# ﻿﻿**Módulo 7**

# **POO: implementación**

**Getters , Setters**

**Encapsulamiento**

name mangling

Ejemplo con el uso de métodos encapsulados.

**Métodos Mágicos**

• \_\_init\_\_(self, ...)

• \_\_str\_\_(self)

• \_\_repr\_\_(self)

• \_\_len\_\_(self)

• \_\_getitem\_\_(self, key)

• \_\_setitem\_\_(self, key, value)

• \_\_del\_\_(self)

• \_\_delitem\_\_(self, key)

• \_\_iter\_\_(self)

• \_\_next\_\_(self)

• \_\_eq\_\_(self, other)

• \_\_ne\_\_(self, other)

• \_\_lt\_\_(self, other)

• \_\_le\_\_(self, other)

• \_\_gt\_\_(self, other)

• \_\_ge\_\_(self, other)

• \_\_add\_\_(self, other)

• \_\_sub\_\_(self, other)

• \_\_mul\_\_(self, other)

• \_\_truediv\_\_(self, other)

• \_\_mod\_\_(self, other)

• \_\_divmod\_\_(self, other)

• \_\_format\_\_(self, format\_spec)

• \_\_contains\_\_(self, item)

**Métodos de operadores.**

Métodos de operador recíprocos.

* \_\_rsub\_\_(self, other)
* \_\_rmul\_\_(self, other)
* \_\_rdiv\_\_(self, other)
* \_\_rmod\_\_(self, other)
* \_\_rand\_\_(self, other)

**Métodos estáticos y métodos de clase**

**Métodos Estáticos**

* @staticmethod

**Métodos de clase**

* @classmethod

**Propiedades**

* @property

**Herencia**

Herencia simple

superclase

subclase

Funcion `super()`

\_issubclass()\_

\_subclass()\_

**Herencia Múltiple**

**Mixin**

**﻿﻿Polimorfismo**

**Duck Typing**

* Flexibilidad y Adaptabilidad
* Menos Restricciones en la Jerarquía de Clases
* Interfaces Implícitas
* Facilita la Creación de APIs Genéricas
* Menos Dependencia de Tipos Específicos
* Prueba de Conceptos Más Rápida
* Uso de Métodos Mixins
* Desafíos Potenciales

**\_\_len\_₍)**

## **Composición**

# **7-POO: implementación**

## **Getters , Setters**

En Python es posible hacer que ciertos métodos se comporten de como si fueran atributos mediante el uso de \*property\*.

|  |
| --- |
| En Python, **no es necesario implementar getters y setters** explícitos como en algunos otros lenguajes de programación, como Java. |

Las propiedades en Python corresponden a métodos que se comportan como "getters" y "setters" de un atributo.

Un "getter" regresa el contenido de un atributo.

Un "setter" crea o modifica el contenido de un atributo.

La sintaxis para la definición de propiedades es la siguiente:La primera función definida es un getter regresará un valor de la propiedad cuando se invoque y la segunda asignará el valor a la propiedad cuando se utilice un operador de asignación.

Si no se define un setter, el valor de la propiedad no podrá ser cambiado.

Ejemplo:

|  |
| --- |
| class Mi\_Grupo\_Python:  ...  ...  @property  def <nombre de la propiedad>(self):  ...  ...  return <objeto>  @<nombre de la propiedad>.setter  def<nombre de la propiedad>(self, <parametro>):  ...  ... |

·

|  |  |  |
| --- | --- | --- |
| **Sin getters y setters**  Código Python   |  | | --- | | class Mi\_Grupo\_Python:  def \_\_init\_\_(self, valor, string):  self.edad = valor  self.nombre = string.title()  # Crear una instancia de la clase  objeto = Mi\_Grupo\_Python(49,"Juan")  print(f"{objeto.nombre} tiene {objeto.edad }") |   Salida esperada por consola   |  | | --- | | Juan tiene 49 |   · |

·

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Los getters y setters son métodos utilizados en programación orientada a objetos para acceder y modificar los atributos (propiedades) de una clase de manera controlada. En muchos lenguajes de programación, incluido Python, los atributos de una clase suelen ser accesibles directamente desde fuera de la clase. Sin embargo, en algunos casos, es deseable tener un mayor control sobre cómo se acceden y modifican estos atributos, lo cual se logra mediante los getters y setters.  **Getter:**  Un getter es un método que se utiliza para obtener el valor de un atributo. Su función principal es proporcionar una interfaz para acceder al valor de un atributo privado o protegido de una clase. Los getters generalmente tienen un nombre que comienza con "get" seguido del nombre del atributo que se desea obtener.   |  |  | | --- | --- | | ·Código Python   |  | | --- | | class Persona:  def \_\_init\_\_(self, nombre):  self.\_\_nombre = nombre # Atributo privado  def get\_nombre(self):  return self.\_\_nombre  persona = Persona("Anita")  print(persona.get\_nombre()) # Accediendo al nombre a través del getter |   · |   ·  **Setter:**  Un setter es un método que se utiliza para modificar el valor de un atributo. Los setters son útiles para validar los nuevos valores asignados a un atributo y para realizar acciones adicionales cuando el valor cambia. Los setters generalmente tienen un nombre que comienza con "set" seguido del nombre del atributo.   |  |  | | --- | --- | | ·Código Python   |  | | --- | | class Persona:  def \_\_init\_\_(self, nombre):  self.\_\_nombre = nombre # Atributo privado  def get\_nombre(self):  return self.\_\_nombre  def set\_nombre(self, nuevo\_nombre):  if len(nuevo\_nombre) >= 2:  self.\_\_nombre = nuevo\_nombre  else:  print("El nombre debe tener al menos 2 caracteres")  persona = Persona("Anita")  persona.set\_nombre("Juan") # Modificando el nombre a través del setter  persona.set\_nombre("A") # Esto imprimirá el mensaje de validación |   · |   ·  **Es importante mencionar que en Python, no es necesario utilizar getters y setters en todos los casos.** La convención en Python es permitir el acceso directo a los atributos, a menos que haya una razón específica para controlar el acceso o realizar validaciones. Esto se debe al principio de "Pythonic" que enfatiza la simplicidad y la legibilidad del código.  Siempre que se utilicen getters y setters, es recomendable nombrarlos de manera descriptiva para que su función sea clara. Sin embargo, en muchos casos, se prefiere el acceso directo a los atributos para mantener el código simple y claro. |

