# Taller Integrador – Python.

# Unidad 5. P.O.O.
# Programación orientada a objetos.

Ya es por todos conocido el paradigma de programación orientada a objetos, y en Python tenemos la posibilidad de implementarlo de manera sencilla.
Hasta acá hemos avanzado en el lenguaje y estamos en condiciones de escribir nuestros propios programas, eso sí, en forma estructurada y algo de funciones, lo que nos limita un poco a la hora de desarrollar.

Como ya se vio, la finalidad de la POO es resolver las necesidades de un sistema informático mediante las relaciones de sus objetos, posibilitando la sencillez del código y su reutilización.
La POO nos proporciona diferentes elementos y características, entendiendo por elementos a los materiales necesarios para diseñar y programar un sistema, mientras que las características serán las herramientas que nos permitirán trabajar con dichos materiales.
Resumiendo, la POO se caracteriza por ser:

- Más ágil.
- Más intuitiva.
- Más organizada.
- Más escalable.

Veremos que, en Python, casi todo es un objeto.

## Clases.

Antes de proseguir explicando la Programación Orientada a Objetos debemos tener más o menos claro la diferencia entre una clase y un objeto, como es normal, estos conceptos los afianzaremos por medio de la práctica, pero también es necesaria una base teórica para comprenderlo realmente.

Una clase es una plantilla para la creación de objetos según un modelo definido previamente.
Las clases se utilizan para la definición de atributos (variables) y métodos (funciones).
Las clases son los modelos sobre los cuales se construirán nuestros objetos.
Para definir una clase en Python, utilizamos la palabra reservada CLASS y el nombre que queremos darle a la clase.
Se acostumbre colocar la primera letra en mayúscula, aunque no es obligatorio.

```
class Persona:
    pass

class Cliente:
    pass
```

## Atributos.

Los atributos o propiedades de una clase son las características que pueden tener, por ejemplo, para la clase Cliente que pusimos, un atributo podría ser el nombre, otro el apellido y así con todo lo que necesitemos guardar de nuestro cliente.

```
class Cliente:
    nombre=''
    apellido=''
    domicilio=''
```

## Métodos.

Los métodos son funciones y representan acciones propias de la clase que solo pueden ser realizadas por los objetos.
Describen el comportamiento de los objetos de una clase. La ejecución de un método puede modificar los atributos del mismo.
Tienen el mismo formato que las funciones tradicionales que vimos hasta ahora, con la diferencia que se declaran dentro de la clase y siempre, el primer argumento hace referencia a la instancia que la llama.

Dentro de las clases, los métodos llevan siempre como primer atributo a SELF.
Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local de un atributo del objeto.
El método espera un argumento y Python, internamente, se lo está pasando.

## Objetos.

En Python los objetos se crean instanciando la clase de la siguiente manera:

```
# Clase
class Auto:
    color = 'sin datos'
    puertas = 'sin datos'
    combustible = 'sin datos'

    def pintar(self,pintura):
        self.color = pintura

# Objeto
vw = Auto()
fiat = Auto()
```

Lo que hicimos fue declarar la clase Auto con los atributos color, puertas y combustible con un valor por defecto, en nuestro caso el valor 'sin datos' y el método 'pintar' para poder asignarle un valor al atributo 'color'.
Para hacer referencia a un determinado atributo de la clase, dentro de los métodos, deberemos anteponer el prefijo SELF.
Una vez creado nuestro objeto, podemos acceder a sus atributos:

```
# Ver atributos
print(vw.color)
```

Veamos todos los atributos antes de ver cómo podemos modificarlos.

```
# Objeto
```

```
vw = Auto()
fiat = Auto()

# Ver atributos
print('VW Color: ',vw.color)
print('VW Puertas: ',vw.color)
print('VW Combustible: ',vw.color)
```

Esto vemos por pantalla:

VW Color:  sin datos
VW Puertas:  sin datos
VW Combustible:  sin datos

Así como accedemos a los atributos de un objeto, de similar manera podemos hacer uso de sus métodos, por ejemplo, para nuestra clase Auto, tenemos el método 'pintar' que se encarga de asignarle al atributo 'color' el valor que le pasemos como argumento al invocarlo.
Veamos un ejemplo:

```
# Objeto
vw = Auto()
fiat = Auto()

# Ver atributos
print('VW Color original: ',vw.color)
vw.pintar('azul')
print('VW nuevo color: ',vw.color)
```

Esto veremos por pantalla:

VW Color original:  sin datos
VW nuevo color:  azul

El primer valor de color que se muestra es el que viene por defecto con cada instancia de la clase, el segundo es el que cargamos mediante el uso del método 'pintar'.

