

# MÓDULO 4 - Parte 1

Definiciones.

Abstracción.

Clase

y

objetos.

Atributos

y

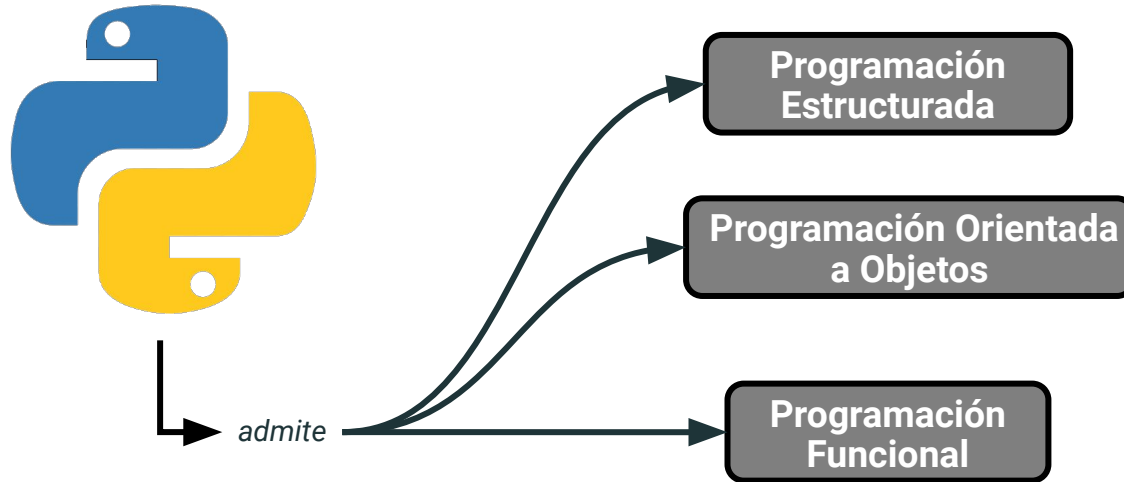
métodos.

Constructores.

Métodos de Acceso Setter y Getter.

1000  
PROGRAMADORES

## >Python - Multiparadigma



# >Programación Estructurada

## Características

- Los programas son más fáciles de entender.
- Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica.
- La estructura del programa es más clara, las instrucciones están más relacionadas entre sí, y es más fácil comprender lo que hace cada función.
- Favorece la reducción del esfuerzo en las pruebas.
- Aumenta la productividad del programador.

# >Programación Estructurada

## Enfoque del Paradigma

Los programas consisten en una sucesión de instrucciones o conjunto de sentencias, como si el programador diera órdenes concretas.

# >Programación Orientada a Objetos - P00

## Características

- Es una forma especial de programar, más cercana a la forma de expresar las cosas en la vida real que otros tipos de programación.
- Hay que pensar de una manera distinta, para escribir programas en términos de objetos, propiedades, métodos y otros conceptos nuevos.

# >Programación Orientada a Objetos - P00

## Características

- El adecuado diseño de clases favorece la reusabilidad.
- La facilidad de añadir, suprimir o modificar nuevos objetos nos permite hacer modificaciones de una forma muy sencilla.