·

|  |  |  |
| --- | --- | --- |
| Código Python   |  | | --- | | class MiPub:  def \_\_init\_\_ (self,nombre,edad):  self.nombre =nombre.upper()  self.edad=round(edad)  print (f"objeto {self}-{self.nombre } creado")  @property  def donde(self):  print ("en property")  if self.nombre is None:  salida = f" {self.nombre} no han ingresado al pub"  else:  salida = f" {self.nombre} se esta diviertiendo en el pub ({self.edad})"  return salida  @donde.setter  def donde(self,datos):  print ("en setter")  self.nombre =datos[0].upper()  self.edad=round(datos[1])  @donde.deleter  def donde(self):  print (f"en deleter adios {self.nombre}")  del self  objetos=[["Juan Manuel",20],["Ana Manuela",25],["Luca Garcia",18],["Luciana De Maria",15]]  lista\_objetos= []  #------------------------------------------------------------------  print ("carga original".center(40))  for nombre,edad in objetos:  obj = MiPub(nombre,edad)  lista\_objetos.append(obj)  #------------------------------------------------------------------  print ("mostrar todo lo cargado".center(40))  for obj in lista\_objetos:  print (f"{obj.donde}")  #------------------------------------------------------------------  print("\*"\*50)  print ("carga nuevo".center(40))  obj=MiPub("Pancho",99)  lista\_objetos.append(obj)  #------------------------------------------------------------------  print("\*"\*50)  print ("mostrar todo lo cargado".center(40))  for obj in lista\_objetos:  print (f"{obj.donde}")  #------------------------------------------------------------------  print("\*"\*50)  print ("renombrar con setter".center(40))  print (lista\_objetos[-1])  datos = ("Francisco",19)  lista\_objetos[-1].donde=(datos)  #------------------------------------------------------------------  print("\*"\*50)  print ("mostrar todo lo cargado".center(40))  print(f"{lista\_objetos}")  for obj in lista\_objetos:  print (f"{obj.donde}")  #------------------------------------------------------------------  print("\*"\*50)  for obj in lista\_objetos[::2]:  del obj.donde  lista\_objetos.remove(obj)  #------------------------------------------------------------------  print("\*"\*50)  print ("mostrar todo lo cargado".center(40))  print(f"{lista\_objetos}")  for obj in lista\_objetos:  print (f"{obj.donde}") |   Salida esperada por consola   |  | | --- | | carga original  objeto <\_\_main\_\_.MiPub object at 0x00000**aaaaa**>-JUAN MANUEL creado  objeto <\_\_main\_\_.MiPub object at 0x00000**bbbbb**>-ANA MANUELA creado  objeto <\_\_main\_\_.MiPub object at 0x00000**ccccc**>-LUCA GARCIA creado  objeto <\_\_main\_\_.MiPub object at 0x00000**ddddd**>-LUCIANA DE MARIA creado  mostrar todo lo cargado  en property  JUAN MANUEL se esta diviertiendo en el pub (20)  en property  ANA MANUELA se esta diviertiendo en el pub (25)  en property  LUCA GARCIA se esta diviertiendo en el pub (18)  en property  LUCIANA DE MARIA se esta diviertiendo en el pub (15)  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  carga nuevo  objeto <\_\_main\_\_.MiPub object at 0x00000**eeeee**>-PANCHO creado  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  mostrar todo lo cargado  en property  JUAN MANUEL se esta diviertiendo en el pub (20)  en property  ANA MANUELA se esta diviertiendo en el pub (25)  en property  LUCA GARCIA se esta diviertiendo en el pub (18)  en property  LUCIANA DE MARIA se esta diviertiendo en el pub (15)  en property  PANCHO se esta diviertiendo en el pub (99)  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  renombrar con setter  <\_\_main\_\_.MiPub object at 0x00000**eeeee**>  en setter  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  mostrar todo lo cargado  [<\_\_main\_\_.MiPub object at 0x00000**aaaaa**>, <\_\_main\_\_.MiPub object at 0x00000**bbbbb**>, <\_\_main\_\_.MiPub object at 0x00000**ccccc**>, <\_\_main\_\_.MiPub object at 0x00000**ddddd**>, <\_\_main\_\_.MiPub object at 0x00000**eeeee**>]  en property  JUAN MANUEL se esta diviertiendo en el pub (20)  en property  ANA MANUELA se esta diviertiendo en el pub (25)  en property  LUCA GARCIA se esta diviertiendo en el pub (18)  en property  LUCIANA DE MARIA se esta diviertiendo en el pub (15)  en property  FRANCISCO se esta diviertiendo en el pub (19)  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  en deleter adios JUAN MANUEL  en deleter adios LUCA GARCIA  en deleter adios FRANCISCO  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  mostrar todo lo cargado  [<\_\_main\_\_.MiPub object at 0x00000aaaaa>, <\_\_main\_\_.MiPub object at 0x00000bbbbb>, <\_\_main\_\_.MiPub object at 0x00000ccccc>, <\_\_main\_\_.MiPub object at 0x00000ddddd>, <\_\_main\_\_.MiPub object at 0x00000**eeeee**>]  en property  JUAN MANUEL se esta diviertiendo en el pub (20)  en property  ANA MANUELA se esta diviertiendo en el pub (25)  en property  LUCA GARCIA se esta diviertiendo en el pub (18)  en property  LUCIANA DE MARIA se esta diviertiendo en el pub (15)  en property  FRANCISCO se esta diviertiendo en el pub (19) |   · |

·

Aunque es menos habitual vemos que también esta implementado el decorador deleter

miremos que pasa al tratar de eliminar el método mostrar

## **Encapsulamiento**

|  |  |  |  |
| --- | --- | --- | --- |
| El encapsulamiento es un principio de la programación orientada a objetos que se refiere a la ocultación de los detalles internos de un objeto y la exposición de una interfaz pública para interactuar con él. En otras palabras, el encapsulamiento busca limitar el acceso directo a los atributos y métodos internos de un objeto y en su lugar proporciona métodos públicos para interactuar con él de manera controlada. Esto tiene varios beneficios, como reducir la complejidad, mejorar la seguridad y facilitar el mantenimiento del código.  El encapsulamiento se logra a través de tres niveles de acceso a los atributos y métodos de una clase:  Público: Los atributos y métodos públicos son accesibles desde cualquier parte del programa. Se definen de manera normal y no requieren ningún prefijo especial.  Privado: Los atributos y métodos privados se definen utilizando un guión bajo al principio del nombre (por ejemplo, \_atributo\_privado). Aunque Python no impide el acceso a ellos desde fuera de la clase, es una convención no hacerlo. El acceso a estos elementos se considera "prohibido" desde fuera de la clase.  Protegido: Los atributos y métodos protegidos se definen utilizando un solo guión bajo al principio del nombre (por ejemplo, \_atributo\_protegido). Al igual que los privados, se considera una convención que solo se accedan desde dentro de la clase y sus subclases.  El encapsulamiento permite que una clase controle cómo sus atributos se modifican y acceden desde fuera. Esto reduce la posibilidad de errores y hace que sea más fácil realizar cambios internos en una clase sin afectar el código que la utiliza.  ·   |  |  |  | | --- | --- | --- | | ·Código Python   |  | | --- | | class Persona:  def \_\_init\_\_(self, nombre, edad):  self.\_nombre = nombre # Atributo protegido  self.\_\_edad = edad # Atributo privado  def obtener\_nombre(self):  return self.\_nombre  def cambiar\_nombre(self, nuevo\_nombre):  self.\_nombre = nuevo\_nombre  def obtener\_edad(self):  return self.\_\_edad  def cambiar\_edad(self, nueva\_edad):  if nueva\_edad >= 0:  self.\_\_edad = nueva\_edad  persona = Persona("Anita", 30)  print(f"{persona.obtener\_nombre()=}") # Acceso a través de método público  persona.cambiar\_nombre("Juan") # Modificación a través de método público  print(f"{persona.obtener\_edad()=}") # Acceso a través de método público  persona.cambiar\_edad(35) # Modificación a través de método público  print (f"{persona.\_\_dict\_\_}") |   ·Salida esperada por consola   |  | | --- | | persona.obtener\_nombre()='Anita'  persona.obtener\_edad()=30  {'\_nombre': 'Juan', '\_Persona\_\_edad': 35} |   · |   ·  En resumen, el encapsulamiento es un principio que fomenta la organización, la seguridad y la flexibilidad en la programación orientada a objetos al ocultar los detalles internos de una clase y proporcionar una interfaz controlada para interactuar con ella. |

El encapsulamiento es una técnica que consiste en exponer los atributos de un objeto exclusivamente mediante interfaces, restringir el acceso tanto a los atributos como a las implementaciones. Es así que en otros lenguajes de programación los atributos y métodos pueden ser restringidos definiéndolos como públicos, privados, protegidos, etc.