Podemos probar de ver el atributo en el otro objeto creado, el que llamamos Fiat, y lo que veremos será el valor que trae por defecto al instanciar la clase.

*** Ejercicio propuesto 1.
Hacer una clase calculadora, con un método llamado "sumar" que reciba dos números y devuelva la suma de ambos.

*** Ejercicio propuesto 2.
Modificar la clase del ejercicio anterior para que reciba la operación a realizar, que puede ser suma, resta o producto y dos números con los cuáles hará dicha operación y muestre el resultado.

## Creación de atributos.

Una característica interesante de Python es que podemos crear atributos de un objeto en tiempo de ejecución.
Por ejemplo, vamos a crear una clase moto sin ningún método ni atributo.

```
# Clase
class Moto:
    pass
```

A continuación, instanciamos la clase, creando un objeto, y acto seguido, creamos dos atributos y les damos valores.

```
# instanciamos la clase.
honda = Moto()
honda.ruedas = 2
honda.color = 'verde'
```

Así de simple, sin declararlos en la clase, podemos tener nuestro objeto con sus atributos.
Podemos acceder a ellos como vimos anteriormente:

```
# Accedemos a los atributos.
print('La moto Honda tiene ',honda.ruedas,' ruedas.')
print('La moto Honda es de color ',honda.color)
```

Y esto deberíamos ver por pantalla:

```
La moto Honda tiene  2  ruedas.
La moto Honda es de color  verde
```

## Método INIT.

El método INIT es el constructor de nuestra clase y sirve, entre otras cosas, para inicializar los atributos de cada objeto creado.
Es el método que se ejecuta apenas creamos nuestro objeto.
La palabra INIT va entre cuatro guiones bajos y el parámetro será siempre SELF.
Creamos una clase Persona a la que le damos, por defecto, un valor inicial a sus atributos.

```python
# Clase
class Persona:
    ojos = 2

    def __init__(self):
        print('Hemos creado una persona')

# Instanciamos la clase.
jose = Persona()
```

Cada vez que instanciemos la clase, creando un objeto, el método INIT será invocado y el código que contiene, será ejecutado.
Un uso adecuado para este método es el de darle valores a los atributos de los objetos que son creados.

*** Ejercicio propuesto 3.
Crear una clase Persona con un método INIT que haga posible que, al instanciarla, es decir, al crear un objeto, nos solicite un valor para el atributo nombre, un valor para el atributo apellido y finalmente muestre un mensaje saludando, con nombre y apellido, a la persona recién creada.

## Método INIT con parámetros.

El método INIT admite el pasaje de argumentos si le especificamos que va a recibir parámetros, al declarar la clase.
Veamos un ejemplo, nuevamente con la clase Auto.

```
class Auto:
    color=''
    puertas=0

    def __init__(self,color,puertas):
        self.color=color
        self.puertas=puertas

# Creamos el objeto
vw = Auto('verde',2)

# Accedemos a los atributos
print('VW color: ',vw.color)
print('VW puertas: ',vw.puertas)
```

Al crear el objeto llamado "vw" también pasamos por parámetros, los valores de sus atributos, en este caso, el valor del atributo color y del atributo puertas.
Podemos modificar el código para hacerlo mas simple, ya que los atributos están en el método constructor, no es necesario declararlos fuera de él:

```
class Auto:
    def __init__(self,color,puertas):
        self.color=color
        self.puertas=puertas
```

## Método destructor.

Este tipo de método, así como el método INIT, se denominan métodos especiales.
Siempre se ejecutan, cuando creamos una instancia o cuando la eliminamos y lo que hacemos al especificar como debe ser el método, es sobrescribirlo.