# >Programación Orientada a Objetos - P00

## Enfoque del Paradigma

**P00** propone resolver un problema computacional a partir de la colaboración entre objetos.

## >Abstracción

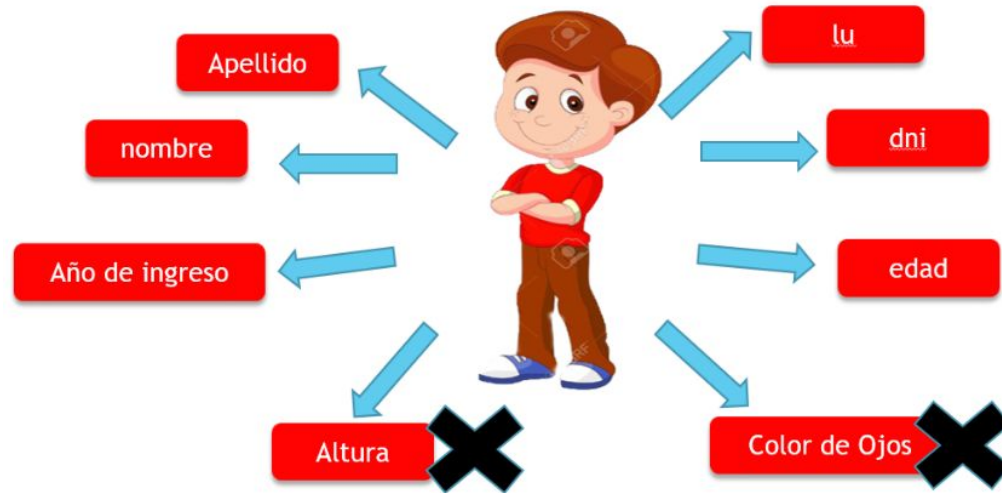
En un problema computacional, la **abstracción** brinda a los programas orientados a objetos sencillez de leer y comprender y mantener, permiten ocultar detalles de implementación dejando visibles sólo los detalles más relevantes.

Haremos una abstracción de la realidad para diseñar un objeto. Es decir, expresamos las características esenciales de un objeto, las cuales permiten distinguirlo de los demás.



## >Abstracción

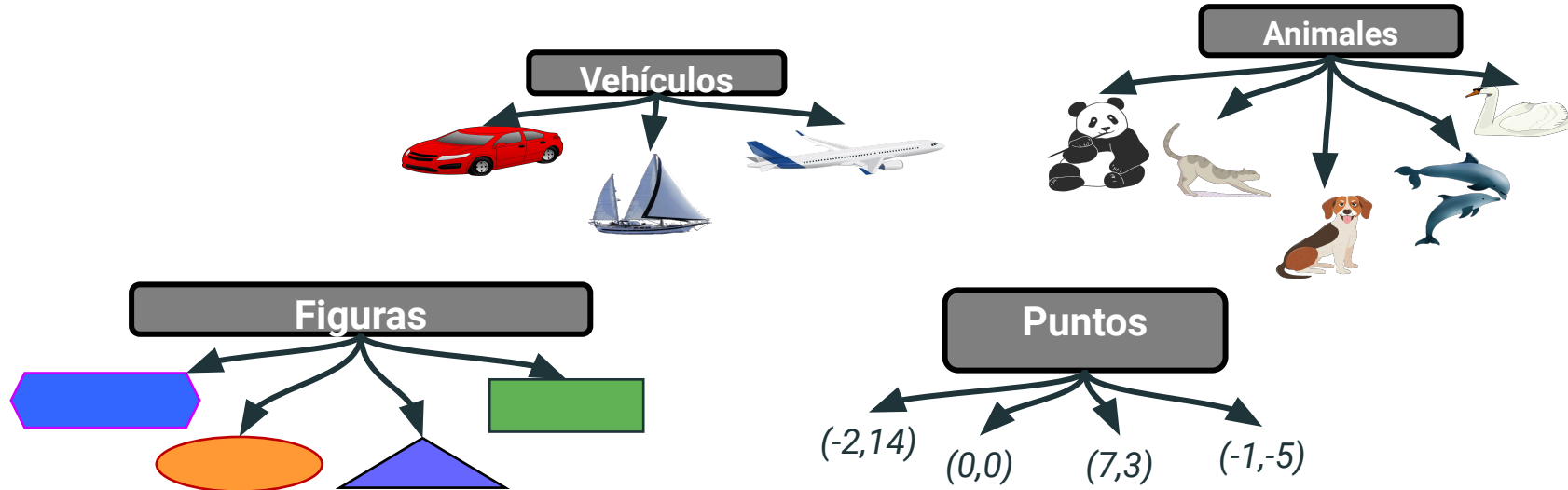
Por ejemplo, podemos abstraer a una **Persona**



## >¿A qué nos referimos con Objetos?

Tratamos de establecer una equivalencia entre un objeto del mundo real con un componente software.