No existen atributos privados o protegidos en Python, pero se puede ofuscar el acceso a un atributo mediante "name mangling".

El name mangling se realiza anteponiendo dobles guiones bajos "\_\_" antes del nombre del atributo a esconder.

Sintaxis:

|  |
| --- |
| class Mi\_Grupo\_Python:  ...  \_\_mi\_atributo = <valor>  def \_\_mi\_metodo(self, parámetros):  ...  …  objeto= Mi\_Grupo\_Python() |

Los atributos escondidos son accesibles como cualquier otro atributo para los métodos del objeto, pero no son accesibles desde fuera.

Para acceder a un atributo escondido fuera del objeto es necesario utilizar la siguiente sintaxis:

|  |
| --- |
| objeto\_\_mi\_metodo = valor  objeto.\_Mi\_Grupo\_Python.\_\_mi\_metodo(parámetros) |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | class Objeto(object):  def \_\_init\_\_(self, nombre, privado):  self.nombre = nombre  self.\_\_privado = privado  def comer(self):  print("me hicieron dar ganas de pizza")  def \_xx(self):  print("salida con \_xx")  def \_\_yy(self):  print("salida con \_\_yy")  Objeto.comer(1)  # ~ print ("dic:",dir(Objeto))  # ~ print ("dict:",Objeto.\_\_dict\_\_)  objeto\_1 = Objeto("Pepe", "canta en la ducha")  print ("Atributo privado:", objeto\_1.\_Objeto\_\_privado)  objeto\_1.\_Objeto\_\_privado = "no se baña"  print ("Atributo privado:", objeto\_1.\_Objeto\_\_privado)  objeto\_1.\_xx()  objeto\_1.\_Objeto\_\_yy() |   ·Salida esperada por consola   |  | | --- | | me hicieron dar ganas de pizza  Atributo privado: canta en la ducha  Atributo privado: no se baña  salida con \_xx  salida con \_\_yy |   · |

·

# **Métodos Mágicos**

|  |
| --- |
| Los métodos mágicos, también conocidos como métodos especiales o métodos dunder (del inglés "double underscore" o doble guion bajo), son métodos predefinidos en Python que tienen nombres especiales que comienzan y terminan con doble guion bajo (\_\_). Estos métodos permiten definir comportamientos específicos para operaciones y comportamientos estándar en objetos de clases definidas por el usuario. Algunos de los métodos mágicos más comunes son:   * **\_\_init\_\_(self, ...):** Método de inicialización, se ejecuta cuando se crea una nueva instancia de la clase. El constructor de la clase, se llama al crear una nueva instancia.. * **\_\_str\_\_(self)**: Método que devuelve una representación en forma de cadena (string) del objeto. Se utiliza cuando se llama a la función str(objeto) o cuando se imprime el objeto directamente, para su uso por parte del usuario final. * **\_\_repr\_\_(self):** Método que devuelve una representación "oficial" del objeto en forma de cadena. Se utiliza cuando se llama a la función repr(objeto) , para su uso en la consola o depuración. * **\_\_len\_\_(self):** Método que devuelve la longitud del objeto. Se utiliza cuando se llama a la función len(objeto). * **\_\_getitem\_\_**(self, key): Método que permite acceder a un elemento del objeto utilizando la notación de indexación, como objeto[key]. * **\_\_setitem\_\_**(self, key, value): Método que permite asignar un valor a un elemento del objeto utilizando la notación de indexación, como objeto[key] = value. * **\_\_del\_\_(self):** Destructor de la clase, se llama cuando una instancia es eliminada. * **\_\_delitem\_\_(self, key)**: Método que permite eliminar un elemento del objeto utilizando la notación de indexación, como del objeto[key]. * **\_\_iter\_\_(self):** Método que permite que el objeto sea iterable en bucles. Debe devolver un iterador. * **\_\_next\_\_(self):** Método que se utiliza en conjunción con \_\_iter\_\_ para implementar la iteración. True si es correcto. * **\_\_eq\_\_(self, other):** Método que define el comportamiento de igualdad (==) entre objetos. True si es correcto. * **\_\_ne\_\_(self, other):** Método que define el comportamiento de desigualdad (!=) entre objeto True si es correcto. * **\_\_lt\_\_(self, other):** Método que define el comportamiento de comparación "menor que" (<) entre objetos. True si es correcto. * **\_\_le\_\_(self, other):** Método que define el comportamiento de comparación "menor o igual que " (<=) entre objetos. True si es correcto. * **\_\_gt\_\_(self, other):** Método que define el comportamiento de comparación "mayor que" (>) entre objetos. True si es correcto. * **\_\_ge\_\_(self, other):** Método que define el comportamiento de comparación "mayor o igual que que" (>=) entre objetos. True si es correcto. * **\_\_add\_\_(self, other):** Método que define el comportamiento de adición (+) entre objetos. * **\_\_sub\_\_(self, other):** Método que define el comportamiento de sustracción (-) entre objetos. * **\_\_mul\_\_(self, other):** Método que define el comportamiento de multiplicación (\*) entre objetos. * **\_\_div\_\_(self, other**)**:** Método que define el comportamiento de multiplicación (/) entre objetos. * **\_\_mod\_\_(self, other):** Método que define el comportamiento de multiplicación (%) entre objetos. * **\_\_divmod\_\_(self, other):** Método que define la operación de división y módulo (divmod(self, other)).   (para mas información ver recíprocos)   * **\_\_format\_\_(self, format\_spec):** Método que define cómo se debe formatear una instancia de la clase cuando se utiliza con la función format(). La función format() toma una cadena de formato y aplica la especificación de formato a través de este método. * **\_\_contains\_\_(self, item):** Método que define el comportamiento de la palabra clave in cuando se utiliza para verificar si un elemento / item está presente en un objeto. Este método se llama cuando utilizas elemento in objeto. |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | import time  class MiClase:  \_formatos\_de\_fechas={"aaaammdd":"{fecha.año}-{fecha.mes}-{fecha.dia}",  "ddmmaaaa":"{fecha.dia}-{fecha.mes}-{fecha.año}",  "mmddaaaa":"{fecha.mes}-{fecha.dia}-{fecha.año}" }  def \_\_init\_\_(self,año,mes,dia):#constructor  self.año=año  self.mes=mes  self.dia=dia  def \_\_repr\_\_(self):  salida = f"fecha: {self.dia} de {self.mes} de {self.año} "  return salida  def \_\_str\_\_(self):  salida = f"a los {self.dia} dias del mes {self.mes} del año {self.año} "  return salida  def \_\_format\_\_ (self,codigo):  codigo=codigo.lower()  if codigo == "":  codigo="ddmmaaaa"  if codigo in MiClase.\_formatos\_de\_fechas.keys():  fecha\_formateada = self.\_formatos\_de\_fechas[codigo]  salida = fecha\_formateada.format(fecha=self)  else:  return (":sin datos")  return salida  def hoy ():  today = time.localtime()  return MiClase(today.tm\_year,today.tm\_mon,today.tm\_mday)  obj = MiClase(1969,7,16)  print (f"salida desde clase {obj}")  print (f"salida con formato ddmmaaaa {format(obj,'ddmmaaaa')}")  print (f"salida con formato mmddaaaa {format(obj,'mmddaaaa')}")  print (f"salida con formato aaaammdd {format(obj,'aaaammdd')}")  print ("\*"\*50)  obj\_hoy = MiClase.hoy()  print (f"salida con formato ddmmaaaa {format(obj\_hoy,'ddmmaaaa')}")  print (f"salida con formato mmddaaaa {format(obj\_hoy,'mmddaaaa')}")  print (f"salida con formato aaaammdd {format(obj\_hoy,'aaaammdd')}") |   ·Salida esperada por consola   |  | | --- | | salida desde clase 16-7-1969  salida con formato ddmmaaaa 16-7-1969  salida con formato mmddaaaa 7-16-1969  salida con formato aaaammdd 1969-7-16  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  salida con formato ddmmaaaa 01-01-2025  salida con formato mmddaaaa 01-01-2025  salida con formato aaaammdd 2025-01- 01 |   · |