Este método es el DEL y como todo método especial, va encerrado entre cuatro guiones bajos y también debe colocarse el parámetro SELF.
Veamos un ejemplo en el que creamos una clase Persona, con un método INIT que recibe el nombre como argumento y muestra por pantalla un mensaje avisando que el objeto fue creado.
A dicha clase le agregaremos un método DEL que nos muestre un mensaje cuando destruyamos cada objeto creado.

```
class Persona:
    def __init__(self,nombre):
        self.nombre=nombre

    def __del__(self):
        print('La persona ',self.nombre,' ha sido eliminada.')

jose = Persona('jose')
print('El nombre de la persona creada es ',jose.nombre)
del(jose)
```

Dijimos que ambos métodos se ejecutan (INIT al crear el objeto y DEL al eliminarlo) y nosotros podremos sobrescribirlos para que hagan determinadas acciones.
En el ejemplo anterior lo que hicimos fue modificar el destructor para que muestre un mensaje por pantalla cada vez que eliminamos un objeto creado con dicha clase

# Encapsulado.

Dentro de Python, no tenemos la posibilidad de realizar un encapsulamiento mediante herramientas del lenguaje, por lo que deberemos recurrir a diferentes técnicas.
Para poder declarar métodos y atributos privados, lo que hacemos es preceder el nombre con dos guiones bajos de la siguiente manera:

```
class Persona:
    __nombre = 'Atributo privado'

    def __saludar(self):
        print('Metodo privado')
```

Si intentamos acceder al atributo o al método definidos de esta manera, el intérprete nos devolverá un mensaje de error diciéndonos que dichos elementos no están definidos.

```
jose = Persona()
print(jose.__nombre)
jose.__saludar()
```

Para poder hacer uso de ambos, deberemos declarar un método puente que tome el atributo o el método privados y lo saque fuera de la clase.
Veamos como hacer métodos del tipo GET para poder hacer uso del atributo o del método en el objeto creado a partir de la clase Persona.

```
class Persona:
    __nombre = 'Atributo privado'

    def __saludar(self):
        print('Metodo privado')

    def get_nombre(self):
        return self.__nombre

    def get_saludar(self):
        return self.__saludar()

jose = Persona()
print(jose.get_nombre())
jose.get_saludar()
```

Ahora tenemos dos nuevos métodos, del tipo GET que nos permiten acceder al atributo y al método que declaramos de manera privada.

*** Ejercicio propuesto 4.
Modificar el ejemplo anterior, agregando un método del tipo SET que me permita modificar el valor del atributo __nombre y probarlo con un par de valores diferentes.

# Herencia.

Hay tres conceptos que son básicos para cualquier lenguaje de programación orientado a objetos: el encapsulamiento, la herencia y el polimorfismo.

En un lenguaje orientado a objetos cuando hacemos que una clase (subclase) herede de otra clase (superclase) estamos haciendo que la subclase contenga todos los atributos y métodos que tenía la superclase.

No obstante, al acto de heredar de una clase también se le llama a menudo "extender una clase".

La herencia en POO es la particularidad de crear nuevas clases a partir de clases existentes, tomando todos sus atributos y métodos tal como están con la posibilidad de sobrescribirlos o agregarles nuevos.

Para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis después del nombre de la clase

Veamos un ejemplo en el que tenemos una clase empleado que al ser instanciada me pide nombre y apellido del empleado, y una clase Gerente, que hereda de la clase Empleado.

Dentro de la clase Gerente, vamos a declarar el método INIT y dentro del mismo vamos a invocar el INIT de la clase padre mediante la función SUPER.

La función incorporada super() permite invocar un método de una clase padre desde una clase hija.

```
class Empleado():
    def __init__(self):
        self.nombre=input('Nombre? : ')
        self.apellido=input('Apellido? : ')

class Gerente(Empleado):
    def __init__(self):
        super().__init__()
        self.gerencia=input('Ingrese gerencia: ')
```

A partir de la clase Empleado, creamos la clase Gerente, pero queremos agregar una modificación al método INIT para que me permita ingresar un atributo llamado GERENCIA.
Si no fuese necesaria dicha modificación, no sería necesario recurrir a la función SUPER.
Cuando instanciamos la clase Empleado, estamos invocando su método constructor que solicita el ingreso por teclado del nombre y apellido del empleado a crear.
Cuando instanciamos la clase Gerente, lo primero que hace es llamar, mediante la función SUPER al constructor de la clase Empleado (solicitando nombre y apellido) pero, como este llamado está dentro de su propio método constructor, podemos sobrescribirlo y agregar nuevas funcionalidades, como, por ejemplo, en nuestro caso, solicitar el nombre de la gerencia a la cual pertenece el nuevo gerente.