Por ejemplo:



## >¿A qué nos referimos con Objetos?

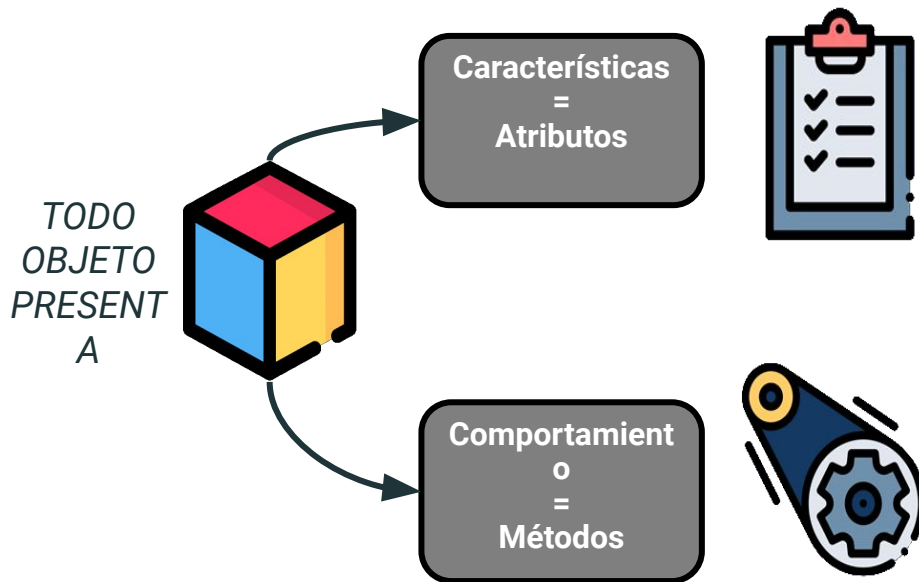
Todos los objetos presentan dos componentes principales:

- Un conjunto de **características**
- Un **comportamiento** determinado

Como vemos en estos ejemplos, un auto, una moto, un velero y un avión tienen características en común como **color, marca y modelo**.

Su comportamiento puede ser descrito con operaciones como **frenar, acelerar o girar**.

## >Componentes de un Objeto



### Atributo

*Contenedor de un tipo de dato asociado a un objeto, cuyo valor puede ser alterado por la ejecución de algún método.*

### Método

*Algoritmo asociado a un objeto cuya ejecución se desencadena tras la recepción de un "mensaje".*

## >Clase

Las clases son plantillas para la creación de objetos. Como tal, la clase forma la base para la programación orientada a objetos, la cual es una de los principales paradigmas de desarrollo de software en la actualidad.

Una clase es una plantilla o molde para crear objetos. También a una clase se le llama modelo.

```
class ClassName:  
    pass
```

## >Clase

Como convención en el paradigma orientado a objetos, la primera letra del nombre de una clase empieza con mayúscula.

¿Cómo crearemos una clase en Python? La estructura de clase más simple en Python luciría de la siguiente manera:

```
class ClassName:  
    pass
```

## >Clase

Como puedes ver, la definición de una clase comienza con la palabra clave **class**, y ClassName sería el nombre de la clase (identificador)

Ahora vamos a definir una clase Persona (persona), que por el momento no contendrá nada, excepto la declaración de **pass**. Según la documentación de Python:

La sentencia **pass** no hace nada. Puede ser utilizada cuando se requiere una sentencia sintácticamente pero el programa no requiere acción alguna.

## >Clase

La sentencia **pass** no hace nada. Puede ser utilizada cuando se requiere una sentencia sintácticamente pero el programa no requiere acción alguna.

```
class Persona:  
    pass
```



## >Instancia de una Clase - Objeto

Teniendo una **clase** podremos crear a partir de ella **objetos** con características específicas. Es decir, emplearemos esa “*plantilla*” o “*molde*” para crear un objeto.

A este proceso se lo conoce como **instanciar**, y nos permite generar una **instancia** u **objeto**.

En **P00** decimos que *las instancias tienen vida*.

Para nuestra clase **Persona**, podemos generar tantas instancias como sean necesarias para resolver un problema.