·

#### **Métodos de operadores.**

Python es un lenguaje de muy alto nivel de abstracción que por ende, realiza operaciones de manera distinta a otros. Uno de esos casos es cuando el intérprete identifica un operador.

Cuando el intérprete identifica a un operador en una expresión, no realiza la operación relacionada con el operador directamente, sino que invoca un método especial.

Para que me tome el comportamiento de la suma debemos implementar los Métodos de operador recíprocos.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | class MiClase:  def \_\_init\_\_(self, valor):  self.valor = valor  def \_\_add\_\_(self, new\_self):  return MiClase(self.valor + new\_self.valor)  def \_\_sub\_\_(self, new\_self):  return MiClase(self.valor - new\_self.valor)  def \_\_mul\_\_(self, new\_self):  return MiClase(self.valor \* new\_self.valor)  def \_\_truediv\_\_(self, new\_self):  if new\_self.valor == 0:  print ("error")  return MiClase(new\_self.valor)  return MiClase(self.valor / new\_self.valor)  num\_1ro = MiClase(9)  num\_2do = MiClase(2)  #------------------------------------------------------------------  resultado = num\_1ro + num\_2do #con + llama al metodo \_\_add\_\_ para sumar los objetos  print (f"{num\_1ro.valor}+{num\_2do.valor}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------  resultado = num\_1ro - num\_2do #con - llama al metodo \_\_sub\_\_ para restar los objetos  print (f"{num\_1ro.valor}-{num\_2do.valor}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------  resultado = num\_1ro \* num\_2do #con \* llama al metodo \_\_sub\_\_ para multiplicar los objetos  print (f"{num\_1ro.valor}\*{num\_2do.valor}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------  resultado = num\_1ro / num\_2do #con / llama al metodo \_\_truediv\_\_ para dividir los objetos  print (f"{num\_1ro.valor}/{num\_2do.valor}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------  num\_2do = MiClase(0)  #------------------------------------------------------------------  resultado = num\_1ro / num\_2do #con / llama al metodo \_\_truediv\_\_ para dividir los objetos  print (f"{num\_1ro.valor}/{num\_2do.valor}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------ |   ·Salida esperada por consola   |  | | --- | | 9+2=11  --------------------------------------------------  9-2=7  --------------------------------------------------  9\*2=18  --------------------------------------------------  9/2=4.5  --------------------------------------------------  error  9/0=0  -------------------------------------------------- | |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | class MiClase:  def \_\_init\_\_(self, valor):  self.valor = valor  def \_\_add\_\_(self, new\_value):  if isinstance (self ,MiClase) and isinstance (new\_value ,MiClase):  salida = MiClase(self.valor + new\_value.valor).valor  print ("obj+obj")  elif isinstance (self ,MiClase) and isinstance (new\_value ,(int,float)):  salida = MiClase(self.valor + new\_value)  print ("obj+valor")  else:  exit()  raise "Error"  return salida  #------------------------------------------------------------------  num\_1ro = MiClase(9)  num\_2do = MiClase(2)  resultado = num\_1ro + num\_2do #con + llama al metodo \_\_add\_\_ para sumar los objetos  print (f"{num\_1ro.valor}+{num\_2do.valor}={resultado}")  print ("-"\*50)  #------------------------------------------------------------------  num\_1ro = MiClase(9)  num\_2do=8  resultado = num\_1ro + num\_2do  print (f"{num\_1ro.valor}+{num\_2do}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------  num\_1ro = 8  num\_2do = MiClase(2)  resultado = num\_1ro + num\_2do  print (f"{num\_1ro}+{num\_2do.valor}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------ |   ·Salida esperada por consola   |  | | --- | | obj+obj  9+2=11  --------------------------------------------------  obj+valor  9+8=17  --------------------------------------------------  **valor+obj**  **Traceback (most recent call last):**  **resultado = num\_1ro + num\_2do**  **~~~~~~~~^~~~~~~~~**  **TypeError: unsupported operand type(s) for +: 'int' and 'MiClase'**  **--------------------------------------------------** |   · |

#### **Métodos de operador recíprocos.**

|  |  |  |  |
| --- | --- | --- | --- |
| Los métodos de operador recíprocos, también conocidos como operadores de reflexión, son métodos mágicos que permiten que un objeto actúe como operando derecho en una operación donde normalmente actuaría como operando izquierdo. Estos métodos son útiles para definir cómo se debe comportar un objeto cuando se usa en una operación donde normalmente el otro objeto sería el operando izquierdo.  Aquí hay algunos ejemplos de métodos de operador recíprocos:  ·   |  |  |  | | --- | --- | --- | | \_\_radd\_\_(self, other): Método que define el comportamiento de la adición (+) cuando el objeto es el operando derecho. Se llama cuando el operando izquierdo no implementa el método \_\_add\_\_.  ·Código Python   |  | | --- | | class MiClase:  def \_\_init\_\_(self, valor):  self.valor = valor  def \_\_add\_\_(self, new\_value):  if isinstance (self ,MiClase) and isinstance (new\_value ,MiClase):  salida = MiClase(self.valor + new\_value.valor).valor  print ("obj+obj")  elif isinstance (self ,MiClase) and isinstance (new\_value ,(int,float)):  salida = MiClase(self.valor + new\_value)  print ("obj+valor")  else:  exit()  raise "Error"  return salida  def \_\_radd\_\_(self, new\_value):  print ("valor+obj")  salida = MiClase(self.valor + new\_value)  return salida    num\_1ro = MiClase(9)  num\_2do = MiClase(2)  #------------------------------------------------------------------  resultado = num\_1ro + num\_2do #con + llama al metodo \_\_add\_\_ para sumar los objetos  print (f"{num\_1ro.valor}+{num\_2do.valor}={resultado}")  print ("-"\*50)  #------------------------------------------------------------------  num\_2do=8  resultado = num\_1ro + num\_2do  print (f"{num\_1ro.valor}+{num\_2do}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------  num\_1ro = 8  num\_2do = MiClase(2)  resultado = num\_1ro + num\_2do  print (f"{num\_1ro}+{num\_2do.valor}={resultado.valor}")  print ("-"\*50)  #------------------------------------------------------------------ |   ·Salida esperada por consola   |  | | --- | | obj+obj  9+2=11  --------------------------------------------------  obj+valor  9+8=17  --------------------------------------------------  valor+obj  8+2=10  -------------------------------------------------- |   · |   ·   * \_\_rsub\_\_(self, other): Método que define el comportamiento de la sustracción (-) cuando el objeto es el operando derecho. * \_\_rmul\_\_(self, other): Método que define el comportamiento de la multiplicación (\*) cuando el objeto es el operando derecho. * \_\_rdiv\_\_(self, other): Método que define el comportamiento de la división (/) cuando el objeto es el operando derecho. * \_\_rmod\_\_(self, other): Método que define el comportamiento del módulo (%) cuando el objeto es el operando derecho. * \_\_rand\_\_(self, other): Método que define el comportamiento del operador de bits "y" (&) cuando el objeto es el operando derecho.   Estos métodos son útiles cuando deseas que tus objetos interactúen de manera coherente con tipos de datos incorporados o personalizados en operaciones donde normalmente no serían el operando izquierdo |