De haber querido hacer solo una herencia simple, hubiera alcanzado con declarar la clase Gerente de la siguiente manera:

```
class Gerente(Empleado):
    pass
```

## Método especial STR.

Si le pasamos un objeto como argumento a la función PRINT, veremos por pantalla una referencia poco legible y sin sentido del objeto.

```
class Empleado():
    def __init__(self):
        self.nombre=input('Nombre? : ')
        self.apellido=input('Apellido? : ')

empleado_uno = Empleado()
print(empleado_uno)
```

Esto vemos por pantalla:

```
Nombre? : Guillermo
Apellido? : Alfaro
<__main__.Empleado object at 0x03C9FFF0>
```

Podemos modificarlo si sobrescribimos el método especial STR, veamos un ejemplo.

```
class Empleado():
    def __init__(self):
        self.nombre=input('Nombre? : ')
        self.apellido=input('Apellido? : ')

    def __str__(self):
        mensaje = self.apellido+' , '+self.nombre
        return mensaje

empleado_uno = Empleado()
print(empleado_uno)
```

En el código anterior, a nuestra clase Empleado le sobrescribimos el método STR para que devuelva una cadena de texto compuesta por el apellido y nombre que se cargue en el objeto creado al instanciar la clase.
Vemos que al utilizar la función PRINT con el objeto creado, ya no vemos el anterior mensaje, ahora vemos el mensaje con el formato definido en nuestro método especial:

```
Nombre? : Guillermo
Apellido? : Alfaro
Alfaro , Guillermo
```

Podemos definir y formatear la salida del método especial STR para que nos de información precisa acerca de cada objeto.
Si utilizamos objetos para manejar datos de cada empleado, cliente o usuario de nuestro sistema, podemos establecer un formato específico para mostrar el contenido de cada objeto.

*** Ejercicio propuesto 5.
Crear una clase Alumno que al instanciarse pida nombre, apellido y nota final de un alumno.
Sobrescribir el método especial STR para que, al imprimir los objetos, se vea por pantalla el nombre, apellido, nota final y un mensaje (‘Promoción’

si la nota es igual o mayor a 7, 'Final' si la nota es mayor o igual a 4 y menor a 7 y 'Recursa' si la nota es menor a 4).
Instanciar la clase y probar el método.

## Herencia múltiple.

En Python, a diferencia de otros lenguajes como Java o C#, se permite la herencia múltiple, es decir, una clase puede heredar de varias clases a la vez.
Basta con enumerar las clases de las que se hereda separándolas por comas.
Veamos un ejemplo sencillo en el que creamos las clases Promocion_a y Promocion_ b, luego crearemos una clase llamada Cliente que heredará a ambas clases.

```
class Promocion_a():
    descuento_a = 10

class Promocion_b():
    descuento_b = 15

class Cliente(Promocion_a,Promocion_b):
    pass

alfaro = Cliente()
print(alfaro.descuento_a)
print(alfaro.descuento_b)
```

Podemos ver por pantalla que al instanciar la clase Cliente, esta hereda el valor de los atributos de las clases Promocion_a y Promocion_b.

## Método especial DOC.

El método especial DOC nos permite agregar una sencilla documentación a las clases que vamos creando mediante el agregado de un comentario descriptivo en la primera línea del código de una clase.

Por ejemplo, creamos la clase llamada Cliente y queremos que la misma, contenga una descripción de los atributos y métodos que la componen, para que este disponible entre quienes la utilicen.
Esta primera línea de código es un texto que va encerrado entre dos grupos de tres comillas dobles cada uno.

```
class Cliente():
    """
    Esta clase permite crear nuevos clientes.
    Sus atributos son: Nombre / Apellido / DNI
    Tiene un metodo constructor que informa cada vez que la
    clase es instanciada.
    Tiene un metodo destructor que informa cada vez que un
    objeto es destruido.
    """
    pass

print(Cliente.__doc__)
```

Y con esto podremos ver la descripción completa que el programador de la clase dejó incorporada a esta:

```
Esta clase permite crear nuevos clientes.
Sus atributos son: Nombre / Apellido / DNI
Tiene un metodo constructor que informa cada vez que la
clase es instanciada.
Tiene un metodo destructor que informa cada vez que un
objeto es destruido.
```