## >Instancia de una Clase - Objeto

```
personal = Persona()
```

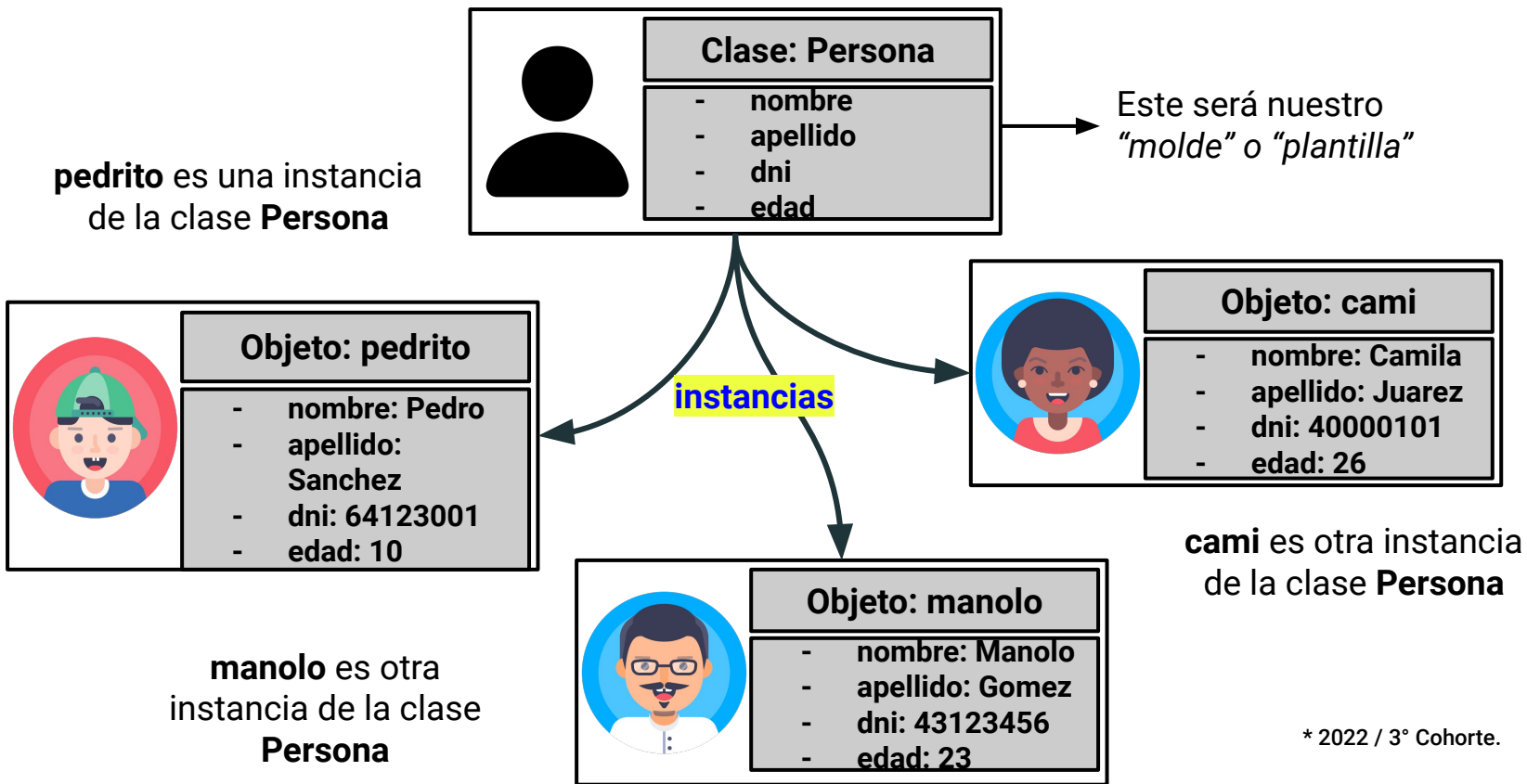


Para **instanciar un objeto** lo haremos como si se tratase de la asignación de una variable normal.

*Nombre de la variable igual a la clase con los ().*

Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

# >Instancia de una Clase - Objeto



## >Atributos

Como vimos previamente un **objeto** cuenta con atributos, que son las características que puede tener.