En el ejemplo previo, el orden del operador tiene relevancia, ya que los objetos de tipo int y de tipo float, los cuales se invocan primero por estar a la izquierda del operador, no cuentan con una implementación de \_\_add\_\_ para la clase SuperficieCuadrada.

o mismo ocurre con la función sum(), la cual comienza a realizar la suma desde 0, el cual es de tipo int.

uando un objeto de tipo numérico no encuentra una implementación adecuada, busca un método recíproco en el objeto a la derecha del operador. En este caso busca al método \_\_radd\_\_().

os métodos de operador recíprocos de son:

* \_\_radd\_\_()
* \_\_rdiv\_\_()
* \_\_rmod\_\_()
* \_\_rmul\_\_()
* \_\_rsub\_\_()

Ejemplo:

### **El método \_\_iter\_\_()**

Convierte a las instancias de la clase en objetos iterables.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | |  |   ·Salida esperada por consola   |  | | --- | |  |   · |

·

## **Métodos estáticos y métodos de clase**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Los métodos estáticos y los métodos de clase son dos tipos de métodos que se pueden definir en una clase en Python, pero tienen propósitos y comportamientos diferentes.  Métodos Estáticos:  Los métodos estáticos son métodos que pertenecen a la clase en sí, en lugar de una instancia específica de la clase. No requieren acceder ni modificar atributos de instancia ni de clase, y no reciben automáticamente el parámetro self (que se usa para referirse a la instancia) ni el parámetro cls (que se usa para referirse a la clase). Se definen utilizando el decorador @staticmethod y se pueden llamar directamente desde la clase, sin necesidad de crear una instancia.  Los métodos estáticos son útiles cuando necesitas una función dentro de una clase que no dependa de ningún estado de instancia o de clase. Por ejemplo, funciones auxiliares que reAnitan cálculos independientes.  python  ·   |  |  |  | | --- | --- | --- | | ·Código Python   |  | | --- | | class Calculadora:  @staticmethod  def sumar(a, b):  return a + b  resultado = Calculadora.sumar(5, 7)  print(resultado) |   ·Salida esperada por consola   |  | | --- | | 12 |   · |   ·  Métodos de Clase:  Los métodos de clase son métodos que pertenecen a la clase en sí, pero también tienen acceso a la clase y pueden modificar sus atributos de clase. Se definen utilizando el decorador @classmethod. Los métodos de clase reciben el parámetro cls, que se usa para acceder y modificar los atributos de clase.  Los métodos de clase son útiles cuando necesitas realizar operaciones que involucran la clase en su conjunto, en lugar de instancias individuales. Por ejemplo, para realizar cálculos basados en atributos de clase.  ·   |  |  |  | | --- | --- | --- | | ·Código Python   |  | | --- | | class Persona:  contador = 0  def \_\_init\_\_(**self**, nombre):  self.nombre = nombre  Persona.contador += 1  **@classmethod**  def obtener\_contador(**cls**):  return **cls.**contador  persona1 = Persona("Anita")  persona2 = Persona("Juan")  print(Persona.obtener\_contador()) # Imprime 2 |   ·Salida esperada por consola   |  | | --- | | 2 |   · |   ·  En resumen, los métodos estáticos no dependen de instancias ni de la clase y se definen con @staticmethod, mientras que los métodos de clase pueden acceder a la clase y se definen con @classmethod. Ambos tipos de métodos son útiles para diferentes escenarios en la programación orientada a objetos. |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | |  |   ·Salida esperada por consola   |  | | --- | |  |   · |

·

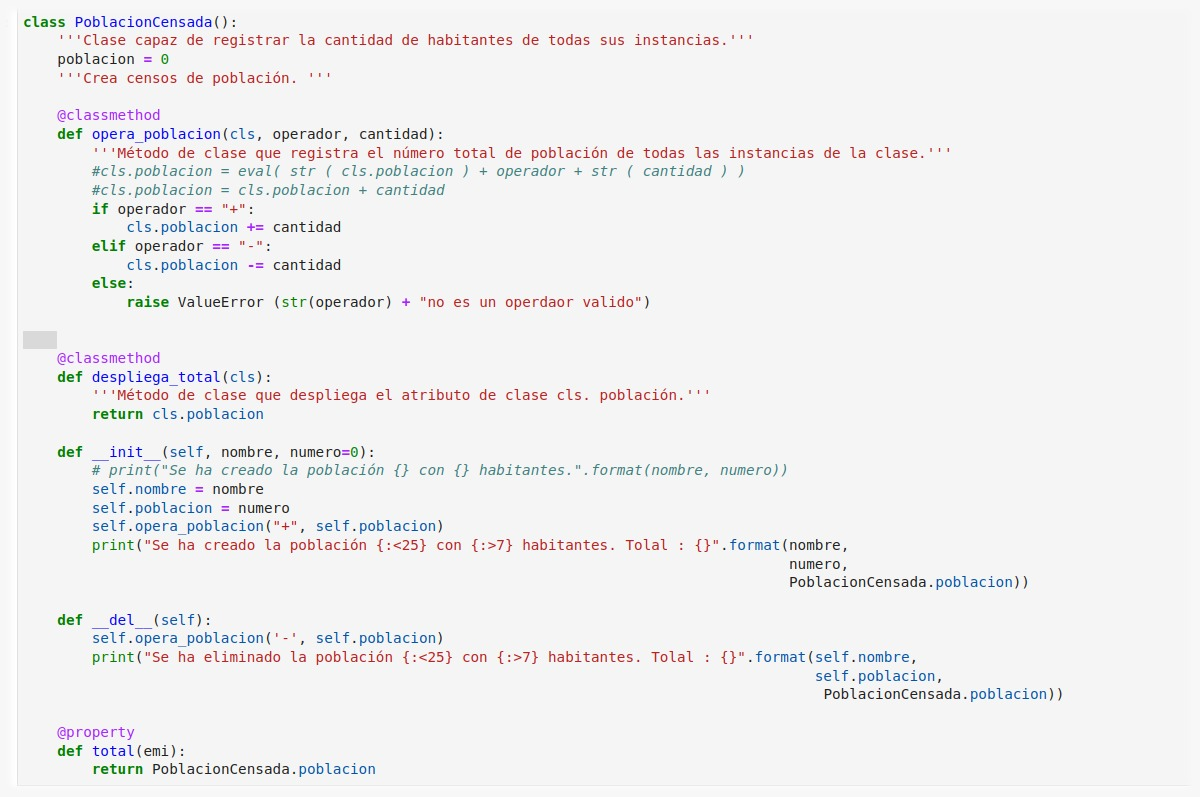
Existen algunos casos en los que se requiera de extender o restringir el ámbito de un método. Los métodos de clase y los métodos estáticos son ejemplo de este tipo de casos.

