamos lo que realmente significa esto en Python. En pocas palabras, a Python le dan igual los tipos de los objetos, lo único que le importan son los métodos. En Python se prioriza el comportamiento por sobre el tipo.

Definamos una clase Pato con un método hablar().

```
class Pato:  
    def hablar(self):  
        print("¡Cua!, Cua!")
```

Y llamamos al método de la siguiente forma.

```
p = Pato()  
p.hablar()  
# ¡Cua!, Cua!
```

Hasta aquí nada nuevo, pero vamos a definir una función llama\_hablar(), que llama al método hablar() del objeto que se le pase.

```
def llama_hablar(x):  
    x.hablar()
```

Como puedes observar, en Python no es necesario especificar los tipos, simplemente decimos que el parámetro de entrada tiene el nombre x, pero no especificamos su tipo.

Cuando Python entra en la función y evalúa x.hablar(), le da igual el tipo al que pertenezca x siempre y cuando tenga el método hablar(). Esto es el duck typing en todo su esplendor.

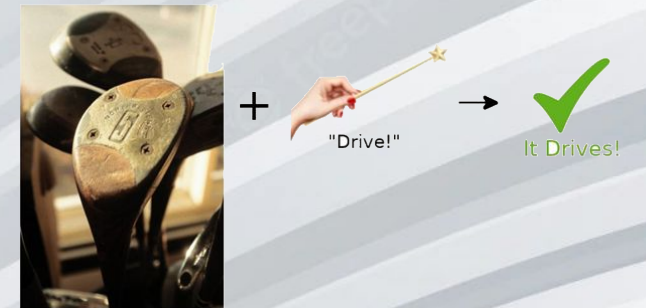
```
p = Pato()  
llama_hablar(p)  
# ¡Cua!, Cua!
```

¿Y qué pasa si usamos otros objetos que no son de la clase Pato? Pues bien, como hemos dicho, a la función llama\_hablar() le da igual el tipo. Lo único que le importa es que el objeto tenga el método hablar().



## Duck Typing

(by Ben Koshy)





# Duck Typing en Python

Definamos tres clases de animales distintas que implementan el método hablar(). Nótese que no existe herencia entre ellas, son clases totalmente independientes. De haberla estaríamos hablando de polimorfismo.

```
class Perro:  
    def hablar(self):  
        print("¡Guau, Guau!")
```

```
class Gato:  
    def hablar(self):  
        print("¡Miau, Miau!")
```

```
class Vaca:  
    def hablar(self):  
        print("¡Muuu, Muuu!")
```

Y como es de esperar la función llama\_hablar() funciona correctamente con todos los objetos.

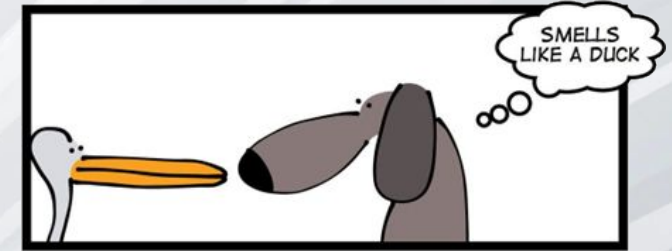
```
llama_hablar(Perro())  
llama_hablar(Gato())  
llama_hablar(Vaca())
```

```
# ¡Guau, Guau!  
# ¡Miau, Miau!  
# ¡Muuu, Muuu!
```

Otra forma de verlo, es iterando una lista con diferentes animales, donde animal va tomando los valores de cada objeto animal. Todo un ejemplo del duck typing y del tipado dinámico en Python.

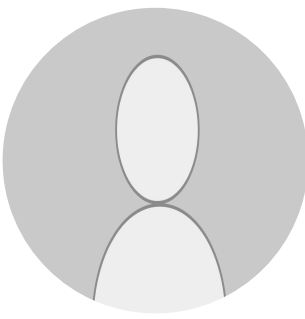
```
lista = [Perro(), Gato(), Vaca()]  
for animal in lista:  
    animal.hablar()
```

```
# ¡Guau, Guau!  
# ¡Miau, Miau!  
# ¡Muuu, Muuu!
```



DUCKFOODING

# Duck Typing en Python - Ejemplos



## Ejemplo len()

Podemos ver el duck typing en todo su esplendor con la función len(). Dicha función lo único que realiza por debajo es llamar al método mágico `__len__()`. Definamos dos clases:

Foo implementa el método `__len__()`.

Bar no lo implementa.

```
class Foo():
    def __len__(self):
        return 99
```

```
class Bar():
    pass
```

Como ya hemos explicado, a la función len() no le importa el tipo del objeto que se le pase, siempre y cuando tenga el método `__len__()` implementado. Por ello, en el segundo caso falla.

```
print(len(Foo())) # 99
```

```
print(len(Bar())) # Error
```

## Ejemplo multiplicar

Por otro lado, cuando hacemos una multiplicación utilizando el operador aritmético `*` el resultado depende de los tipos que estemos usando.

No es lo mismo multiplicar dos enteros que un entero y cadena.

```
print(3*3) # 9
```

```
print(3*"3") # 333
```

Una vez más, podemos ver el duck typing en Python. Simplemente se busca que los objetos a la izquierda y derecha del `*` tengan implementado el `__rmul__` o `__mul__`.


## Conclusiones

Python es un lenguaje que soporta el duck typing, lo que hace que el tipo de los objetos no sea tan relevante, siendo más importante lo que pueden hacer (sus métodos).

Otros lenguajes como Java no soporta el duck typing, pero se puede conseguir un comportamiento similar cuando los objetos comparten un interface (si existe herencia entre ellos). Este concepto relacionado es el polimorfismo.

El duck typing está en todos lados, desde la función len() hasta el uso del operador `*`.





# Language - Agnostic

## Language-agnostic programing

La programación o secuencias de comandos independientes del lenguaje es un paradigma de desarrollo de software en el que se elige un lenguaje en particular debido a su idoneidad para una tarea en particular, y no simplemente por el conjunto de habilidades disponibles dentro de un equipo de desarrollo.

**Modern Portfolio Designed**





# THANK YOU

Let's Python Code...