Si el objeto es **Persona**, los atributos podrían ser: dni, nombre, apellido, edad, etc...

Los atributos describen el estado de un objeto. Los atributos pueden ser de cualquier tipo de dato.

```
class Persona:  
    pass
```



```
#Dentro de __init__() definiremos. Los atributos  
#de Persona y le daremos un estado inicial  
#realizando asignaciones sobre estos  
def __init__(self):  
    self.nombre = ''  
    self.apellido = ''  
    self.dni = 0  
    self.edad = 0
```

## >Métodos

La definición de un método (de instancia) es análoga a la de una función ordinaria, pero incorporando un parámetro **self** es una variable que representa la instancia de la **clase**.

Dentro de una clase definirán métodos como ser el constructor, métodos de acceso (setter y getter), y más adelante métodos personalizados creados por nosotros mismos.



Los métodos representan la funcionalidad de un objeto. Hay métodos como el constructor, setter, getter, y los métodos que podemos definir nosotros mismos dentro de una clase para darle funcionalidad

## >Constructor

El **constructor** es un método que se utiliza para inicializar todos los atributos de las instancias.

Particularmente en **Python**, un constructor se define mediante el método de nombre `__init__()` para todas las clases.

## >Constructor

Al instanciar objetos de esta clase **Persona**, todos ellos tendrán que realizar el comportamiento que vemos en `__init__()`. Es decir, cualquier objeto de esta clase, al momento de ser instanciado tendrá nombre y apellido nulos o vacíos, dni y edad serán 0.

```
class Persona:

    def __init__(self):
        self.nombre = ''
        self.apellido = ''
        self.dni = 0
        self.edad = 0
```

Por supuesto podemos definir un comportamiento de inicialización distinto si así lo necesitamos

## >Constructor

Al momento de *inicializar* los atributos de un objeto podemos asignar ciertos datos mediante los parámetros del método `__init__()`.

Para la clase **Persona**, podemos definir el constructor indicando cuales serán su **nombre**, **apellido**, **dni** y **edad** de la siguiente manera:

```
class Persona:

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



## >Constructor

Desafortunadamente en **Python**, no podemos definir varios constructores a la vez. Es decir, no podemos definir más de un método `__init__()` en la misma clase. Y si lo intentamos, solo se ejecutará el último en ser definido.

En su lugar podemos emplear los diferentes tipos de parámetros, como vimos en la unidad anterior (parámetros **posicionales**, **por default** u **opcionales**). Así podemos definir un **constructor** que permitirá establecer un estado predeterminado a los atributos de un objeto.

## >Constructor

El siguiente fragmento de código muestra cómo eliminar la necesidad de varios constructores con el constructor con parámetros por default.

```
class Persona:

    def __init__(self,nombre="Carlos",apellido="Santana",dni=10321444,edad=75):
        self.nombre = nombre
        self.apellido = apellido
        self.dni = dni
        self.edad = edad
```

## >Encapsulamiento

Como hemos visto previamente, los atributos definidos en un objeto son accesibles públicamente. Esto puede parecer extraño a personas desarrolladoras de otros lenguajes. En **Python** existe un cierto «sentido de responsabilidad» a la hora de programar y manejar este tipo de situaciones.

Una posible solución «pythónica» para la privacidad de los atributos es el uso de propiedades. La forma más común de aplicar propiedades es mediante el uso de decoradores **@property** y **@nombreAtributo.setter**

## >Encapsulamiento


Es el ocultamiento del estado de un objeto. Para lograr el encapsulamiento de los objetos, los atributos de los mismos deben ser únicamente modificados mediante operaciones. Para que los atributos de una clase no sean accedidos de manera directa debemos ***protegerlos***.

Para indicar el nivel de *visibilidad* o *acceso* de un atributo en Python empleamos la siguiente notación:

- Anteponer (“\_”) un guión bajo, nos indica que el atributo es ***protegido***, es decir, sólo deberíamos acceder a él dentro de la definición de la clase o desde una clase hija.
- Anteponer (“\_\_”) doble guión bajo, nos indica que el atributo es ***privado***, es decir, sólo deberíamos acceder a él dentro de la definición de la clase

## >Encapsulamiento