### **Métodos de clase**

Un método de clase puede modificar el estado de una clase, accediendo a los atributos de dicha clase, aún cuando el método es invocado desde un objeto. En lugar de definirse utilizando \_self\_ como primer parámetro, se utiliza \_cls\_.

Los métodos de clase se definen con la siguiente sintaxis:

|  |
| --- |
| @classmethod  def <nombre>(cls , <parámetros>):  ... |

·

Ejemplo:

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | |  |   ·Salida esperada por consola   |  | | --- | |  |   · |

·

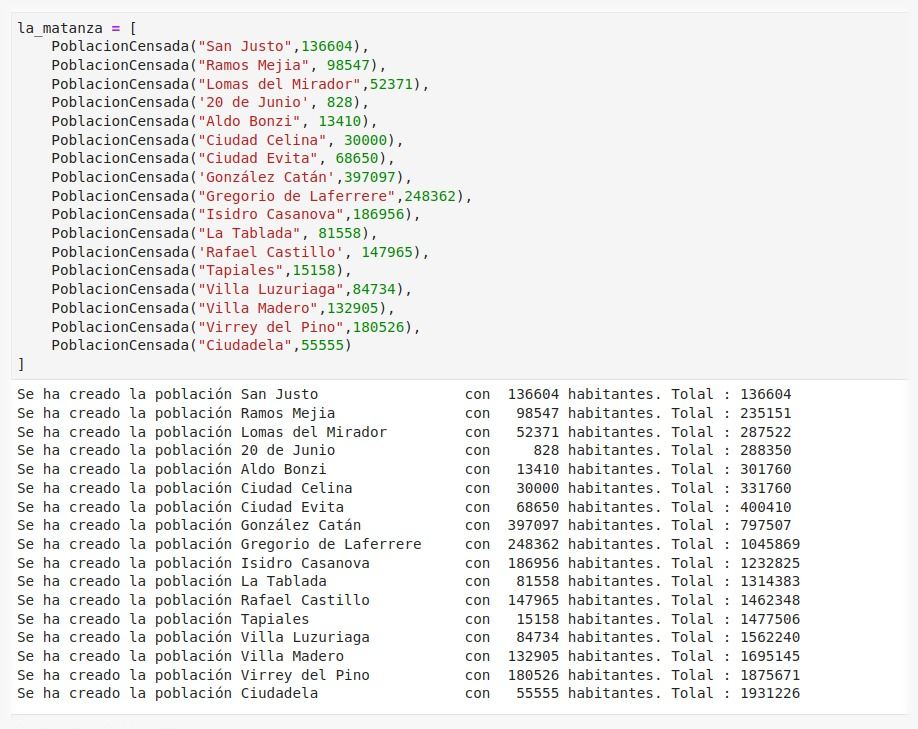
Esta es una clase llamada `PoblacionCensada` que registra la cantidad total de habitantes de todas sus instancias a través del atributo de clase `poblacion`. Tiene dos métodos de clase, `opera\_poblacion` que agrega o resta un número a `poblacion` y `despliega\_total` que imprime el valor actual de `poblacion`. Además, tiene un constructor `\_\_init\_\_` que recibe el nombre de la población y su número de habitantes, registra esta información en las propiedades de la instancia y luego agrega el número de habitantes al atributo de clase `poblacion`. También tiene un destructor `\_\_del\_\_` que resta el número de habitantes de la instancia eliminada al atributo de clase `poblacion`. Al final, hay un decorador `@property` para el método `total` que muestra el valor actual de `poblacion`. En resumen, esta clase proporciona una manera conveniente de hacer un seguimiento del total de habitantes de todas las poblaciones creadas a través del atributo de clase `poblacion`.

veamos como funciona:

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | |  |   ·Salida esperada por consola   |  | | --- | |  |   · |

·



·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | |  |   ·Salida esperada por consola   |  | | --- | |  |   · |

·

### **Métodos Estáticos**

Los métodos estáticos están restringidos en su ámbito, de tal manera que no tienen acceso a los atributos del objeto. Se definen de forma idéntica a una función, sin necesidad de ingresar el parámetro inicial *self*.

Sintaxis:

|  |
| --- |
| @staticmethod  def <nombre>(<parámetros>):  … |

·

El método estático es un método común "como una función mas", pero solamente la puede llamar una instancia de la clase o la clase misma

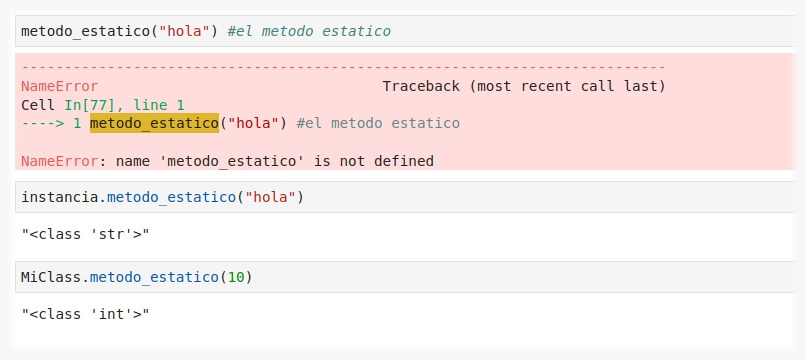
****Ejemplo:****

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | |  |   ·Salida esperada por consola   |  | | --- | |  |   · |

**·**

hay un método estático `metodo\_estatico`, que no depende de la clase ni de una instancia de la misma. Este método simplemente devuelve el tipo de su argumento `x` en una cadena. Pero solo puede ser llamado por la clase o una instancia de la clase



#### **ejemplo de métodos estáticos (adaptación)**

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | |  |   ·Salida esperada por consola   |  | | --- | |  |   · |

·

## **Herencia**

La herencia es un concepto fundamental en la programación orientada a objetos, que permite a un objeto o clase "heredar" propiedades y comportamientos de otro objeto o clase. En otras palabras, una clase puede derivarse de otra clase existente y heredar sus propiedades y métodos, y además puede agregar nuevas propiedades y métodos propios.

Por ejemplo, podemos tener una clase "Animal" con propiedades como "nombre" y "edad", y métodos como "comer" y "dormir", y luego crear una nueva clase "Perro" que hereda las propiedades y métodos de la clase "Animal", pero también incluye sus propios métodos y propiedades, como "ladrar" y "raza".

La herencia permite organizar el código de forma jerárquica y modular, facilitando la reutilización de código y permitiendo la creación de aplicaciones más complejas y escalables.

### **Herencia simple**

Es posible crear nuevas clases a partir de una o varias clases mediante la herencia.

La clase original se denomina "superclase" y la clase que hereda los atributos y métodos de la superclase se denomina "subclase".

Se pueden definir atributos y métodos adicionales a la superclase e incluso se pueden sobrescribir los atributos y métodos heredados en la subclase.