```
class Persona():  
    def __init__(self, nombre, apellido, dni, edad):  
        self.__nombre=nombre  
        self.__apellido=apellido  
        self.__dni=dni  
        self.__edad=edad
```



Aquí podemos ver que los atributos se encuentran **encapsulados**

## >Métodos de Acceso - Setter y Getter

Como dijimos anteriormente el **encapsulamiento** solo permite el acceso a un atributo mediante un método, estos métodos se conocen como **método de acceso** o bien **getter** y **setter**.

Los **getters** y **setters** se utilizan en **P00** para garantizar el principio de la encapsulación de datos.

El **getter** se emplea para obtener los datos y el **setter** para la asignación de los datos.

## >Métodos de Acceso - Setter y Getter

Para definir estos métodos en **Python** emplearemos decoradores y se identifican por tener un **@** (como lo veremos en el ejemplo).

El propósito principal de cualquier decorador es cambiar los métodos o atributos de la clase de tal manera que el usuario de la clase no necesite hacer ningún cambio en el código.

## >Métodos de Acceso - Setter y Getter

Una vez que los atributos están encapsulados para poder acceder a ellos emplearemos los siguientes decoradores: Ya sea obtener o modificar los datos de un objeto en particular, usaremos el decorador

- **@property** para un **getter**, que nos permitirá obtener un dato asociado a un atributo en concreto.
- **@nombreAtributo.setter** para un **setter**, que nos permitirá modificar un atributo de en particular.



## >Métodos de Acceso - Setter y Getter

```
@property  
def nombre(self):  
    return self.__nombre
```



Así definimos un **getter** para el atributo nombre, este método *retorna* dicho atributo que, como vemos se encuentra encapsulado.

```
@nombre.setter  
def nombre(self, nuevoNombre):  
    self.__nombre= nuevoNombre
```



Así definimos un **setter** para el atributo nombre, este método *asigna* un nuevo nombre al atributo que se encuentra encapsulado.

## >Objetos

Una vez creado el modelo o **Clase** con sus atributos y métodos, podemos instanciar objetos usando el constructor definido dentro de la clase.

Se pueden crear los objetos que uno necesite para resolver un problema en cuestión. Cada objeto es único e irrepetible por lo cual no puede haber objetos iguales. En **Python** instanciamos objetos de la siguiente manera:

## >Objetos

```
class Persona():  
    def __init__(self, nombre, apellido, dni, edad):  
        self.__nombre=nombre  
        self.__apellido=apellido  
        self.__dni=dni  
        self.__edad=edad
```

instanciamos  
3 objetos



```
pedrito= Persona("Pedro","Sanchez",46412301,10)  
cami= Persona("Camila","Juarez",382376168,26)  
manolo= Persona("Manolo","Gomez",43762129,23)
```

## >Método `__str__`

Como hemos visto con anterioridad es una tarea común el mostrar ciertos datos por consola, y para esto recurrimos a `print()`. En **P00** suele ocurrir algo similar, necesitaremos conocer una representación de nuestros objetos, pero si queremos mostrarlos a través de `print()` nos encontraremos con un resultado como este:

```
class Persona():  
    def __init__(self, nombre, apellido, dni, edad):  
        self.__nombre=nombre  
        self.__apellido=apellido  
        self.__dni=dni  
        self.__edad=edad
```

## >Método `__str__`

```
print(pedrito)
print(cami)
print(manolo)
```



```
<__main__.Persona object at 0x00000214E9BFBD0>
<__main__.Persona object at 0x00000214E9BFBD60>
<__main__.Persona object at 0x00000214E9BFBD00>
```

Como podemos apreciar al ***mostrar nuestros objetos, el resultado no es muy descriptivo***. Esto ocurre porque no hemos definido como representaremos a los objetos de la clase **Persona**, y para ello deberemos implementar un nuevo método llamado `__str__`

## >Método `__str__`

Este método retorna la representación de nuestro objeto como una cadena de texto.