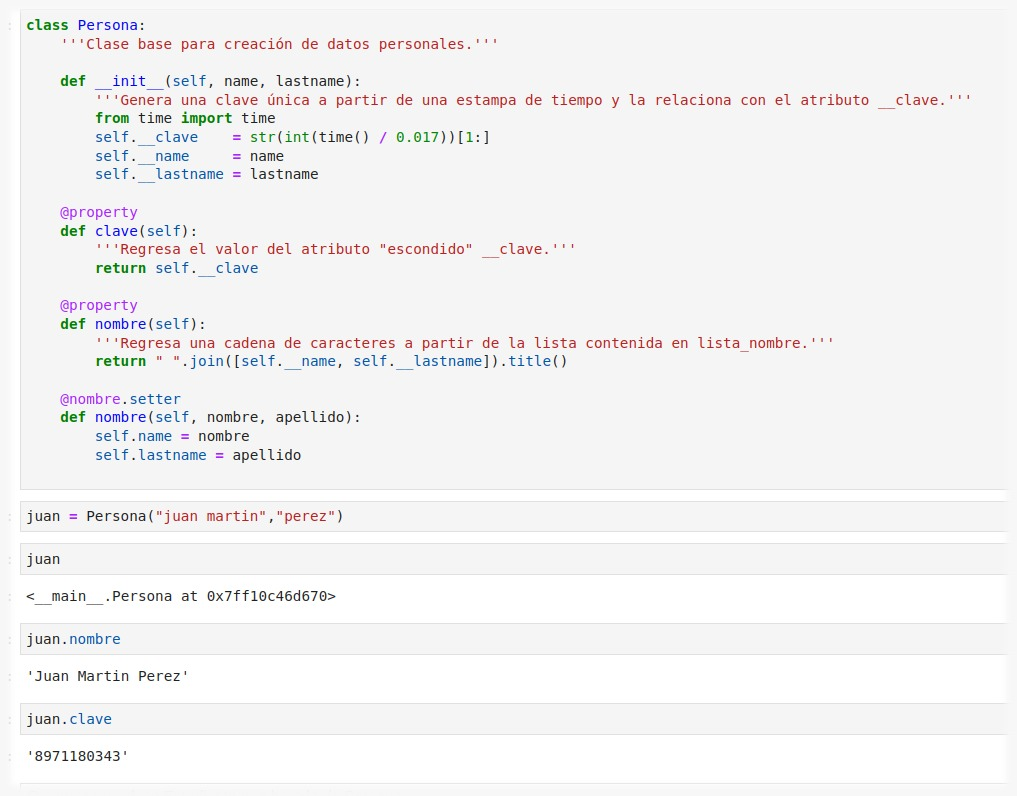
La sintaxis de herencia es la siguiente:

|  |
| --- |
| class <nombre>(<nombre de la superclase>):  ...  ... |

·

Ejemplo:

Vamos a empezar creando una clase padre Persona



Creamos una clase Estudiante que hereda de Persona

### **Funcion `super()`**

Se utiliza para acceder a los métodos y atributos de la clase padre en una clase derivada (una subclase). Es útil cuando necesitas llamar a un método de la clase padre que ha sido modificado en una subclase en particular.

Un ejemplo sencillo sería el siguiente:

|  |
| --- |
| class Animal:  def \_\_init\_\_(self, especie):  self.especie = especie  def caminar(self):  print("Estoy caminando")  class Perro(Animal):  def \_\_init\_\_(self, especie, nombre):  super().\_\_init\_\_(especie)  self.nombre = nombre  def ladrar(self):  print("Guau! Soy un perro.") |

·

En este ejemplo, la clase `Perro` hereda de la clase `Animal`. La clase `Animal` contiene un método `caminar`. La clase `Perro` define su propio método `ladrar`. Al definir su propio constructor `\_\_init\_\_`, debemos llamar al constructor de la clase padre `Animal` utilizando `super().\_\_init\_\_(especie)` para asegurarnos de que estamos inicializando correctamente la clase `Perro`.

Siguiendo con nuestro ejemplo:



Este es un ejemplo de una clase llamada "Estudiante" que hereda de la clase "Persona" utilizando la palabra clave "super()". La clase tiene un constructor que toma dos argumentos "name" y "lastname" y los pasa al constructor de la clase padre "Persona", a través del método "super().\_\_init\_\_(name,lastname)". Además, el constructor de la clase "Estudiante" inicializa el atributo "self.\_materias" como un conjunto vacío para almacenar las materias que el estudiante va a inscribir.

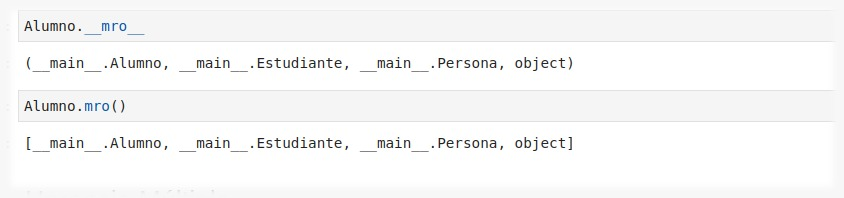
La clase "Estudiante" tiene un método llamado "inscripcion" que toma un argumento "materia" y lo añade al conjunto "self.\_materias". Este método permite agregar nuevas materias a la lista de materias en la que el estudiante está inscrito.

La clase tiene también un método decorado con el decorador "@property" llamado "materias", que se utiliza para acceder al conjunto de materias en el que el estudiante está inscrito. Este método devuelve una tupla de materias utilizando el método "tuple(self.\_materias)", lo que permite acceso de solo lectura a las materias.

También se puede heredar de una clase que a su vez hereda de otra.

ejemplo:

Para conoces el orden de la herencia se puede ver con el método \_\_mro\_\_ y va de derecha a izquierda



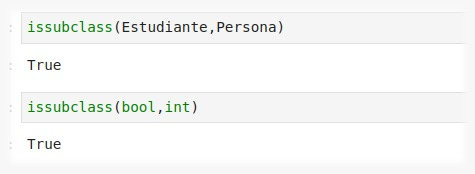
### **La función \_issubclass()\_.**

Para saber si una clase es subclase de otra, se utiliza la función \_subclass()\_.

Sintaxis:

issubclass(<clase\_1>, <clase\_2>)

Ejemplo:



## **Herencia Múltiple**

En Python, por ejemplo, es posible heredar de varias clases simultáneamente definiendo una lista separada por comas con los nombres de las clases padres. Los atributos y métodos definidos en las clases padres pueden utilizarse en la clase hija (subclase) como si hubieran sido definidos allí mismo.

La herencia múltiple puede ser una herramienta poderosa en ciertos contextos, ya que permite crear nuevas clases que combinan características de varias clases padres. Sin embargo, también puede ser complicada y potencialmente generar conflictos si las clases padres tienen atributos o métodos con el mismo nombre. Por lo tanto, es recomendable utilizar la herencia multiple con precaución y siempre tener en cuenta la compatibilidad y la coherencia de las clases padre y las subclases creadas.

Sintaxis:

|  |
| --- |
| class <nombre>(<superclase 1>, <superclase 2>, ..., <superclase n>) |

·

La primera superclase que se ingrese sobrescribirá los atributos de la siguiente y así sucesivamente.