```
def __str__(self):  
    cadena= "\nNombre: "+self.__nombre  
    cadena+= "\nApellido: "+self.__apellido  
    cadena+= "\nDNI: "+str(self.__dni)  
    cadena+= "\nEdad: "+str(self.__edad)  
    return cadena
```

## >Método `__str__`

Entonces si quisiéramos mostrar las mismas instancias del ejemplo anterior tendríamos este resultado

```
print(pedrito)  
print(cami)  
print(manolo)
```



```
Nombre: Pedro  
Apellido:  
Sanchez  
DNI: 46412301  
Edad: 10
```

```
Nombre: Camila  
Apellido:  
Juarez  
DNI: 382376168  
Edad: 26
```

```
Nombre: Manolo  
Apellido: Gomez  
DNI: 43762129  
Edad: 23
```

## >Objetos

Si definieramos un constructor con parámetros por default, como vimos anteriormente, podemos tener la siguiente instancia:

```
def __init__(self, nombre= "Carlos",  
apellido="Santana", dni=12372364, edad=75):  
    self.__nombre=nombre  
    self.__apellido=apellido  
    self.__dni=dni  
    self.__edad=edad
```

instanciamos 1 objeto

```
unaPersona= Persona()
```

Si mostramos el objeto **unaPersona**, veremos lo siguiente:

```
print(unaPersona)
```

```
Nombre: Carlos  
Apellido: Santana  
DNI: 12372364  
Edad: 75
```



## >Objetos

Por supuesto podemos acceder a un atributo en particular si contamos con un **método de acceso**.

Por ejemplo, podemos mostrar el atributo **nombre** de las instancias creada anteriormente:

```
@property
def nombre(self):
    return self.__nombre

@nombre.setter
def nombre(self,nuevoNombre):
    self.__nombre= nuevoNombre
```

```
print(pedrito.nombre)
print(cami.nombre)
print(manolo.nombre)
print(unaPersona.nombre)
```

```
Pedro
Camila
Manolo
Carlos
```

# Objetos con Atributo `datetime` (fecha)

Para crear un objeto **`datetime`** usaremos lo siguiente:

1. Importar **`datetime`**

```
from datetime import datetime
```

1. Crear un objeto fecha pasando el año, mes y día a elección

```
fecha = datetime(año,mes,día)
```

## >Objetos con Atributo datetime (fecha)

3. En el constructor de la clase **Persona** asignaremos la fecha con el atributo correspondiente.

```
class Persona():  
    def __init__(self, nombre, apellido, dni, edad, fecha):  
        self.__nombre=nombre  
        self.__apellido=apellido  
        self.__dni=dni  
        self.__edad=edad  
        self.__fecha=fecha  
  
    @property  
    def fecha(self):  
        return self.__fecha  
  
    @fecha.setter  
    def fecha(self, fecha):  
        self.__fecha = fecha
```

## >Objetos con Atributo datetime (fecha)

4. Crearemos un método para aplicar un formato personalizado a la fecha:

```
def fechaString(self):  
    date_string= datetime.strftime(self.__fecha,"%d/%m/%y")  
    return date_string
```

## >Objetos con Atributo datetime (fecha)

Creamos el objeto **datetime** e instanciamos nuestra **Persona**. Podemos probar el método para formatear el atributo de tipo fecha

```
from datetime import datetime
fecha= datetime(2022,10,12)
print("*** Fecha creada con datatime ***")
print(fecha)
p1= Persona("Ballesteros","Cristian",12345632,30,fecha)
informacion= p1
print(informacion)
print("*** Fecha del objeto llamando al metodo fechaString de la clase persona ***")
fechaCadena= p1.fechaString()
print(fechaCadena)
```

A person is holding a yellow sticky note with the word "CODE" written on it in blue ink. The person is smiling and pointing towards the camera. The background is a blurred image of a person's face and hands. The entire image has a green overlay.

CODE

🏠 web: <http://milprogramadores.unsa.com.ar>

📍 telegram: <https://t.me/milprogramadoressaltenios>

💖 centro de ayuda: <http://ayudamilprogramadores.com/>