Ejemplo:





* En este caso, la clase Ornitorrinco es subclase de Reptil y Mamífero, las cuales a su vez son subclases de Animal.
* El método \_\_init\_\_() de Ornitorrinco sobrescribe al método \_\_init\_\_() de Animal, pero es recuperado mediante la función super().
* Debido a que la superclase Reptil fue ingresado antes que Mamífero en Ornitorrinco, el método reproduccion() de Reptil es el que va a sobrescribir al resto.

### **Mixin**

﻿﻿

Los mixins son una colección de clases que contienen métodos, los cuales pueden ser intercambiables. Aprovechan la herencia múltiple para conformar clases modulares.

Por lo general estas clases contienen métodos que corresponden a bibliotecas que no sobrescriben a las otras superclases.

El uso de mixins es muy controvertido e incluso algunos lo consideran un "antipatrón", pero aún así es aceptable en Python

ejemplo:



﻿﻿  
Es un ejemplo simple para demostrar como una clase puede usar herencia múltiple y utilizar métodos de las clases padres.

## **Polimorfismo y Duck Typing**

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?

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

#### **Duck Typing en Python**

|  |  |  |  |
| --- | --- | --- | --- |
| Duck Typing es un concepto en programación que se refiere a la forma en que Python trata los tipos de datos y la compatibilidad de operaciones basada en su comportamiento en lugar de su tipo específico. La idea detrás de Duck Typing es que si un objeto se comporta de una manera particular (como un pato), entonces se le considera como tal, independientemente de su tipo real.  La famosa frase que resume el Duck Typing es: "Si parece un pato, nada como un pato y grazna como un pato, entonces probablemente es un pato".  En Python, esto significa que no importa el tipo exacto de un objeto, sino más bien si el objeto admite ciertas operaciones. Si un objeto puede realizar las operaciones requeridas, se considera que tiene el "tipo correcto" para ese contexto.  Por ejemplo, si tienes una función que espera un objeto que tenga un método quack() para hacer un sonido similar al graznido de un pato, no importa si el objeto es una instancia de una clase llamada "Pato" o de cualquier otra clase. Lo que importa es si el objeto tiene el método quack().   |  |  |  | | --- | --- | --- | | **Duck Typing**  ·Código Python   |  | | --- | | class Pato:  def quack(self):  print("Quack, quack!")  class Persona:  def quack(self):  print("Hola, soy una persona haciendo un quack ficticio.")  def hacer\_sonido(objeto):  objeto.quack()  pato = Pato()  persona = Persona()  hacer\_sonido(pato)  hacer\_sonido(persona) |   Salida esperada por consola   |  | | --- | | Imprime "Quack, quack!"  Imprime "Hola, soy una persona haciendo un quack ficticio." |   · |   ·  En este ejemplo, tanto la clase Pato como la clase Persona son compatibles con la función hacer\_sonido() porque ambas tienen un método quack(), a pesar de que la clase Persona no tiene ninguna relación con patos en la realidad. Esto es un ejemplo de cómo Python implementa el Duck Typing, ya que se centra en el comportamiento (la presencia del método quack()) en lugar del tipo específico de objeto.  **1. Flexibilidad y Adaptabilidad:**  Duck Typing permite que el código sea más flexible y adaptable a diferentes situaciones. No es necesario preocuparse por los tipos exactos de los objetos; en su lugar, solo se necesita asegurarse de que los objetos tengan el comportamiento requerido.  **2. Menos Restricciones en la Jerarquía de Clases:**  A diferencia de otros lenguajes de programación donde la herencia determina la compatibilidad, en Python, los objetos pueden compartir comportamientos sin necesidad de heredar de una misma clase base. Esto promueve una estructura de código más simple y menos rígida.  **3. Interfaces Implícitas:**  En lugar de definir interfaces de manera explícita como en algunos lenguajes, Python permite que las interfaces se definan de manera implícita a través del comportamiento compartido entre objetos.  **4. Facilita la Creación de APIs Genéricas:**  Duck Typing es especialmente útil al crear APIs genéricas que pueden funcionar con múltiples tipos de objetos. Esto aumenta la reutilización y evita la necesidad de escribir código específico para cada tipo.  **5. Menos Dependencia de Tipos Específicos:**  Duck Typing reduce la necesidad de depender en exceso de tipos de datos específicos. Esto hace que el código sea más independiente de las implementaciones concretas y más enfocado en lo que realmente importa: el comportamiento.  **6. Prueba de Conceptos Más Rápida:**  Duck Typing puede agilizar la fase de prueba de concepto y prototipado, ya que puedes intercambiar objetos de manera más sencilla y ver cómo funcionan en un contexto específico.  **7. Uso de Métodos Mixins:**  Los mixins son clases pequeñas que proporcionan ciertas funcionalidades adicionales a otras clases. Duck Typing es esencial para la efectiva implementación y uso de mixins, ya que los objetos solo necesitan tener el comportamiento esperado para aprovechar estas funcionalidades.  **8. Desafíos Potenciales:**  Aunque Duck Typing ofrece muchas ventajas, es importante mencionar que puede llevar a errores en tiempo de ejecución si no se tiene cuidado. Si un objeto no tiene el comportamiento esperado, se generará una excepción cuando se intente acceder a ese comportamiento. |

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?



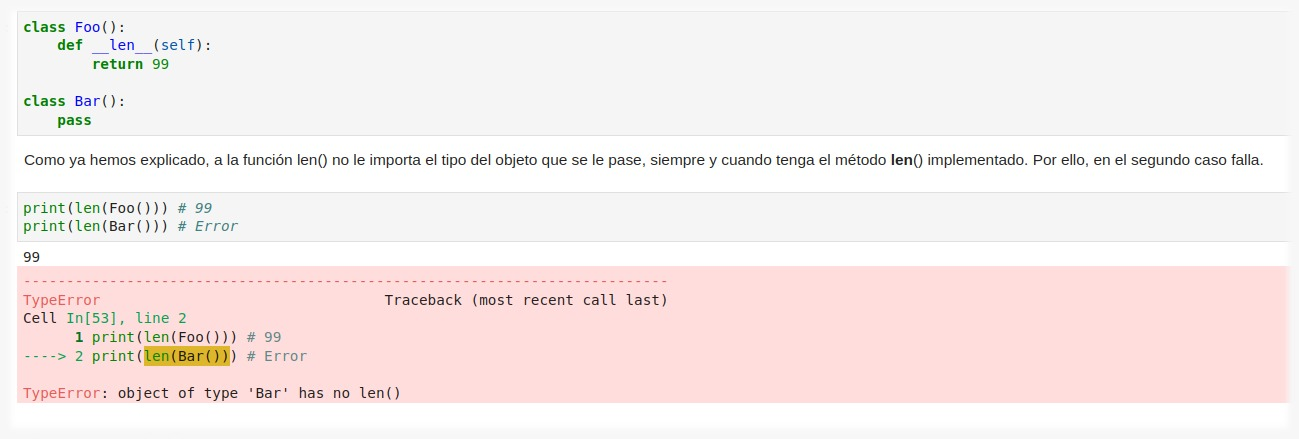


**Ejemplo len()**

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.



## **Composición**

La composición es una técnica que permite crear objetos complejos combinando objetos más simples. Esto se logra mediante la creación de clases que contienen instancias de otras clases como atributos. En lugar de heredar de una clase base, una clase utiliza objetos de otras clases para proporcionar funcionalidad.

Ejemplo :

En esta clase utilizamos herencia de las clases Previas de Alumno y composición de la clase Fecha que vimos previamente en la sección de métodos mágicos

